Библиотека профессионала

Java™ 2

Том 1. Основы

ВОСЬМОЕ ИЗДАНИЕ

Java_2_title.indd 1 15.02.2008 12:57:21

Core Java

Volume I – Fundamentals

Eighth Edition

Cay S. Horstmann
Gary Cornell



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

Java_2_title.indd 2 15.02.2008 12:57:21

Библиотека профессионала

Java™ 2

Том 1. Основы

ВОСЬМОЕ ИЗДАНИЕ

Кей Хорстманн • Гари Корнелл



Москва • Санкт-Петербург • Киев
2013

Java_2_title.indd 3 15.02.2008 12:57:22

ББК 32.973.26-018.2.75
Х82
УДК 681.3.07

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

Хорстманн, Кей **С., Корнелл**, Гари.

Х82 Java 2. Библиотека профессионала, том 1. Основы. 8-е издание. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2013. — 816 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1378-4 (рус.)

Книга ведущих специалистов по программированию на языке Java представляет собой обновленное издание фундаментального труда, учитывающее всю специфику новой версии платформы Java SE 6. Подробно рассматриваются такие темы, как организация и настройка среды программирования на Java, основные структуры данных, объектно-ориентированное программирование и его реализация в Java, интерфейсы, программирование графики, обработка событий, Swing, развертывание приложений и аплетов, отладка, обобщенное программирование, коллекции и построение многопоточных приложений. Книга изобилует множеством примеров, которые не только иллюстрируют концепции, но также демонстрируют способы правильной разработки, применяемые в реальных условиях.

Книга рассчитана на программистов разной квалификации, а также будет полезна студентам и преподавателям дисциплин, связанных с программированием на языке Java.

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

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

Authorized translation from the English language edition published by Prentice Hall, Ptr., Copyright © 2008 Sun Microsystems, Inc.

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

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2013

ISBN 978-5-8459-1378-4 (рус.)
ISBN 978-0-1323-5476-9 (англ.)

© Издательский дом "Вильямс", 2013
© Sun Microsystems, Inc., 2008

Оглавление

Предисловие 16
Глава 1. Введение в язык Java 21
Глава 2. Среда программирования Java 36
Глава 3. Основные конструкции языка Java 56
Глава 4. Объекты и классы 125
Глава 5. Наследование 189
Глава 6. Интерфейсы и вложенные классы 259
Глава 7. Программирование графики 299
Глава 8. Обработка событий 339
Глава 9. Компоненты пользовательского интерфейса Swing 375
Глава 10. Развертывание приложений и аплетов 502
Глава 11. Исключения, протоколирование, утверждения и отладка 558
Глава 12. Обобщенное программирование 619
Глава 13. Коллекции 652
Глава 14. Многопоточность 716
Приложение. Ключевые слова Java 805
Предметный указатель 807

Содержание

Об авторах 14
Благодарности 14

Предисловие 16
Краткий тур по книге 17
Соглашения 19
Коды примеров 20
От издательства 20

Глава 1. Введение в язык Java 21
Программная платформа Java 22
Характерные особенности языка Java 22
Простота 23
Поддержка объектов 24
Поддержка распределенных вычислений 24
Надежность 24
Безопасность 25
Независимость от архитектуры компьютера 26
Переносимость 26
Интепретируемость 27
Производительность 27
Многопоточность 28
Динамичность 28
Аплеты Java и Internet 28
Краткая история языка Java 30
Распространенные заблуждения относительно языка Java 32

Глава 2. Среда программирования Java 36
Инсталляция Java Development Kit 37
Где найти JDK 37
Установка пути к исполняемым файлам 39
Инсталляция библиотек и документации 41
Установка примеров программ 42
Навигация по каталогам 42
Выбор среды разработки программ 43
Использование инструментов командной строки 43
Возможные ошибки 45
Использование интегрированной среды разработки 46
Сообщения об ошибках компиляции 48
Выполнение графического приложения 50
Создание и запуск аплетов 52

Глава 3. Основные конструкции языка Java 56
Простая программа на языке Java 57
Комментарии 60
Типы данных 61
Целые числа 61
Числа с плавающей точкой 62
Тип `char` 63

Тип boolean 65
Переменные 65
Инициализация переменных 66
Константы 67
Операции 68
Операции инкрементирования и декрементирования 69
Операции отношений и логические операции 69
Битовые операции 70
Математические функции и константы 71
Преобразование числовых типов 72
Приведение числовых типов 73
Скобки и иерархия операций 73
Перечислимые типы 74
Строки 75
Подстроки 75
Конкатенация 75
Неизменность строк 76
Проверка эквивалентности строк 77
Кодовые точки и кодовые единицы 78
API класса String 79
Интерактивная документация по API 81
Построение строк 84
Ввод и вывод 85
Чтение входных данных 85
Форматирование выходных данных 87
Файловый ввод и вывод 92
Поток управления 94
Блоки 94
Условные выражения 95
Неопределенные циклы 98
Определенные циклы 101
Многовариантное ветвление — оператор switch 104
Прерывание потока управления 106
Большие числа 109
Массивы 111
Цикл “for each” 112
Инициализация массивов и анонимные массивы 113
Копирование массивов 114
Параметры командной строки 115
Сортировка массива 116
Многомерные массивы 119
“Зубчатые” массивы 122

Глава 4. Объекты и классы 125
Введение в объектно-ориентированное программирование 126
Классы 127
Объекты 128
Идентификация классов 128
Отношения между классами 129
Использование готовых классов 131
Объекты и объектные переменные 131

Класс GregorianCalendar из библиотеки Java 134
Модифицирующие методы и методы доступа 136
Определение собственных классов 142
Класс Employee 142
Использование нескольких исходных файлов 145
Анализ класса Employee 146
Первые шаги с конструкторами 146
Явные и неявные параметры 147
Преимущества инкапсуляции 148
Доступ к данным из различных экземпляров класса 150
Приватные методы 151
Неизменяемые поля экземпляра 152
Статические поля и методы 152
Статические поля 152
Статические константы 153
Статические методы 154
Методы-фабрики 155
Метод main() 155
Параметры методов 158
Конструирование объектов 163
Перегрузка 164
Инициализация полей по умолчанию 164
Конструктор по умолчанию 165
Явная инициализация полей 165
Имена параметров 166
Вызов одного конструктора из другого 167
Инициализационные блоки 167
Уничтожение объекта и метод finalize() 172
Пакеты 172
Импортирование классов 173
Статический импорт 174
Добавление классов в пакеты 175
Область видимости пакета 177
Путь поиска классов 179
Установка пути к классам 181
Комментарии и документирование 181
Включение комментариев 182
Комментарии к классу 182
Комментарии к методам 183
Комментарии к полям 184
Комментарии общего характера 184
Пакетные и обзорные комментарии 185
Извлечение комментариев 185
Рекомендации по проектированию классов 186

Глава 5. Наследование 189

Классы, суперклассы и подклассы 190
Иерархия наследования 196
Полиморфизм 197
Динамическое связывание 198

Предотвращение наследования: финальные классы и методы 201
Приведение типов 202
Абстрактные классы 204
Защищенный доступ 209
`Object`: глобальный суперкласс 210
Метод `equals()` 210
Проверка эквивалентности объектов и наследование 211
Метод `hashCode()` 214
Метод `toString()` 216
Обобщенные массивы-списки 222
Доступ к элементам списочных массивов 224
Совместимость между типизированными и “сырыми” списочными массивами 228
Объектные оболочки и автоупаковка 229
Методы с переменным числом параметров 232
Классы-перечисления 233
Рефлексия 235
Класс `Class` 236
Основы обработки исключений 238
Использование рефлексии для анализа свойств классов 239
Использование рефлексии для анализа объектов во время выполнения 244
Использование рефлексии для написания кода обобщенного массива 249
Указатели на методы! 253
Рекомендации по применению наследования 256

Глава 6. Интерфейсы и вложенные классы 259

Интерфейсы 260
Свойства интерфейсов 265
Интерфейсы и абстрактные классы 267
Клонирование объектов 268
Интерфейсы и обратный вызов 274
Вложенные классы 276
Использование вложенных классов для доступа к состоянию объекта 278
Специальные синтаксические правила для работы с вложенными классами 281
Чем полезны вложенные классы, нужны ли они и насколько они безопасны 282
Локальные вложенные классы 285
Доступ к финальным переменным из внешних методов 285
Анонимные вложенные классы 288
Статические вложенные классы 290
Прокси-классы 293
Свойства прокси-классов 297

Глава 7. Программирование графики 299

Общие сведения о пакете Swing 300
Создание фрейма 304
Позиционирование фрейма 306
Определение подходящего размера фрейма 308
Отображение информации в компоненте 312
Двумерные фигуры 316
Работа с цветом 324
Использование шрифтов при выводе текста 327
Вывод графических изображений 335

Глава 8. Обработка событий 339
Общие сведения об обработке событий 340
Пример: обработка щелчка на кнопке 342
Удобство вложенных классов 346
Создание слушателей, содержащих единственный вызов метода 349
Пример: изменение визуального стиля 350
Классы-адаптеры 353
Действия 357
События мыши 364
Иерархия событий AWT 372
Семантические и низкоуровневые события в библиотеке AWT 373

Глава 9. Компоненты пользовательского интерфейса Swing 375
Swing и шаблон проектирования "Модель–представление–контроллер" 376
Шаблоны проектирования 376
Шаблон "Модель-представление-контроллер" 377
Кнопки Swing с точки зрения шаблона "Модель–представление–контроллер" 381
Общие сведения о компоновке элементов 382
Диспетчер компоновки `BorderLayout` 384
Диспетчер компоновки `GridLayout` 386
Ввод текста 390
Текстовые поля 391
Метки и маркирующие компоненты 393
Поля для ввода пароля 394
Текстовые области 395
Панели прокрутки 395
Компоненты для выбора вариантов 398
Флажки опций 398
Переключатели 401
Рамки 405
Выпадающие списки 409
Линейные регуляторы 412
Меню 418
Построение меню 419
Пиктограммы в пунктах меню 421
Флажки опций и переключатели в меню 422
Всплывающие меню 423
Клавиши быстрого доступа и клавиатурные сокращения 425
Блокирование и разблокирование пунктов меню 427
Панели инструментов 431
Всплывающие подсказки 433
Расширенные средства компоновки 435
Диспетчер компоновки `GridBagLayout` 437
Альтернативные способы указания параметров `gridx`, `gridy`, `gridwidth` и `gridheight` 440
Диспетчер компоновки `GroupLayout` 447
Компоновка без использования диспетчера 456
Диспетчер компоновки, определяемый разработчиком 457
Порядок обхода компонентов 461
Диалоговые окна 463
Диалоговые окна для выбора опций 463

Создание диалоговых окон 473
Передача данных 478
Файловые диалоги 485
Диалоговое окно для выбора цвета 496

Глава 10. Развертывание приложений и аплетов 502

Файлы JAR 503
Файл манифеста 504
Исполняемые JAR-файлы 505
Ресурсы 506
Герметизация пакета 509
Технология Java Web Start 510
"Песочница" 513
Подписанный код 514
JNLP API 516
Аплеты 524
Простой аплет 525
HTML-дескрипторы и атрибуты для работы с аплетами 529
Дескриптор `<object>` 532
Передача информации аплетам через параметры 532
Обращение к файлам изображений и звуковым файлам 538
Контекст аплета 539
Хранение параметров настройки 547
Карты свойств 547
Интерфейс Preferences API 552

Глава 11. Исключения, протоколирование, утверждения и отладка 558

Обработка ошибок 559
Классификация исключений 561
Объявление контролируемых исключений 562
Как сгенерировать исключение 564
Создание классов исключений 565
Перехват исключений 566
Перехват нескольких исключений 568
Повторная генерация исключения 569
Блок `finally` 570
Анализ элементов трассировки стека 573
Как использовать исключения 575
Использование диагностических утверждений 578
Включение и отключение диагностических утверждений 579
Использование диагностических утверждений для проверки параметров 580
Использование утверждений для документирования предположений 581
Протоколирование 582
Базовое протоколирование 583
Расширенное протоколирование 583
Настройка диспетчера протоколирования 585
Локализация 586
Обработчики 587
Фильтры 591
Средства форматирования 591
"Рецепт" протоколирования 591
Приемы отладки 599

Использование консольного окна 605
Трассировка событий AWT 606
Позволим роботу AWT выполнять его работу 610
Применение отладчика 614

Глава 12. Обобщенное программирование 619

Назначение обобщенного программирования 620
Кто хочет быть обобщенным программистом? 621
Определение простого обобщенного класса 622
Обобщенные методы 624
Ограничения переменных типов 625
Обобщенный код и виртуальная машина 627
Трансляция обобщенных выражений 628
Трансляция обобщенных методов 629
Вызов унаследованного кода 630
Ограничения и лимиты 632
Параметры типа не могут принимать примитивные типы 632
Исследование типов времени выполнения работает только с “сырыми” типами 632
Генерировать и перехватывать исключения-экземпляры обобщенного класса нельзя 632
Массивы параметризованных типов не разрешены 633
Нельзя создавать экземпляры переменных типов 633
Переменные типа в статическом контексте обобщенных классов не разрешены 635
Остерегайтесь конфликтов после подчистки 635
Правила наследования обобщенных типов 636
Подстановочные типы 638
Ограничения супертипа для подстановок 639
Неограниченные подстановки 641
Захват подстановок 642
Рефлексия и обобщения 645
Использование параметров `Class<T>` для сравнения типов 646
Информация об обобщенных типах в виртуальной машине 647

Глава 13. Коллекции 652

Интерфейсы коллекций 653
Отделение интерфейса от реализации коллекций 653
Интерфейсы `Collection` и `Iterator` из библиотеки Java 656
Конкретные коллекции 661
Специализированные классы `Set` и `Map` 686
Каркас коллекций 690
Представления и “оболочки” 694
Пакетные операции 700
Преобразования между коллекциями и массивами 701
Алгоритмы 702
Сортировка и перетасовка 703
Бинарный поиск 705
Простые алгоритмы 706
Написание собственных алгоритмов 708
Унаследованные коллекции 709
Класс `Hashtable` 709
Интерфейс `Enumeration` 709
Карты свойств 710

Стеки 711
Битовые наборы 711

Глава 14. Многопоточность 716
Что такое потоки? 717
Использование потоков для предоставления шанса другим задачам 722
Прерывание потоков 727
Состояния потока 730
Новые потоки 730
Работоспособные потоки 730
Блокированные и ожидающие потоки 731
Завершенные потоки 731
Свойства потока 733
Приоритеты потоков 733
Потоки-демоны 734
Обработчики неперехваченных исключений 734
Синхронизация 736
Пример состояния гонок 736
Объяснение состояния гонок 740
Блокирующие объекты 742
Объекты условий 745
Ключевое слово `synchronized` 749
Синхронизированные блоки 753
Концепция монитора 754
Поля `volatile` 755
Взаимные блокировки 756
Тестирование блокировок и таймауты 759
Блокировки чтения/записи 760
Почему методы `stop()` и `suspend()` не рекомендованы к применению 761
Блокирующие очереди 763
Коллекции, безопасные к потокам 770
Эффективные карты, наборы и очереди 770
Копирующиеся при записи массивы 772
Устаревшие безопасные к потокам коллекции 772
Интерфейсы `Callable` и `Future` 773
Класс `Executors` 777
Пулы потоков 778
Плановое выполнение 782
Управление группами задач 782
Синхронизаторы 784
Семафоры 784
Защелки с обратным отсчетом 785
Барьеры 785
Обменники 786
Синхронные очереди 786
Пример: приостановка и возобновление анимации 786
Потоки и Swing 792
Выполнение задач, требующих много времени 793
Использование класса `SwingWorker` 797
Правило единственного потока 803

Приложение. Ключевые слова Java 805

Предметный указатель 807

Об авторах

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

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

Благодарности

Написание книги всегда требует титанических усилий, причем переписывание не намного легче, особенно учитывая постоянные изменения технологии Java. Чтобы сделать книгу полезной, необходимы совместные усилия многих преданных делу людей, и мы с удовольствием выражаем признательность всей команде за их вклад в наше общее дело.

Огромное количество людей из Prentice Hall и Sun Microsystems Press оказали неоценимую помощь, хотя и остались в тени. Я хотел бы выразить им свою признательность за их усилия. Как всегда, мою самую горячую благодарность заслуживает редактор Prentice Hall Грег Доенч (Greg Doench) — за сопровождение книги на протяжении всего процесса ее написания и издания, а также за то, что позволил мне пребывать в блаженном неведении о многих деталях этого процесса, происходящих за сценой. Я благодарен Ванессе Мур (Vanessa Moore) за блестящую продуктивную поддержку. Также приношу свои благодарности моему соавтору по ранним изданиям Гари Корнеллу (Gary Cornell), который с тех пор перешел на другое предприятие.

Благодарю многих и многих читателей ранних изданий, которые сообщали о найденных ошибках и внесли массу ценных предложений по улучшению книги. Я особенно благодарен блестящей команде рецензентов, которые тщательно прошлись по рукописи и избавили ее от многих неприятных ошибок.

Среди рецензентов этого и предыдущих изданий хочу отметить Чака Аллисона (Chuck Allison, выпускающий редактор *C/C++ Users Journal*), Алека Битона (Alec Beaton, PointBase, Inc.), Клиффа Берга (Cliff Berg, iSavvix Corporation), Джошуа Блоха (Joshua Bloch, Sun Microsystems), Дэвида Брауна (David Brown), Корки Картрайта (Corky Cartwright), Френка Коена (Frank Cohen, PushToTest), Криса Крейна (Chris Crane, devXsolution), доктора Николаса Дж. Де Лилло (Dr. Nicholas J. De Lillo, Manhattan College), Ракеша Дхупара (Rakesh Dhoopar, Oracle), David Geary (David Geary, Sabreware), Брайана Гоетца (Brian Goetz, ведущий консультант в Quiotix Corp.), Анжелу Гордон (Angela Gordon, Sun Microsystems), Дэна Гордона (Dan Gordon, Sun Microsystems), Роба Гордона (Rob Gordon), Джона Грэя (John Gray, University of Hartford), Камерон Грегори (Cameron Gregory, olabs.com), Марти Холл (Marty Hall, The Johns Hopkins University Applied Physics Lab), Винсента Харди (Vincent Hardy, Sun Microsystems), Дэна Харки (Dan Harkey, San Jose State University), Вильяма Хиггинса (William Higgins, IBM), Владимира Ивановича (Vladimir Ivanovic, PointBase), Джерри Джексона (Jerry Jackson, ChannelPoint Software), Тима Киммета (Tim Kimmet, Preview

Systems), Криса Лаффра (Chris Laffra), Чарли Лаи (Charlie Lai, Sun Microsystems), Анжелику Лангер (Angelika Langer), Дуга Лэнгстона (Doug Langston), Ханг Лау (Hang Lau, McGill University), Марка Лоуренса (Mark Lawrence), Дуга Ли (Doug Lea, SUNY Oswego), Грегори Лонгшора (Gregory Longshore), Боба Линча (Bob Lynch, Lynch Associates), Филиппа Милна (Philip Milne, консультант), Марка Моррисси (Mark Morrissey, The Oregon Graduate Institute), Махеш Нилаканта (Mahesh Neelakanta, Florida Atlantic University), Хао Фам (Hao Pham), Пола Филиона (Paul Philion), Блейка Рагсдейла (Blake Ragsdell), Стюарта Реджеса (Stuart Reges, University of Arizona), Рича Росена (Rich Rosen, Interactive Data Corporation), Питера Сандерса (Peter Sanders, ESSI University, Ницца, Франция), доктора Пола Сангеру (Dr. Paul Sanghera, San Jose State University and Brooks College), Поля Сэвинка (Paul Sevinc, Teamup AG), Деванг Ша (Devang Shah, Sun Microsystems), Бредли А. Смита (Bradley A. Smith), Стивена Стелтинга (Steven Stelting, Sun Microsystems), Кристофера Тэйлора (Christopher Taylor), Люка Тэйлора (Luke Taylor, Valtech), Джорджа Тхируватукала (George Thiruvathukal), Кима Топли (Kim Topley, автор Core JFC), Джанет Трауб (Janet Traub), Пола Тиму (Paul Tyma консультант), Питера Ван Дер Линдена (Peter van der Linden, Sun Microsystems) и Берта Велша (Burt Walsh).

Кей Хорстманн (Cay Horstmann)
Сан-Франциско, 2007 г.

ПРЕДИСЛОВИЕ

В конце 1995 г. язык программирования Java вырвался на Internet-сцену и моментально завоевал популярность. Технология Java обещала стать *универсальным клеем*, соединяющим пользователей с информацией, откуда бы эта информация ни поступала — от Web-серверов, из баз данных, поставщиков информации, либо любого другого источника, который только можно было себе вообразить. И на самом деле, Java обладает всеми возможностями соответствовать этим обещаниям. Это исключительно основательно спроектированный язык, который получил признания всех основных участников рынка за исключением Microsoft. Его встроенные средства защиты и безопасности обнадежили как программистов, так и пользователей Java-программ. Java изначально обладала встроенной поддержкой таких сложных задач, как сетевое программирование, подключаемость к базам данных и многопоточность, обеспечивая простоту реализации всех этих вещей.

С 1995 г. Sun Microsystems выпустила семь главных версий Java Development Kit. За последние 11 лет интерфейс прикладного программирования (API) вырос с 200 до 3000 классов. Теперь API охватывает такие различные области, как конструирование пользовательских интерфейсов, управление базами данных, интернационализация, безопасность и обработка XML.

Книга, которую вы держите в руках — это первый том восьмого издания. С выходом каждого издания мы старались как можно быстрее следовать очередному выпуску Java Development Kit, каждый раз переписывая ее, чтобы вы могли воспользоваться преимуществами новейших средств Java. Настоящее издание отражает средства, имеющиеся в составе Java Standard Edition (SE) 6.

Как и все предыдущие издания этой книги, мы по-прежнему *ориентируем ее на аудиторию серьезных программистов*, которые хотят использовать Java для разработки реальных проектов. Мы воспринимаем вас, дорогой читатель, как программиста с солидным опытом в языках программирования, отличных от Java, и надеемся, что вы не любите книг, полных игрушечных примеров (вроде программ управления тостерами или животными в зоопарке либо "прыгающим текстом"). Ничего подобного в этой книге вы не найдете. Наша цель — достижение полного понимания языка Java и его библиотеки, а не создание иллюзии такого понимания.

В книге вы найдете массу примеров кода, демонстрирующих почти все обсуждаемые языковые и библиотечные средства. Эти примеры мы намеренно сделали насколько возможно простыми, чтобы сосредоточиться на основных моментах, но, тем не менее, в большинстве своем, они вовсе не игрушечные и не "срезают острых углов". Все они могут стать хорошей отправной точкой для разработки вашего собственного кода.

Мы предполагаем, что вы хотите, а то и жаждете узнать обо всех расширенных средствах, которые предлагает Java вашему вниманию. Так, например, мы предлагаем детальное рассмотрение следующих тем.

- Объектно-ориентированное программирование.
- Рефлексия и прокси-классы.
- Интерфейсы и вложенные классы.

- Модель слушателей событий.
- Проектирование графического интерфейса пользователя с использованием инструментария Swing.
- Обработка исключений.
- Обобщенное программирование.
- Каркас коллекций.
- Параллелизм.

В связи со стремительным ростом библиотеки классов Java, одного тома оказалось недостаточно для описания всех средств Java, о которых следует знать серьезным программистам. Первый том, который вы держите в руках, сосредоточен на фундаментальных концепциях языка Java, наряду с основами программирования пользовательского интерфейса. Второй том посвящен средствам разработки приложений масштаба предприятия и усовершенствованному программированию пользовательских интерфейсов. В нем вы найдете детальное обсуждение следующих тем.

- Файлы и потоки.
- Распределенные объекты.
- Базы данных.
- Расширенные компоненты графического пользовательского интерфейса.
- "Родные" (native) методы.
- Обработка XML.
- Сетевое программирование.
- Расширенная графика.
- Интернационализация.
- JavaBeans.
- Аннотации.

В этом издании мы перегруппировали содержимое двух томов. В частности, многопоточность теперь рассматривается в первом томе, поскольку ее значение существенно возросло.

При написании книги ошибки и неточности неизбежны. Мы очень заинтересованы в том, чтобы знать о них. Но, конечно же, мы бы предпочли узнавать о каждой из них только однажды. Поэтому мы поместили список часто задаваемых вопросов, исправлений ошибок и "обходных путей" по адресу `http://horstmann.com/corejava`.

Краткий тур по книге

Глава 1 предоставляет обзор возможностей Java, которые отличают его от других языков программирования. Мы объясним, что было задумано проектировщиками языка, и в какой мере им удалось воплотить задуманное. Затем приведем краткую историю того, как развивался язык Java, и как он стал тем, чем он есть сегодня.

В **главе 2** мы расскажем о том, как загрузить и инсталлировать JDK, а также примеры программ для книги. Затем мы проведем вас через компиляцию и запуск трех типичных программ Java — консольного приложения, графического приложения и аплета — все это с использованием только JDK, специального Java-ориентированного текстового редактора и Java IDE.

В **главе 3** начинается дискуссия о языке программирования Java. В этой главе мы раскроем основы: переменные, циклы и простые функции. Если вы — программист на C или C++, для вас все должно пройти гладко, поскольку синтаксис этих языковых средств по существу очень похож. Если вы обладаете опытом в языке, не похожем на C, таком как Visual Basic, вам следует прочесть эту главу особенно внимательно.

Объектно-ориентированное программирование (ООП) в наши дни — доминирующая программистская практика, и Java — в полной мере объектно-ориентированный язык. В **главе 4** представлена инкапсуляция — первый из двух фундаментальных строительных блоков объектной ориентации, а также механизмы Java, реализующие ее — классы и методы. В дополнение к правилам языка Java мы также приводим советы по правильному объектно-ориентированному дизайну. И, наконец, мы опишем изумительный инструмент `javadoc`, который форматирует комментарии из вашего кода в набор Web-страниц с перекрестными ссылками. Если вы знакомы с C++, то можете лишь бегло просмотреть эту главу. Программисты, пришедшие из мира не объектно-ориентированных языков, должны потратить больше времени для овладения концепциями ООП, прежде чем дальше изучать Java.

Классы и инкапсуляция — это лишь часть идеологии ООП, и потому **глава 5** представляет еще один ее краеугольный камень — *наследование*. Наследование позволяет взять существующий класс и модифицировать его в соответствии с вашими потребностями. Это — фундаментальная техника в программировании на Java. Механизм наследования в Java довольно похож на тот же механизм в C++. Опять-таки, программисты на C++ могут сосредоточиться лишь на отличиях между языками.

В **главе 6** будет показано, как использовать в Java понятие *интерфейса*. Интерфейсы позволяют вам выйти за пределы простого наследования, описанного в главе 5. Овладение интерфейсами позволит вам получить полный доступ ко всей мощи объектно-ориентированного подхода Java к программированию. Мы также раскроем удобное техническое средство Java, называемое *встроенными классами*. Встроенные классы помогают сделать ваш код яснее и более согласованным.

С **главы 7** начинается серьезное прикладное программирование. Каждый программист Java должен знать хоть немного о программировании графических пользовательских интерфейсов, и здесь вы найдете его основы. Мы покажем, как создаются окна, как выполняется рисование в них, как рисуются геометрические фигуры, как форматируется текст с применением множества шрифтов и как отображаются графические изображения.

Глава 8 содержит детальную дискуссию о модели событий AWT — *абстрактного оконного инструментария*. Вы увидите, как писать код, реагирующий на такие события, как щелчки кнопками мыши или нажатия клавиш. По ходу дела вы ознакомитесь с обработкой базовых элементов графического пользовательского интерфейса, таких как кнопки и панели.

Глава 9 посвящена более детальному обсуждению инструментария Swing. Пакет инструментов Swing позволяет строить межплатформенный графический интерфейс пользователя. Вы познакомитесь с различными типами кнопок, текстовых компонентов, рамок, ползунков, окон списков, меню и диалоговых окон. Однако некоторые из наиболее совершенных компонентов мы отложим до второго тома.

В **главе 10** будет показано, как развертывать ваши программы — будь то приложения или аплеты. Мы опишем, как пакетировать программы в файлы JAR, и как доставлять приложения через Internet с применением Java Web Start и механизмов аплетов. И, наконец, мы объясним, как программы Java могут сохранять и извлекать конфигурационную информацию после своего развертывания.

Глава 11 посвящена *обработке исключений* — устойчивому механизму Java, предназначенному противостоять тому факту, что плохие вещи случаются и с хорошими программами. Исключения обеспечивают эффективный способ отделения кода нормальной обработки от кода обработки ошибок. Конечно, даже после укрепления вашей программы проверкой всех возможных исключительных ситуаций, может произойти сбой в ожидаемой работе. Во второй половине этой главы мы предоставим огромное количество полезных советов по организации отладки. И, наконец, приведем пример полного сценария отладки.

В **главе 12** представлен обзор *обобщенного программирования* — главного преимущества Java SE 5.0. Обобщенное программирование делает ваши программы легче читаемыми и более безопасными. Мы покажем, как используется строгая типизация, и исключаются необходимость в некрасивых и небезопасных приведениях типа, а также как справиться со сложностями, происходящими из-за необходимости сохранения совместимости с предыдущими версиями Java.

Тема **главы 13** — каркас коллекций платформы Java. Всякий раз, когда вы хотите собирать множество объектов и извлекать их позднее, вам понадобится использовать коллекцию, которая наилучшим образом подходит к вашим конкретным условиям, вместо того, чтобы сбрасывать их в обычный массив. В этой главе будет продемонстрировано, какие преимущества дают стандартные коллекции, которые заранее подготовлены для вас.

Глава 14 завершает книгу обсуждением многопоточности, которая позволяет программировать выполняемые параллельно задачи. (Под потоком понимается поток управления внутри программы.) Мы покажем, как создаются потоки и как осуществляется их синхронизация. Средства многопоточности существенно изменились после Java SE 5.0, и мы расскажем вам о новых ее механизмах.

В **приложении** вы найдете зарезервированные слова языка Java.

Соглашения

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



На заметку!
Такой пиктограммой выделяются замечания.



Совет.
Такой пиктограммой выделяются советы.



Внимание!
Такой пиктограммой выделяются предупреждения об опасности.



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

API **Интерфейс прикладного программирования**

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

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

Листинг 1.1. Содержимое файла `WellcomeApplet.java`

Коды примеров

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

От издательства

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

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

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

Наши координаты:

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

Информация для писем из:

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

Глава 1

ВВЕДЕНИЕ В ЯЗЫК JAVA

В ЭТОЙ ГЛАВЕ...

- Программная платформа Java
- Характерные особенности языка Java
- Аплеты Java и Internet
- Краткая история языка Java
- Распространенные заблуждения относительно языка Java

На появление первой версии Java в 1996 г. откликнулись не только специализированные компьютерные газеты и журналы, но даже такие издания, как *The New York Times*, *The Washington Post* и *Business Week*. Java — единственный язык программирования, удостоившийся десятиминутного репортажа на National Public Radio. Специально для поддержки продуктов, основанных на применении *специфического* языка программирования, был создан венчурный фонд в 100 млн. долларов. Это было удивительное время. Тем дням и последующей истории языка Java посвящена данная глава.

Программная платформа Java

В первой редакции этой книги говорилось следующее.

"Как язык программирования, Java перевыполнил рекламные обещания. Определенно Java — хороший язык программирования. Несомненно, это один из лучших языков, доступных серьезным программистам. Потенциально Java имел все предпосылки, чтобы стать великим языком программирования, однако, вероятно, сейчас время для этого уже упущено. Когда появляется новый язык, немедленно возникает проблема его совместимости с программным обеспечением, созданным ранее".

По поводу этого абзаца наш редактор долго спорил с одним из руководителей компании Sun. Но и сейчас, по прошествии долгого времени, такая оценка кажется нам правильной. Действительно, Java обладает целым рядом преимуществ, о которых мы поговорим чуть позже. Однако более поздние дополнения далеко не так элегантны, как исходный вариант языка, и виной тому пресловутые требования совместимости.

Как уже говорилось в первой редакции, Java никогда не был только языком. Хорошие языки — не редкость; появление некоторых из них вызвало в свое время сенсацию в компьютерном мире. В отличие от них, Java — это программная платформа, включающая мощную библиотеку, большой объем кода, пригодного для повторного использования, и среду для выполнения программ, которая обеспечивает безопасность, независимость от операционной системы и автоматическую "сборку мусора".

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

Характерные особенности языка Java

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

- простой;
- объектно-ориентированный;
- распределенный;
- надежный;
- безопасный;

- не зависит от архитектуры компьютера;
- переносимый;
- интерпретируемый;
- высокопроизводительный;
- многопотоковый;
- динамичный.

В данном разделе мы:

- приведем цитаты из руководства по языку Java, раскрывающие особенности языка;
- поделимся с читателями соображениями по поводу отдельных свойств языка, основываясь на собственном опыте работы с его последней версией.



На заметку! Руководство, о котором идет речь, можно найти по адресу: `http://java.sun.com/docs/white/langenv/`. Характерные особенности Java описаны в документе, который находится по адресу `ftp://ftp.javasoft.com/docs/papers/java-overview.ps`.

Простота

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

Синтаксис языка Java, по существу, представляет собой упрощенный вариант синтаксиса языка C++. В этом языке нет необходимости в файлах заголовков, арифметике указателей (и даже в синтаксисе указателей), структурах, объединениях, перегрузке операций, виртуальных базовых классах и т.п. (Различия между Java и C++ упоминаются на протяжении всей книги.) Однако разработчики не стремились исправить все недостатки языка C++. Например, синтаксис оператора `switch` в языке Java остался неизменным. Зная C++, несложно перейти к языку Java.

Если вы привыкли к визуальной среде программирования (например, Visual Basic), язык Java может показаться сложным. Его синтаксис некоторые считают странным (хотя понять смысл выражения не составляет труда). И что самое главное, при работе на языке Java приходится намного больше программировать. Начинающие программисты нередко предпочитают Visual Basic, так как его визуальная среда программирования позволяет почти автоматически создавать инфраструктуру приложения. Чтобы достичь того же результата с помощью языка Java, необходимо вручную написать достаточно большой объем кода. Существуют, однако, среды разработки от независимых поставщиков, которые позволяют программировать в стиле перетаскивания (drag-and-drop).

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

На то время это было впечатляющим достижением. Конечно же, с тех пор библиотеки разрослись до гигантских масштабов. Однако теперь существует отдельный

комплект Java Micro Edition с уменьшенными библиотеками, более подходящий для встроенных устройств.

Поддержка объектов

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

За прошедшие 30 лет объектно-ориентированный подход уже доказал свое право на жизнь, и без него невозможно представить себе современный язык программирования. Действительно, особенности Java, связанные с объектами, сравнимы с языком C++. Основное различие между ними заключается в механизме множественного наследования, который в языке Java заменен более простой концепцией интерфейсов, а также в модели метаклассов Java (о которых речь пойдет в главе 5).



На заметку! Если вы никогда не использовали объектно-ориентированные языки, внимательно изучите главы 4–6. В них излагаются основы объектно-ориентированного подхода и демонстрируются его преимущества над такими традиционными, процедурно-ориентированными языками, как C или Basic, проявляющиеся при разработке сложных проектов.

Поддержка распределенных вычислений

"Язык Java предоставляет разработчику обширную библиотеку программ для передачи данных на базе протоколов TCP/IP (Transmission Control Protocol/Internet Protocol — протокол управления передачей/Internet-протокол), HTTP (Hypertext Transfer Protocol — протокол передачи гипертекстовой информации) и FTP (File Transfer Protocol — протокол передачи файлов). Приложения, написанные на языке Java, могут открывать объекты и получать к ним доступ по сети; при этом для адресации используются URL (Uniform Resource Location — унифицированный локатор ресурсов)", что открывает к ним такой же легкий доступ, как при работе с локальной файловой системой.

Язык Java предоставляет мощные и удобные средства для работы в сети. Каждый, кто когда-либо пытался писать программы на других языках для работы в глобальной сети Internet, будет приятно удивлен тем, как легко решаются на Java самые трудные задачи, например открытие сокетных соединений (socket connections). (Работа в сети будет описана во втором томе данной книги.) Связь между распределенными объектами в языке Java обеспечивается механизмом вызова удаленных методов (эта тема также затрагивается во втором томе).

Надежность

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

Это средство также очень полезно. Компилятор языка Java выявляет такие ошибки, которые в других языках обнаруживаются только на этапе выполнения программы. Кроме того, программисты, потратившие многие часы на поиски ошибки, вызвавшей повреждение памяти из-за неверного указателя, будут очень рады тому, что при работе с Java такие проблемы возникнуть в принципе не могут.

Если раньше вы программировали на Visual Basic либо на другом языке, в котором указатели явным образом не используются, возможно, вам будет непонятно, почему этот вопрос настолько важен. Программистам, использующим язык C, нужны указатели для доступа к строкам, массивам, объектам и даже файлам. При программировании на Visual Basic ничего этого не требуется, и разработчик может не беспокоиться о распределении памяти для хранения объектов. С другой стороны, многие структуры данных в языке, не имеющем указателей, реализовать очень трудно. Для простых конструкций, вроде строк и массивов, указатели не нужны. Возможности указателей в полной мере проявляются лишь там, где без них нельзя обойтись, например, при создании связных списков. Программист на языке Java полностью защищен, поскольку он никогда не обратится к неправильному указателю, не допустит ошибок выделения памяти, и ему не придется искать причин утечек памяти.

Безопасность

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

В первом издании мы написали: "Никогда не говори *никогда*", — и оказались правы. Вскоре после выхода первой версии Java Development Kit группа экспертов по вопросам безопасности из Принстонского университета обнаружила первые ошибки в системе защиты Java 1.0. Компания Sun Microsystems развернула исследования в области безопасности Java-программ. В частности, она обнародовала спецификацию и код виртуальной машины и библиотек, ответственных за защиту. Это ускорило выявление и устранение ошибок. В любом случае, разработчики Java здорово затруднили задачу обхода ее механизмов безопасности. Обнаруженные с тех пор ошибки носили очень технический характер и весьма немногочисленны.

Перечислим некоторые ситуации, возникновение которых с самого начала предотвращает система безопасности языка Java.

- Намеренное переполнение стека выполняемой программы — один из распространенных способов атаки, используемых вирусами и червями.
- Повреждение участков памяти, находящихся за пределами пространства, выделенного процессу.
- Несанкционированное чтение файлов и их модификация.

Со временем в язык были добавлен ряд новых средств защиты. В версии 1.1 в Java появилось понятие классов с цифровой подписью (см. том II). Пользуясь таким классом, вы получаете сведения об авторе. Если вы доверяете автору класса, то можете предоставить этому классу все необходимые привилегии.



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

Независимость от архитектуры компьютера

"Компилятор генерирует объектный файл, формат которого не зависит от архитектуры компьютера. Скомпилированная программа может выполняться на любых процессорах; для ее работы необходима лишь исполняющая система Java. Код, генерируемый компилятором Java, называется байт-кодом. Он разработан таким образом, чтобы на любой машине его можно было легко интерпретировать либо в процессе работы перевести в набор команд, ориентированных на конкретный процессор".

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

Очевидно, что байт-код, интерпретируемый с помощью виртуальной машины, всегда будет работать медленнее, чем машинные инструкции, поэтому целесообразность такого подхода у многих вызывает сомнение. Однако эффективность байт-кода можно существенно повысить за счет компиляции, осуществляемой в процессе выполнения программы. Этот механизм доказал свою эффективность и даже использовался при создании платформы .NET, также полагающейся на виртуальную машину.

Следует отметить, что виртуальная машина имеет ряд других преимуществ по сравнению с непосредственным выполнением программы. Она существенно повышает безопасность, поскольку в процессе работы можно оценить последствия выполнения каждой конкретной команды. Некоторые программы даже способны генерировать байт-код "на лету", динамически расширяя возможности исполняющегося приложения.

Переносимость

"В отличие от языков C и C++, ни один из аспектов спецификации Java не зависит от реализации. И размер основных типов данных, и арифметические операции над ними строго определены".

Например, тип `int` в языке Java всегда означает 32-разрядное целое число. В языках C и C++ тип `int` может означать как 16-разрядное, так и 32-разрядное целое число. Единственное ограничение заключается в том, что размер типа `int` не может быть меньше размера типа `short int` и больше размера типа `long int`. Фиксированный размер числовых типов позволяет избежать многих неприятностей, связанных с выполнением программ на разных компьютерах. Бинарные данные хранятся и передаются в неизменном формате, что также позволяет избежать недоразумений, связанных с разным порядком следования байтов на разных платформах. Строки сохраняются в стандартном формате Unicode.

"Библиотеки, представляющие собой часть системы, предоставляют переносимые интерфейсы. Например, в языке предусмотрен абстрактный класс `Window` *и его реализации для операционных систем Unix, Windows и Macintosh".*

Каждый, кто когда-либо пытался написать программу, которая одинаково хорошо работала бы под управлением операционных систем Windows, Macintosh и десятка разновидностей системы Unix, знает, что это очень трудная задача. Разработчики версии Java 1.0 предприняли героическую попытку решить эту проблему, предоставив простой инструментальный набор, адаптирующий обычные элементы пользовательского интерфейса к различным платформам. К сожалению, библиотека, на которую было затрачено немало труда, не позволила достичь приемлемых результатов в разных системах. (В результате для каждой системы в графических программах проявлялись свои характерные ошибки.) Однако это было лишь началом. Во многих приложениях машинная независимость оказалась намного важнее изысканности графического пользовательского интерфейса. Именно эти приложения выиграли от появления версии Java 1.0. Однако теперь инструментальный набор для создания графического пользовательского интерфейса полностью переработан и больше не зависит от интерфейсных средств, используемых на конкретном компьютере. Новая версия более осмысленна и, по нашему мнению, более привлекательна для пользователя, чем предыдущие.

Интепретируемость

"Интерпретатор Java может выполнять байт-код Java непосредственно на любой машине, на которую перенесен интерпретатор. Поскольку компоновка — в большей степени инкрементный и относительно легкий процесс, процесс разработки программ может стать намного быстрее и в большей степени творческим".

Инкрементная компоновка имеет свои преимущества, однако ее выгоды для процесса разработки, несомненно, переоцениваются. Ранние инструменты разработки Java были довольно медленными. Сегодня байт-код транслируется в машинный код.

Производительность

"Хотя обычно интерпретируемые байт-коды имеют достаточную производительность, бывают ситуации, когда требуется еще более высокая эффективность. Байт-коды можно транслировать во время выполнения программы в машинные команды для процессора, на котором выполняется данное приложение".

В ранние годы Java многие пользователи были не согласны с утверждением, что производительность "более чем достаточна". Однако, сегодня, когда оперативные (just-in-time — JIT) компиляторы стали настолько хороши, что могут конкурировать с традиционными компиляторами, в некоторых случаях они дают даже выигрыш в производительности, поскольку имеют больше доступной информации. Так, например, JIT-компилятор может отслеживать, какой код выполняется чаще и оптимизировать по скорости только эту часть кода. JIT-компилятор знает, какие классы были загружены. Он может применить встраивание (inlining), когда до загрузки текущего набора классов определенная функция никогда не переопределяется, а затем, при необходимости, отменить такую оптимизацию.

Многопоточность

"Преимуществами многопоточности является более высокая интерактивность и поведение программ в реальном масштабе времени".

Если вы когда-либо пытались организовать многопоточные вычисления на каком-нибудь из языков программирования, вы будете приятно удивлены тем, насколько это легко делать на Java. Потоки в языке Java могут использовать преимущества многопроцессорных систем, если операционная система позволяет это сделать. К сожалению, реализации потоков на большинстве систем сильно отличаются друг от друга, а разработчики языка Java не предпринимают должных усилий, чтобы достичь независимости от платформы и в этом вопросе. Только код для вызова потоков остается одинаковым для всех машин, а ответственность за их поддержку возлагается на операционную систему или библиотеку потоков. Несмотря на это, именно простота организации многопоточных вычислений делает язык Java таким привлекательным для разработки программного обеспечения серверов.

Динамичность

"Во многих отношениях язык Java является более динамичным, чем языки C и C++. Он был разработан так, чтобы легко адаптироваться к постоянно изменяющейся среде. В библиотеки можно свободно включать новые методы и объекты, никак не затрагивая этим приложения, пользующиеся библиотеками. Язык Java позволяет легко получать информацию о ходе выполнения программы".

Это очень важно в тех случаях, когда требуется добавить код в уже выполняемую программу. Ярким примером этого является код, который загружается из Internet для выполнения браузером. В версии Java 1.0 получить информацию о работающей программе было непросто, однако последняя версия языка Java раскрывает перед программистом как структуру, так и поведение объектов. Это весьма ценно для систем, которые должны анализировать объекты в ходе выполнения программы. К таким системам относятся средства создания графического пользовательского интерфейса, интеллектуальные отладчики, включаемые компоненты и объектные базы данных.



На заметку! Компания Microsoft выпустила продукт под названием J++, с языком программирования и виртуальной машиной, который очень похож на Java. В то же время библиотеки для работы с графикой, создания пользовательских интерфейсов и доступа к удаленным объектам существенно отличаются друг от друга. В настоящее время Microsoft прекратила поддержку J++ и сосредоточила свои усилия на другом языке, который называется C# и несколько напоминает Java, но использует другую виртуальную машину. Создан даже язык J#, предназначенный для переноса приложений J++ на виртуальную машину, используемую C#. Языки J++, C# и J# в данной книге рассматриваться не будут.

Аплеты Java и Internet

Здесь идея была проста — пользователи загружают байт-коды Java по сети и выполняют их на своих машинах. Программы Java, работающие под управлением Web-браузеров, называются *аплетами* (applet). Для использования аплета нужен Web-браузер, поддерживающий язык Java и способный интерпретировать байт-код. Поскольку лицензия на исходные коды языка Java принадлежит компании Sun, настаивающей на неизменности как самого языка, так и структуры его основных библиотек, аплеты Java

должны запускаться в любом браузере, который поддерживает Java. Всякий раз, посещая Web-страницу, содержащую аплет, вы получаете последнюю версию программы. Но что более важно, благодаря средствам безопасности виртуальной машины, вам не приходится беспокоиться об угрозах, исходящих от вредоносного кода.

Загрузка аплета напоминает вставку рисунка на Web-страницу. Аплет становится частью страницы, а текст обтекает занимаемое им пространство. Изображение, реализуемое аплетом, становится *живым*. Оно реагирует на команды пользователя, изменяет, в зависимости от них, свой внешний вид и выполняет обмен данными между компьютером, на котором выполняется аплет, и компьютером, где этот аплет постоянно хранится.

На рис. 1.1 показан хороший пример динамической Web-страницы, выполняющей сложные вычисления и применяющей аплет для отображения моделей молекул. Чтобы лучше понять структуру молекулы, можно вращать ее либо изменять масштаб изображения, используя мышь. Подобные эффекты нельзя реализовать на статических Web-страницах, однако аплеты делают это возможным. (Этот аплет можно найти по адресу `http://jmol.sourceforge.net`.)

Когда аплеты появились впервые, они наделали немало шума. Многие считают, что именно привлекательность аплетов стала причиной ошеломляющего успеха Java. Однако первоначальный энтузиазм быстро перешел в разочарование. Разные версии браузеров Netscape и Internet Explorer поддерживают разные версии языка Java, причем некоторые из этих версий значительно устарели. Эта неприятная ситуация создает все больше препятствий при разработке аплетов, которые позволяли бы использовать преимущества последней версии языка Java. Сегодня большинство Web-страниц просто используют JavaScript или Flash, когда требуется получить динамические эффекты в браузере. С другой стороны, Java стала наиболее популярным языком для разработки приложений серверной стороны, которые генерируют Web-страницы и поддерживают логику заднего плана.

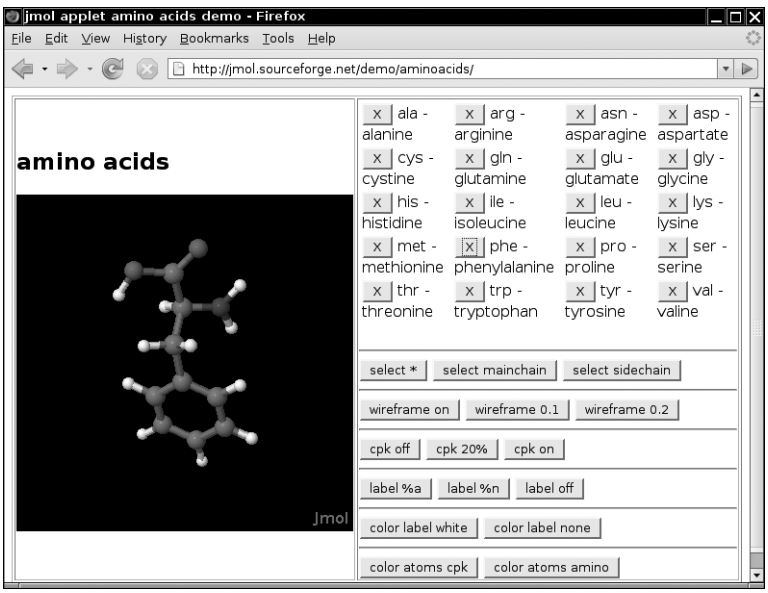


Рис. 1.1. Аплет Jmol

Краткая история языка Java

В данном разделе кратко описывается история языка Java. В основу раздела положены различные опубликованные материалы (в частности, интервью с создателями языка Java в июльском выпуске электронного журнала *SunWorld* за 1995 г.).

История Java восходит к 1991 г., когда группа инженеров из компании Sun под руководством Патрика Нотона (Patrick Naughton) и члена Совета директоров (и разностороннего специалиста) Джеймса Гослинга (James Gosling) занялась разработкой языка, который можно было бы использовать для программирования бытовых устройств, например, контроллеров для переключения каналов кабельного телевидения. Так как подобные устройства не обладают большой мощностью и объемом памяти, язык должен был быть маленьким и генерировать очень компактный код. Кроме того, поскольку разные производители могут выбирать разные процессоры, было важно не привязаться к конкретной архитектуре. Проект получил кодовое название "Green".

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

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

В 1992 г. в рамках проекта "Green" был выпущен первый продукт, названный "*7". Это было устройство интеллектуального дистанционного управления. (Помещаясь в объеме 6×4×4 дюйма, оно имело мощность рабочей станции SPARK.) К сожалению, ни одна из компаний — производителей электронной техники не заинтересовалась этой разработкой. Затем группа стала заниматься созданием устройства для кабельного телевидения, которое могло бы осуществлять новые виды услуг, например, включать видеосистему по требованию. И снова они не получили ни одного контракта. Примечательно, что одной из компаний, отказавшихся подписать с ними контракт, руководил Джим Кларк (Jim Clark) — основатель компании Netscape, впоследствии сделавшей очень много для развития языка Java.

Весь 1993 г. и половину 1994 г. продолжались безрезультатные поиски покупателей продукции, разработанной в рамках проекта "Green" (группа, работающая над проектом, выделилась под названием "First Person, Inc."). Патрик Нотон, один из основателей группы, в основном занимавшийся маркетингом, налетал в общей сложности более 300 тысяч миль, пытаясь продать разработанную технологию. Работа группы "First Person, Inc." была прекращена в 1994 г.

Тем временем в рамках глобальной сети Internet развивалась система World Wide Web. Ключевым элементом этой системы является браузер, превращающий гипертекстовые данные в изображение на экране. В 1994 г. большинство пользователей применяли некоммерческий Web-браузер Mosaic, разработанный в суперкомпьютерном центре Университета штата Иллинойс в 1993 г. Частично этот браузер был написан Марком Андреессеном (Mark Andreessen) за 6,85$ в час. В то время Марк заканчивал университет, и браузер был его дипломной работой. (Затем он достиг известности и успеха как один из основателей и ведущих специалистов компании Netscape.)

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

Реальный браузер был разработан Патриком Нотоном и Джонатаном Пэйном (Johnatan Payne). Позднее он был доработан и получил имя HotJava. Чтобы продемонстрировать все возможности Java, браузер был написан на этом языке. Однако разработчики не забывали о таких средствах, которые теперь называются аплетами, наделив свой продукт способностью выполнять код внутри Web-страниц. Продукт, созданный в рамках новой технологии, был представлен на выставке SunWorld '95 23 мая 1995 г. и вызвал всеобщий интерес к Java, сохраняющийся по сей день.

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

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

Кроме Standard Edition, были предложены еще два варианта: Micro Edition ("микроредакция") для портативных устройств, например для мобильных телефонов, и Enterprise Edition (редакция для корпоративных приложений). В нашей книге внимание в основном уделено Standard Edition.

Версии 1.3 и 1.4 пакета инструментальных средств Standard Edition являются последовательными шагами по усовершенствованию первоначальной реализации Java 2. Они обладают новыми возможностями, производительность их повышена, и, разумеется, содержат намного меньше ошибок. В процессе развития Java многие взгляды на аплеты и серверные программы были пересмотрены. В частности, оказалось, что на базе Java можно разрабатывать высококачественные серверные приложения.

В версии 5.0 язык Java подвергся наиболее существенной модификации с момента реализации версии 1.1. (Первоначально версия 5.0 имела номер 1.5, но на конференции JavaOne в 2004 г. была принята новая нумерация версий.) После многолетних исследований были добавлены обобщенные типы (которые приблизительно соответствуют шаблонам C++), хотя при этом не были выдвинуты требования модификации виртуальной машины. Ряд других языковых элементов, например циклы "for each", автоматическое представление простых типов в виде объектов (упаковка) и метаданные были явно "навеяны" языком C#. Изменение и дополнение базовых языковых конструкций — болезненный процесс, но многие из средств версии 5.0 настолько удобны, что разработчики наверняка примут их с энтузиазмом.

Версия 6 (без суффикса .0) была реализована в конце 2006 г. Опять-таки, язык не претерпел существенных изменений, но была внесены усовершенствования, связанные с производительностью, и расширения библиотек.

В табл. 1.1 приведена информация об этапах развития *языка и библиотек* Java. Как видите, объем интерфейса программирования приложений (API) рос ужасающе.

Таблица 1.1. Этапы развития языка Java

Версия	Год	Новые языковые средства	Количество классов и интерфейсов
1.0	1996	Реализация языка	211
1.1	1997	Внутренние классы	477
1.2	1998	Отсутствуют	1 524
1.3	2000	Отсутствуют	1 840
1.4	2004	Директивы	2 723
5.0	2004	Универсальные классы, цикл "for each", представление простых типов в виде объектов, метаданные, нумерация, статическое импортирование	3 279
6	2006	Отсутствуют	3 777

Распространенные заблуждения относительно языка Java

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

Язык Java — это расширение языка HTML.

Java — это язык программирования, а язык HTML — это способ описания структуры Web-страниц. Между ними нет ничего общего, за исключением HTML-дескрипторов, позволяющих размещать на Web-страницах аплеты, написанные на языке Java.

Я использую XML, поэтому мне не нужен язык Java.

Java — это язык программирования, а XML — средство описания данных. Данные, описанные с помощью языка XML, можно обрабатывать программами, написанными на любом языке программирования, но лишь библиотека API языка Java содержит превосходные средства для обработки таких данных. Кроме того, на языке Java реализованы многие программы независимых производителей, предназначенные для

работы с XML-документами. Более подробную информацию по этому вопросу можно найти во втором томе данной книги.

Язык Java легко выучить.

Нет ни одного языка программирования, сопоставимого по возможностям с Java, который можно было бы легко выучить. Простейшие программы написать несложно, однако выполнять серьезную работу всегда трудно. Кроме того, обратите внимание на то, что в этой книге обсуждению собственно языка Java посвящено только четыре главы. В остальных главах рассматривается работа с библиотеками, содержащими тысячи классов и интерфейсов, а также многие тысячи методов. К счастью, вам не нужно знать каждый из них, но все же знать нужно достаточно много, чтобы использовать Java в реальной работе.

Язык Java со временем станет универсальным языком программирования для всех платформ.

Теоретически это возможно. Именно об этом мечтают все производители программного обеспечения, кроме компании Microsoft. Однако есть много приложений, прекрасно работающих на персональных компьютерах, которые не будут так же хорошо работать на других устройствах или под управлением браузера. Кроме того, эти приложения написаны так, чтобы максимально использовать возможности процессоров и платформенно-ориентированных библиотек. В любом случае они уже перенесены на все важные платформы. К таким приложениям относятся текстовые и графические редакторы, а также Web-браузеры. Обычно эти приложения пишутся на языке C или C++, и пользователь ничего не выиграет, если переписать их на языке Java.

Язык Java — это не более чем еще один язык программирования.

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

Почему так происходит? Успех любого языка программирования в большей степени определяется его *системой поддержки*, чем элегантностью его синтаксиса. Существуют ли стандартные библиотеки, позволяющие сделать именно то, что необходимо программисту? Разработана ли удобная среда для создания и отладки программ? Интегрирован ли язык и его инструментарий в остальную инфраструктуру компьютера? Язык Java достиг успехов в области серверных приложений, поскольку его библиотеки классов позволяют легко сделать то, что раньше было трудно реализовать, например, поддерживать работу в сети и обеспечивать выполнение нескольких потоков. Тот факт, что язык Java уменьшил количество ошибок, связанных с указателями, также говорит в его пользу. Благодаря этому производительность труда программистов повысилась. Однако не в этом кроется причина его успеха.

С появлением языка C# Java устарел.

Разработчики C# использовали многие удачные технические решения, применявшиеся в языке Java, такие как полностью определенный язык, использование виртуальной машины и систему “сборки мусора”. Однако по ряду характеристик C# уступает Java, это вопросы безопасности и переносимости программ. Самое очевидное преимущество C# — это прекрасная среда разработки. Если вы привязаны к системе Windows, может быть, вам стоит использовать C#. Если же оценить реальную ситуацию, то на сегодняшний день Java остается выбором очень многих программистов.

Java является собственностью компании Sun, поэтому пользоваться им не следует.

Sun Microsystems лицензирует Java для разработчиков и конечных пользователей. Несмотря на то что Sun сохраняет полный контроль над Java через Java Community Process (Процесс сообщества Java), в работу над данным языком и его библиотеками уже оказались вовлечены многие другие компании. Исходных код виртуальной машины и библиотек доступен всем желающим, но его можно использовать лишь для ознакомления, вносить в него изменения запрещено. До настоящего момента язык Java имел "закрытый исходный код, но прекрасно работал".

Ситуация изменилась кардинально в 2007 г., когда Sun объявила, что будущие версии Java будут доступны на условиях General Public License (GPL) — той же лицензии открытого кода, по которой распространяется Linux. Похоже, что Sun будет и впредь управлять развитием Java, но несомненно, что открытие кода Java — это сильный шаг, который продлит жизнь Java на многие годы.

Программы на языке Java работают под управлением интерпретатора, значит, серьезные приложения будут выполняться слишком медленно.

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

Для Java характерны дополнительные накладные расходы по сравнению с C. Запуск виртуальной машины требует времени, и графический пользовательский интерфейс, написанный на Java, медленнее, чем конкурирующий "родной" код, поскольку он рисуется в платформенно-независимой манере.

Годами пользователи жалуются на низкое быстродействие Java. Однако с того времени, когда начались эти жалобы, мощность компьютеров многократно возросла. И те самые медленные Java-программы теперь выполняются быстрее, чем несколько лет назад выполнялись те самые "молниеносные" программы на C++. Теперь эти жалобы выглядят надуманными, и потому некоторые скептики вместо этого начали жаловаться на "уродливый" пользовательский интерфейс Java.

Все программы на языке Java выполняются под управлением Web-браузеров.

Все *аплеты*, написанные на языке Java, действительно выполняются под управлением Web-браузеров. Однако большинство Java-программ являются независимыми приложениями, которые никак не связаны с Web-браузером. Фактически многие Java-программы исполняются на Web-серверах и генерируют код для Web-страниц.

Большинство программ, рассматриваемых в этой книге, являются независимыми приложениями. Конечно, аплеты очень забавны. Однако Java-приложения более важны и полезны.

Java-аплеты представляют большую опасность для системы защиты.

В свое время было опубликовано несколько отчетов об ошибках в системе защиты языка Java. Большинство из них касалось реализаций языка Java в конкретных браузерах. Исследователи восприняли это как вызов и принялись искать глобальные недостатки в системе защиты Java, которые делали бы бессмысленной модель безопасности аплетов. Их усилия не увенчались успехом. Обнаруженные ошибки в конкретных реализациях вскоре были исправлены, и, после этого, насколько мы знаем, ни одна реальная система не была взломана посредством аплета. Чтобы оценить значение этого

факта, вспомните о миллионах вирусных атак на исполняемые файлы операционной системы Windows и макросы редактора Word, действительно вызвавшие немало бед. При этом критика недостатков платформы была удивительно беззубой. Кроме того, механизм ActiveX в браузере Internet Explorer мог бы вызвать много нареканий, однако способы его взлома настолько очевидны, что лишь немногие специалисты потрудились опубликовать результаты своих исследований.

Некоторые системные администраторы даже стали отключать системы защиты языка Java в своих браузерах, чтобы пользователи могли, как и прежде, загружать исполняемые файлы, элементы управления ActiveX и документы, созданные с помощью текстового процессора Word. Можно утверждать, что в настоящее время вероятность преодоления системы защиты аплетов на языке Java сравнима с вероятностью погибнуть в авиационной катастрофе, в то время как риск заразить компьютер вирусом, открыв документ, созданный текстовым процессором Word, куда больше, чем риск погибнуть под колесами автомобиля, перебегая дорогу в час пик.

Язык JavaScript — упрощенная версия языка Java.

Язык JavaScript — это язык сценариев, который можно *использовать в составе Web-страницы*. Он был разработан компанией Netscape и сначала назывался LiveScript. Синтаксис языка JavaScript напоминает синтаксис языка Java, однако на этом их сходство заканчивается (за исключением имени, конечно). Подмножество JavaScript было стандартизовано как ECMA-262. Язык JavaScript более тесно привязан к браузерам, чем аплеты Java. В частности, программа JavaScript может модифицировать отображаемый документ, в то время как аплет контролирует только свою ограниченную область отображения.

Пользуясь языком Java, можно заменить компьютер Internet-устройством стоимостью 500 долларов.

Когда появился язык Java, некоторые люди были уверены, что так и случится. Однако даже со времени выхода в свет первой редакции этой книги мы убедились в абсурдности ожидания, что пользователи, работающие дома, решат заменить мощные и удобные персональные компьютеры устройствами без жестких дисков с ограниченными возможностями. Тем не менее, сетевой компьютер, оснащенный средствами языка Java, предоставляет вполне реальную возможность реализовать “нулевую административную инициативу”, направленную на уменьшение стоимости компьютеров, занятых в бизнесе, но даже это не получило большого распространения.

С другой стороны, Java широко распространился в мире мобильных телефонов. Следует признать, что мы пока не видели серьезных приложений, выполняющихся на мобильных телефонах, но игры и хранители экрана для них достаточно хорошо продаются на многих рынках.



Совет. Ответы на часто задаваемые вопросы о языке Java можно найти в документе по адресу `http://www.apl.jhu.edu/~hall/java/FAQs-and-Tutorials.html`.

Глава 2

СРЕДА ПРОГРАММИРОВАНИЯ JAVA

В ЭТОЙ ГЛАВЕ...

- Инсталляция Java Development Kit
- Выбор среды разработки программ
- Использование инструментов командной строки
- Использование интегрированной среды разработки
- Выполнение графического приложения
- Создание и запуск аплетов

В этой главе вы научитесь инсталлировать Java Development Kit (JDK), а также компилировать и запускать на выполнение программы различных типов: консольные программы, графические приложения и аплеты. Мы будем использовать средства пакета JDK, набирая команды в окне, предназначенном для работы с оболочкой. Однако многие программисты предпочитают комфорт интегрированной среды разработки. Вы узнаете, как использовать бесплатно доступную интегрированную среду разработки для компиляции и выполнения программ, написанных на языке Java. Изучать интегрированные среды разработки и пользоваться ими довольно легко, однако они долго загружаются и предъявляют высокие требования к ресурсам компьютера, так что применять их для разработки небольших программ не имеет смысла. В качестве компромисса мы покажем вам, как можно использовать текстовый редактор, из которого вызывается компилятор и запускаются Java-программы. Овладев приемами, рассмотренными в этой главе, и выбрав инструмент для разработки программ, вы сможете перейти к главе 3, с которой и начинается изучение языка Java.

Инсталляция Java Development Kit

Наиболее полные и современные версии Java Development Kit (JDK) доступны от Sun Microsystems для операционных систем Solaris, Linux и Windows. Версии, находящиеся в разных стадиях разработки для Macintosh и многих других платформ лицензированы и поставляются производителями соответствующих систем.



На заметку! Некоторые дистрибутивы Linux имеют предварительно пакетированные версии JDK. Например, в Ubuntu вы можете инсталлировать JDK, просто установив пакет `sun-java6-jdk` с помощью `apt-get` или Synaptic GUI.

Где найти JDK

Если вам необходимо скопировать на свой компьютер Java Development Kit, вам надо обратиться на Web-страницу Sun и, приложив немалые усилия, чтобы разобраться в жаргоне, найти требуемое программное обеспечение. Подробности смотрите в табл. 2.1.

Аббревиатура JDK вам уже знакома. Она, как нетрудно догадаться, означает Java Development Kit. Некоторые трудности может вызвать тот факт, что в версиях 1.2–1.4 пакет называется Java SDK (Software Development Kit). Иногда вы встретите ссылки на старый термин. Существует также Java Runtime Environment (JRE), включающая виртуальную машину, но без компилятора. Однако вам, как разработчику, это не подходит. Пакет JRE предназначен для конечных пользователей, которым компилятор ни к чему.

Далее везде вы будете встречать термин Java SE. Он означает Java Standard Edition, в отличие от Java EE (Enterprise Edition) и Java ME (Micro Edition).

Иногда вы встретите термин “Java 2”, который был введен в 1998 г., когда специалисты компании Sun по маркетингу поняли, что очередной дробный номер реализации никак не отражает глобальных отличий между JDK 1.2 и предыдущими версиями. Но, поскольку такое мнение сформировалось уже после выхода в свет JDK, было решено, что номер версии 1.2 останется за *пакетом разработки*. Последующие реализации JDK имеют номера 1.3, 1.4 и 5.0. Платформа же была переименована из “Java” в “Java 2”. Таким образом, последний инструментальный пакет называется Java 2 Standard Edition Software Development Kit Version 5.0, или J2SE SDK 5.0.

Таблица 2.1. Жаргон Java

Наименование	Аббревиатура	Пояснение
Java Development Kit	JDK	Программное обеспечение для программистов, которые хотят писать приложения на Java.
Java Runtime Environment	JRE	Программное обеспечение для потребителей, которые желают запускать программы Java.
Standard Edition	SE	Платформа Java для использования в настольных системах и простых серверных приложениях.
Enterprise Edition	EE	Платформа Java для сложных серверных приложений.
Micro Edition	ME	Платформа Java для использования в мобильных телефонах и других компактных устройствах.
Java 2	J2	Устаревший термин для версий Java, выпущенных в период с 1998 до 2006 гг.
Software Development Kit	SDK	Устаревший термин для версий JDK, выпущенных в период с 1998 до 2006 гг.
Update	u	Термин Sun для выпусков с исправлениями ошибок.
NetBeans	–	Интегрированная среда разработки Sun.

Для нас, инженеров, эти манипуляции с именами версий выглядят, по меньшей мере, странно. Именно поэтому мы никогда не сможем достичь успеха в маркетинге. Наконец, в 2006 г. здравый смысл возобладал. Неудобный термин “Java 2” был отброшен, и текущая версия Java Standard Edition была названа Java SE 6. Иногда вам попадутся случайные ссылки на версии 1.5 и 1.6 — это просто синонимы версий 5.0 и 6.

И, наконец, когда Sun вносит небольшие изменения, исправляя неотложные проблемы, она называет такие изменения обновлениями (update). Например, первое обновление пакета разработки для Java SE 6 официально называется JDK 6u1 и имеет внутренний номер версии 1.6.0_01.

Если вы используете операционную систему Solaris, Linux или Windows, можете загрузить Java Development Kit, обратившись по адресу `http://java.sun.com/j2se`. Выберите версию 6 или более позднюю и укажите вашу платформу. Пусть вас не тревожит, что программное обеспечение называется “update”. Пакет обновления содержит наиболее свежую версию всего JDK.

Время от времени компания Sun выпускает наборы, содержащие, помимо Java Development Kit, интегрированную среду разработки. Это интегрированное окружение в разные периоды времени называлось Forte, Sun ONE Studio, Sun Java Studio и Netbeans. Трудно предугадать, какое имя для него выберут напряженно работающие маркетологи на тот момент, когда вы обратитесь на Web-узел компании Sun. На данном этапе нам нужен только Java Development Kit. Если впоследствии вы решите поработать с интегрированной средой разработки, то сможете скопировать ее с сервера `http://netbeans.org`.

После того как JDK окажется на вашем компьютере, следуйте инструкциям по инсталляции для вашей платформы. На момент написания данной книги инструкции были представлены в документе `http://java.sun.com/javase/6/webnotes/install/index.html`.

От конкретной системы зависит только процесс инсталляции и компиляции. После того как вы убедились в работоспособности Java, вы сможете непосредственно

выполнять рекомендации, приведенные в данной книге. Независимость от операционной системы является одним из главных преимуществ Java перед многими другими языками.



На заметку! Процедура инсталляции предлагает по умолчанию имя каталога для установки JDK. В имя каталога входит номер версии, например `jdk1.6.0`. Это может показаться не столь важным, но данный подход упрощает работу с новыми версиями JDK, установленными для тестирования.

Если вы работаете в системе Windows, мы рекомендуем не принимать предлагаемый каталог. Обычно это `c:\Program Files\jdk1.6.0`. По крайней мере, откажитесь от `Program Files` в пути инсталляции.

В данной книге мы используем для ссылки на каталог, содержащий Java Development Kit, имя *jdk*. Например, если в тексте указано имя *jdk*`/bin`, значит, речь идет о каталоге `/usr/local/jdk1.6.0/bin` или `c:\jdk1.6.0\bin`.

Установка пути к исполняемым файлам

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

- В системе Unix (включая Solaris и Linux) процедура редактирования путей к выполняемым файлам зависит от используемой оболочки. В случае C shell (эта оболочка принимается на платформе Solaris по умолчанию) добавьте в конец файла `~/.cshrc` следующую строку:

  ```
  set path=(/usr/local/jdk/bin $path)
  ```

  Если вы работаете с оболочкой Bourne Again (она принимается по умолчанию на платформе Linux), добавьте в конец файла `~/.bashrc` или `~/.bash_profile` строку, приведенную ниже.

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

- Под Windows войдите в систему как администратор. Запустите панель управления, выберите пиктограмму Система. Под Windows NT/2000/XP вы немедленно попадете в диалог свойств системы. В Vista вам нужно будет выбрать Advanced System Settings (Дополнительные настройки системы), как показано на рис. 2.1. В диалоге свойств системы перейдите на вкладку Advanced (Дополнительно), затем щелкните на кнопку Environment (Среда). Прокрутите содержимое окна System Variables (Системные переменные), пока не найдете переменную `PATH`. Щелкните на кнопке Edit (Правка), как показано на рис. 2.2. Добавьте имя каталога *jdk*`/bin` в начало списка; новый элемент списка отделяется от уже существующих точкой с запятой, например:

  ```
  c:\jdk\bin; остальное
  ```

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

Правильность установок можно проверить следующим образом.

Откройте окно оболочки. Как вы это сделаете, зависит от операционной системы. Введите следующую строку:

```
java -version
```

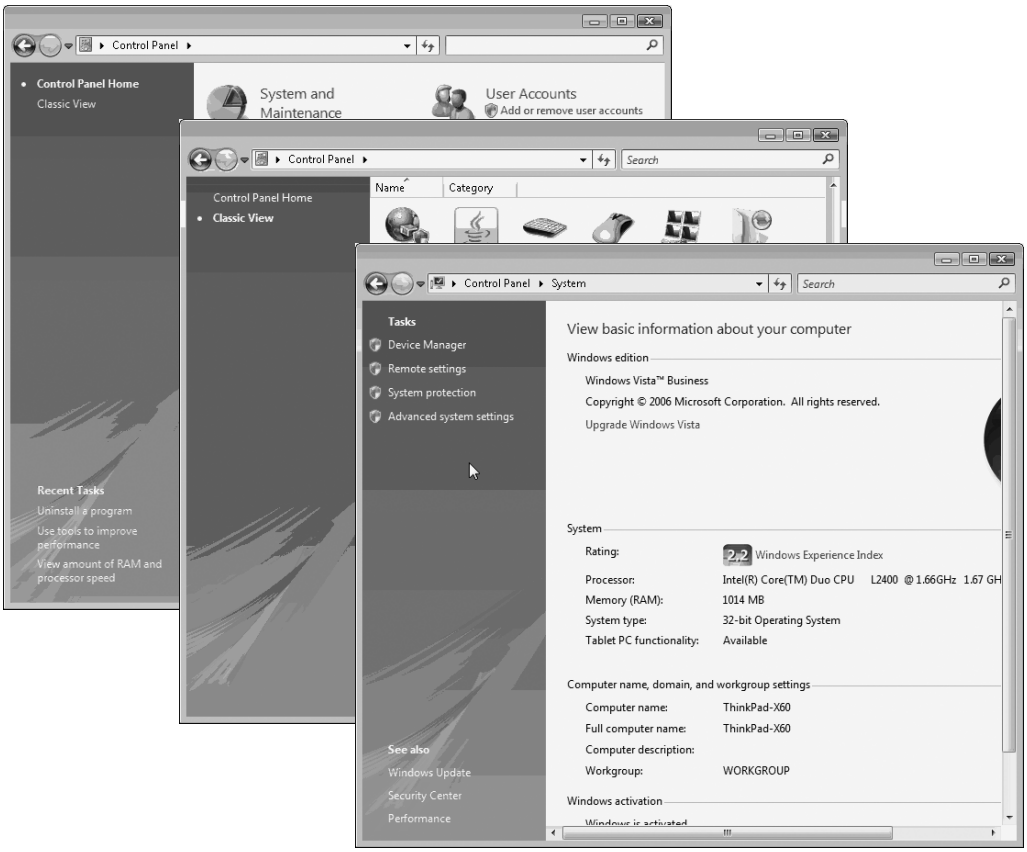


Рис. 2.1. Запуск диалога свойств системы в Windows Vista

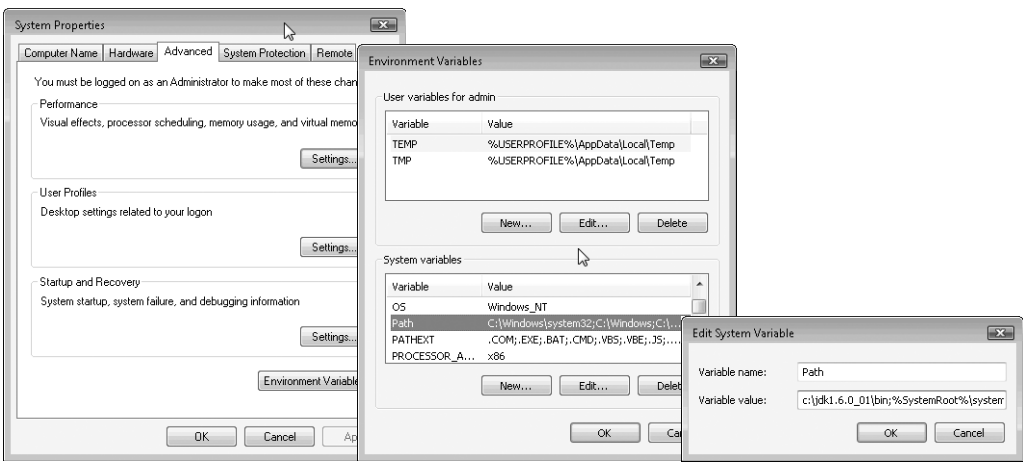


Рис. 2.2. Установка переменной окружения Path в Windows Vista

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

```
java version "1.6.0_01"
Java(TM) SE Runtime Environment (build 1.6.0_01-b06)
Java HotSpot(TM) Client VM (build 1.6.0_01-b06, mixed mode, sharing)
```

Если вместо этого сообщения появится строка наподобие "`java:command not found`" или "`The name specified is nor recognized as an internal or external command, operable program or batch file`", нужно еще раз проверить, правильно ли выполнена инсталляция и установлен путь.



На заметку! Для открытия окна оболочки в Windows выполните следующие инструкции. Если вы используете Windows NT/2000/XP, выберите пункт Run (Выполнить) в меню Start (Пуск) и наберите `cmd`. В Vista просто введите `cmd` в поле Start Search (Запустить поиск) меню Start. Нажмите <Enter>, и окно оболочки немедленно появится.

Если вы никогда не имели дело с командной оболочкой, мы советуем вам ознакомиться с руководством, которое научит основам использования командной строки. Многие учебные заведения публикуют такие руководства в Web, например, `http://www.cs.sjsu.edu/faculty/horstman/CS46A/windows/tutorial.html`.

Инсталляция библиотек и документации

Библиотечные файлы поставляются в пакете JDK в виде архива `src.zip`. Вам следует распаковать этот файл, чтобы получить доступ к исходным текстам программ. Мы настоятельно рекомендуем сделать это. Выполните перечисленные ниже действия.

1. Убедитесь, что пакет JDK установлен, а имя каталога *jdk*`/bin` находится в списке путей к исполняемым файлам.
2. Откройте окно для ввода команд оболочки.
3. Перейдите в каталог *jdk* (т.е. в каталог `/usr/local/jdk1.6.0` или `C:/jdk1.6.0`).
4. Создайте подкаталог `src`.

   ```
   mkdir src
   cd src
   ```

5. Выполните следующую команду:

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

   (или, если вы работаете в системе Windows, `jar xvf ..\src.zip`).



Совет. Файл `src.sip` содержит исходный код всех общедоступных библиотек. Чтобы получить дополнительные исходные тексты (компилятора, виртуальной машины, платформенно-ориентированных методов и вспомогательных классов), посетите Web-страницу `http://download.java.net/jdk6`.

Документация содержится в отдельном от JDK архиве. Вы можете загрузить документацию по адресу `http://java.sun.com/javase/downloads`. Для этого просто выполните следующие шаги.

1. Убедитесь, что пакет JDK установлен правильно, а имя каталога *jdk*`/bin` находится в списке путей к исполняемым файлам.

2. Загрузите файл документации с расширением `.zip`, содержащий документацию, на свой компьютер и поместите его в каталог *jdk*. Файл с документацией называется `jdk-`*версия*`-doc.zip`, где вместо слова *версия* указан конкретный номер реализации, например 6.
3. Откройте окно командной оболочки.
4. Перейдите в каталог *jdk*.
5. Выполните следующую команду:

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

Здесь, как и ранее, *версия* означает номер реализации.

Установка примеров программ

В процессе работы вам, скорее всего, понадобятся примеры программ, приведенные в книге. Вы можете найти их на Web-странице `http://horstmann.com/corejava` или на сайте издательства. Программы хранятся в архиве `corejava.zip`, который нужно распаковать в отдельный каталог, — мы рекомендуем назвать его `CoreJavaBook`. Выполните следующие действия.

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

   ```
   jar xvf corejava.zip
   ```

Навигация по каталогам

Изучая язык Java, вам может понадобиться просмотреть исходные тексты программ. Кроме того, вам, без сомнения, придется интенсивно работать с документацией. В табл. 2.2 показано дерево каталогов JDK

Таблица 2.2. Дерево каталогов Java

Структура каталогов	Описание
jdk	(Имя может быть другим, например `jdk5.0`)
`bin`	Компилятор и инструментальные средства
`demo`	Демонстрационные программы
`docs`	Библиотечная документация в формате HTML (после распаковки архива `jdk-`*версия*`-doc.zip`)
`include`	Файлы для создания платформенно-ориентированных методов (см. том II)
`jre`	Файлы среды исполняющей системы Java
`lib`	Файлы библиотек
`src`	Исходные тексты библиотек (после распаковки файла `src.zip`)

Для начинающих разработчиков, только приступающих к изучению языка Java, наиболее важными среди каталогов являются `docs` и `src`. Каталог `docs` содержит документацию о библиотеках Java в формате HTML. Их можно просматривать с помощью Web-браузера, например Netscape Navigator.



Совет. Установите в браузере закладку на локальную версию файла `docs/api/index.html`. При изучении языка Java вам придется не раз обращаться к этой странице.

Каталог `src` содержит исходные тексты программ, находящихся в общедоступных библиотеках Java. Освоившись с языком, вы можете оказаться в ситуации, когда ни информация из глобальной сети, ни эта книга не помогут вам решить проблему. В этом случае воспользуйтесь исходным кодом. По исходному коду вы всегда сможете понять, что именно делает та или иная библиотечная функция. Например, если вы интересуетесь, как работает класс `System`, загляните в файл `src/java/lang/System.java`.

Выбор среды разработки программ

Если ваш опыт программирования включает применение Microsoft Visual Studio, значит, вы уже знакомы со средой разработки, содержащей встроенный текстовый редактор, меню для компиляции и запуска программ, а также отладчик. Пакет JDK не имеет никаких средств, даже отдаленно напоминающих интегрированную среду разработки программ. *Все* команды выполняются в командной строке. Это может показаться обременительным, но, тем не менее, это важнейшее умение. Когда вы впервые устанавливаете Java, вам придется преодолеть некоторые проблемы, прежде чем вы инсталлируете среду разработки. Более того, выполняя базовые шаги самостоятельно, вы получаете лучшее представление о том, что делает среда разработки у вас за спиной.

Однако после того как вы освоите основные шаги компиляции и выполнения Java-программ, вы наверняка захотите использовать профессиональную среду разработки. За последние 10 лет такие среды стали настолько мощны и удобны, что теперь просто нет особого смысла обходиться без них. Два блестящих выбора, доступных бесплатно — это программы Eclipse и NetBeans. В этой главе мы покажем, с чего начать работу с Eclipse, поскольку эта среда немного более "гладкая", чем NetBeans, хотя и NetBeans быстро распространяется. Конечно, если вы предпочитаете другую среду разработки, вы вполне можете применять ее для работы с примерами из нашей книги.

В прошлом мы рекомендовали для простых программ использовать текстовый редактор, такой как Emacs, JEdit или TextPad. Теперь мы отказались от таких рекомендаций, поскольку среды разработки стали намного быстрее и удобнее.

Итак, мы полагаем, что вы должны знать, как используются базовые инструменты JDK, тогда вы почувствуете себя комфортабельно, имея дело с интегрированной средой разработки.

Использование инструментов командной строки

Для начала выберем самый трудный путь: будем вызывать компилятор и запускать программы на выполнение из командной строки.

1. Откройте окно для ввода команд.
2. Перейдите в каталог `CoreJavaBook/v1ch02/Welcome`. (Если вы помните, каталог `CoreJavaBook` был специально создан для хранения кодов примеров из данной книги, как объяснялось в разделе "Установка примеров программ").
3. Введите следующие команды:

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

На экране должен появиться вывод, показанный на рис. 2.3.

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

Что же произошло? Программа `javac` — это компилятор языка Java. Она скомпилировала файл `Welcome.java` и преобразовала его в файл `Welcome.class`. Программа `java` запускает виртуальную машину Java. Она выполняет байт-код, который компилятор поместил в файл с расширением `.class`.



На заметку! Если вы получили сообщение об ошибке, указывающее на приведенную ниже строку, то, вероятнее всего, используете старую версию компилятора Java. В Java SE 5.0 язык был дополнен новыми возможностями, и мы воспользуемся их преимуществами.

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

Если вы все же хотите продолжать работу со старой версией Java, вам надо переписать цикл следующим образом:

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



Рис. 2.3. Компиляция и выполнение программы `Welcome.java`

Программа Welcome чрезвычайно проста и лишь выводит сообщение на экран. Текст этой программы представлен в листинге 2.1 (как она работает, мы объясним в следующей главе).

Листинг 2.1. Содержимое файла Welcome.java

```
1. /**
2. * Эта программа отображает приветствие от авторов.
3. * @version 1.20 2004-02-28
4. * @author Cay Horstmann
5. */
6. public class Welcome
7. {
8.   public static void main(String[] args)
9.   {
10.    String[] greeting = new String[3];
11.    greeting[0] = "Welcome to Core Java";
12.    greeting[1] = "by Cay Hortsmann";
13.    greeting[2] = "and Gary Cornell";
14.
15.    for (String g : greeting)
16.      System.out.println(g);
17.  }
18. }
```

Возможные ошибки

В эпоху визуальных сред разработки программ многие программисты не умеют работать в режиме командной строки. Такое неумение чревато неприятными ошибками.

Работая с командной строкой, надо учитывать перечисленные ниже моменты.

- Если вы вводите код программы вручную, внимательно следите за употреблением прописных и строчных букв. В частности, в рассмотренном выше примере имя класса — Welcome, а не welcome или WELCOME.
- Компилятор требует указывать имя файла, в данном случае Welcome.java. При запуске требуется *имя класса*, в данном случае это Welcome, без расширения .java или .class.
- Если вы получили сообщение "Bad command or file name" или "javac:command not found", проверьте, правильно ли выполнена инсталляция, в частности, верно ли указаны пути к исполняемым файлам.
- Если компилятор javac выдал сообщение "cannot read: Welcome.java", нужно проверить, имеется ли файл в соответствующем каталоге.

  При работе под управлением операционной системы Unix проверьте, правильно ли использованы прописные буквы в имени файла Welcome.java.

  При работе в среде Windows просматривайте содержимое каталогов с помощью команды dir, а не графическим проводником. Некоторые текстовые редакторы (в частности, Notepad) сохраняют текст в файлах с расширением .txt. Если вы используете эту программу для редактирования файла Welcome.java, она сохранит его под именем Welcome.java.txt. По умолчанию в среде Windows проводник скрывает расширение .txt, поскольку оно предполагается по умолчанию. В этом случае нужно переименовать файл, пользуясь командой ren, либо повторно сохранить его, указав имя в кавычках, например "Welcome.java".

- Если при запуске вашей программы вы получаете сообщение об ошибке `java.lang.NoClassDefFoundError`, проверьте, правильно ли вы указали имя файла.

  Если вы получаете сообщение касательно имени `welcome` (начинающегося со строчной буквы), повторно выполните команду `java Welcome`, написав имя с прописной буквы `W`. В языке Java учитывается регистр символов. Если интерпретатор выдает сообщение об ошибке, в котором фигурирует имя `Welcome/java`, значит, вы случайно ввели команду `java Welcome.java`. Укажите `java Welcome`.

- Если вы задали имя `Welcome`, а виртуальная машина не может найти класс с этим именем, то проверьте, не установил ли кто-нибудь в вашей системе переменную окружения `CLASSPATH`. (Обычно не стоит устанавливать эту переменную глобально, но некоторые плохо написанные инсталляторы программного обеспечения в Windows это делают.) Вы можете временно отменить переменную окружения `CLASSPATH` в текущей командной оболочке, введя команду:

  ```
  set CLASSPATH=
  ```

  Данная программа работает в Windows, а также в Unix/Linux с оболочкой C shell. Если в Unix/Linux применяется оболочка Bourne/bash, надо использовать следующее выражение:

  ```
  export CLASSPATH=
  ```

- Если сообщение об ошибках касается новых языковых конструкций, убедитесь, что ваш компилятор поддерживает Java SE 5.0.

- Если в вашей программе слишком много ошибок, то все сообщения о них мелькают на экране очень быстро. Компилятор `java` выводит подобные сообщения в стандартный поток ошибок, и если информация занимает больше одного экрана, то ее желательно записать в файл.

  Воспользуйтесь операцией перенаправления `2>` для сохранения сообщений об ошибках в файле:

  ```
  javac MyProg.java 2> errors.txt
  ```



Совет. Превосходное учебное пособие приведено на Web-странице `http://java.sun.com.docs/books/tutorial/getStarted/cupojava`. В нем подробно описываются проблемы, с которыми сталкиваются начинающие программисты.

Использование интегрированной среды разработки

В этом разделе мы покажем, как скомпилировать программу в интегрированной среде Eclipse. Данный продукт распространяется свободно; вы можете скопировать его с Web-страницы `http://eclipse.org`. Программа Eclipse написана на языке Java, но вследствие применения нестандартных библиотек для работы с окнами переносимость ее ограничена. Тем не менее, существуют версии Eclipse для операционных систем Linux, Mac OS X, Solaris и Windows.

Существуют и другие популярные интегрированные среды, но Eclipse распространена наиболее широко. Ниже описаны шаги, которые вам потребуется выполнить.

1. После запуска Eclipse выберите пункт меню File⇨New Project (Файл⇨Создать проект).

2. Выберите Java Project (Проект Java) в диалоговом окне мастера (рис. 2.4). На данном рисунке показан образ экрана из Eclipse 3.2. В вашей версии Eclipse окно может несколько отличаться.

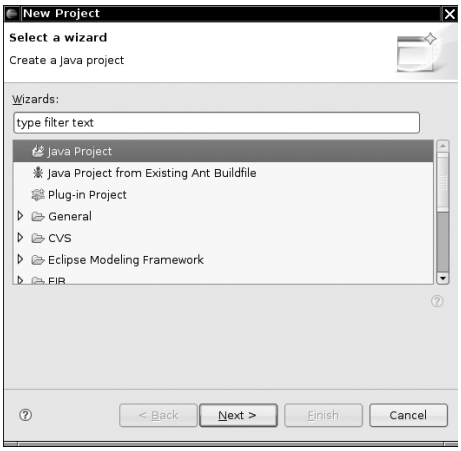


Рис. 2.4. Диалоговое окно Eclipse, предназначенное для создания проекта

3. Щелкните на кнопке Next (Далее), укажите в качестве имени проекта `Welcome` и введите полный путь к каталогу, содержащему файл `Welcome.java` (рис. 2.5).
4. Убедитесь, что переключатель Create project in workspace (Создать проект в рабочем пространстве) не выбран.
5. Щелкните на кнопке Finish (Готово). Проект создан.



Рис. 2.5. Настройка проекта Eclipse

6. Щелкните на треугольном маркере, соответствующем окну проекта, а затем на маркере рядом с пунктом (default package) (Пакет по умолчанию). Дважды щелкните на `Welcome.java`. На экране отобразится окно с кодом программы, показанное на рис. 2.6.

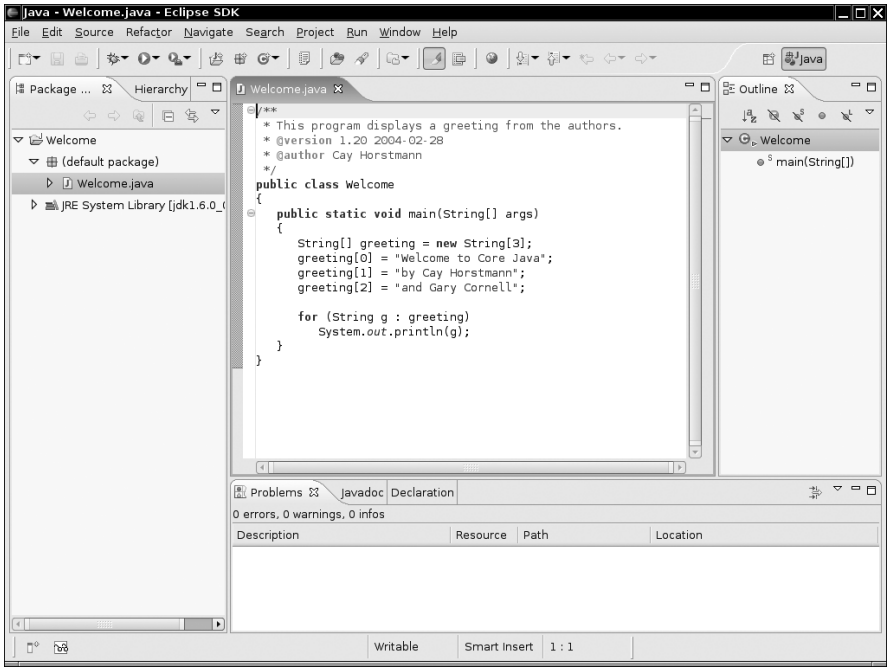


Рис. 2.6. Редактирование исходного кода при работе с Eclipse

7. Щелкните правой кнопкой мыши в левой панели на имени проекта (`Welcome`). В появившемся контекстном меню выберите пункт Run⇨Run As⇨Java Application (Выполнить⇨Выполнить как⇨Приложение Java). В нижней части окна появится окно для отображения выходных данных, в котором будут выведены результаты выполнения программы (рис. 2.7).

Сообщения об ошибках компиляции

Поскольку программа состоит из нескольких строк, вряд ли в ней есть ошибка или даже опечатка. Но для того, чтобы продемонстрировать обработку ошибок, предположим, что в имени `String` вместо прописной буквы вы указали строчную:

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

Снова попытайтесь скомпилировать программу. Вы получите сообщение об ошибке, которое информирует о том, что использован неизвестный тип `string` (рис. 2.8). Просто щелкните на сообщении. Курсор перейдет на соответствующую строку в окне редактирования, и вы сможете быстро исправить ошибку.



Совет. Часто в Eclipse сообщение об ошибке сопровождается пиктограммой с изображением лампочки. Щелкните по этой лампочке, чтобы получить список рекомендуемых способов исправления ошибки.

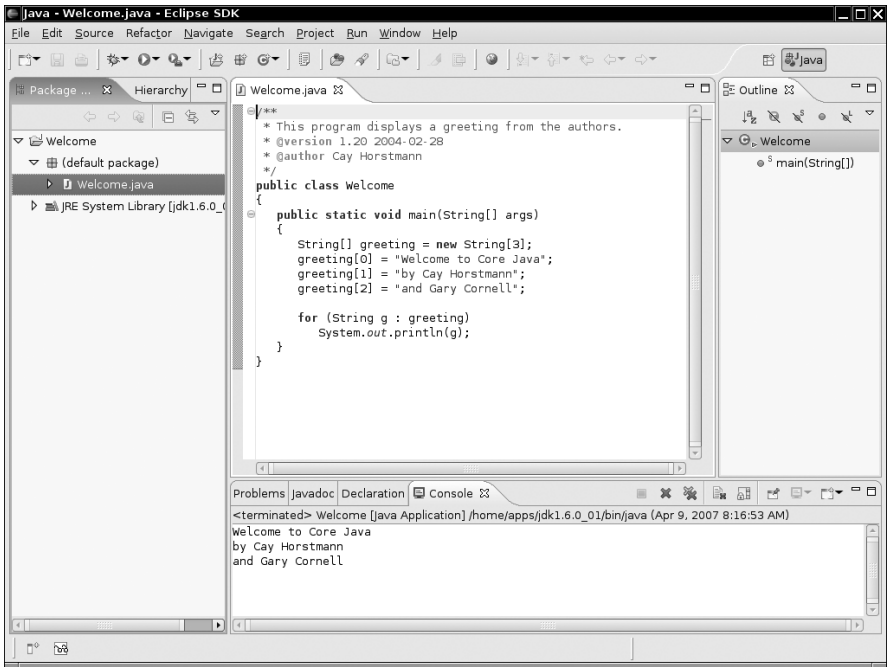


Рис. 2.7. Выполнение программы в среде Eclipse

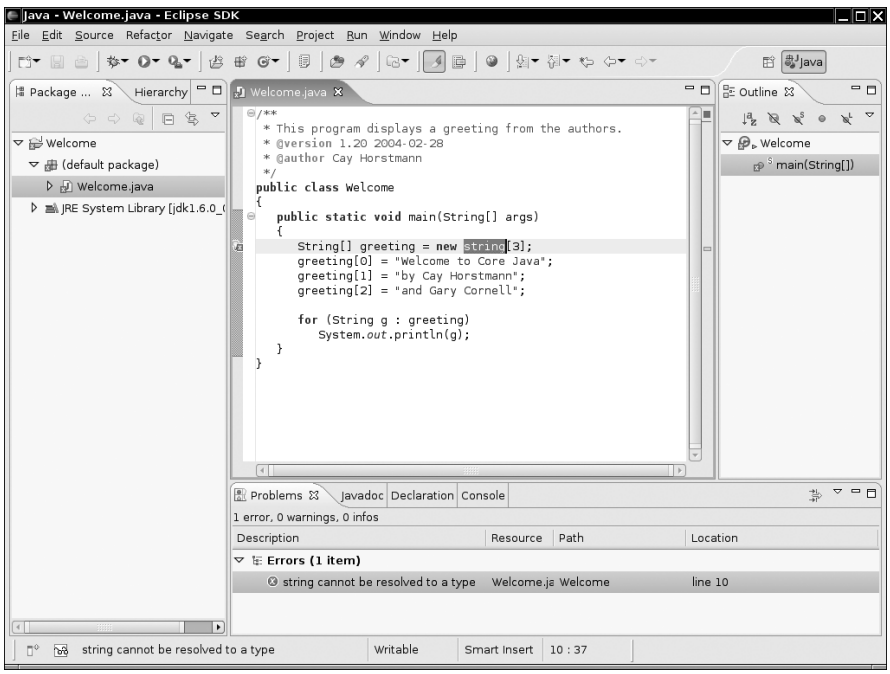


Рис. 2.8. Сообщение об ошибке, отображаемое в среде Eclipse

Таким образом, на простом примере вы получили представление о работе в среде Eclipse. Отладчик Eclipse будет рассмотрен в главе 11.

Выполнение графического приложения

Программа `Welcome` не особо впечатляет. Перейдем теперь к графическому приложению. Это приложение представляет собой очень простую программу, которая выводит на экран изображение из файла. Как и прежде, скомпилируем и выполним это приложение в режиме командной строки.

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

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

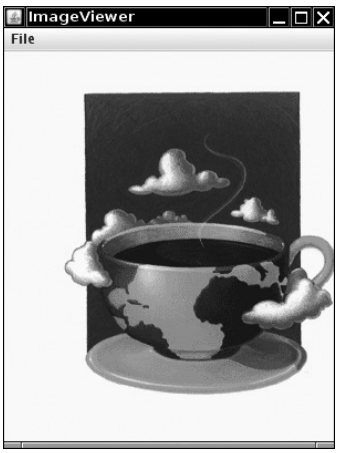


Рис. 2.9. Окно выполняющегося приложения `ImageViewer`

На экране появится новое окно приложения `ImageViewer` (рис. 2.9).

Выберите пункт меню File⇨Open (Файл⇨Открыть) и найдите файл с расширением `.gif`, который вы хотите открыть. (Мы поместили в каталог, в котором находится программа, несколько графических файлов.)

Чтобы завершить выполнение программы, щелкните на кнопке Close (Закрыть) в строке заголовка или откройте системное меню и закройте программу. (Чтобы скомпилировать и выполнить эту программу в текстовом редакторе или интегрированной среде, нужно сделать то же, что и прежде. Например, при работе с текстовым редактором Emacs выберите пункт меню JDE⇨Compile (JDE⇨Скомпилировать), а затем JDE⇨Run App (JDE⇨Выполнить приложение).)

Мы надеемся, что эта программа будет для вас интересной и полезной. Рассмотрим вкратце ее исходный текст. Эта программа существенно длиннее, чем первая, однако она не слишком сложна, особенно если представить себе, сколько строк кода на языке C или C++ нужно было бы написать, чтобы достичь того же эффекта. Конечно, на Visual Basic такую программу написать совсем легко, для этого достаточно перетащить несколько компонентов с помощью мыши и добавить несколько строк, чтобы код стал функциональным. В пакет JDK не входит инструмент для создания интерфейса, поэтому для всех элементов нужно писать соответствующий код, как показано в листинге 2.2. Написанию подобных программ посвящены главы 7–9.

Листинг 2.2. Содержимое файла `ImageViewer.java`

```
1. import java.awt.EventQueue;
2. import java.awt.event.*;
3. import java.io.*;
4. import javax.swing.*;
5.
6. /**
```

```
7.  * Программа для просмотра изображений.
8.  * @version 1.22 2007-05-21
9.  * @author Cay Horstmann
10. */
11. public class ImageViewer
12. {
13.  public static void main(String[] args)
14.  {
15.    EventQueue.invokeLater(new Runnable()
16.     {
17.       public void run()
18.       {
19.          JFrame frame = new ImageViewerFrame();
20.          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21.          frame.setVisible(true);
22.       }
23.     });
24.  }
25. }
26.
27. /**
28.   Фрейм с текстовой меткой для вывода изображения.
29. */
30. class ImageViewerFrame extends JFrame
31. {
32.   public ImageViewerFrame()
33.   {
34.     setTitle("ImageViewer");
35.     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
36.
37.     // Для вывода изображений используется метка.
38.     label = new JLabel();
39.     add(label);
40.
41.     // Диалоговое окно для выбора файлов.
42.     chooser = new JFileChooser();
43.     chooser.setCurrentDirectory(new File("."));
44.
45.     // Создание строки меню.
46.     JMenuBar menuBar = new JMenuBar();
47.     setJMenuBar(menuBar);
48.
49.     JMenu menu = new JMenu("File");
50.     menuBar.add(menu);
51.
52.     JMenuItem openItem = new JMenuItem("Open");
53.     menu.add(openItem);
54.     openItem.addActionListener(new ActionListener()
55.       {
56.         public void actionPerformed(ActionEvent event)
57.         {
58.           // Отображение диалогового окна для выбора файла.
59.           int result = chooser.showOpenDialog(null);
60.
61.           // Если файл выбран, он задается в качестве пиктограммы для метки.
62.           if (result == JFileChooser.APPROVE_OPTION)
```

```
63.             {
64.                String name = chooser.getSelectedFile().getPath();
65.                label.setIcon(new ImageIcon(name));
66.             }
67.          }
68.       });
69.    JMenuItem exitItem = new JMenuItem("Exit");
70.    menu.add(exitItem);
71.    exitItem.addActionListener(new ActionListener()
72.       {
73.          public void actionPerformed(ActionEvent event)
74.          {
75.             System.exit(0);
76.          }
77.       });
78. }
79.
80. private JLabel label;
81. private JFileChooser chooser;
82. private static final int DEFAULT_WIDTH = 300;
83. private static final int DEFAULT_HEIGHT = 400;
84. }
```

Создание и запуск аплетов

Первые два примера, представленные в этой книге, — это *приложения*, написанные на языке Java, т. е. независимые программы, аналогичные обычным платформенно-ориентированным продуктам. С другой стороны, как упоминалось в предыдущей главе, всеобщий интерес к языку Java был вызван в основном его возможностями выполнять *аплеты* под управлением Web-браузеров. В данном разделе мы рассмотрим создание и выполнение аплета в режиме командной строки. Затем мы загрузим этот аплет в специальную программу просмотра (`appletviewer`), входящую в состав пакета JDK. В завершение мы отобразим его в окне Web-браузера.

Для начала откройте окно командной оболочки и перейдите в каталог `CoreJavaBook/v1ch2/WelcomeApplet`, а затем введите следующие команды:

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

Внешний вид окна программы для просмотра аплетов показан на рис. 2.10.



Рис. 2.10. Аплет `WelcomeApplet`, отображающийся в окне `appletviewer`

Первая команда нам уже знакома — она вызывает компилятор языка Java. В процессе компиляции исходный текст аплета, содержащийся в файле `WelcomeApplet.java`, преобразуется в байт-код, который помещается в файл `WelcomeApplet.class`.

Однако на этот раз мы не запускаем программу `java`. Вместо этого мы пользуемся `appletviewer`. Эта программа специально предназначена для быстрого тестирования аплетов и входит в пакет JDK. Исходными данными для `appletviewer` является файл в формате HTML. Содержимое файла `WelcomeApplet.html` показано в листинге 2.3.

Листинг 2.3. Содержимое файла `WelcomeApplet.html`

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

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

Однако при работе с браузерами не исключены проблемы.

- Firefox поддерживают Java в системах Windows, Linux и Mac OS X. Если вы хотите поэкспериментировать с аплетами, вам надо просто загрузить последнюю версию, зайти на `http://java.com` и использовать средство проверки версий (version checker), чтобы узнать, нужно ли инсталлировать подключаемый модуль Java Plug-In.
- Некоторые версии браузера Internet Explorer вообще не поддерживают язык Java. Другие версии поддерживают лишь очень старую виртуальную машину Java компании Microsoft. Если вы работаете с браузером Internet Explorer под управлением операционной системы Windows, обратитесь по адресу `http://java.com` и установите Java Plug-In.
- Если вы работаете на платформе Macintosh под управлением OS X, то здесь браузер Safari уже интегрирован с реализацией Macintosh Java, которая на момент написания нашей книги поддерживает версию Java SE 5.0.

Убедившись, что ваш браузер поддерживает современную версию Java, попробуйте загрузить в него аплет. Для этого выполните описанные ниже шаги.

1. Запустите браузер.
2. Выберите пункт меню File⇨Open File (Файл⇨Открыть файл) или откройте файл другим способом.
3. Перейдите в каталог `CoreJavaBook/v1ch2/WelcomeApplet`. Здесь вы должны увидеть файл `WelcomeApplet.html`. Загрузите этот файл.
4. Теперь ваш браузер загрузит аплет, включая окружающий текст документа. Web-страница будет выглядеть приблизительно так, как показано на рис. 2.11.

Нетрудно заметить, что это приложение действительно интерактивно и взаимодействует с Internet. Щелкните на кнопке Cay Horstmann, и на экране отобразится страница Кея Хорстманна. Щелкните кнопку Gary Cornell. Аплет откроет окно для ввода электронной почты, в котором уже указан адрес Гари Корнелла.

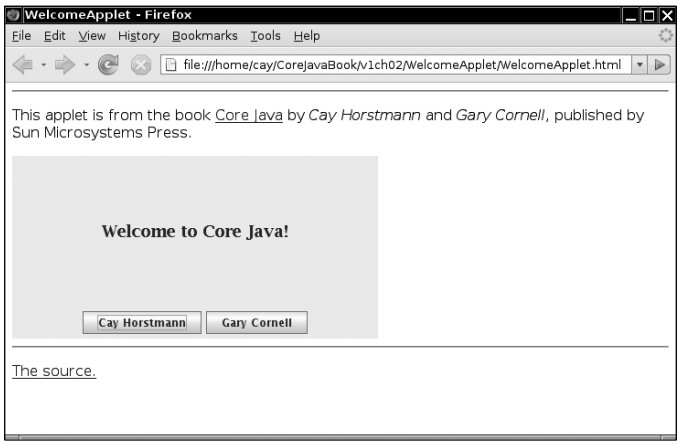


Рис. 2.11. Просмотр аплета `WelcomeApplet` с помощью браузера

Обратите внимание, что ни одна из этих кнопок не будет работать с программой просмотра аплетов. Программа просмотра аплетов не может посылать почту или отображать на экране Web-страницу, поэтому она попросту игнорирует соответствующие запросы. Программа просмотра аплетов хороша для тестирования независимых аплетов, однако для проверки реального взаимодействия аплета по сети с другими программами следует выполнить его в среде, реализуемой браузером и Internet.



Совет. Аплет можно запустить из текстового редактора или интегрированной среды разработки. При работе с редактором Emacs выберите пункт меню JDE⇨Run Applet (JDE⇨Выполнить аплет). Работая с Eclipse, выберите пункт Run⇨Run as⇨Java Applet (Выполнить⇨Выполнить как⇨Аплет Java).

Код аплета `Welcome` показан в листинге 2.4. В данный момент его нужно лишь бегло просмотреть. К созданию аплетов мы еще вернемся в главе 10.

Листинг 2.4. Содержимое файла `WelcomeApplet.java`

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

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

Итак, в этой главе мы рассмотрели механизмы компиляции и запуска программ, написанных на языке Java. Теперь мы готовы перейти к главе 3, в которой приступим к изучению языка Java.

Глава 3

ОСНОВНЫЕ КОНСТРУКЦИИ ЯЗЫКА JAVA

В ЭТОЙ ГЛАВЕ...

- Простая программа на языке Java
- Комментарии
- Типы данных
- Переменные
- Операции
- Строки
- Ввод и вывод
- Поток управления
- Большие числа
- Массивы

Будем считать, что вы успешно инсталлировали JDK и смогли запустить простые программы, приведенные в главе 2. Настало время приступать непосредственно к программированию. В этой главе вы узнаете, как в языке Java реализуются основные концепции программирования, например, типы данных, ветви и циклы.

К сожалению, на языке Java нелегко написать программу с графическим интерфейсом, — для этого нужно изучить множество вопросов, связанных с окнами, полями ввода, кнопками и т.п. Поскольку описание таких технических подробностей увело бы нас далеко в сторону от нашей основной цели — анализа основных языковых конструкций, в этой главе мы рассмотрим лишь простые программы, иллюстрирующие то или иное понятие. Все эти программы используют для ввода и вывода данных окно командной оболочки.

Отметим, что программисты, имеющие опыт работы на языке C/C++, могут просто бегло просмотреть эту главу. Сосредоточьте внимание на комментариях C/C++, которые разбросаны по тексту. Разработчикам, использующим другие языки, например Visual Basic, многие понятия также окажутся знакомыми, хотя синтаксис выражений будет существенно отличаться. Таким читателям мы советуем тщательно изучить эту главу.

Простая программа на языке Java

Рассмотрим самую простую программу, какую только можно себе представить. В процессе работы она лишь выводит сообщение в консольное окно.

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

Этому примеру стоит посвятить столько времени, сколько потребуется, чтобы привыкнуть к особенностям языка и посмотреть на характерные особенности Java-программ, которые будут встречаться во всех приложениях. Первое, на что надо обратить внимание, — *в языке Java учитывается регистр символов*. Если вы перепутаете их (например, введете `Main` вместо `main`), программа выполняться не будет.

Теперь просмотрим исходный код построчно. Ключевое слово `public` называется *модификатором доступа* (access modifier); такие модификаторы управляют обращением к коду из других частей программы. Более подробно мы рассмотрим этот вопрос в главе 5. Ключевое слово `class` напоминает нам, что все элементы Java-программ находятся в составе классов. Классы будут детально рассматриваться в следующей главе, а пока мы будем считать их некими "контейнерами", в которых реализована логика программы, определяющая работу приложения. Как указывалось в главе 1, классы — это "строительные блоки", из которых состоят все приложения и аплеты, написанные на языке Java. *Все*, что есть в Java-программе, должно находиться внутри класса.

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

Согласно соглашениям об именовании, имя класса должно начинаться с прописной буквы (именно так сформировано имя `FirstSample`). Если имя состоит из нескольких слов, каждое из них должно начинаться с прописной буквы. (Правила, по которым в середине слова может стоять символ верхнего регистра, иногда называют "camel case" ("верблюжий стиль"), или, в соответствии с этими же правилами, `CamelCase`.)

Файл, содержащий исходный текст, должен называться так же, как и общедоступный (`public`) класс, и иметь расширение `.java`. Таким образом, код рассматриваемого здесь класса мы должны поместить в файл `FirstSample.java`. (Как и следовало ожидать, регистр символов учитывается, поэтому имя `firstsample.java` не подходит.)

Если вы правильно назвали файл и не допустили ошибок в исходном тексте программы, то в результате компиляции получите файл, содержащий байт-код данного класса. Компилятор языка Java автоматически назовет этот файл `FirstSample.class` и сохранит его в том же каталоге, в котором содержится исходный файл. Осталось запустить программу с помощью интерпретатора языка Java, набрав команду

```
java FirstSample
```

(Расширение `.class` не указывается!) Выполняясь, программа выведет на экран строку `"We will not use 'Hello, World'!"`.

Когда для запуска скомпилированной программы используется команда

```
java имя_класса
```

виртуальная машина Java всегда начинает свою работу с выполнения метода `main()` указанного класса (Термин "метод" — это принятый в Java эквивалент функции.) Следовательно, чтобы программа могла выполняться, в классе *должен присутствовать* метод `main()`. Разумеется, в класс можно добавить и другие методы и вызывать их из метода `main()`. (Мы покажем, как создавать такие методы, в следующей главе.)



На заметку! В соответствии со спецификацией языка Java метод `main()` должен быть объявлен как `public`. (Спецификация языка Java является официальным документом. Его можно скопировать, обратившись по адресу `http://java.sun.com/docs/books/jls`.) Однако некоторые версии интерпретатора Java допускали выполнение программ, даже когда метод `main()` не имел модификатора `public`. Эта ошибка была помещена в список замеченных ошибок, представленный на сайте `http://bugs.sun.com/bugdatabase/index.jsp`, и получила номер 4252539. Однако она была помечена как "исправлению не подлежит". Разработчики компании Sun выступили с разъяснениями, что спецификация виртуальной машины языка Java не требует, чтобы метод `main()` был общедоступным (см. Web-страницу `http://java.sun.com/docs/books/vmspec`), а попытка исправить эту ошибку "может вызвать проблемы". К счастью, здравый смысл в итоге восторжествовал. Интерпретатор языка Java в пакете Java SE 1.4 требует, чтобы метод `main()` был общедоступным.

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

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

Стиль расстановки фигурных скобок всегда вызвал споры. Обычно мы стараемся располагать скобки одну под другой, выравнивая их с помощью пробелов. Поскольку пробелы не имеют значения для компилятора Java, вы можете применять любой стиль расположения скобок, который вам нравится. Позднее, изучая различные операторы цикла, мы поговорим о скобках более подробно.

Пока мы не будем обращать внимание на ключевые слова `static void`, считая их просто необходимой частью программы на языке Java. В конце главы 4 мы полностью раскроем смысл этих слов. Сейчас важно помнить, что каждое приложение на языке Java должно иметь метод `main()`, заголовок которого приведен ниже.

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



На заметку C++! Если вы программируете на языке C++, то, конечно же, знаете, что такое класс. Классы в Java похожи на классы в C++, однако между ними есть существенные различия. Например, в языке Java *все* функции являются методами того или иного класса. (Термин метод в Java соответствует термину функция-член в C++.) Следовательно, в языке Java должен существовать класс, которому принадлежит метод `main()`. Вероятно, вы знакомы с понятием *статических функций*-членов в языке C++. Это функции-члены, определенные внутри класса и не принадлежащие ни одному объекту этого класса. Метод `main()` в языке Java всегда является статическим. В заключение, как и в языке C/C++, ключевое слово `void` означает, что метод не возвращает никакого значения. В отличие от C/C++, метод `main()` не передает операционной системе код завершения. Если данный метод корректно завершает свою работу, код завершения равен 0. Чтобы изменить код завершения, надо использовать метод `System.exit()`.

Теперь обратите внимание на следующий фрагмент кода:

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

Фигурные скобки отмечают начало и конец *тела* метода, содержащего лишь одну строку кода. Как и в большинстве языков программирования, операторы Java можно считать предложениями обычного языка. В языке Java каждый оператор должен заканчиваться точкой с запятой. В частности, символ конца строки не означает конец оператора, поэтому оператор может занимать столько строк, сколько потребуется.

В нашем случае при выполнении метода `main()` на консоль выводится одна строка текста.

В данном примере мы используем объект `System.out` и вызываем его метод `println()`. Обратите внимание, что метод отделяется от объекта точкой. В общем случае вызов метода в Java имеет следующий вид:

объект.метод(*параметры*)

что эквивалентно вызову функции.

В нашем примере мы вызываем метод `println()`, передавая ему в качестве параметра текстовую строку. Метод выводит строку текста на консоль, дополняя ее символом перевода строки. В языке Java, как и в языке C/C++, строковый литерал помещается в двойные кавычки. (Далее в этой главе мы рассмотрим работу со строками подробнее.)

Методам в языке Java, как и функциям в любом другом языке программирования, может передаваться ноль, один или несколько *параметров* (В некоторых языках параметры принято называть *аргументами*). Даже если метод не имеет параметров, после его имени надо ставить пустые скобки. Например, при вызове метода `println()` без параметров на экран выводится пустая строка. Такой вызов выглядит следующим образом:

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


На заметку! В объекте `System.out` есть метод `print()`, который выводит строку текста, не добавляя к ней символ перехода на новую строку. Например, выражение `System.out.print("Hello")` выводит текст "Hello" и оставляет курсор в конце строки. Следующие данные, выводимые на экран, появятся сразу за буквой "o".

Комментарии

Комментарии в языке Java, как и в большинстве языков программирования, игнорируются при выполнении программы. Таким образом, в программу можно добавлять столько комментариев, сколько потребуется, не опасаясь увеличить ее объем. В языке Java есть три способа выделения комментариев в тексте. Чаще всего используются две косые черты (//), при этом комментарий начинается сразу за символами // и продолжается до конца строки.

```
System.out.println("We will not use 'Hello, World!'");
// Нам не нужен 'Hello, World!' Остроумно, не правда ли?
```

Если нужны комментарии, состоящие из нескольких строк, можно каждую строку начинать символами //. Кроме того, для создания больших блоков комментариев можно использовать разделители /* и */, как показано в листинге 3.1.

Листинг 3.1. Содержимое файла `FirstSample.java`

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

В заключение отметим, что в языке Java есть и третья разновидность комментариев, которую можно использовать для автоматической генерации документации. Эти комментарии начинаются символами /** и заканчиваются символами */. Более подробную информацию об этом виде комментариев и автоматической генерации документации можно найти в главе 4.



Внимание! Комментарии, выделяемые символами `/*` и `*/`, в языке Java не могут быть вложенными. Это значит, что фрагмент кода нельзя отключить, просто окружив его парами символов `/*` и `*/`, поскольку в составе этого кода в свою очередь могут содержаться разделители `/*` и `*/`.

Типы данных

Язык Java является *строго типизированным*. Это значит, что тип каждой переменной должен быть объявлен. В языке Java есть восемь *основных, или простых, типов* (primitive types) данных. Четыре из них представляют целые числа, два — действительные числа с плавающей точкой, один — символы в формате Unicode (см. раздел "Тип `char`") и последний — логические значения.



На заметку! В языке Java предусмотрен пакет для выполнения арифметических действий с произвольной точностью. Однако так называемые "большие числа" в языке Java являются *объектами* и не являются новым типом Java. Позднее в этой главе мы покажем, как с ними работать.

Целые числа

Целочисленные типы используются для представления как положительных, так и отрицательных чисел, не имеющих дробной части. В языке Java есть четыре целочисленных типа. Они представлены в табл. 3.1.

Таблица 3.1. Целочисленные типы в языке Java

Тип	Требуемый объем памяти (байты)	Диапазон (включительно)
`int`	4	от –2147483648 до 2147483647 (больше 2 миллиардов)
`short`	2	от –32768 до 32767
`long`	8	от –9223372036854775808 до –9223372036854775807
`byte`	1	от –128 до 127

В большинстве случаев тип `int` наиболее удобен. Если нужно задать количество жителей на вашей планете, нет никакой необходимости прибегать к типу `long`. Типы `byte` и `short` в основном используются в специальных приложениях, например, при низкоуровневой обработке файлов или с целью экономии памяти при формировании больших массивов, когда важен размер хранилища.

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

Длинные целые числа имеют суффикс L (например, 4000000000L). Шестнадцатеричные числа имеют префикс 0x (например, 0xCAFE). Восьмеричные числа имеют префикс 0. Например, 010 — это число 8. Такая запись иногда приводит к недоразумениям, поэтому мы не рекомендуем применять восьмеричные числа.



На заметку! В языках C и C++ int означает целочисленный тип, зависящий от машины, для которой предназначена программа. На 16-разрядном процессоре, например процессоре 8086, целые числа занимают 2 байта. На 32-разрядном процессоре, например процессоре Sun SPARK, они занимают 4 байта. На процессоре Intel Pentium размер целого типа в языках C и C++ зависит от операционной системы: в DOS и Windows 3.1 целые числа занимают 2 байта. При использовании 32-разрядного режима работы в системе Windows целые числа занимают 4 байта. В языке Java размеры всех числовых типов не зависят от платформы, на которой выполняется программа.

Заметим, что в языке Java нет беззнаковых типов unsigned.

Числа с плавающей точкой

Типы с плавающей точкой представляют значения, имеющие дробную часть. В языке Java есть два типа для чисел с плавающей точкой, они приведены в табл. 3.2.

Таблица 3.2. Числа с плавающей точкой в языке Java

Тип	Требуемый объем памяти (байты)	Диапазон
float	4	Приблизительно ±3,40282347E+38F (6-7 значащих десятичных цифр)
double	8	Приблизительно ±1,7976931348623157E+308F (15 значащих десятичных цифр)

Имя double означает, что точность этих чисел вдвое превышает точность чисел типа float. (Некоторые называют их числами *с двойной точностью*.) В большинстве приложений тип double является наиболее удобным. Ограниченной точности чисел типа float во многих случаях попросту недостаточно. Семи значимых (десятичных) цифр, возможно, хватит для того, чтобы точно выразить вашу годовую зарплату в долларах и центах, но не зарплату президента вашей компании. Причины, по которой тип float все еще используется, — это скорость обработки чисел (для чисел типа float она выше) и экономия памяти при хранении (это важно для больших массивов действительных чисел).

Числа типа float имеют суффикс F, например 3.402F. Числа с плавающей точкой, не имеющие суффикса F (например, 3.402), всегда рассматриваются как числа типа double. Для их представления можно (но не обязательно) использовать суффикс D, например 3.402D.



На заметку! В Java SE 5.0 вы можете специфицировать числа с плавающей точкой в шестнадцатеричном виде! Например, 0.125 = 2^{-3} можно записать как 0x1.0p-3. В шестнадцатеричной нотации вы используете p, а не e, чтобы отделить экспоненту. Обратите внимание, что мантисса записывается в шестнадцатеричном виде, а экспонента — в десятичном. База экспоненты — 2, а не 10.

Все вычисления, производящиеся над числами с плавающей точкой, следуют стандарту IEEE 754. В частности, в языке Java есть три специальных значения с плавающей точкой:

- положительная бесконечность;
- отрицательная бесконечность;
- NaN (не число).

Например, результат деления положительного числа на 0 равен положительной бесконечности. Вычисление 0/0 или извлечение квадратного корня из отрицательного числа равно NaN.



На заметку! В языке Java существуют константы `Double.POSITIVE_INFINITY`, `Double.NEGATIVE_INFINITY` и `Double.NaN` (а также соответствующие константы типа `float`). Однако на практике они редко используются. В частности, для того, чтобы убедиться, что некий результат равен константе `Double.NaN`, нельзя выполнить проверку

```
if (x == Double.NAN) // никогда не будет true
```

Все величины, "не являющиеся числами", считаются разными. Однако можно вызывать метод `Double.isNaN()`:

```
if (Double.isNaN(x)) // проверка, является ли не числом
```



Внимание! Числа с плавающей точкой нельзя использовать в финансовых вычислениях, где ошибки округления недопустимы. Например, в результате выполнения команды `System.out.println(2.0 - 1.1)` будет выведено не `0.9`, как логично ожидать, а `0.8999999999999999`. Подобные ошибки связаны с внутренним двоичным представлением чисел. Подобно тому, как в десятичной системе счисления нельзя точно представить результат деления 1/3, так и в двоичной системе невозможно точно представить 1/10. Если вам надо исключить ошибки округления, следует использовать класс `BigDecimal`, который будет рассмотрен далее в этой главе.

Тип `char`

Тип `char` используется для представления индивидуальных символов. Чаще всего это будут символьные константы. Например, символьной константой является `'A'`, которой соответствует значение 65. Не следует путать символ `'A'` со строкой `"A"`, состоящей из одного символа. Кодовые единицы Unicode можно выражать в виде шестнадцатеричных чисел в диапазоне от `\u0000` до `\uFFFF`. Например, значение `\u2122` соответствует символу торговой марки (™), а `\u03C0` — греческой букве π.

Кроме префикса `\u`, который предваряет кодовую единицу Unicode, существует также несколько специальных символьных последовательностей, показанных в табл. 3.3. Эти последовательности можно применять в составе символьных констант или строк, например `'\u2122'` или `"Hello\n"`. Последовательности, начинающиеся с `\u` (и никакие другие), можно даже указывать за пределами символьных констант или строк. Приведенный ниже пример корректен, так как последовательности `\u005B` и `\u005D` соответствуют символам `[` и `]`.

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

Таблица 3.3. Специальные символы

Специальный символ	Описание	Значение Unicode
`\b`	Возврат на одну позицию	`\u0008`
`\t`	Табуляция	`\u0009`
`\n`	Переход на новую строку	`\u000a`
`\r`	Возврат каретки	`\u000d`
`\"`	Двойная кавычка	`\u0022`
`\'`	Одинарная кавычка	`\u0027`
`\\`	Обратная косая черта	`\u005c`

Для того чтобы понимать тип `char`, надо иметь представление о принципах кодировки Unicode. Кодировка Unicode была изобретена для преодоления ограничений традиционных символьных схем. До появления Unicode существовало несколько различных стандартов: ASCII, ISO 8859-1, KOI-8, GB18030, BIG-5 и т.д. При этом возникали две проблемы. Во-первых, один и тот же код в разных кодировках соответствовал различным символам. Во-вторых, в языках с большим набором символов использовался код различной длины: часто употребляющиеся символы представлялись одним байтом, другие знаки — двумя, тремя и бо́льшим количеством байтов.

Для решения этих проблем была разработана кодировка Unicode. В результате исследований, начавшихся в 80-х годах, выяснилось, что двухбайтового кода более чем достаточно для представления всех символов, использующихся во всех языках; при этом оставался достаточный резерв для любых мыслимых расширений. В 1991 г. была выпущена спецификация Unicode 1.0, в которой было использовано меньше половины из возможных 65536 кодов. В Java изначально были приняты 16-битовые символы Unicode, что стало еще одним преимуществом перед другими языками, использующими 8-битовые символы.

Однако впоследствии случилось непредвиденное: количество символов превысило допустимые 65536. Причиной тому стали чрезвычайно большие наборы иероглифов китайского, японского и корейского языков. Поэтому в настоящее время 16-битового типа `char` недостаточно для описания всех символов Unicode.

Чтобы понять, как эта проблема решается в Java, начиная с Java SE 5.0, надо ввести несколько терминов. Назовем *кодовой точкой* (code point) значение, связанное с символом в схеме кодирования. Согласно стандарту Unicode, кодовые точки записываются в шестнадцатеричном формате и предваряются символами U+. Например, для буквы A кодовая точка равна U+0041. В Unicode кодовые точки объединяются в 17 *кодовых плоскостей* (code plane). Первая кодовая плоскость, называемая *основной многоязыковой плоскостью* (basic multilingual plane), состоит из "классических" символов Unicode с кодовыми точками от U+0000 до U+FFFF. Шестнадцать дополнительных плоскостей с кодовыми точками от U+10000 до U+10FFFF содержат *дополнительные символы* (supplementary character).

Кодировка UTF-16 — это способ представления всех кодов Unicode последовательностью переменной длины. Символы из основной многоязыковой плоскости представляются 16-битовыми значениями, называемыми *кодовыми единицами* (code unit). Дополнительные символы обозначаются последовательными парами кодовых единиц. Каждое из значений пары попадает на используемую 2048-байтовую область основной многоязыковой плоскости, называемой *областью подстановки* (surrogates area); от

U+D800 до U+DBFF для первой кодовой единицы и от U+DC00 до U+DFFF для второй кодовой единицы. Такой подход позволяет сразу определить, соответствует ли значение коду конкретного символа или является ли частью кода дополнительного символа. Например, математическому коду символов, обозначающему множество целых чисел, соответствует кодовая точка U+1D56B и две кодовых единицы, U+D835 и U+DD6B (описание алгоритма кодирования можно найти по адресу `http://en.wikipedia.org/wiki/UTF-16`).

В Java тип `char` описывает кодовую единицу UTF-16.

Начинающим программистам мы советуем использовать коды UTF-16 лишь в случае крайней необходимости. Если возможно, используйте строки (они будут рассмотрены ниже в разделе "Строки"), как абстрактные типы данных.

Тип `boolean`

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



На заметку! В языке C++ вместо логических значений можно использовать числа и даже указатели. Значение 0 эквивалентно логическому значению `false`, а ненулевые величины — значению `true`. В языке Java представлять логические значения посредством других типов невозможно. Следовательно, программист на языке Java защищен от недоразумений, подобных следующему:

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

В языке C++ эта строка компилируется и выполняется проверка, причем выражение всегда равно `false`. В языке Java наличие такой строки приведет к ошибке на этапе компиляции, поскольку целочисленное выражение `x = 0` нельзя преобразовать в логическое значение.

Переменные

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

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

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

Имя переменной должно начинаться с буквы и представлять собой сочетание букв и цифр. Термины "буквы" и "цифры" в Java имеют более широкое значение, чем в большинстве других языков программирования. Буквами считаются символы `'A'–'Z'`, `'a'–'z'`, `'_'` и *любой другой* символ в кодировке Unicode, соответствующий букве. Например, немецкие пользователи в именах переменных могут использовать символ `'ä'`, а греческие пользователи могут воспользоваться буквой π. Аналогично цифрами считаются как обычные десятичные цифры, `'0'–'9'`, так и любые символы в кодировке Unicode, использующиеся для обозначения цифры в каком-либо языке. Символы наподобие `'+'` или `'©'`, а также пробел нельзя использовать в именах пере-

менных. Все символы в имени переменной важны, причем *регистр также учитывается*. Длина имени переменной не ограничена.



Совет. Если вы действительно хотите знать, какие символы в формате Unicode считаются "буквами" в языке Java, воспользуйтесь методами `isJavaIdentifierStart()` и `isJavaIdentifierPart()` класса `Character`.

В качестве имен переменных нельзя использовать зарезервированные слова. (Список зарезервированных слов приведен в приложении.)

В одной строке программы можно размещать несколько объявлений, например:

```
int i,j; // Обе переменные — целочисленные.
```

Однако мы не рекомендуем следовать такому стилю. Если объявить каждую переменную в отдельной строке, читать программу станет гораздо легче.



На заметку! Как вы уже видели, в языке Java различаются прописные и строчные буквы, например имена `hireday` и `hireDay` считаются разными. Вообще не стоит использовать два имени, различающиеся только регистром символов. Однако иногда для переменной трудно подобрать подходящее имя. Многие программисты в этих случаях дают переменной имя, совпадающее с именем типа, но отличающееся регистром символов. Например:

```
Box box; // Box - это тип, а box - имя переменной.
```

На наш взгляд, намного лучше использовать в имени переменной префикс `a`:

```
Box aBox;
```

Инициализация переменных

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

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

Для присвоения ранее объявленной переменной какого-либо значения нужно указать слева ее имя, поставить знак равенства (=), а справа записать некоторое выражение на языке Java, задающее требуемое значение.

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

При желании вы можете одновременно объявить и инициализировать переменную. Например:

```
int vacationDays = 12;
```

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

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

При создании программ на Java рекомендуется объявлять переменную как можно ближе к той точке кода, где она будет использована.



На заметку! В языках C и C++ различаются *объявление* и *определение* переменной. Ниже приведен пример определения переменной.

```
int i = 10;
```

Объявление переменной выглядит следующим образом:

```
extern int i;
```

В языке Java объявление и определение переменных не различаются.

Константы

В языке Java для обозначения констант используется ключевое слово `final`, например:

```
public class Constants
{
   public static void main(String[] args)
   {
      final double CM_PER_INCH = 2.54;
      double paperWidth = 8.5;
      double PaperHeight = 11;
      System.out.println("Размер страницы в сантиметрах: "
         + paperWidth * CM_PER_INCH + "на" + paperheight * CM_PER_INCH);
   }
}
```

Ключевое слово `final` означает, что присвоить какое-либо значение данной переменной можно лишь один раз и изменять его нельзя. Использовать в именах констант только прописные буквы необязательно.

В языке Java часто возникает необходимость в константах, доступных нескольким методам внутри одного класса. Обычно они называются *константами класса* (class constant). Константы класса объявляются с помощью ключевых слов `static final`. Ниже приведен пример использования константы класса.

```
public class Constants2
{
   public static void main(String[] args)
   {
      double paperWidth = 8.5;
      double PaperHeight = 11;
      System.out.println("Размер страницы в сантиметрах: "
         + paperWidth * CM_PER_INCH + "на" + paperHeight * CM_PER_INCH);
   }

   public static final double CM_PER_INCH = 2.54;
}
```

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



На заметку! В языке Java слово `const` является зарезервированным, однако сейчас оно уже не употребляется. Для объявления констант следует использовать ключевое слово `final`.

Операции

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

Заметим, что целочисленное деление на 0 генерирует исключение, в то время как результатом деления на 0 чисел с плавающей точкой является бесконечность или NaN.

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

```
x += 4;
```

эквивалентен

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

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



На заметку! Одной из заявленных целей языка Java является переносимость. Вычисления должны приводить к одинаковому результату, независимо от того, какая виртуальная машина их выполняет. Для арифметических вычислений над числами с плавающей точкой это неожиданно оказалось трудной задачей. Тип `double` использует для хранения числовых значений 64 бита, однако некоторые процессоры применяют 80-разрядные регистры с плавающей точкой. Эти регистры обеспечивают дополнительную точность на промежуточных этапах вычисления. Рассмотрим в качестве примера следующее выражение:

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

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

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

В этом случае все команды внутри метода `main()` будут выполнять точные операции над числами с плавающей точкой.

Детали выполнения этих операций тесно связаны с особенностями работы процессоров Intel. По умолчанию промежуточные результаты могут использовать расширенный показатель степени, но не расширенную мантиссу. (Микросхемы компании Intel поддерживают округление мантиссы без потери производительности.) Следовательно, единственное различие между вычислениями по умолчанию и точными вычислениями состоит в возможности переполнения.

Если сказанное кажется вам слишком сложным, не волнуйтесь. Переполнение при вычислениях чисел с плавающей точкой в большинстве случаев не возникает. В примерах, рассматриваемых в этой книге, ключевое слово `strictfp` использоваться не будет.

Операции инкрементирования и декрементирования

Программисты, конечно, знают, что одной из наиболее распространенных операций с числовыми переменными является добавление или вычитание единицы. В языке Java, как и в языках C и C++, есть операции инкрементирования и декрементирования: в результате вычисления выражения `n++` к текущему значению переменной `n` прибавляется единица, а `n--` уменьшает `n` на единицу. Например, в результате обработки следующего фрагмента значение переменной `n` становится равным 13.

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

Поскольку эти операции `++` и `--` изменяют значение переменной, их нельзя применять к самим числам. Например, выражение `4++` является недопустимым.

Существуют два вида операций инкрементирования и декрементирования. Выше показана постфиксная форма, в которой символы операции размещаются после операнда. Есть и префиксная форма — `++n`. Обе эти операции изменяют значение переменной на единицу. Разница между ними проявляется, только когда эти операции присутствуют внутри выражений. Префиксная форма сначала изменяет значение переменной и использует новое значение для дальнейших вычислений, а постфиксная форма использует старое значение этой переменной и лишь после этого изменяет его.

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

Мы не рекомендуем использовать операции инкрементирования и декрементирования внутри выражений, поскольку это зачастую запутывает код и приводит к досадным ошибкам.

(Поскольку именно операция `++` дала имя языку C++, это послужило поводом к первой шутке о нем. Недоброжелатели отмечают, что даже имя этого языка содержит в себе ошибку: “Язык следовало бы назвать ++C, потому что мы хотим использовать его только после улучшения”.)

Операции отношений и логические операции

Язык Java содержит полный комплект операций отношений. Чтобы проверить равенство, следует использовать символы ==. Например, значение выражения `3 == 7` равно `false`.

Для проверки неравенства используются символы `!=`. Так, значение выражения `3 != 7` равно `true`.

Кроме того, в языке Java есть обычные операции < (меньше), > (больше), <= (меньше или равно) и => (больше или равно).

В языке Java, как и в C++, используются символы && для обозначения логической операции "и", а также символы || для обозначения логической операции "или". Как обычно, знак восклицания означает логическую операцию отрицания. Операции && и || задают вычисление по сокращенной схеме, согласно которой, если первый элемент определяет значение всего выражения, то остальные элементы не вычисляются. Рассмотрим два выражения, объединенных операцией &&.

выражение_1 && *выражение_2*

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

```
x != 0 && 1/x > x+y  // Не делить на 0.
```

Таким образом, деление на нуль не происходит.

Аналогично, значение *выражение_1* || *выражение_2* истинно, если истинным является значение первого выражения. В этом случае вычислять второе выражение нет необходимости.

В языке Java есть также тернарная операция ?:, которая иногда оказывается полезной.

условие ? *выражение_1* : *выражение_2*

Если условие истинно, то вычисляется первое выражение, а если ложно — второе выражение. Например, x < y ? x : y возвращает меньшее из чисел x и y.

Битовые операции

Работая с любыми целочисленными типами, можно применять операции, непосредственно обрабатывающие биты, из которых состоят целые числа. Это значит, что для определения состояния отдельных битов числа можно использовать маски. В языке Java есть следующие битовые операции: & ("и") , | ("или"), ^ ("исключающее или"), ~("не"). Например, если n — это целое число, то приведенное ниже выражение равно единице только в том случае, если четвертый бит в двоичном представлении числа равен единице.

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

Используя операцию & в сочетании с соответствующей степенью двойки, можно замаскировать все биты, кроме одного.



На заметку! В применении к логическим переменным операции & и | дают логические значения. Эти операции аналогичны операциям && и ||, за исключением того, что вычисление производится по полной схеме, т.е. обрабатываются все элементы выражения.

В языке Java есть также операции >> и <<, сдвигающие битовое представление числа вправо или влево. Эти операции удобны, если нужно построить битовое представление на основе маски:

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

Имеется даже операция >>>, заполняющая старшие разряды нулями, в то время как операция >> восстанавливает в старших разрядах знаковый бит. Операции вроде <<< в языке Java нет.



Внимание! Значение, присутствующее в правой части операций битового сдвига, сокращается по модулю 32 (если левая часть является числом типа `long`, правая часть сокращается по модулю 64). Например, значение выражения `1<<35` эквивалентно выражению `1<<3`, или 8.



На заметку! В языках C/C++ не определено, какой сдвиг выполняет операция >>: арифметический (при котором знаковый бит восстанавливается) или логический (при котором старшие разряды заполняются нулями). Разработчики средств реализации языка могут выбрать тот вариант, который покажется им более эффективным. Это значит, что результат выполнения операции сдвига вправо в языке C/C++ определен лишь для неотрицательных чисел. В языке Java эта неоднозначность удалена.

Математические функции и константы

Класс `Math` содержит набор математических функций, которые часто оказываются необходимыми при решении практических задач.

Чтобы извлечь квадратный корень из числа, применяют метод `sqrt()`.

```
double x = 4;
double y = Math.sqrt(x);
System.out.println(y); // Выводит число 2.0.
```



На заметку! Между методами `println()` и `sqrt()` есть тонкая разница. Метод `println()` принадлежит объекту `System.out`, определенному в классе `System`. Но метод `sqrt()` принадлежит классу `Math`, а не объекту. Такие методы называются *статическими*. Они будут рассматриваться в главе 4.

В языке Java нет операции возведения в степень: для этого нужно использовать метод `pow()` класса `Math`. В результате выполнения следующей строки кода переменной `y` присваивается значение переменной `x`, возведенное в степень `a`.

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

Оба параметра метода `pow()`, а также возвращаемое им значение имеют тип `double`.

Класс `Math` содержит методы для вычисления обычных тригонометрических функций:

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

Кроме того, в него включены экспоненциальная и обратная к ней логарифмическая функции (натуральный логарифм):

```
Math.exp
Math.log
```

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

```
Math.PI
Math.E
```

Совет. Начиная с Java SE 5.0, при вызове методов для математических вычислений класс `Math` можно не указывать, включив вместо этого в начало файла с исходным кодом следующее выражение:

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

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

```
System.out.println("Квадратный корень из \u03C0 равен " + sqrt(PI));
```

Подробно вопросы статического импортирования мы рассмотрим в главе 4.

На заметку! Для повышения своей производительности функции в классе `Math` используют программы из аппаратного модуля, предназначенного для вычислений с плавающей точкой. Если для вас не очень существенна скорость работы, а важнее получить предсказуемые результаты, используйте класс `StrictMath`. Он реализует алгоритмы из свободно распространяемой математической библиотеки `fdlibm`, гарантирующей идентичность результатов на всех платформах. Исходные тексты программ, реализующих эти алгоритмы, можно найти на Web-странице `http://www.netlib.org/fdlibm/index.html`. (Поскольку в библиотеке `fdlibm` каждая функция определена неоднократно, в классе `StrictMath` в соответствии с IEEE 754 имена функций начинаются с буквы “e”.)

Преобразование числовых типов

Часто возникает необходимость преобразовать один числовой тип в другой. На рис. 3.1 показаны допустимые преобразования.

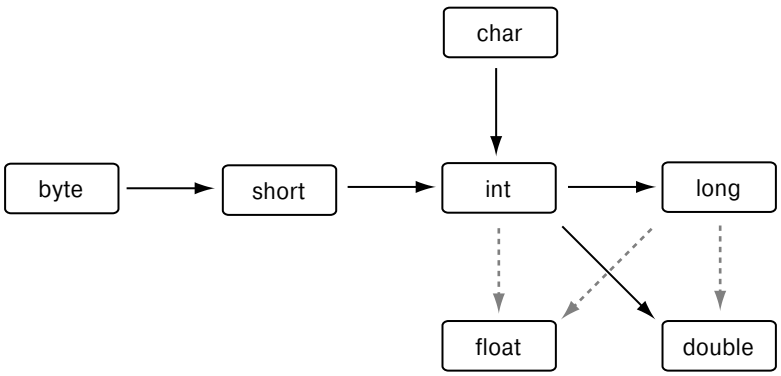


Рис. 3.1. Допустимые преобразования числовых типов

Пять сплошных линий со стрелками обозначают преобразования, которые выполняются без потери информации. Три штриховые линии, также со стрелками, означают преобразования, при которых может произойти потеря точности. Например, количество цифр в длинном целом числе 123456789 превышает количество цифр, которое может быть представлено типом `float`. Число, преобразованное в тип `float`, имеет тот же порядок, но несколько меньшую точность.

```
int n = 123456789;
float f = n; // Содержимое f равно 1.234567892E8.
```

Если два значения объединяются бинарной операцией (например, n+f, где n — целое число, а f — число с плавающей точкой), то перед выполнением операции оба операнда преобразовываются в числа одинакового типа.

- Если хотя бы один из операндов имеет тип double, то второй тоже преобразовывается в число типа double.
- В противном случае, если хотя бы один из операндов имеет тип float, то второй тоже преобразовывается в тип float.
- В противном случае, если хотя бы один из операндов имеет тип long, то второй тоже преобразовывается в число типа long.
- В противном случае оба операнда преобразовываются в числа типа int.

Приведение числовых типов

Как уже было сказано, при необходимости значения типа int автоматически преобразовываются в значения типа double. С другой стороны, в ряде ситуаций число типа double должно рассматриваться как целое. Преобразования чисел в языке Java возможны, однако, разумеется, при этом может происходить потеря информации. Такие преобразования называются *приведением типов* (cast). Синтаксически приведение типа задается парой скобок, внутри которых указывается желательный тип, а затем имя переменной. Например:

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

Теперь в результате приведения значения с плавающей точкой к целому типу переменная nx равна 9, поскольку при этом дробная часть числа отбрасывается.

Если нужно *округлить* число с плавающей точкой до *ближайшего* целого числа (что во многих случаях является намного более полезным), используется метод Math.round().

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

Теперь переменная nx равна 10. При вызове метода round() по-прежнему нужно выполнять приведение (int), поскольку возвращаемое им значение имеет тип long. Причина этого в том, что возвращаемое значение метода round() имеет тип long, а long может быть присвоен int только с явным приведением, поскольку существует возможность утери информации.



Внимание! При попытке приведения типов результат может выйти за пределы допустимого диапазона. В этом случае произойдет усечение. Например, при вычислении выражения (byte)300 будет получено значение 44.



На заметку! Приведение логических значений к целым и наоборот невозможно. Это предотвращает появление ошибок. В редких случаях, когда действительно необходимо представить логическое значение в виде целого, можно использовать условное выражение вроде b ? 1 : 0.

Скобки и иерархия операций

В табл. 3.4 приведена информация о приоритетах операций. Если скобки не используются, сначала выполняются более приоритетные операции. Операции, находящиеся на одном уровне иерархии, выполняются слева направо, за исключением опера-

ций, имеющих правую ассоциативность, как показано в таблице. Например, поскольку операция && приоритетнее ||, выражение a && b || c эквивалентно (a && b) || c. Так как операция += ассоциируется справа налево, выражение a += b += c означает a += (b += c). В данном случае значение b += c (значение b после прибавления к нему значения c) прибавляется к переменной a.



На заметку! В отличие от C и C++, язык Java не имеет операции "запятая". Однако в операторе for в качестве первой и третьей конструкций можно использовать список выражений, разделенных запятыми.

Таблица 3.5. Приоритеты операций

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

Перечислимые типы

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

Начиная с Java SE 5.0, появилась возможность использовать в подобных ситуациях *перечислимый тип*. Перечислимый тип имеет конечный набор именованных значений. Например:

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

После этого можно определять переменные данного типа:

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

Переменная типа Size может содержать только предопределенные значения. Допустимо также значение null, указывающее на то, что данной переменной не присвоена никакая величина.

Подробно перечислимые типы будут рассмотрены в главе 5.

Строки

Строка Java — это последовательность символов Unicode.

Например, строка "Java\u2122" состоит из пяти символов: J, a, v, a и ™. В языке Java нет встроенного типа для строк. Вместо этого стандартная библиотека языка содержит класс String. Каждая строка, помещенная в кавычки, представляет собой экземпляр класса String.

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

Подстроки

С помощью метода substring() класса String можно выделить подстроку данной строки. Например, в результате выполнения приведенного ниже кода формируется строка "Hel".

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

Второй параметр метода substring() — это позиция символа, который *не следует* включать в состав подстроки. В данном примере мы хотим скопировать символы из позиций 0, 1 и 2 (от позиции 0 до позиции 2 включительно), поэтому задаем при вызове метода substring() значения 0 и 3 *исключительно*.

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

Конкатенация

Язык Java, как и большинство языков программирования, дает возможность использовать знак + для объединения (конкатенации) двух строк.

```
String expletive = "Вставка";
String PG13 = "удаленная";
String message = expletive + PG13;
```

Код, приведенный выше, присваивает переменной message строку "Вставкаудаленная". (Обратите внимание на отсутствие пробела между словами: знак + объединяет в *точности* те строки, которые были заданы.)

При конкатенации строки со значением, отличным от строкового, это значение преобразовывается в строку. (Как вы убедитесь в главе 5, каждый объект Java может быть конвертирован в строку.) Например, в результате выполнения приведенного ниже кода переменной rating присваивается строка "PG13".

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

Это свойство широко используется в операторах вывода; например, приведенное ниже выражение выводит результат выполненных ранее вычислений (обратите внимание на пробел после слова Ответ).

```
System.out.println("Ответ " + answer);
```

Неизменность строк

В классе String отсутствуют методы, которые позволяли бы *изменять* символы в существующей строке. Если, например, вы хотите изменить строку greeting с "Hello" на "Help!", то заменить требуемые два символа невозможно. Для разработчиков, использующих язык C, это звучит, по крайней мере, странно. "Как же модифицировать строку?" — спросят они. В Java внести необходимые изменения можно, выполнив конкатенацию подстроки greeting и символов "p!":

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

В результате текущим значением переменной greeting становится строка "Help!".

Поскольку, программируя на языке Java, вы не можете изменять отдельные символы в строке, в документации для описания объектов String используется термин *неизменяемые, или немодифицируемые* (immutable). Как число 3 всегда равно 3, так и строка "Hello" всегда содержит символы 'H', 'e', 'l', 'l' и 'o'. Изменить эти значения невозможно. Однако, как мы только что убедились, можно изменить содержимое строковой *переменной* greeting и заставить ее ссылаться на другую строку так же, как числовой переменной, в которой хранится число 3, можно присвоить число 4.

Не снижается ли при этом эффективность? Кажется, было бы намного проще изменять символы, чем создавать новую строку заново. Возможно, это и так. Действительно, неэффективно создавать новую строку посредством конкатенации строк "Hel" и "p!". Однако неизменяемые строки имеют одно большое преимущество: компилятор может делать строки *совместно используемыми*.

Чтобы понять, как это работает, представьте, что в совместно используемом пуле находятся разные строки. Строковые переменные указывают объекты в этом пуле. При копировании строки переменной и оригинал, и копия содержат одну и ту же последовательность символов. Логично не прибегать к дублированию строк, а поместить в переменные ссылки одну и ту же область памяти. Одним словом, разработчики языка Java решили, что эффективность совместного использования памяти перевешивает неэффективность редактирования строк путем выделения подстрок и конкатенации.

Посмотрите на свои программы; мы подозреваем, что бо́льшую часть времени вы не изменяете строки, а сравниваете их. Разумеется, бывают случаи, когда непосредственные манипуляции со строками более эффективны. (Одна из таких ситуаций возникает, когда нужно образовать строку из отдельных символов, поступающих из файла или клавиатуры. Для этих ситуаций в языке Java предусмотрен отдельный класс StringBuffer, который будет описан в разделе "Построение строк".) Если же эффективность обработки строк для вас не важна (как это часто случается во многих приложениях на языке Java), вы можете не применять класс StringBuffer, а использовать лишь класс String.



На заметку! Когда программисты, работающие на языке C, видят строки в Java-программе, они обычно попадают в тупик, поскольку привыкли рассматривать строки как массивы символов:

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

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

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

При замене содержимого `greeting` другой строкой Java-программа выполняет примерно следующее:

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

Разумеется, теперь переменная `greeting` указывает на строку `"Help!"`. И даже самые убежденные поклонники языка C должны признать, что синтаксис Java более элегантен, чем последовательность вызовов функции `strncpy()`. А что будет, если мы присвоим строке `greeting` еще одно значение?

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

Не появятся ли утечки памяти? К счастью, в языке Java есть механизм автоматической "сборки мусора". Если память больше не нужна, она вскоре будет освобождена.

Если вы программируете на языке C++ и применяете класс `string`, определенный в стандарте ANSI C++, вам будет намного легче работать с объектами `String` в Java. Объекты класса `string` в языке C++ также обеспечивают автоматическое выделение и освобождение памяти. Однако строки в языке C++ могут изменяться — отдельные символы в строке можно модифицировать.

Проверка эквивалентности строк

Чтобы проверить, совпадают ли две строки, следует использовать метод `equals()`. Приведенное ниже выражение возвращает значение `true`, если строки `s` и `t` равны между собой, в противном случае возвращается значение `false`.

```
s.equals(t)
```

Заметим, что в качестве `s` и `t` могут быть использованы как переменные, так и константы. Например, следующее выражение вполне допустимо:

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

Чтобы проверить идентичность строк, игнорируя различие между прописными и строчными буквами, следует применять метод `equalsIgnoreCase()`.

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

Для проверки строк на равенство нельзя применять операцию ==! Она лишь определяет, хранятся ли обе строки в одной и той же области памяти. Разумеется, если обе строки хранятся в одном и том же месте, они должны совпадать между собой. Однако вполне возможна ситуация, при которой идентичные строки хранятся в разных местах.

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

Если виртуальная машина всегда обеспечивает совместное использование одинаковых строк, то для проверки их равенства можно применять операцию ==. Однако совместно использовать можно лишь *константы*, а не строки, являющиеся результатами таких операций, как + или substring. Следовательно, либо нужно *навсегда* отказаться от проверок строк на равенство с помощью операции ==, либо вы получите программу, содержащую наихудшую из возможных ошибок, проявляющуюся лишь время от времени, — ошибку, возникновение которой невозможно предсказать.



На заметку! Если вы привыкли использовать класс string в языке C++, будьте особенно внимательны при проверке совпадения строк. В классе string операция == перегружена и позволяет проверять идентичность содержимого строк. Возможно, в языке Java разработчики напрасно отказались от возможности работать со строками как с числовыми значениями, однако это позволило сделать строки похожими на указатели. Разработчики могли переопределить операцию == для строк так, как они это сделали с операцией +. Что ж, каждый язык имеет свои недостатки.

Программисты, создающие продукты на языке C, никогда не используют для проверки строк на равенство операцию ==, вместо этого они вызывают функцию strcmp(). Метод compareTo() в языке Java представляет собой точный аналог функции strcmp(). Можно, конечно, использовать выражения вида

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

Однако нам кажется, что применение метода equals() делает программу более читабельной.

Кодовые точки и кодовые единицы

В языке Java строки реализованы как последовательности значений типа char. Как было сказано ранее в разделе "Тип char", тип char позволяет задавать кодовые единицы, представляющие кодовые точки Unicode в кодировке UTF-16. Наиболее часто используемые символы Unicode представляются одной кодовой единицей. Дополнительные символы задаются парами кодовых единиц.

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

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

Чтобы определить реальную длину, представляющую собой число кодовых точек, надо использовать следующий вызов:

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

Метод s.charAt(n) возвращает кодовую единицу в позиции n, где n находится в интервале от 0 до s.length() - 1. Ниже приведены примеры вызова данного метода.

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

Для получения i-й кодовой точки надо использовать приведенные ниже выражения.

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



На заметку! Java подсчитывает кодовые единицы в строках специфическим образом: первая кодовая единица в строке расположена в позиции 0. Это соглашение пришло из языка C, где по техническим причинам подсчет позиций начинается с 0. Эта причина давно ушла в прошлое, а неудобство осталось. Однако этому соглашению следует настолько много программистов, что проектировщики Java решили сохранить его.

Зачем мы подняли шум относительно кодовых единиц? Рассмотрим приведенную ниже строку:

```
𝕫 is the set of integers
```

Для представления символа 𝕫 используются две кодовые единицы UTF-16. Приведенный ниже вызов метода даст не код пробела, а второй код символа 𝕫. Чтобы избежать возникновения данной проблемы, не следует применять тип `char`, так как он представляет символы на слишком низком уровне.

Если вы хотите просмотреть строку посимвольно, т.е. получить по очереди каждую кодовую точку, вам надо использовать фрагмент кода, подобный показанному ниже.

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

К счастью, метод `codePointAt()` определяет, является ли кодовая единица первой или второй частью дополнительного символа, и всегда возвращает корректный результат. Поэтому вы можете организовать просмотр строки и в обратном направлении.

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

API класса `String`

Класс `String` в языке Java содержит свыше 50 методов. Многие из них оказались очень полезными и используются довольно часто. Приведенный ниже фрагмент описания API содержит наиболее полезные из них.



На заметку! Время от времени мы будем включать в текст книги фрагменты описания интерфейса программирования приложений Java (Application Programming Interface — API). Каждый такой фрагмент начинается с имени класса, например `java.lang.String`. В данном случае `java.lang` — это пакет; механизм пакетов будет подробно рассмотрен в главе 4. После имени класса следуют имена конкретных методов и их описание.

Обычно в тексте книги не перечисляются все методы отдельного класса; мы ограничиваемся лишь наиболее часто используемыми методами. Полный список методов можно найти в справочной системе (см. раздел “Интерактивная документация по API” далее в главе).

Мы также приводим номер версии, в которой был реализован описываемый класс. Если тот или иной метод был добавлен позже, это указывается отдельно.

API java.lang.String 1.0

- `char charAt(int index)`

  Возвращает символ, расположенный в указанной позиции. Вызывать этот метод следует только в том случае, если вас интересуют низкоуровневые кодовые единицы.

- `int codePointAt(int index)` **5.0**

  Возвращает кодовую точку, начало или конец которой находится в указанной позиции.

- `int offsetByCodePoints(int startIndex, int cpCount)` **5.0**

  Возвращает индекс кодовой точки, которая определяется `cpCount` относительно `startIndex`.

- `int compareTo(String other)`

  Возвращает отрицательное значение, если данная строка лексикографически предшествует строке `other`, положительное значение — если строка `other` предшествует данной строке, и 0 — если строки идентичны.

- `boolean endsWith(String suffix)`

  Возвращает значение `true`, если строка заканчивается подстрокой `suffix`.

- `boolean equals(Object other)`

  Возвращает значение `true`, если данная строка совпадает со строкой `other`.

- `boolean equalsIgnoreCase(String other)`

  Возвращает значение `true`, если данная строка совпадает со строкой `other` без учета регистра символов.

- `int indexOf(String str)`
- `int indexOf(String str, int fromIndex)`
- `int indexOf(int cp)`
- `int indexOf(int cp, int fromIndex)`

  Возвращает индекс начала первой подстроки, совпадающей со строкой `str`, либо индекс указанной кодовой точки `cp`. Отсчет начинается с позиции `0` или `formIndex`. Если указанная подстрока в составе строки отсутствует, возвращается значение, равное –1.

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

  Возвращает начало последней подстроки, равной строке `str`, либо индекс указанной кодовой точки `cp`. Отсчет начинается с позиции `0` или `formIndex`. Если указанная подстрока в составе строки отсутствует, возвращается значение, равное –1.

- `int length()`

  Возвращает длину строки.

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

  Возвращает число кодовых точек между `startIndex` и `endIndex - 1`. Половина пары, обозначающей дополнительный индекс, считается как полноправная кодовая точка.

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

  Возвращает новую строку, которая получается путем замены всех подстрок, соответствующих `oldString`, строкой `newString`. В качестве параметров `CharSequence` могут выступать объекты `String` или `StringBuilder`.

- `boolean startWith(String prefix)`

  Возвращает значение `true`, если строка начинается подстрокой `prefix`.

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

  Возвращает новую строку, состоящую из всех кодовых единиц, начиная с позиции `BeginIndex` и заканчивая концом строки или позицией `endIndex - 1`.

- `String toLowerCase()`

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

- `String toUpperCase()`

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

- `String trim()`

  Возвращает новую строку, из которой исключены все предшествующие и завершающие пробелы.

Интерактивная документация по API

Как вы уже знаете, класс `String` имеет много методов. Более того, в стандартной библиотеке существуют тысячи классов, содержащих огромное количество методов. Совершенно невозможно запомнить всю информацию, которая может понадобиться при написании программ. Следовательно, очень важно уметь пользоваться интерактивной справочной системой, содержащей документацию об API, и находить нужные классы и методы. Документация является составной частью набора инструментальных средств JDK. Она представлена в формате HTML. Обратитесь с помощью вашего браузера к документу `docs/api/index.html` в каталоге, в котором установлен JDK. Вы увидите информацию, показанную на рис. 3.2.

Окно браузера разделено на три фрейма. В небольшом фрейме в верхнем левом углу отображаются имена всех доступных пакетов. Под ним во фрейме побольше перечислены все классы. Щелкните мышью на любом из имен классов, и информация об этом классе будет показана в большом фрейме, расположенном справа (рис. 3.3). Например, чтобы получить дополнительные сведения о методах класса `String`, прокрутите содержимое левого нижнего фрейма, пока не увидите ссылку `String`, и щелкните на ней.

Затем прокрутите содержимое правого фрейма до тех пор, пока не увидите краткое описание всех методов. Методы расположены в алфавитном порядке (рис. 3.4). Щелкните на имени интересующего вас метода, чтобы получить его детальное описание (рис. 3.5). Например, если вы щелкнете на ссылке `compareToIgnoreCase`, то получите описание метода с этим именем.



Совет. Сразу же сделайте закладку в браузере, указывающую на документ `docs/api/index.html`.

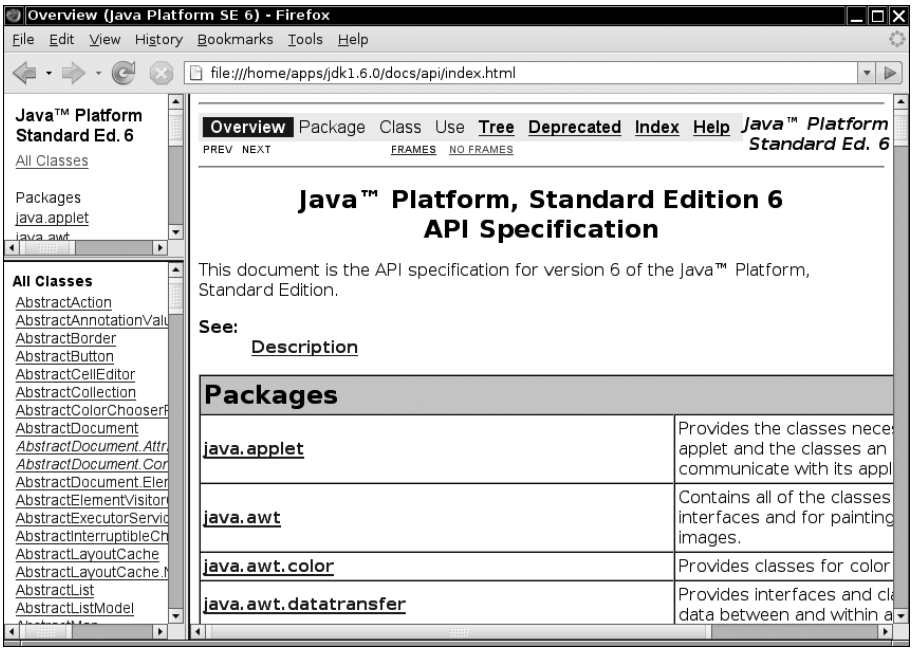


Рис. 3.2. Окно с документацией по API

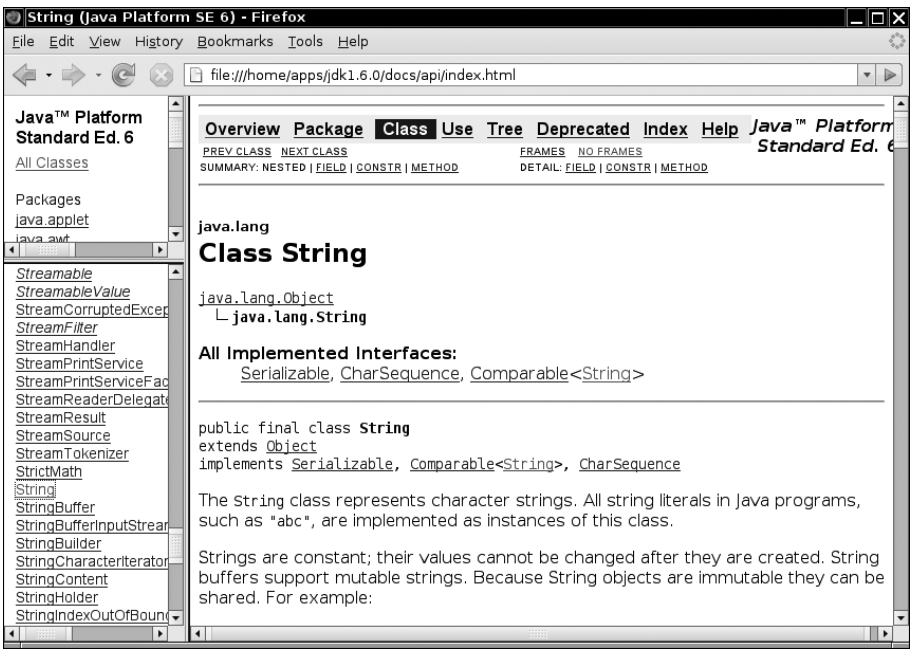


Рис. 3.3. Описание класса `String`

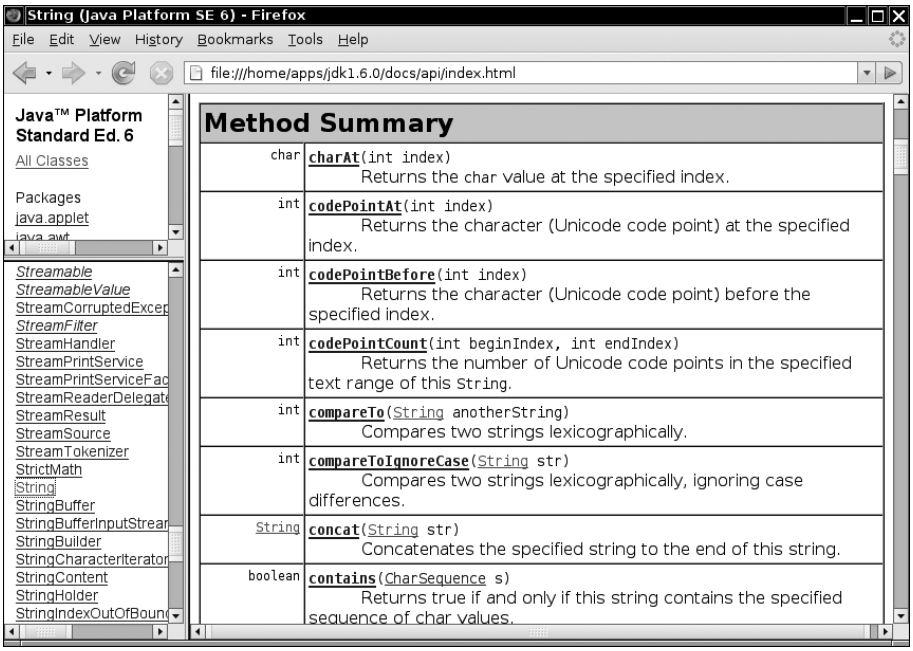


Рис. 3.4. Перечень методов класса `String`

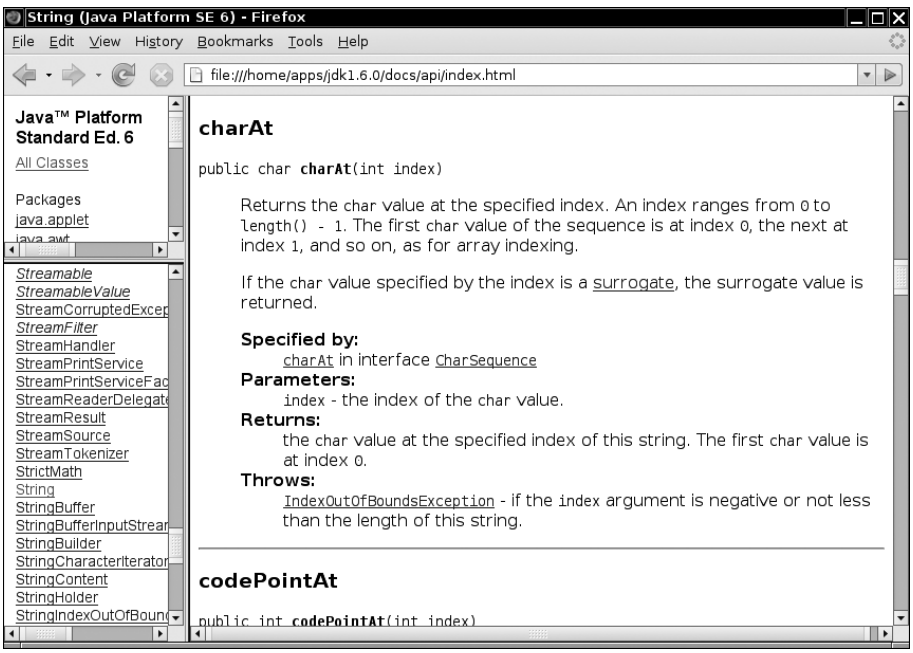


Рис. 3.5. Подробное описание метода `compareToIgnoreCase()` *класса* `String`

Построение строк

Однажды вам понадобится собирать одни строки из более коротких других строк — поступающих от клавиатуры или из файла. Было бы неэффективно постоянно использовать для этой цели конкатенацию. При каждой конкатенации строк конструируется новый объект `String`. Это требует времени и расходует память. Применение класса `StringBuilder` позволяет избежать этой проблемы.

Если вам нужно построить строку из нескольких маленьких кусочков, выполните следующие шаги. Во-первых, сконструируйте пустой построитель строки:

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

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

Всякий раз, когда вам понадобится добавить новую часть, вызываете метод `append()`.

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

Завершив сборку строки, вызовите метод `toString()`. Так вы получите объект `String`, состоящий из последовательности символов, содержащихся в объекте построителя строк:

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



На заметку! Класс `StringBuilder` появился в JDK 5.0. Его предшественник `StringBuffer` несколько менее эффективен, но позволяет множеству потоков добавлять и удалять символы. Если все редактирование строки происходит в единственном потоке (как это обычно бывает), вы должны использовать вместо него класс `StringBuilder`. API обоих классов идентичны.

Следующее описание API содержит наиболее важные методы класса `StringBuilder`.



java.lang.StringBuilder 5.0

- `StringBuilder()`

  Конструирует пустой построитель строки.

- `int length()`

  Возвращает количество кодовых единиц построителя или буфера.

- `StringBuilder append(String str)`

  Добавляет строку и возвращает `this`.

- `StringBuilder append(char c)`

  Добавляет кодовую единицу и возвращает `this`.

- `StringBuilder appendCodePoint(int cp)`

  Добавляет кодовую точку, конвертируя ее в одну или две кодовых единицы, и возвращает `this`.

- `void setCharAt(int i, int c)`

  Устанавливает `i`-ю кодовую единицу в `c`.

- `StringBuilder insert(int offset, String str)`

  Вставляет строку в позицию `offset` и возвращает `this`.

- `StringBuilder insert(int offset, char c)`

  Вставляет кодовую единицу в позицию `offset` и возвращает `this`.
- `StringBuilder delete(int startIndex, int endIndex)`

  Удаляет кодовые единицы со смещениями от `startIndex` до `endIndex-1` и возвращает `this`.
- `String toString()`

  Вставляет строку, содержащую те же данные, что и построитель или буфер.

Ввод и вывод

Для того чтобы немного "оживить" программы, рассматриваемые в качестве примеров, организуем ввод информации и форматирование выходных данных. Конечно, в современных приложениях для ввода используются средства графического пользовательского интерфейса, однако в данный момент вы еще не обладаете знаниями, достаточными для формирования интерфейсных элементов. Поскольку наша текущая цель — лучше узнать языковые средства Java, мы ограничимся вводом и выводом посредством консоли. Вопросы программирования графического интерфейса будут подробно рассмотрены в главах 7–9.

Чтение входных данных

Вы уже знаете, что информацию можно легко вывести на стандартное устройство вывода (т.е. в консольное окно), вызвав метод `System.out.println()`. Чтение из "стандартного входного потока" `System.in` не так просто. Для того чтобы организовать чтение информации с консоли, вам надо создать объект `Scanner` и связать его со стандартным входным потоком `System.in`.

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

(Конструкторы и операцию `new` мы обсудим подробно в главе 4.)

Сделав это, вы получите в свое распоряжение многочисленные методы класса `Scanner`, предназначенные для чтения входных данных. Например, метод `nextLine()` обеспечивает прием строки текста.

```
System.out.print("Как вас зовут? ");
String name = in.nextLine();
```

В данном случае мы использовали метод `nextLine()`, потому что входная строка может содержать пробелы. Для того чтобы прочитать одно слово (разделителями между словами считаются пробелы), можно использовать следующий вызов:

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

Для чтения целочисленного значения предназначен метод `nextInt()`.

```
System.out.print("Сколько вам лет? ");
int age = in.nextInt();
```

Как нетрудно догадаться, метод `nextDouble()` читает очередное число в формате с плавающей точкой.

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

```
Инкогнито, в следующем году вам будет 46
```

В первой строке содержится выражение

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

Класс Scanner принадлежит пакету java.util. Если вы собираетесь использовать в программе класс, не содержащийся в базовом пакете java.lang, вам надо включить в состав кода директиву import. Подробно пакеты и директива import будут рассмотрены в главе 4.

Листинг 3.2. Содержимое файла InputTest.java

```
1. import java.util.*;
2.
3. /**
4. * Эта программа демонстрирует консольный ввод.
5. * @version 1.10 2004-02-10
6. * @author Cay Horstmann
7. */
8. public class InputTest
9. {
10.   public static void main(String[] args)
11.   {
12.      Scanner in = new Scanner(System.in);
13.
14.      // Получение первой строки входных данных
15.      System.out.print("Как вас зовут? ");
16.      String name = in.nextLine();
17.
18.      // Получение целочисленного значения
19.      System.out.print("Сколько вам лет? ");
20.      int age = in.nextInt();
21.
22.      // Отображение информации в консольном окне
23.      System.out.println("name + ", в следующем году вам будет " + (age + 1));
24.   }
25. }
```



На заметку! Класс Scanner не подходит для чтения с консоли паролей, поскольку такой ввод просто-напросто будет виден любому. В Java SE 6 появился класс Console, специально предназначенный для этих целей. Чтобы прочесть пароль, используйте следующий код:

```
Console cons = System.console();
String username = cons.readLine("Имя пользователя: ");
char[] passwd = cons.readPassword("Пароль: ");
```

Из соображений безопасности пароль возвращается в виде массива символов, а не в виде строки. После того, как вы обработаете пароль, вы должны немедленно перезаписать элементы массива значением-заполнителем. (Обработку массивов мы обсудим далее в этой главе.)

Обработка ввода с объектом Console не так удобна, как со Scanner. Вы можете читать только по одной строке ввода за раз. Здесь нет методов для чтения отдельных слов или чисел.



java.util.Scanner 5.0

- Scanner(InputStream in)

  Создает объект Scanner на базе заданного входного потока.

- `String nextLine()`

  Читает очередную строку.

- `String next()`

  Читает очередное слово (разделителями между словами считаются пробелы).

- `int nextInt()`
- `double nextDouble()`

  Читает очередную последовательность символов, представляющую целое число или число с плавающей точкой, и выполняет соответствующее преобразование.

- `boolean hasNext()`

  Проверяет, существует ли во входном потоке еще одно слово.

- `boolean hasNextInt()`
- `boolean hasNextDouble()`

  Проверяет, существует ли во входном потоке последовательность символов, целое число или число с плавающей точкой.

API **java.lang.System 1.0**

- `static Console console()`

  Возвращает объект `Console` для взаимодействия с пользователем через консольное окно, и если такое взаимодействие невозможно, возвращает `null`. Объект `Console` доступен любой программе, запущенной в консольном окне. В противном случае его доступность зависит от системы.

API **java.lang.Console 6**

- `static char[] readPassword(String prompt, Object... args)`
- `static String readLine(String prompt, Object... args)`

  Отображает приглашение и читает ввод пользователя до тех пор, пока не получит конец строки ввода. Параметры `args` могут быть использованы для применения аргументов форматирования, как описано в следующем разделе.

Форматирование выходных данных

Число `x` можно вывести на консоль с помощью выражения `System.out.println(x)`. В результате на экране отобразится число с максимальным количеством значащих цифр, допустимых для данного типа. Например, в результате выполнения приведенного ниже фрагмента кода на экран будет выведено число `3333.3333333333335`.

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

В ряде случаев это создает проблемы. Так, например, если вы хотите вывести на экран сумму в долларах и центах, большое количество цифр затруднит восприятие.

В ранних версиях Java процесс форматирования чисел был сопряжен с определенными трудностями. К счастью, в B Java SE 5.0 был реализован практически бесценный метод `printf()`, привычный всем программистам, которые имели дело с языком C.

Например, с помощью приведенного ниже оператора мы можем вывести значение x в виде числа, размер поля которого составляет 8 цифр, а дробная часть равна двум цифрам. (Число цифр дробной части называют также точностью.)

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

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

```
3333.33
```

Метод printf() позволяет задавать произвольное количество параметров. Пример вызова с несколькими параметрами приведен ниже.

```
System.out.printf("%s, в следующем году вам будет %d", name, age);
```

Каждый *спецификатор формата*, начинающийся с символа %, заменяется соответствующим параметром. *Символ преобразования*, которым завершается спецификатор формата, задает тип форматируемого значения: f — число с плавающей точкой; s — строка; d — десятичное число. Символы преобразования описаны в табл. 3.5.

Таблица 3.5. Символы преобразования для метода printf()

Символ преобразования	Тип	Пример
d	Десятичное целое	159
x	Шестнадцатеричное целое	9f
o	Восьмеричное целое	237
f	Число с фиксированной или плавающей точкой	15.9
e	Число с плавающей точкой в экспоненциальной форме	1.59e+01
g	Число с плавающей точкой в общем формате (чаще всего используется формат e или f, в зависимости от того, какой из них дает более короткую запись)	
a	Шестнадцатеричное представление числа с плавающей точкой	0x1.fccdp3
s	Строка	Hello
c	Символ	H
b	Логическое значение	true
h	Хеш-код	42628b2
tx	Дата и время	См. табл. 3.7
%	Символ процента	%
n	Разделитель строк, зависящий от платформы	

В составе спецификатора формата могут присутствовать флаги, управляющие форматом выходных данных. Назначение всех флагов описано в табл. 3.6. Например, запятая, используемая в качестве флага, формирует разделитель групп. Так, в результате выполнения приведенного ниже оператора на экран будет выведена строка 3,333.33.

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

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



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

Таблица 3.6. Флаги для метода `printf()`

Флаг	Назначение	Пример
`+`	Выводит знак не только для отрицательных, но и для положительных чисел	`+3333.33`
Пробел	Добавляет пробел перед положительными числами	`| 3333.33|`
`0`	Выводит ведущие нули	`003333.33`
`-`	Выравнивает поле по левому краю	`|3333.33 |`
`(`	Помещает отрицательные числа в скобки	`(3333.33)`
`,`	Задает использование разделителя групп	`3,333.33`
`#` (для формата `f`)	Всегда отображает десятичную точку	`3,333.`
`#` (для формата `x` или `o`)	Добавляет префикс `0x` или `0`	`0xcafe`
`$`	Определяет индекс параметра, предназначенного для форматирования. Например, выражение `%1$d %1$x` указывает на то, что первый параметр должен быть выведен в десятичном формате, а затем он же представлен в шестнадцатеричном виде	`159 9F`
`<`	Задает форматирование того же значения, к которому относился предыдущий спецификатор. Например, выражение `%d %<x` указывает на то, что одно и то же значение должно быть представлено как в десятичном, так и в шестнадцатеричном виде	`159 9F`

Для создания форматированной строки без вывода ее можно использовать статический метод `String.format()`.

```
String message = String.format("%s, в следующем году вам будет %d", name, age);
```

Несмотря на то что тип `Date` будет подробно рассмотрен лишь в главе 4, нам, чтобы закончить разговор о методе `printf()`, желательно подробно рассмотреть средства форматирования даты и времени. Для этой цели используется последовательность из двух символов, начинающаяся с буквы `t`, за которой следует одна из букв, приведенных в табл. 3.7. Пример такого форматирования приведен ниже.

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

В результате выполнения данного оператора выводится текущая дата и время.

```
Mon May 16 18:05:19 PST 2004
```

Таблица 3.7. Символы для форматирования даты и времени

Символ преобразования	Выходная информация	Пример
c	Полные сведения о дате и времени	Mon Feb 04 18:05:19 PST 2008
F	Дата в формате ISO 8601	2008-02-04
D	Дата в формате, принятом в США (месяц/день/год)	02/04/2008
T	Время при 24-часовом цикле	18:05:19
r	Время при 12-часовом цикле	06:05:19 pm
R	Время при 24-часовом цикле (секунды не выводятся)	18:05
Y	Год, представленный четырьмя цифрами (при необходимости отображаются ведущие нули)	2008
y	Последние две цифры года (при необходимости отображаются ведущие нули)	08
C	Первые две цифры года (при необходимости отображаются ведущие нули)	20
B	Полное название месяца	February
b или h	Сокращенное название месяца	Feb
m	Месяц, представленный двумя цифрами (при необходимости отображаются ведущие нули)	02
d	День, представленный двумя цифрами (при необходимости отображаются ведущие нули)	09
e	День, представленный одной или двумя цифрами (ведущие нули не отображаются)	4
A	Полное название дня недели	Monday
a	Сокращенное название дня недели	Mon
j	День в году, представленный тремя цифрами (при необходимости отображаются ведущие нули); номер дня может лежать в пределах от 001 до 366	035
H	Номер часа, представленный двумя цифрами (при необходимости отображаются ведущие нули); номер часа может лежать в пределах от 00 до 23	18
k	Номер часа, представленный одной или двумя цифрами (ведущие нули не отображаются); номер часа может лежать в пределах от 0 до 23	18
I	Номер часа, представленный двумя цифрами (при необходимости отображаются ведущие нули); номер часа может лежать в пределах от 01 до 12	06
l	Номер часа, представленный одной или двумя цифрами (ведущие нули не отображаются); номер часа может лежать в пределах от 1 до 12	6
M	Число минут, представленное двумя цифрами (при необходимости отображаются ведущие нули)	05

Окончание табл. 3.7

Символ преобразования	Выходная информация	Пример
S	Число секунд, представленное двумя цифрами (при необходимости отображаются ведущие нули)	19
L	Число миллисекунд, представленное двумя цифрами (при необходимости отображаются ведущие нули)	047
N	Число наносекунд, представленное девятью цифрами (при необходимости отображаются ведущие нули)	047000000
P	AM и PM отображаются символами верхнего регистра	PM
p	AM и PM отображаются символами нижнего регистра	pm
z	Смещение относительно гринвичского времени (GMT) согласно RFC 822	-0800
Z	Временной пояс	PST
s	Число секунд с 1970-01-01 00:00:00 GMT	1078884319
Q	Число миллисекунд с 1970-01-01 00:00:00 GMT	1078884319047

Как видно в табл. 3.7, некоторые форматы предполагают отображение лишь отдельных компонентов даты — дня или месяца. Было бы неразумно многократно задавать дату лишь для того, чтобы отформатировать различные ее элементы. По этой причине в строке, определяющей формат, может задаваться индекс форматируемого параметра. Индекс должен следовать непосредственно за символом `%` и завершаться знаком `$`. Пример использования индекса приведен ниже.

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

В результате обработки данного оператора будет выведена следующая строка:

```
Дата: February 4, 2008
```

Вы также можете использовать флаг <, который означает, что форматированию подлежит тот же параметр, который был сформатирован последним. Так, приведенный ниже оператор дает тот же результат, что и рассмотренное ранее.

```
System.out.printf("%s %tB %<te, %<tY", "Дата:", new Date());
```



Внимание! Индексы начинаются с единицы. Так, выражение `%1$` задает форматирование первого параметра. Это сделано для того, чтобы избежать конфликтов с флагом `0`.

В данном разделе были рассмотрены не все особенности метода `printf()`. На рис. 3.6 показана диаграмма, которая строго задает синтаксис спецификатора формата.



На заметку! Ряд правил форматирования зависит от специфических условий конкретной страны или региона. Так, например, в Германии разделителем групп является не запятая, а точка, а вместо `Monday` выводится имя `Montag`. Вопросы интернационализации приложений будут рассмотрены во втором томе.

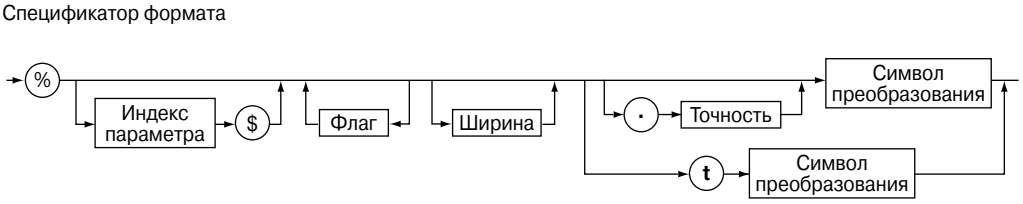


Рис. 3.6. Правила записи спецификатора формата

Файловый ввод и вывод

Чтобы прочитать из файла, сконструируйте объект `Scanner` из объекта `File`, как показано ниже:

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

Если имя файла содержит в себе обратные косые черты, не забудьте защитить их дополнительными обратными косыми чертами: `"c:\\mydirectory\\myfile.txt"`.

После этого вы можете выполнять чтение из файла, используя любые методы `Scanner`, которые мы описали выше.

Чтобы выполнить запись в файл, сконструируйте объект `PrintWriter`. В конструкторе просто укажите имя файла:

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

Если файл еще не существует, вы можете просто использовать команды `print`, `println` и `printf`, как вы поступали, осуществляя вывод в `System.out`.



Внимание! Вы можете конструировать `Scanner` со строковым параметром, но при этом сканер интерпретирует эту строку как данные, а не как имя файла. Например, если вы вызовете конструктор так, как показано ниже, то `Scanner` увидит символы ‘m’, ‘y’, ‘f’ и т.д. Вероятно, это не то, что вы подразумевали.

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



На заметку! Когда вы специфицируете относительное имя файла, такое как `"myfile.txt"`, `"mydirectory/myfile.txt"` или `"../myfile.txt"`, то файл ищется относительно каталога, в котором была запущена виртуальная машина Java. Если вы запустили вашу программу из командной оболочки, введя

```
java MyProg
```

То стартовым каталогом будет текущий каталог командной оболочки. Однако если вы используете интегрированную среду разработки, то стартовый каталог определяется этой средой. Вы можете определить расположение стартового каталога следующим вызовом:

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

Если вы испытываете трудности, связанные с нахождением файлов, попробуйте применять абсолютные путевые имена наподобие `"c:\\mydirectory\\myfile.txt"` или `"/home/me/mydirectory/myfile.txt"`.

Как видите, вы можете обращаться к файлам так же легко, как используете `System.in` и `System.out`. Правда, здесь есть одна ловушка: если вы конструируете `Scanner` с файлом, который еще не существует, либо `PrintWriter` с именем файла, который не может быть создан, возникает исключение. Компилятор Java рассматривает

эти исключения как более серьезные, чем, например исключение "деления на ноль". В главе 11 вы узнаете о разных способах обработки исключений. А пока вам следует просто сказать компилятору, что вы знаете о возможности возникновения исключения типа "файл не найден". Это делается посредством оснащения метода `main()` конструкцией `throws`, как показано ниже:

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

Теперь вы знаете, как читать и записывать в файлы, содержащие текстовую информацию. Более сложные темы, связанные с файловым вводом-выводом, — такие как работа с различными символьными кодировками, обработка двоичных данных, чтение каталогов и запись zip-файлов — рассматриваются во втором томе нашей книги.



На заметку! Когда вы запускаете программу из командной оболочки, то можете использовать синтаксис перенаправления ввода-вывода и присоединить любой файл в `System.in` и `System.out`:

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

В этом случае вам не придется беспокоиться об обработке исключения `FileNotFoundException`.



java.util.Scanner 5.0

- `Scanner(File f)`

  Создает объект `Scanner`, который читает данные из указанного файла.

- `Scanner(String data)`

  Создает объект `Scanner`, который читает данные из указанной строки.



java.io.PrintWriter 1.1

- `PrintWriter(File f)`

  Создает объект `PrintWriter`, который записывает данные в указанный файл.

- `PrintWriter(String fileName)`

  Создает объект `PrintWriter`, который записывает данные в файл с указанным именем.



java.io.File 1.0

- `File (String fileName)`

  Создает объект `File`, описывающий файл с заданным именем. Обратите внимание, что файл не обязательно должен существовать.

Поток управления

В языке Java, как и в любом другом языке программирования, есть условные операторы и циклы. Начнем с условных операторов, а затем перейдем к циклам. Рассмотрение потока управления мы завершим довольно неуклюжим оператором `switch`, который можно применять при проверке большого количества значений одного выражения.



На заметку! Поток управления в языке Java построен точно так же, как и в языках C и C++, за исключением двух особенностей. В нем нет оператора `goto`, однако существует версия оператора `break` с метками, который можно использовать для выхода из вложенного цикла (в C для этого пришлось бы применять `goto`). Кроме того, в Java SE 5.0 реализован вариант цикла `for`, который не имеет аналогов в C или C++. Его можно сравнить с циклом `foreach` в C#.

Блоки

Перед изучением управляющих структур нам необходимо рассмотреть *блоки*.

Блок, или составной оператор, — это произвольное количество простых операторов языка Java, заключенных в фигурные скобки. Блоки определяют область видимости переменных. Блоки могут быть *вложенными* один в другой. Ниже приведен пример блока, вложенного в другой блок в методе `main()`.

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

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

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



На заметку! В языке C++ переменные во вложенных блоках можно переопределять. Внутреннее определение маскирует внешнее. Это может привести к ошибкам, поэтому в языке Java подобный подход не реализован.

Условные выражения

Условный оператор в языке Java имеет следующий вид:

```
if (условие) оператор
```

Условие должно указываться в скобках.

В языке Java, как и в большинстве других языков программирования, часто необходимо выполнять много операторов в зависимости от выполнения одного условия. В этом случае используется *блок*, формируемый следующим образом:

```
{
  Оператор_1;
  Оператор_2;

  ...
}
```

Например,

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

В этом фрагменте все операторы, помещенные в фигурные скобки, будут выполнены, если значение переменной `yourSales` больше значения переменной `target` или равно ему (рис. 3.7).

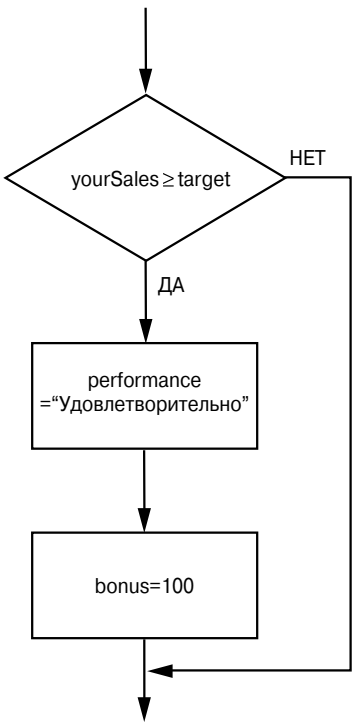


Рис. 3.7. Блок-схема, иллюстрирующая действие оператора `if`



На заметку! Блок (иногда называемый *составным оператором*) позволяет включать несколько (простых) операторов в любую структуру языка Java, которая в ином случае состояла бы лишь из одного (простого) оператора.

Условный оператор в языке Java может также иметь приведенную ниже общую форму. Его действие показано на рис. 3.8.

```
if (условие) оператор_1 else оператор_2
```

Например:

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

Часть `else` не является обязательной. Она объединяется с ближайшим оператором `if`. Таким образом, в следующем фрагменте кода оператор `else` соответствует второму оператору `if`:

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

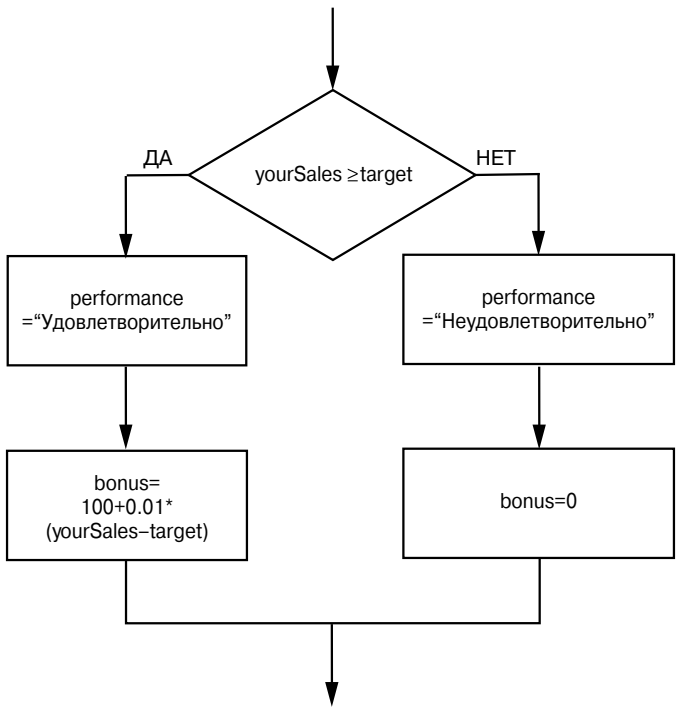


Рис. 3.8. Блок-схема, демонстрирующая действие оператора `if/else`

В программах часто встречаются также операторы типа `if...else if...` (рис. 3.9). Пример такой языковой конструкции приведен ниже.

```
if (yourSales >= 2 * target)
{
   performance = "Великолепно";
   bonus = 1000;
}
else if (yourSales >= 1.5 * target)
{
   performance = "Хорошо";
   bonus = 500;
}
else if (yourSales >= target)
{
   performance = "Удовлетворительно";
   bonus = 100;
}
else
{
   System.out.println("Вы уволены");
}
```

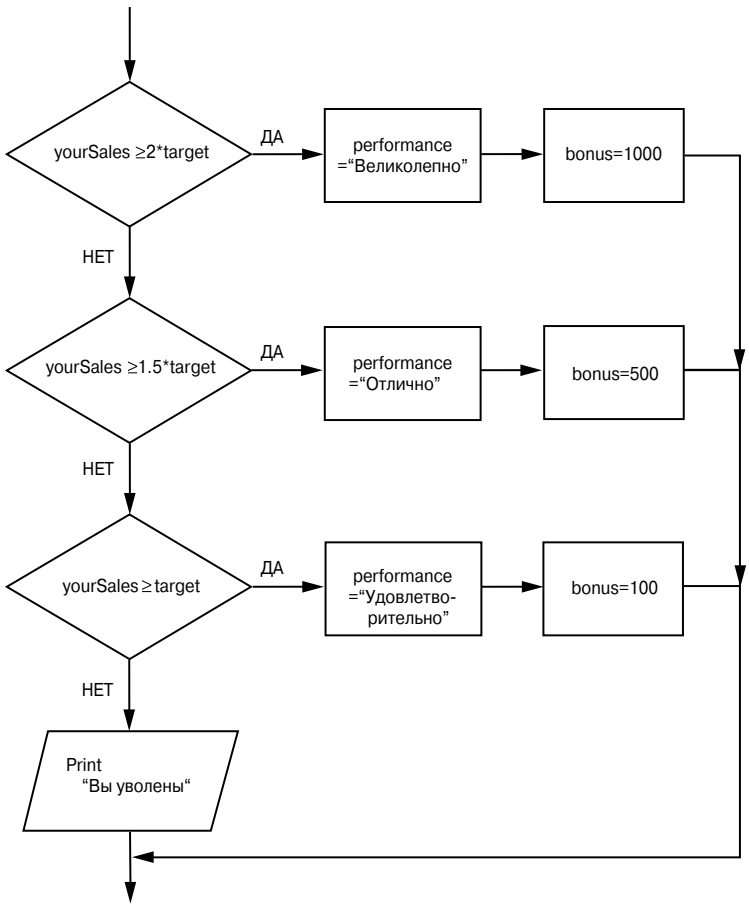


Рис. 3.9. Блок-схема, демонстрирующая действие оператора `if/else if` *(множество ветвей)*

Неопределенные циклы

Цикл `while` обеспечивает выполнение выражения (или группы операторов, составляющих блок) до тех пор, пока условие равно `true`. Данный цикл записывается в следующем виде:

```
while (условие) оператор
```

Тело цикла `while` не будет выполнено ни разу, если его условие изначально равно `false` (рис. 3.10).

В листинге 3.3 показана программа, подсчитывающая, сколько лет надо вносить деньги на счет, чтобы накопить заданную сумму. Считается, что каждый год вносится одна и та же сумма и процентная ставка не изменяется.

В этом примере мы увеличиваем счетчик и обновляем в теле цикла сумму, накопленную к данному моменту, пока общий итог не превысит заданную величину.

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

(Не полагайтесь на эту программу при планировании своей пенсии. Мы отбросили такие детали, как инфляция и ожидаемая продолжительность вашей жизни.)

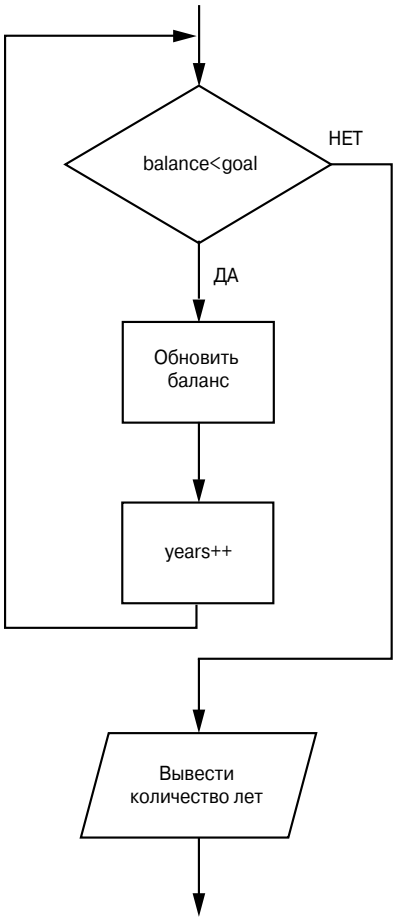


Рис. 3.10. Блок-схема оператора `while`

Условие цикла `while` проверяется в самом начале. Следовательно, возможна ситуация, при которой код, содержащийся в блоке, не будет выполнен никогда. Если вы хотите, чтобы блок выполнялся хотя бы один раз, проверку условия нужно перенести в конец. Это можно сделать с помощью цикла `do/while`, который записывается следующим образом:

```
do оператор while (условие);
```

Условие проверяется лишь после выполнения тела цикла. Затем тело цикла повторяется, вновь проверяет условие и т.д. Например, код в листинге 3.4 вычисляет новый баланс вашего счета, а затем спрашивает вас, не собираетесь ли вы на заслуженный отдых.

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

```
   // Вывести текущий баланс
   ...
   // Спросить, не собирается ли пользователь на пенсию,
   // и получить ответ.
   ...
}
while (input.equals("N"));
```

Пока пользователь отвечает "N", цикл повторяется (рис. 3.11). Эта программа является хорошим примером применения циклов, которые нужно выполнить хотя бы один раз, поскольку пользователь должен увидеть баланс прежде, чем решит, хватит ли ему денег на пенсию.

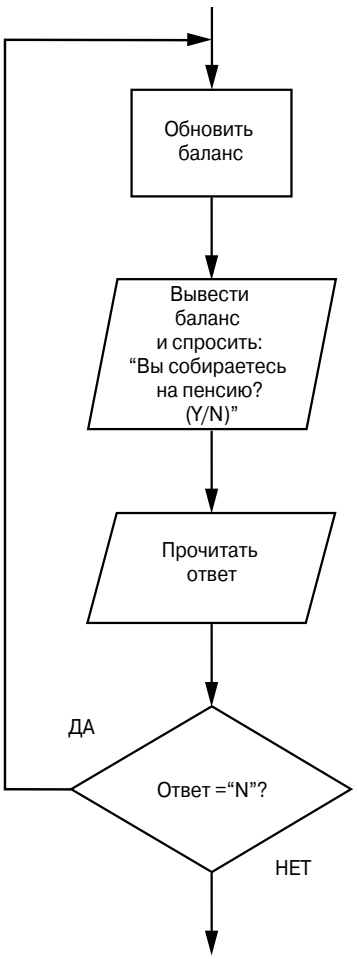


Рис. 3.11. Блок-схема, поясняющая работу оператора `do/while`

Листинг 3.3. Содержимое файла `Retirement.java`

```
1. import java.util.*;
2.
3. /**
4. * Эта программа демонстрирует работу цикла <code>while</code>.
5. * @version 1.20 2004-02-10
6. * @author Cay Horstmann
7. */
8. public class Retirement
9. {
10.    public static void main(String[] args)
11.    {
12.       // Чтение входных данных
13.       Scanner in = new Scanner(System.in);
14.
15.       System.out.print("Какая сумма вам необходима?");
16.       double goal = in.nextDouble();
17.
18.       System.out.print("Сколько вы можете вносить в год? ");
19.       double payment = in.nextDouble();
20.
21.       System.out.print("Процентная ставка: ");
22.       double interestRate = in.nextDouble();
23.
24.       double balance = 0;
25.       int years = 0;
26.
27.       // Баланс обновляется до тех пор, пока не будет
28.       // достигнута требуемая сумма.
29.       while (balance < goal)
30.       {
31.          // Добавление ежегодного взноса и процентов
32.          balance += payment;
33.          double interest = balance * interestRate / 100;
34.          balance += interest;
35.          years++;
36.       }
37.
38.       System.out.println("Вы можете уйти на отдых через " + years + " лет.");
39.    }
40. }
```

Листинг 3.4. Содержимое файла `Retirement2.java`

```
1.  import java.util.*;
2.
3. /**
4. * Эта программа демонстрирует работу цикла <code>do/while</code>.
5. * @version 1.20 2004-02-10
6. * @author Cay Horstmann
7. */
8.  public class Retirement2
9.  {
10.    public static void main(String[] args)
11.    {
```

```
12.       Scanner in = new Scanner(System.in);
13.
14.       System.out.print("Сколько вы можете вносить в год? ");
15.       double payment = in.nextDouble();
16.
17.       System.out.print("Процентная ставка: ");
18.       double interestRate = in.nextDouble();
19.
20.       double balance = 0;
21.       int year = 0;
22.
23.       String input;
24.
25.       // Обновлять баланс, пока пользователь вводит N
26.       do
27.       {
28.          // Добавление ежегодного взноса и процентов
29.          balance += payment;
30.          double interest = balance * interestRate / 100;
31.          balance += interest;
32.
33.          year++;
34.
35.          // Вывод текущего баланса
36.          System.out.printf("Через %d, у вас будет %,.2f%n", year, balance);
37.
38.          // Запрос, готов ли пользователь прекратить работу
39.          System.out.print("Уходите на отдых? (Y/N) ");
40.          input = in.next();
41.       }
42.       while (input.equals("N"));
43.    }
44. }
```

Определенные циклы

Цикл `for` — очень распространенная языковая конструкция. В ней количество повторений контролируется переменной, играющей роль счетчика и обновляемой на каждой итерации. Приведенный ниже цикл выводит на экран числа от 1 до 10. Его выполнение показано на рис. 3.12.

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

Первый элемент оператора `for` обычно выполняет инициализацию счетчика, второй формулирует условие выполнения тела цикла, а третий определяет способ обновления счетчика.

Хотя в языке Java, как и в C++, элементами `for` могут быть практически любые операторы, существуют неписанные правила, согласно которым все три элемента оператора `for` должны только инициализировать, проверять и обновлять один и тот же счетчик. Если не придерживаться этих правил, полученный код станет совершенно непригодным для чтения.

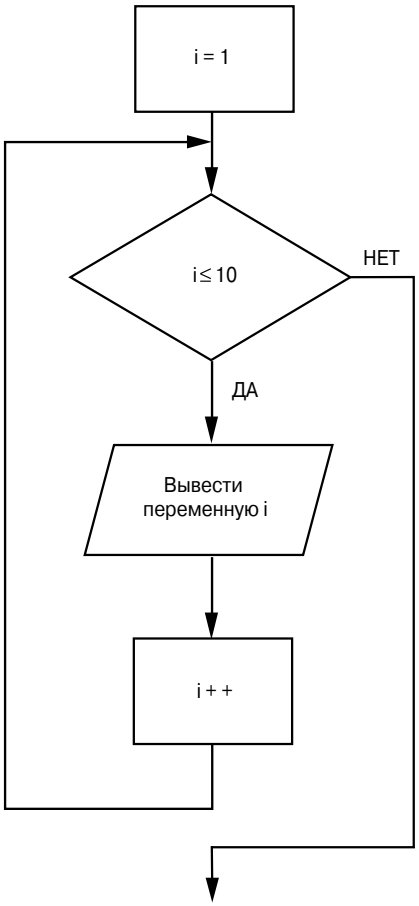


Рис. 3.12. Блок-схема, поясняющая выполнение оператора `for`

Данные правила не слишком сковывают инициативу программиста. Даже придерживаясь их, можно сделать очень многое, например, реализовать цикл с убывающим счетчиком.

```
for (int i = 10; i > 0; i--)
   System.out.prinln("Обратный отсчет ..." + i);
System.out.println("Старт!);
```



Внимание! Будьте осторожны, проверяя в цикле равенство двух чисел с плавающей точкой. Представленный ниже цикл может никогда не завершиться.

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

Из-за ошибок округления окончательный результат никогда не будет достигнут. Например, в приведенном выше цикле переменная `x` изменит свое значение с `9.999999999999998` на `10.099999999999998`. Так произойдет потому, что для числа `0.1` не существует точного двоичного представления.

При объявлении переменной в первой части оператора `for` ее область видимости простирается до конца тела цикла.

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

В частности, если переменная определена в операторе `for`, ее нельзя использовать вне этого цикла. Следовательно, если вы хотите использовать конечное значение счетчика вне цикла `for`, эту переменную нужно объявлять до начала цикла!

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

С другой стороны, можно объявлять переменные, имеющие одинаковое имя в разных циклах `for`.

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

Действия, выполняемые посредством цикла `for`, можно реализовать с помощью цикла `while`. Приведенные ниже два фрагмента кода эквивалентны.

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

и

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

В листинге 3.5 показан типичный пример использования цикла `for`.

Эта программа вычисляет вероятность выигрыша в лотерее. Например, если нужно угадать 6 номеров из 50, количество возможных вариантов равно (50×49×48×47×46×45)/(1×2×3×4×5×6), так что шанс выигрыша равен 1 из 15 890 700. Удачи!

В общем случае, если нужно угадать k номеров из n, количество возможных вариантов равно следующему выражению:

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

Оно вычисляется с помощью следующего цикла `for`:

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



На заметку! В конце данной главы, в разделе “Цикл “for each””, будет рассмотрен обобщенный цикл `for`, называемый также циклом “for each”. Эта языковая конструкция была введена в Java SE 5.0.

Листинг 3.5. Содержимое файла `LotteryOdds.java`

```
1. import java.util.*;
2. /**
3. * Эта программа демонстрирует работу цикла <code>for</code>.
4. * @version 1.20 2004-02-10
5. * @author Cay Horstmann
6. */
7.  public class LotteryOdds
8.  {
9.     public static void main(String[] args)
10.    {
11.       Scanner in = new Scanner(System.in);
12.
13.       System.out.print("Сколько номеров? ");
14.       int k = in.nextInt();
15.
16.       System.out.print("Наибольший номер? ");
17.       int n = in.nextInt();
18.
19.        /*
20.         Вычисление биномиального коэффициента
21.         n * (n - 1) * (n - 2) * . . . * (n - k + 1)
22.         -------------------------------------------
23.         1 * 2 * 3 * . . . * k
24.       */
25.
26.       int lotteryOdds = 1;
27.       for (int i = 1; i <= k; i++)
28.          lotteryOdds = lotteryOdds * (n - i + 1) / i;
29.       System.out.println("Ваши шансы 1 из " + lotteryOdds + ". Успехов!");
30.    }
31. }
```

Многовариантное ветвление — оператор `switch`

Конструкция `if/else` может оказаться неудобной, если вам необходимо реализовать выбор из многих вариантов. В языке Java есть оператор `switch`, эквивалентный одноименному оператору, используемому в C и C++.

Например, программируя выбор из четырех альтернативных вариантов (рис. 3.13), можно использовать следующий код:

```
Scanner in = new Scanner(System.in);
System.out.print("Выберите вариант (1, 2, 3, 4) ");
int choice = in.nextInt();
switch (choice)
{
  case 1:
     ...
     break;
  case 2:
     ...
     break;
```

```
   case 3:
      ...
      break;
   case 4:
      ...
      break;
   default:
      // Неверный выбор.
      ...
      break;
}
```

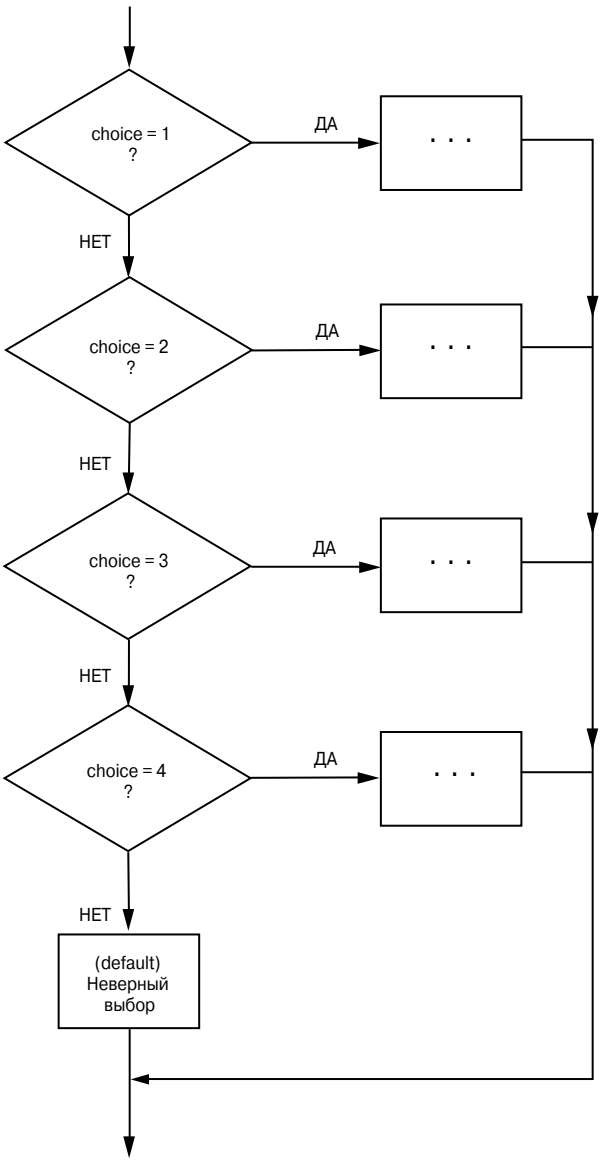


Рис. 3.13. Блок-схема, поясняющая действие оператора `switch`

Выполнение начинается с метки case, соответствующей значению переменной choice, и продолжается до очередного оператора break или конца оператора switch. Если ни одна метка не совпадает со значением переменной, выполняется раздел default (если он предусмотрен).



Внимание! Если вы забудете добавить в конце раздела case оператор break, возможно последовательное выполнение нескольких разделов case. Совершенно очевидно, что эта ситуация чревата ошибками, поэтому мы никогда не используем оператор switch в своих программах.

Заметим, что метка case должна быть целочисленной или перечислимой константой. Нельзя проверять строки. Например, в следующем фрагменте кода присутствует ошибка:

```
String input = ...;
switch (input)    // ОШИБКА
{
   case "A":      // ОШИБКА
      ...
      break;
   ...
}
```

Когда вы используете оператор switch с перечислимыми константами, вы не должны применять имя перечисления в каждой метке — оно выводится из значения switch. Например:

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

Прерывание потока управления

Несмотря на то что разработчики языка Java сохранили зарезервированное слово goto, они решили не включать его в язык. В принципе применение операторов goto считается признаком плохого стиля программирования. Некоторые программисты считают, что борьба с использованием оператора goto ведется недостаточно активно (см., например, знаменитую статью Дональда Кнута “Structured Programming with goto statements”). Они считают, что применение операторов goto может приводить к ошибкам. Однако в некоторых случаях нужно выполнять преждевременный *выход из цикла*. Создатели языка Java согласились с их аргументами и даже добавили в язык новый оператор для поддержки такого стиля программирования — оператор break с меткой.

Рассмотрим обычный оператор break, в котором не применяется метка. Для выхода из цикла можно применять тот же оператор, который использовался для выхода из тела оператора switch. Пример применения оператора break приведен ниже.

```
while (years <= 100)
{
   balance += payment;
   double interest = balance * interestRate / 100;
```

```
   balance += interest;
   if(balance >= goal) break;
   years++;
}
```

Теперь выход из цикла осуществляется при выполнении одного из двух условий: `years > 100` — в начале цикла либо `balance >= goal` — внутри цикла. Разумеется, то же значение для `years` можно было бы вычислить и без применения оператора `break`:

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

Заметим, однако, что проверка условия `balance < goal` в данном варианте программы повторяется дважды. Оператор `break` позволяет избежать этого.

В отличие от языка C++, язык Java содержит также оператор `break` *с меткой*, обеспечивающий выход из вложенных циклов. Если во вложенном цикле выполняется некоторое условие, возможно, потребуется выйти из всех вложенных циклов. Программировать дополнительные условия для проверки каждого вложенного цикла попросту неудобно.

Ниже приводится пример, иллюстрирующий работу оператора `break` с меткой. Заметим, что метка должна предшествовать тому внешнему циклу, из которого требуется выйти. Метка должна оканчиваться двоеточием.

```
Scanner in = new Scanner(System.in);
int n;
read_data:
while (. . .) // Этот цикл помечен
{
   . . .
   for (. . .) // Этот цикл не помечен
   {
      System.out.print("Введите число >= 0: ");
      n = in.nextInt();
      if (n < 0) // Если это не произойдет, цикл продолжится
         break read_data;
         // Прерывание цикла
      . . .
   }
}
// Данный оператор выполняется сразу же после
// оператора break с меткой
if (n < 0) // Проверка на наличие недопустимой ситуации
{
   // Обработка недопустимой ситуации
}
else
{
   // Действия при нормальном выполнении программы
}
```

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



На заметку! Это может показаться странным, но метку можно связать с любым оператором, даже оператором if или блоком. Соответствующий пример приведен ниже.

```
метка:
{
      ...
   if (условие) break метка; // Выход из блока.
   ...
}
// При выполнении оператора break управление передается в эту точку.
```

Итак, если вам крайне необходим оператор goto, вы можете поместить блок, из которого нужно выйти, и применить оператор break! Естественно, мы не рекомендуем использовать этот способ. Заметим также, что подобным образом можно выйти из блока, но невозможно войти в него.

Существует также оператор continue, который, подобно break, прерывает нормальное выполнение программы. Оператор continue передает управление в начало текущего вложенного цикла. Пример использования данного оператора приведен ниже.

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

Если n < 0, то оператор continue выполняет переход в начало цикла, пропуская оставшуюся часть текущей итерации.

Если оператор continue используется в цикле for, он передает управление оператору увеличения счетчика цикла. Например, рассмотрим следующий цикл:

```
for (count = 0; count < 100; count ++)
{
   System.out.print("Введите число, -1 для завершения: ");
   n = in.nextInt();
   if (n < 0) continue;
   sum += n; // Не выполняется, если n < 0.
}
```

Если n < 0, то оператор continue выполнит переход к оператору count++.

В языке Java есть также оператор continue с меткой, передающий управление заголовку оператора цикла, помеченного соответствующей меткой.



Совет. Многие программисты считают, что операторы break и continue неоправданно усложняют текст программы. Применять эти операторы вовсе не обязательно — те же действия можно реализовать, не прибегая к ним. В этой книге мы нигде не используем ни break, ни continue.

Большие числа

Если для решения задачи недостаточно точности встроенных целочисленных типов и чисел с плавающей точкой, можно обратиться к классам `BigInteger` и `BigDecimal` из пакета `java.math`. Эти классы предназначены для выполнения действий с числами, состоящими из произвольного количества цифр. Классы `BigInteger` и `BigDecimal` реализуют арифметические операции произвольной точности соответственно для целых и действительных чисел.

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

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

К сожалению, к большим числам нельзя применять обычные математические операции вроде + или *. Вместо этого надо использовать методы `add()` и `multiply()` из соответствующих классов.

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



На заметку! В отличие от C++, язык Java не поддерживает перегрузку операций. Поэтому разработчики класса `BigInteger` были лишены возможности переопределить операции + и * для методов `add()` и `multiply()` в классе `BigInteger`. Проектировщики языка перегрузили только операцию + для обозначения конкатенации строк. Они решили не перегружать остальные операции и не предоставили такой возможности программистам Java в их собственных классах.

В листинге 3.6 показана модифицированная программа для подсчета шансов выигрыша в лотерее (ее исходный вариант представлен в листинге 3.5). Теперь эта программа может работать с большими числами. Например, если вам предложили сыграть в лотерее, в которой нужно угадать 60 чисел из 490 возможных, то эта программа сообщит вам, что шанс выигрыша составляет 1 из 716395843461995557415116222540092933411717612789263493493351013459481104668848. Удачи!

Программа, представленная в листинге 3.5, вычисляла следующее выражение:

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

При работе с большими числами соответствующая строка кода выглядит так:

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

Листинг 3.6. Содержимое файла `BigIntegerTest.java`

```
1. import java.math.*;
2. import java.util.*;
3.
4. /**
5. * Эта программа использует большие числа для оценки шансов выигрыша в лотерею.
6. * @version 1.20 2004-02-10
7. * @author Cay Horstmann
8. */
9. public class BigIntegerTest
10. {
```

```
11.   public static void main(String[] args)
12.   {
13.      Scanner in = Scanner.create(System.in);
14.
15.      System.out.print("Сколько номеров? ");
16.      int k = in.nextInt();
17.
18.      System.out.print("Наибольший номер? ");
19.      int n = in.nextInt();
20.
21.      /*
22.         Вычисление биномиального коэффициента
23.         n * (n - 1) * (n - 2) * . . . * (n - k + 1)
24.         ---------------------------------------
25.         1 * 2 * 3 * . . . * k
26.      */
27.
28.      BigInteger lotteryOdds = BigInteger.valueOf(1);
29.
30.      for (int i = 1; i <= k; i++)
31.         lotteryOdds = lotteryOdds.multiply(BigInteger.valueOf(n - i + 1))
32.            .divide(BigInteger.valueOf(i));
33.
34.      System.out.println("Ваши шансы 1 из " + lotteryOdds + ". Успехов!");
35.   }
36. }
```

API java.math.BigInteger 1.1

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

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

- `int compareTo(BigInteger other)`

  Возвращает `0`, если данное большое число равно значению переменной `other`, отрицательное число, если данное большое число меньше значения переменной `other`, и положительное число в противном случае.

- `static BigInteger valueOf(long x)`

  Возвращает большое число, значение которого равно значению переменной `x`.

API java.math.BigDecimal 1.1

- `BigDecimal add(BigDecimal other)`
- `BigDecimal subtract(BigDecimal other)`
- `BigDecimal multiply(BigDecimal other)`
- `BigDecimal divide(BigDecimal other, int roundingMode)` **5.0**

Возвращает сумму, разность, произведение и частное, полученные в результате выполнения соответствующих операций над данным большим числом и значением переменной `other`. Чтобы вычислить частное, нужно указать *режим округления*. Режим `roundingMode.ROUND_HALF_UP` означает привычное всем округление (т.е. в сторону уменьшения для цифры меньше 5 и в сторону увеличения, если соответствующая цифра больше или равна 5). Для обычных вычислений этого достаточно. Другие режимы округления описаны в документации.

- `int compareTo(BigDecimal other)`

  Возвращает `0`, если данное число типа `BigDecimal` равно значению переменной `other`, отрицательное число, если данное число меньше значения переменной `other`, и положительное число в противном случае.

- `static BigDecimal valueOf(long x)`
- `static BigDecimal valueOf(long x, int scale)`

  Возвращает большое десятичное число, значение которого равно значению переменной `x` или $x/10^{scale}$.

Массивы

Массив — это структура данных, в которой хранятся величины одинакового типа. Доступ к отдельному элементу массива осуществляется с помощью целочисленного *индекса*. Например, если `a` — массив целых чисел, то значение выражения `a[i]` равно `i`-му целому числу в массиве.

Массив объявляется следующим образом: сначала указывается тип массива, т. е. тип элементов, содержащихся в нем, затем следует пара пустых квадратных скобок, а после них — имя переменной. Ниже приведено объявление массива, состоящего из целых чисел.

```
int[] a;
```

Однако этот оператор лишь объявляет переменную `a`, не инициализируя ее. Чтобы создать массив, нужно применить операцию `new`.

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

Приведенный оператор создает массив, состоящий из 100 целых чисел. При создании массива чисел все элементы его инициализируются нулями. (Массивы значений `boolean` инициализируются `false`, а массивы объектов — `null`.)



На заметку! Объявить массив можно двумя способами:

```
int[] a;
```

или

```
int a[];
```

Большинство программистов, работающих на Java, используют первый способ, так как при этом тип более явно отделяется от имени переменной.

Элементы сформированного выше массива *нумеруются от 0 до 99* (а не от 1 до 100). После создания массив можно заполнять конкретными значениями, в частности, это можно делать в цикле.

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



Внимание! Если, создав массив, состоящий из 100 элементов, вы попытаетесь обратиться к `a[100]` (или любому другому элементу, индекс которого выходит за пределы диапазона от 0 до 99), программа прекратит работу, поскольку будет сгенерировано исключение, соответствующее выходу индекса массива за пределы допустимого диапазона.

Чтобы подсчитать количество элементов в массиве, используйте выражение *имя_массива*`.length`. Например:

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

После создания массива изменить его размер невозможно (хотя можно, конечно, изменять его отдельные элементы). Если в ходе выполнения программы необходимо часто изменять размер массива, лучше использовать другую структуру данных, называемую *списком*. (Подробнее о списочных массивах читайте в главе 5.)

Цикл “for each”

В Java SE 5.0 был реализован новый цикл, позволяющий перебирать все элементы массива (а также любого другого набора данных), не применяя счетчик.

Новый вариант цикла `for` записывается следующим образом:

```
for (переменная : набор_данных) оператор
```

При обработке цикла переменной последовательно присваивается каждый элемент набора данных, после чего выполняется *оператор* (или блок). В качестве набора данных может использоваться массив либо экземпляр класса, реализующего интерфейс `Iterable`, например `ArrayList`. Списки будут обсуждаться в главе 5, а интерфейс `Iterable` — во втором томе.

Ниже приведен пример использования рассматриваемого здесь цикла.

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

В результате выполнения данного фрагмента кода каждый элемент массива будет напечатан в отдельной строке.

Действия данного цикла можно кратко описать как “обработка каждого элемента из `a`”. Разработчики языка Java рассматривали возможность применения в качестве идентификатора данного цикла ключевых слов `foreach` и `in`. Однако данный тип цикла появился намного позже основных языковых средств Java, и введение нового ключевого слова привело бы к необходимости изменять исходный код некоторых готовых приложений, содержащих переменные или методы с такими именами (вроде `System.in`).

Конечно, действия, выполняемые с помощью нового цикла, можно реализовать с помощью традиционного цикла `for`.

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

Однако при использовании нового цикла запись получается более краткой (необходимость в начальном выражении и условии завершения цикла отпадает), и вероятность возникновения ошибок уменьшается.



На заметку! Переменная цикла в “for each” перебирает не значения индекса, а элементы массива.

Несмотря на появление нового цикла “for each”, упрощающего во многих случаях составление программ, традиционный цикл `for` отнюдь не устарел. Без него нельзя обойтись, например, в тех случаях, когда вам надо обработать не весь набор данных, а лишь его часть, либо тогда, когда счетчик явно используется в теле цикла.



Совет. Есть еще более простой способ напечатать все значения из массива — используя метод `toString()` класса `Arrays`. Вызов `Arrays.toString(a)` вернет строку, содержащую все элементы массива, заключенную в квадратные скобки и разделенную запятыми, вот так: `"[2, 3, 5, 7, 11, 13]"`. Чтобы напечатать массив, просто вызовите `System.out.println(Arrays.toString(a));`

Инициализация массивов и анонимные массивы

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

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

Обратите внимание, что в этом случае операция `new` не нужна.

Кроме того, можно даже инициализировать массив, не имеющий имени, или *анонимный массив*.

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

Данное выражение выделяет память для нового массива и заполняет его числами, указанными в фигурных скобках. При этом подсчитывается их количество и соответственно определяется размер массива. Такую синтаксическую конструкцию удобно применять для повторной инициализации массива без образования новой переменной. Например, выражение

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

представляет собой сокращенную запись выражения

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



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

```
new тип_элементов[0]
```

Заметим, что такой массив не эквивалентен объекту `null`. (Подробно этот вопрос будет обсуждаться в главе 4.)

Копирование массивов

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

```
int[] luckyNumbers = smallPrimes;
luckyNuimbers[5] = 12;  // Теперь элемент smallPrimes[5] также равен 12.
```

Результат копирования переменных массивов показан на рис. 3.14.

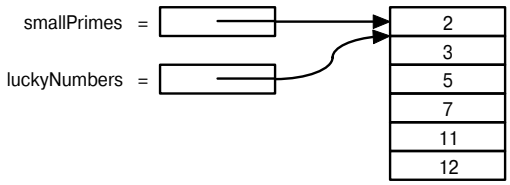


Рис. 3.14. Копирование переменной массива

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

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

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

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

Дополнительные элементы заполняются нулями, если массив содержит числа, и `false` — если значения типа `boolean`. В противоположность этому, если длина нового массива меньше длины исходного, то копируются только начальные элементы.



На заметку! До появления Java SE 6 для копирования элементов массива использовался метод `arrayCopy` класса `System`. Синтаксис этого вызова выглядит следующим образом:

```
System.arraycopy(from, fromIndex, to, toIndex, count);
```

Массив `to` должен иметь достаточный размер, чтобы в нем уместились все копируемые элементы.

Ниже приведен фрагмент кода, результаты выполнения которого показаны на рис. 3.15. Вначале создаются два массива, а затем последние четыре элемента первого массива копируются во второй. Копирование исходного массива начинается с элемента с номером 2, а в целевой массив копируемые данные помещаются, начиная с элемента с номером 3.

```
int[] smallPrimes = {2, 3, 5, 7, 11, 13};
int[] luckyNumbers = {1001, 1002, 1003, 1004, 1005, 1006, 1007};
System.arraycopy(smallPrimes, 2, luckyNumbers, 3, 4);
for ( int i = 0; i < luckyNumbers.length; i++)
   System.println(i + ":  " + luckyNumbers[i]);
```

Выполнение данного фрагмента приводит к следующему результату:

```
0:  1001
1:  1002
2:  1003
3:  5
4:  7
5:  11
6:  13
```

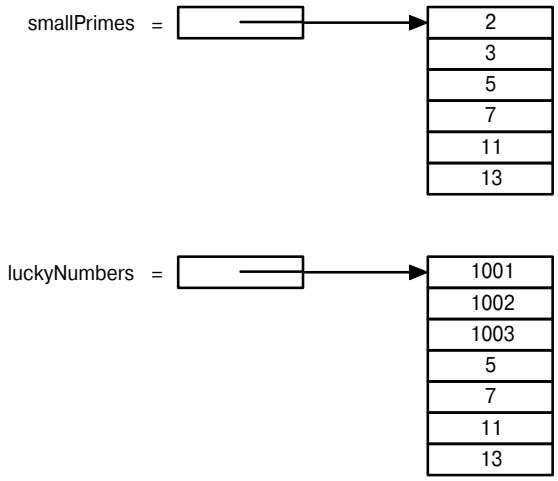


Рис. 3.15. Копирование элементов массива

На заметку! Массив в языке Java значительно отличается от массива в C++, располагающегося в стеке. Однако переменную массива можно условно сравнить с указателем на *динамически созданный* массив. Таким образом, выражение Java

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

можно сравнить с выражением C++

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

но оно существенно отличается от следующего:

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

В языке Java операция `[]` по умолчанию проверяет *диапазон изменения индексов*. Кроме того, в языке Java не поддерживается арифметика указателей, — нельзя увеличить указатель `a`, чтобы обратиться к следующему элементу массива.

Параметры командной строки

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

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

При следующем вызове программы

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

массив `args` будет состоять из таких элементов:

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

Программа выведет сообщение

```
Goodbye, cruel world!
```



На заметку! При запуске программы на языке Java в методе `main` ее имя не помещается в массиве `args`. Например, после запуска из командной строки программы `Message` с помощью команды

```
java Message -h world
```

элемент `args[0]` будет равен `"-h"` , а не `"Message"` или `"java"`.

Сортировка массива

Если нужно упорядочить массив чисел, можно применить метод `sort()` из класса `Arrays`.

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

Этот метод использует усовершенствованную версию алгоритма быстрой сортировки, которая считается наиболее эффективной для большинства наборов данных. Класс `Arrays` содержит несколько удобных методов, предназначенных для работы с массивами. Эти методы приведены в конце данного раздела.

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

```
Попробуйте такую комбинацию, и станете богатым!
  4
  7
  8
  19
  30
  44
```

Чтобы выбрать случайные числа, массив `numbers` сначала заполняется числами 1, 2, ..., `n`:

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

Второй массив предназначен для хранения сгенерированных чисел:

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

Теперь сгенерируем `k` чисел. Метод `Math.random()` возвращает случайное число с плавающей точкой, лежащее в интервале от 0 (включительно) до 1 (это значение не

принадлежит интервалу). Умножая результат на число n, получим случайное число, лежащее между 0 и n-1.

```
int r = (int)(Math.random() * n);
```

Присвоим i-е число i-му элементу массива. Сначала там будет помещено само число r, однако, как будет показано ниже, содержимое массива number будет изменяться после генерации каждого нового числа:

```
result[i] = numbers[r];
```

Теперь мы должны убедиться, что ни одно число не повторится — все номера должны быть разными. Следовательно, нужно поместить в элемент number[r] *последнее* число, содержащееся в массиве, и уменьшить n на единицу.

```
numbers[r] = numbers[n - 1];
n--;
```

Обратите внимание на то, что при каждой генерации мы получаем *индекс*, а не само число. Этот индекс относится к массиву, содержащему числа, которые еще не были выбраны.

После генерации k номеров полученный массив сортируется, чтобы результат выглядел более элегантно:

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

Листинг 3.7. Содержимое файла `LotteryDrawing.java`

```
1. import java.util.*;
2.
3. /**
4. * Эта программа демонстрирует манипуляции с массивами.
5. * @version 1.20 2004-02-10
6. * @author Cay Horstmann
7. */
8.
9. public class LotteryDrawing
10.{
11.   public static void main(String[] args)
12.   {
13.      Scanner in = new Scanner(System.in);
14.
15.      System.out.print("Число номеров? ");
16.      int k = in.nextInt();
17.
18.      System.out.print("Максимальный номер? ");
19.      int n = in.nextInt();
20.
21.      // Заполнение массива числами 1 2 3 . . . n
22.      int[] numbers = new int[n];
23.      for (int i = 0; i < numbers.length; i++)
24.         numbers[i] = i + 1;
25.
26.      // Выбор k номеров и запись их в другой массив
27.      int[] result = new int[k];
```

```
28.      for (int i = 0; i < result.length; i++)
29.      {
30.         // Генерация случайного индекса между 0 и n - 1
31.         int r = (int) (Math.random() * n);
32.
33.         // Выбор случайного элемента
34.         result[i] = numbers[r];
35.
36.         // Копирование последнего элемента
37.         numbers[r] = numbers[n - 1];
38.         n--;
39.      }
40.
41.      // Вывод сортированного массива
42.      Arrays.sort(result);
43.     System.out.println("Попробуйте такую комбинацию, и станете богатым!");
44.      for (int r : result)
45.         System.out.println(r);
46.   }
47. }
```

API java.util.Arrays 1.2

- `static String toString(type[] a)` **5.0**

  Возвращает строку с элементами `a`, заключенную в квадратные скобки и разделенную запятыми.

  Параметры: `a` массив типа `int`, `long`, `short`, `char`, `byte`, `boolean`, `float` или `double`.

- `static type copyOf(type[] a, int length)` **6**
- `static type copyOf(type[] a, int start, int end)` **6**

  Возвращает массив того же типа, что и `a`, длиной либо `length`, либо `end - start`, заполненный значениями из `a`.

  Параметры: `a` массив типа `int`, `long`, `short`, `char`, `byte`, `boolean`, `float` или `double`;

  `start` начальный индекс (включительно);

  `end` конечный индекс (исключительно). Может быть больше чем `a.length`, при этом результирующий массив дополняется нулями или значениями `false`;

  `length` длина копии. Если `length` больше чем `a.length`, результат дополняется нулями или значениями `false`. В противном случае копируется только начальные `length` значений.

- `static void sort(type[] a)` **6**

  Сортирует массив, используя алгоритм быстрой сортировки (QuickSort).

  Параметры: `a` массив типа `int`, `long`, `short`, `char`, `byte`, `boolean`, `float` или `double`.

- `static int binarySearch(`*type*`[] a, `*type*` v)` **6**
- `static int binarySearch(`*type*`[] a, int start, int end, `*type*` v)` **6**

Использует алгоритм бинарного поиска для нахождения значения `v`. В случае успеха возвращается индекс найденного элемента. В противном случае возвращается отрицательное значение `r`; `-r - 1` указывает на индекс позиции, куда должен быть вставлен искомый элемент, чтобы сохранился порядок сортировки.

Параметры: `a` массив типа `int`, `long`, `short`, `char`, `byte`, `boolean`, `float` или `double`;

`start` начальный индекс (включительно);

`end` конечный индекс (исключительно);

`v` значение того же типа, что и элементы `a`.

- `static void fill(`*type*`[] a, `*type*` v)`

Устанавливает все элементы массива в `v`.

Параметры: `a` массив типа `int`, `long`, `short`, `char`, `byte`, `boolean`, `float` или `double`;

`v` значение того же типа, что и элементы `a`.

- `static boolean equals(`*type*`[] a, `*type*`[] b)`

Возвращает `true`, если массивы имеют равную длину и совпадают все их элементы в соответствующих позициях индекса.

Параметры: `a, b` типа `int`, `long`, `short`, `char`, `byte`, `boolean`, `float` или `double`.

API java.lang.System 1.1

- `static void arraycopy(Object from, int fromIndex, Object to, int toIndex, int count)`

Копирует элементы из первого массива во второй.

Параметры `from` Массив любого типа (причины, по которым этот параметр должен иметь тип `Object`, объясняются в главе 5);

`fromIndex` Индекс, с которого начинается копирование элементов исходного массива;

`to` Массив, имеющий тот же тип, что и массив `from`;

`toIndex` Индекс, с которого начинается копирование элементов в целевой массив;

`count` Количество копируемых элементов.

Многомерные массивы

Для доступа к элементам многомерного массива применяется несколько индексов. Такие массивы используются для хранения таблиц и более сложных упорядоченных структур данных. Если многомерные массивы в вашей работе не нужны, можете смело пропустить этот раздел.

Предположим, что вам нужно создать таблицу чисел, показывающих, как возрастет первоначальная инвестиция объемом 10 000 долларов при разных процентных ставках, если прибыль ежегодно выплачивается и реинвестируется. Необходимые данные показаны в табл. 3.8.

Таблица 3.8. Рост вклада при разных процентных ставках

10%	11%	12%	13%	14%	15%
10000,00	10000,00	10000,00	10000,00	10000,00	10000,00
11000,00	11100,00	11200,00	11300,00	11400,00	11500,00
12100,00	12321,00	12544,00	12769,00	12996,00	13225,00
13310,00	13676,31	14049,28	14428,97	14815,44	15208,75
14641,00	15180,70	15735,19	16304,74	16889,60	17490,06
16105,10	16850,58	17623,42	18424,35	19254,15	20113,57
17715,61	18704,15	19738,23	20819,52	21949,73	23130,61
19487,17	20761,60	22106,81	23526,05	25022,69	26600,20
21435,89	23045,38	24759,63	26584,44	28525,86	30590,23
23579,48	25580,37	26730,79	30040,42	32519,49	35178,76

Очевидно, что эту информацию лучше всего хранить в двумерном массиве (или матрице), который мы назовем `balance`.

Объявить двумерный массив в языке Java довольно просто. Например, это можно сделать следующим образом:

```
double[][] balances;
```

Как обычно, мы не можем использовать массив, пока он не инициализирован с помощью операции `new`:

```
balances = new double[NYEARS][NRATES];
```

В других случаях, если элементы массива известны заранее, можно использовать сокращенную запись для его инициализации, в которой не применяется операция `new`.

```
int[][] magicSquare =
   {
     {16, 3, 2, 13},
     {5, 10, 11, 8},
     {9, 6, 7, 12},
     {4, 15, 14, 1}
   };
```

После инициализации массива к его отдельным элементам можно обращаться с помощью двух пар квадратных скобок, например `balances[i][j]`.

Программа, приведенная в качестве примера, сохраняет процентные ставки в одномерном массиве `interest`, а результаты подсчета баланса для каждого года и каждой процентной ставки — в двумерном массиве `balance`. Первая строка массива инициализируется исходной суммой:

```
for (int j = 0; j < balances[0].length; j++)
   balances[0][j] = 10000;
```

Затем мы подсчитываем содержимое остальных строк:

```
for (int i = 1; i < balances.length; i++)
{
   for (int j = 0; j < balances[i].length; j++)
   {
      double oldBalance = balances[i - 1][j];
```

```
      double interest = . . .;
      balances[i][j] = oldBalance + interest;
   }
}
```

В листинге 3.8 код программы приведен полностью.



На заметку! Цикл "for each" не обеспечивает автоматического перебора элементов двумерного массива. Он лишь перебирает строки, которые, в свою очередь, являются одномерными массивами. Для обработки всех элементов двумерного массива `a` нужно два цикла:

```
for (double[] row : a)
   for (double value : row)
      Обработка value
```



Совет. Чтобы напечатать на скорую руку список элементов двумерного массива, используйте

```
System.out.println(Arrays.deepToString(a));
```

Вывод будет отформатирован следующим образом:

```
[[16, 3, 2, 13], [5, 10, 11, 8], [9, 6, 7, 12], [4, 15, 14, 1]]
```

Листинг 3.8. Содержимое файла `CompoundInterest.java`

```
1. /**
2. * Эта программа демонстрирует сохранение табличных данных в двумерном массиве.
3. * @version 1.40 2004-02-10
4. * @author Cay Horstmann
5. */
6. public class CompoundInterest
7. {
8.    public static void main(String[] args)
9.    {
10.      final int STARTRATE = 10;
11.      final int NRATES = 6;
12.      final int NYEARS = 10;
13.
14.      // Установка процентной ставки 10 . . . 15%.
15.      double[] interestRate = new double[NRATES];
16.      for (int j = 0; j < interestRate.length; j++)
17.         interestRate[j] = (STARTRATE + j) / 100.0;
18.
19.      double[][] balances = new double[NYEARS][NRATES];
20.
21.      // Установка исходной суммы в 10000.
22.      for (int j = 0; j < balances[0].length; j++)
23.         balances[0][j] = 10000;
24.
25.      // Вычисление баланса на следующие годы.
26.      for (int i = 1; i < balances.length; i++)
27.      {
28.         for (int j = 0; j < balances[i].length; j++)
29.         {
30.            // Извлечение баланса за прошлый год.
31.            double oldBalance = balances[i - 1][j];
32.
```

```
33.           // Вычисление процентного дохода.
34.           double interest = oldBalance * interestRate[j];
35.
36.           // Вычисление баланса за текущий год.
37.           balances[i][j] = oldBalance + interest;
38.        }
39.     }
40.
41.     // Вывод процентных ставок.
42.     for (int j = 0; j < interestRate.length; j++)
43.        System.out.printf("%9.0f%%", 100 * interestRate[j]);
44.
45.     System.out.println();
46.
47.     // Вывод таблицы.
48.     for (double[] row : balances)
49.     {
50.        // Вывод строки таблицы.
51.        for (double b : row)
52.           System.out.printf("%10.2f", b);
53.
54.        System.out.println();
55.     }
56.  }
57. }
```

"Зубчатые" массивы

Все языковые конструкции, которые мы до сих пор рассматривали, мало отличались от других языков программирования. Однако механизм массивов в Java имеет особенность, предоставляющую совершенно новые возможности. В данном языке вообще *нет* многомерных массивов, только одномерные. Многомерные массивы — это "массивы массивов".

Например, массив `balances` в предыдущем примере фактически представляет собой массив, состоящий из 10 элементов, каждый из которых является массивом из шести элементов, представляющих собой числа с плавающей точкой (рис. 3.16).

Выражение `balance[i]` определяет `i`-й подмассив, т.е `i`-ю строку таблицы. Эта строка сама представляет собой массив, и выражение `balance[i][j]` относится к его `j`-му элементу.

Поскольку строки массива доступны из программы, их можно легко переставлять!

```
double[] temp = balances[i];
balances[i] = balances[i+1];
balances[i=1] = temp;
```

Кроме того, в языке Java легко создавать "зубчатые" массивы, т.е. массивы, у которых разные строки имеют разную длину. Рассмотрим стандартный пример. Создадим массив, в котором элемент, стоящий на пересечении `i`-й строки и `j`-го столбца, равен количеству возможностей "выбрать `j` чисел из `i`".

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
1   5  10  10   5   1
1   6  15  20  15   6   1
```

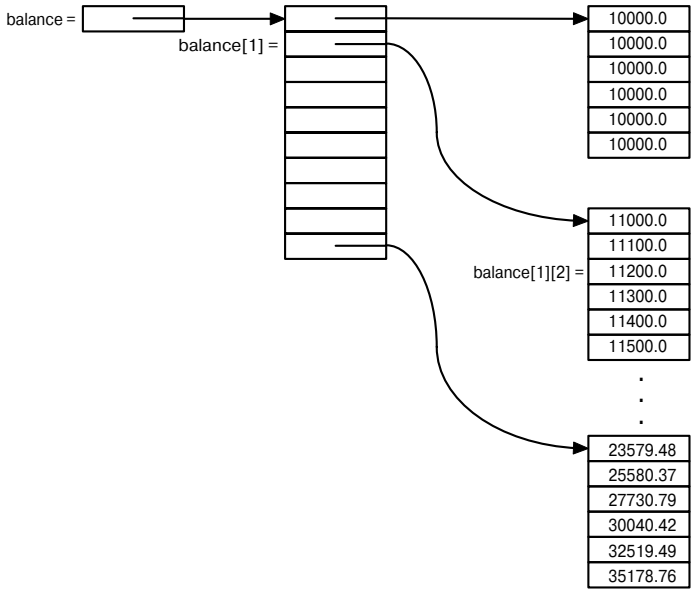


Рис. 3.16. Двумерный массив

Поскольку число `j` не может превышать `i`, получается треугольная матрица. В `i`-й строке этой матрицы находится `i+1` элементов. (Мы можем выбрать и 0 элементов; сделать это можно лишь одним-единственным способом.) Чтобы создать "зубчатый" массив, сначала разместим в памяти массив, хранящий его строки:

```
int[][] odds = new int[NMAX+1][];
```

Затем поместим туда сами строки:

```
for (int n=0; n<=NMAX; n++)
   odds[n] = new int[n+1];
```

Теперь в памяти размещен весь массив, так что мы можем обращаться к его элементам, как обычно, при условии, что индексы не выходят за пределы допустимого диапазона:

```
for (int n=0; n<odds.length; n++)
   for (int k=0; k<odds[n].length; k++)
   {
      // Вычисление вариантов.
      ...
      odds[n][k] = lotteryOdds;
   }
```

В листинге 3.9 приведен полный текст данной программы.



На заметку! В Java объявление

```
double[][] balance = new double[10][6]; // Java
```

не эквивалентно объявлению

```
double balance[10][6];                  // C++
```

и даже не соответствует

```
double (*balance)[6] = new double[10][6];// C++
```

Вместо этого в памяти размещается массив, состоящий из десяти указателей. Средствами C++ это можно выразить следующим образом:

```
double** balance = new double*[10];  // C++
```

Затем каждый элемент в массиве указателей заполняется массивом, состоящим из 6 чисел:

```
for (i = 0; i < 10; i++)
  balance[i] = new double[6];
```

К счастью, этот цикл выполняется автоматически при объявлении массива с помощью оператора `new double[10][6]`. Если вам нужны "зубчатые" массивы, размещайте массивы строк отдельно.

Листинг 3.9. Содержимое файла `LotteryArray.java`

```
1. /**
2. * Эта программа демонстрирует треугольный массив.
3. * @version 1.40 2004-02-10
4. * @author Cay Horstmann
5. */
6. public class LotteryArray
7. {
8.    public static void main(String[] args)
9.    {
10.      final int NMAX = 10;
11.      // Выделение памяти для треугольного массива
12.      int[][] odds = new int[NMAX + 1][];
13.      for (int n = 0; n <= NMAX; n++)
14.         odds[n] = new int[n + 1];
15.      // Заполнение треугольного массива
16.      for (int n = 0; n < odds.length; n++)
17.         for (int k = 0; k < odds[n].length; k++)
18.         {
19.            /*
20.               Вычисление биномиального коэффициента
21.               n * (n - 1) * (n - 2) * . . . * (n - k + 1)
22.               ---------------------------------------
23.               1 * 2 * 3 * . . . * k
24.            */
25.            int lotteryOdds = 1;
26.            for (int i = 1; i <= k; i++)
27.               lotteryOdds = lotteryOdds * (n - i + 1) / i;
28.            odds[n][k] = lotteryOdds;
29.         }
30.      // Вывод треугольного массива
31.      for (int[] row : odds)
32.      {
33.         for (int odd : row)
34.            System.out.printf("%4d", odd);
35.         System.out.println();
36.      }
37.   }
38. }
```

К этому моменту вы ознакомились со всеми основными программными структурами языка Java. Следующая глава посвящена объектно-ориентированному программированию на Java.

Глава 4

ОБЪЕКТЫ И КЛАССЫ

В ЭТОЙ ГЛАВЕ...

- Введение в объектно-ориентированное программирование
- Использование готовых классов
- Определение собственных классов
- Статические поля и методы
- Параметры методов
- Конструирование объектов
- Пакеты
- Путь поиска классов
- Комментарии и документирование
- Рекомендации по проектированию классов

В этой главе рассматриваются следующие вопросы:

- введение в объектно-ориентированное программирование;
- создание объектов классов из стандартной библиотеки Java;
- создание собственных классов.

Если вы недостаточно хорошо ориентируетесь в вопросах объектно-ориентированного программирования, внимательно прочитайте данную главу. Для объектно-ориентированного программирования требуется совершенно иной способ мышления по сравнению подходом, типичным для процедурно-ориентированных языков. Освоить новые принципы создания программ не всегда просто, но сделать это необходимо; для овладения языком Java надо хорошо знать основные понятия объектно-ориентированного программирования.

Программистам, имеющим большой опыт работы с языком C++, информация, содержащаяся в этой и в предыдущей главах, уже известна. Однако между Java и C++ есть существенные различия, поэтому последний раздел настоящей главы следует прочесть очень внимательно. Примечания, посвященные C++, помогут вам перейти от C++ к Java.

Введение в объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) в настоящее время стало доминирующей парадигмой программирования, вытеснив "структурные", процедурно-ориентированные подходы, разработанные в 1970-х годах. Java представляет собой полностью объектно-ориентированный язык, и для успешной работы с ним вам необходимо ознакомиться с принципами ООП.

Объектно-ориентированная программа состоит из объектов. Каждый объект имеет определенную функциональность, которую предоставляет в распоряжение пользователей, а также скрытую реализацию. Многие объекты ваших программ будут взяты в готовом виде из библиотеки, другие вы спроектируете самостоятельно. Строить свои объекты или покупать готовые — зависит от вашего бюджета или времени. Но, как правило, пока объекты соответствуют вашим спецификациям, вам не нужно беспокоиться о том, как они реализованы. В ООП вас не заботит реализация объектов, пока они делают то, что вам нужно.

Традиционное структурное программирование заключается в разработке набора процедур (или *алгоритмов*) для решения поставленной задачи. Определив эти процедуры, программист должен задать подходящий способ хранения данных. Вот почему создатель языка Pascal Никлаус Вирт (Niklaus Wirth) назвал свою знаменитую книгу по программированию *Алгоритмы + Структуры данных = Программы*. Обратите внимание, что в названии этой книги алгоритмы стоят на первом месте, а структуры данных — на втором. Это отражает образ мышления программистов того времени. Во-первых, они решали, как манипулировать данными; затем решали, какую структуру применить для организации этих данных, чтобы работать с ними было легче. Объектный подход в корне изменил ситуацию, поместив на первое место данные и лишь на второе — алгоритмы, предназначенные для их обработки.

Для решения небольших задач процедурный подход работает очень хорошо. Но объекты более приспособлены для более крупных задач. Рассмотрим небольшой Web-браузер. Его реализация может потребовать 2000 процедур, каждая из которых ма-

нипулирует набором глобальных данных. В объектно-ориентированном стиле та же программа может состоять из 100 классов, в каждом из которых в среднем определено по 20 методов (рис. 4.1). Вторая структура программы гораздо удобнее для программиста. В ней легче находить ошибки. Предположим, что данные некоторого объекта находятся в некорректном состоянии. Намного легче найти "виновника" среди 20 методов, имеющих доступ к данным, чем среди 2000 процедур.

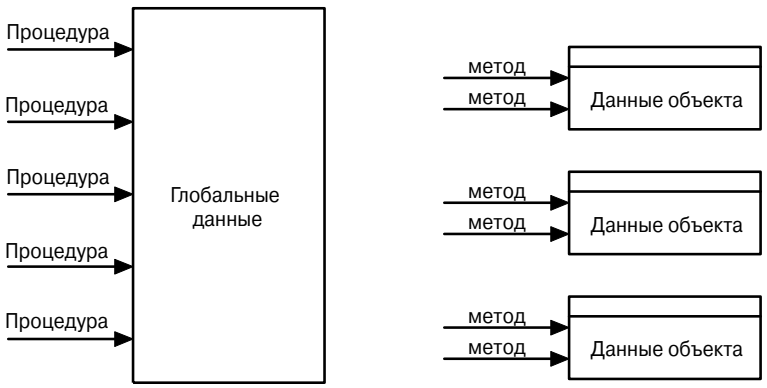


Рис. 4.1. Сравнение процедурного и ОО программирования

Классы

Для дальнейшей работы вам нужно освоить терминологию ООП. Наиболее важным термином является *класс*, который мы уже видели в примерах программ из главы 3. Класс — это шаблон, или проект, по которому будет сделан объект. Обычно класс сравнивают с формой для выпечки печенья. Объект — это само печенье. *Конструирование* объекта на основе некоторого класса называется созданием *экземпляра* (instance) этого класса.

Как вы уже видели, все коды, которые создаются в языке Java, находятся внутри классов. Стандартная библиотека языка Java содержит несколько тысяч классов, предназначенных для решения разных задач, например, для создания пользовательского интерфейса, календарей, установления сетевых соединений и т.д. Несмотря на это, программисты продолжают создавать свои собственные классы на языке Java, чтобы формировать объекты, характерные для разрабатываемого приложения, а также адаптировать классы из стандартной библиотеки для своих нужд.

Инкапсуляция (иногда называемая *сокрытием данных*) — это ключевое понятие при работе с объектами. Формально инкапсуляцией считается обычное объединение данных и операций над ними в одном пакете и сокрытие данных от других объектов. Данные в объекте называются *полями экземпляра* (instance fields), а функции и процедуры, выполняющие операции над данными, — его *методами* (methods). В конкретном объекте, т.е. экземпляре класса, поля экземпляра имеют определенные значения. Множество этих значений называется текущим *состоянием* (state) объекта. Применение любого метода к какому-нибудь объекту может изменить его состояние.

Еще раз подчеркнем, что основной принцип инкапсуляции заключается в запрещении прямого доступа к полям экземпляра данного класса из других классов. Программы должны взаимодействовать с данными объекта *только* с помощью методов этого объекта. Из сказанного следует, что в классе можно полностью изменить способ хранения данных, сохранив методы их обработки, и при этом остальные объекты смогут работать с соответствующими объектами так же, как и прежде.

Еще один принцип ООП облегчает разработку собственных классов в Java: класс можно построить на основе других классов. В этом случае говорят, что вновь созданный класс *расширяет* класс, на основе которого он создан. Язык Java, по существу, создан на основе "глобального суперкласса", называемого `Object`. Все остальные объекты расширяют его. В следующей главе мы рассмотрим этот вопрос подробнее.

Если вы разрабатываете класс на основе существующего, то новый класс содержит все свойства и методы расширяемого класса, кроме того, в нем добавляются новые методы и поля данных. Расширение класса и получение на его основе нового называется *наследованием* (inheritance). Детально принцип наследования будет описан в следующей главе.

Объекты

В объектно-ориентированном программировании определены следующие ключевые свойства объектов.

- *Поведение* (behavior) объекта — что с ним можно делать и какие методы к нему можно применять.
- *Состояние* объекта — как этот объект реагирует на применение методов.
- *Сущность* (identity) объекта — чем данный объект отличается от других, характеризующихся таким же поведением и состоянием.

Все объекты, являющиеся экземплярами одного и того же класса, *ведут* себя одинаково. Поведение объекта определяется методами, которые можно вызвать.

Далее, каждый объект сохраняет информацию о своем *состоянии*. Со временем состояние объекта может измениться, однако спонтанно это произойти не может. Состояние объекта может изменяться только в результате вызовов методов. (Если состояние объекта изменилось вследствие иных причин, значит, принцип инкапсуляции не соблюден.)

Состояние объекта не полностью описывает его, поскольку каждый объект имеет свою собственную *сущность*. Например, в системе обработки заказов два заказа могут отличаться друг от друга, даже если они относятся к одним и тем же товарам. Заметим, что индивидуальные объекты, представляющие собой экземпляры класса, *всегда* отличаются своей сущностью и *обычно* отличаются своим состоянием.

Эти основные характеристики могут влиять друг на друга. Например, состояние объекта может влиять на его поведение. (Если заказ "выполнен" или "оплачен", объект может отказаться выполнить вызов метода, требующего добавить или удалить изделие. И наоборот, если заказ "пуст", т.е. ни одна единица товара не была заказана, он не может быть выполнен.)

Идентификация классов

В традиционной процедурной программе выполнение начинается сверху, с функции `main`. При проектировании объектно-ориентированной системы понятие "верха" не существует, и новички в ООП часто интересуются, с чего начинать. Ответ таков: "Сначала найдите классы и добавьте методы в каждый класс".

Простое эмпирическое правило для идентификации классов заключается в выделении имен существительных при анализе проблемной области. С другой стороны, методы соответствуют глаголам.

Например, при описании системы обработки заказов используются следующие имена существительные:

- изделие;
- заказ;
- адрес доставки;
- оплата;
- счет.

Этим именам соответствуют классы `Item`, `Order` и т.д.

Рассмотрим теперь глаголы. Изделия *добавляются* к заказам. Заказы *выполняются* или *отменяются*. Оплата заказа *осуществляется*. Используя эти глаголы, можно определить объект, выполняющий такие действия. Например, если поступил новый заказ, то ответственность за его обработку должен нести объект `Order`, поскольку именно в нем содержится информация о способе хранения и сортировке заказываемых предметов. Следовательно, в классе `Order` должен существовать метод `add`, получающий объект `Item` в качестве параметра.

Разумеется, "правило имен существительных и глаголов" — это не более чем совет, и только опыт может помочь программисту решить, какие существительные и глаголы важны при создании класса.

Отношения между классами

Между классами существуют три общих отношения:

- *зависимость* ("использует");
- *агрегирование* ("содержит");
- *наследование* ("является").

Отношение зависимости (dependence — "uses-a") наиболее очевидное и распространенное. Например, класс `Order` использует класс `Account`, поскольку объекты `Order` должны иметь доступ к объектам `Account`, чтобы проверить кредитоспособность заказчика. Однако класс `Item` не зависит от класса `Account`, так как объекты `Item` никогда не интересуются состоянием счета заказчика. Следовательно, класс зависит от другого класса, если его методы выполняют какие-либо действия с экземплярами этого класса.

Старайтесь минимизировать количество взаимозависимых классов. Если класс `A` не знает о существовании класса `B`, он тем более ничего не знает о любых его изменениях! (Это значит, что любые изменения класса `B` не повлияют на поведение объектов класса `A`.)

Отношение агрегирования (aggregation — "has-a") понять довольно просто, поскольку оно конкретно. Например, объект `Order` может содержать объекты класса `Item`. Агрегирование означает, что объект класса `A` содержит объекты класса `B`.



На заметку! Некоторые специалисты не признают понятие агрегирования и предпочитают использовать отношение "связи" ("association"). С точки зрения моделирования это разумно. Однако для программистов отношение, при котором один объект содержит другой, гораздо удобнее. Мы предпочитаем использовать понятие агрегирования еще по одной причине: его обозначение проще для восприятия, чем обозначение отношения связи (табл. 4.1).

Наследование (inheritance – "is-a") выражает отношение между более конкретным и более общим классом. Например, класс `RushOrder` наследуется от класса `Order`. Специализированный класс `RushOrder` имеет специальные методы для обработки приоритетов и разные методы для вычисления стоимости доставки, в то время как другие его методы, например для заказа предмета и выписывания счета, унаследованы от класса `Order`. Вообще, если класс `A` расширяет класс `B`, то класс `A` наследует методы класса `B` и, кроме них, имеет дополнительные возможности. (Более подробно наследование описано в следующей главе.)

Многие программисты используют средства UML (Unified Modeling Language – унифицированный язык моделирования) для изображения *диаграмм классов* (class diagrams), описывающих отношения между классами. Пример такой диаграммы приведен на рис. 4.2. Здесь классы изображены с помощью прямоугольников, а отношения между ними – различными стрелками. В табл. 4.1 показаны основные обозначения, принятые в языке UML.

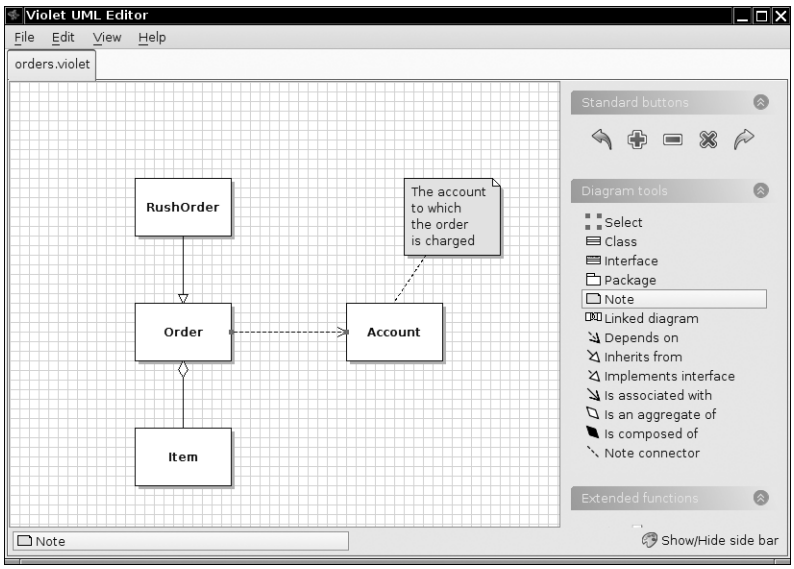
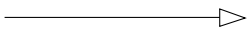
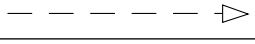
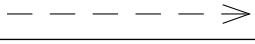
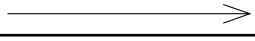


Рис. 4.2. Диаграмма классов



На заметку! Для создания диаграмм на языке UML разработано много инструментов. Некоторые поставщики предлагают очень мощные (и очень дорогие) программы, призванные стать основным средством проектирования. Среди них можно отметить продукты Rational Rose (`http://www.ibm.com/software/awdtools/developer/modeler`) и Together (`http://www.borland.com/together`). В качестве альтернативы можно назвать программу ArgoUML, созданную на принципах открытого кода (`http://argouml.tigris.org/`). Коммерческая версия программы поддержки URL доступна по адресу `http://gentleware.com`. Если вам нужно лишь нарисовать диаграмму, приложив минимум усилий, попробуйте воспользоваться программой Violet (`http://horstmann.com/violet`).

Таблица 4.1. Обозначение отношений между классами принятые в UML

Отношение	Обозначение в UML
Наследование	
Наследование интерфейса	
Зависимость	
Агрегирование	
Связь	
Направленная связь	

Использование готовых классов

Поскольку на языке Java ничего нельзя сделать без классов, мы вкратце обсудили в предыдущих разделах, как работают некоторые из них. Однако в Java есть классы, к которым не совсем подходят приведенные выше рассуждения. Ярким примером является класс `Math`. Как вы уже видели, методы класса `Math`, например метод `Math.random`, можно вызывать, ничего не зная о деталях их реализации. Для обращения к методу достаточно знать его имя и параметры (если они предусмотрены). Это признак инкапсуляции, и, безусловно, это справедливо по отношению ко всем классам. Но в классе `Math` нет данных, он инкапсулирует только функциональность, а, следовательно, нет необходимости в их сокрытии. Таким образом, программист может не заботиться о создании объектов и инициализации их полей, — в классе `Math` ничего подобного нет!

В следующем разделе мы рассмотрим класс `Date`. Мы увидим, как создаются экземпляры этого класса и вызываются методы.

Объекты и объектные переменные

Чтобы работать с объектами, их нужно сначала создать и задать их исходное состояние. Затем к этим объектам можно применять методы.

В языке Java для создания новых экземпляров используются *конструкторы*. Конструктор — это специальный метод, предназначенный для создания и инициализации экземпляра класса. В качестве примера можно привести класс `Date`, содержащийся в стандартной библиотеке Java. С помощью объектов этого класса можно описать текущий или любой другой момент времени, например “December 31, 1999, 23:59:59 GMP”.



На заметку! Может возникнуть вопрос: почему для представления даты и времени применяются классы, а не встроенные типы (как в некоторых языках)? Такой подход применяется, например, в Visual Basic; при этом дата задается в виде `#6/1/1995#`. На первый взгляд, это удобно — программист может использовать встроенный тип и не заботиться о классах. Однако не является ли удобство кажущимся? В некоторых странах даты записываются в виде месяц/день/год, а в других — год/месяц/день. Могут ли разработчики языка предусмотреть все возможные варианты? Даже если это удастся сделать, соответствующие средства будут слишком сложны, причем программисты будут вынуждены применять их. Использование классов позволяет переложить ответственность за решение этих проблем с разработчиков языка на создателей библиотек. Если системный класс плох, то разработчики всегда могут написать свой собственный. (В качестве иллюстрации отметим, что библиотека дат Java довольно запутана, и ее перепроектирование уже началось: см. `http://jcp.org/en/jsr/detail?id=310`.)

Имя конструктора всегда совпадает с именем класса. Следовательно, конструктор класса Date называется Date. Чтобы создать объект Date, конструктор нужно объединить с операцией new, как показано в следующем примере:

```
new Date()
```

Это выражение создает новый объект, который инициализируется текущими датой и временем.

При желании объект можно передать методу:

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

И наоборот, можно вызвать метод вновь созданного объекта. Среди методов класса Date есть метод toString(), позволяющий представить дату в виде строки. Его можно вызвать следующим образом:

```
String = new Date().toString();
```

В этих двух примерах созданный объект использовался только один раз. Обычно объект приходится использовать в дальнейшем. Чтобы это стало возможным, необходимо связать объект с некоторым идентификатором, другими словами, присвоить объект переменной.

```
Date birthday = new Date();
```

На рис. 4.3 условно показана переменная birthday, ссылающаяся на вновь созданный объект.

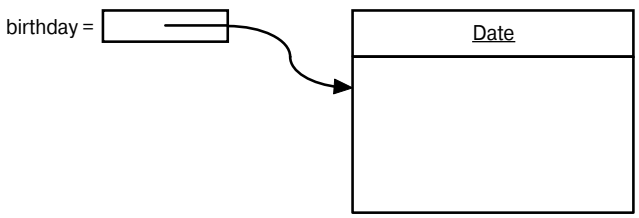


Рис. 4.3. Создание нового объекта

Между объектами и объектными переменными есть существенная разница. Например, приведенное ниже выражение определяет объектную переменную deadline, которая может ссылаться на объекты типа Date.

```
Date deadline;  // Переменная deadline не ссылается
                // ни на один объект
```

Важно понимать, что на данном этапе сама переменная deadline *объектом не является* и даже не ссылается ни на один объект. Поэтому ни один метод класса Date с помощью этой переменной вызывать пока нельзя. Попытка сделать это приведет к появлению сообщения об ошибке.

```
s = deadline.toString(); // Вызывать метод еще рано
```

Сначала переменную deadline нужно инициализировать. У программиста есть две возможности. Разумеется, переменную можно инициализировать вновь созданным объектом:

```
deadline = new Date();
```

Кроме того, можно заставить переменную ссылаться на существующий объект:

```
deadline = birthday;
```

Теперь переменные deadline и birthday ссылаются на *один и тот же* объект (рис. 4.4).

Важно помнить, что объектная переменная фактически не содержит никакого объекта. Она лишь *ссылается* на него.

В языке Java значение любой объектной переменной представляет собой ссылку на объект, размещенный в другом месте. Операция new также возвращает ссылку. Например, приведенная ниже строка кода состоит из двух частей.

```
Date deadline = new Date();
```

Выражение new Date() создает объект типа Date, а значение переменной представляет собой ссылку на вновь созданный объект.

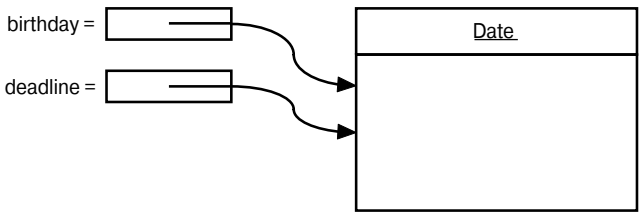


Рис. 4.4. Объектные переменные, ссылающиеся на один и тот же объект

Объектной переменной можно явно присвоить ссылку null, чтобы отметить тот факт, что она пока не ссылается ни на один объект.

```
deadline = null;
...
if(deadline != null)
   System.out.println(deadline);
```

Если вы попытаетесь вызывать метод объекта посредством переменной, значение которой равно null, то при выполнении программы возникнет ошибка.

```
birthday = null;
String s = birthday.toString(); // Ошибка!
```

Локальные переменные не инициализируются автоматически значением null. Программист должен сам инициализировать переменную, либо вызвав операцию new, либо присвоив значение null.



На заметку! Многие ошибочно полагают, что объектные переменные в языке Java похожи на ссылки C++. Однако в языке C++ такая переменная не может иметь значение null. Кроме того, им ничего нельзя присваивать. Объектные переменные в языке Java следует считать аналогами *указателей на объекты*. Например, выражение

```
Date birthday;    // Java
```

почти эквивалентно приведенной ниже строке кода.

```
Date* birthday;   // C++
```

Такая аналогия все расставляет по своим местам. Разумеется, указатель Date* не инициализируется, пока не выполнена операция new. Синтаксис выражений в языках C++ и Java почти совпадает.

```
Date* birthday = new Date(); // C++
```

При копировании одной переменной в другую в обеих переменных появляется ссылка на один и тот же объект. Эквивалентом ссылки null в языке C++ является указатель NULL.

Все объекты в языке Java располагаются в динамической памяти. Если объект содержит другую объектную переменную, она представляет собой всего лишь указатель на другой объект, расположенный в области памяти под названием "куча".

В языке C++ указатели доставляют много хлопот, так как из-за них часто возникают ошибки. Очень легко создать неверный указатель или потерять управление памятью. В языке Java эти проблемы попросту не существуют. Если вы используете неинициализированный указатель, то исполняющая система обязательно сообщит об ошибке, а не продолжит выполнение некорректной программы, выдавая случайные результаты. Программист может не заботиться об управлении памятью, поскольку механизм "сборки мусора" делает это за него автоматически.

В языке C++ большое внимание уделено автоматическому копированию объектов посредством копирующих конструкторов и операций присваивания. Например, копией связного списка является новый связный список, который, имея старое содержимое, содержит совершенно другие связи. Другими словами, копирование объектов осуществляется так же, как и копирование встроенных типов. В языке Java для получения полной копии объекта используется метод `clone`.

Класс `GregorianCalendar` из библиотеки Java

В предыдущих примерах мы использовали класс `Date`, являющийся частью стандартной библиотеки Java. Экземпляр класса `Date` находится в состоянии, которое отражает *конкретный момент времени*.

Хотя при использовании класса `Date` нам необязательно знать о формате даты, отметим, что время представляется количеством миллисекунд (положительным или отрицательным), отсчитанным от фиксированного момента времени, так называемого *начала эпохи*, т.е. от 00:00:00 UTC, 1 января 1970 г. Аббревиатура UTC означает Universal Coordinated Time (универсальное скоординированное время) — научный стандарт времени. UTC применяется наряду с более известным GMT (Greenwich Mean Time — среднее время по Гринвичу).

Однако класс `Date` не очень удобен для работы с датами. Проектировщики библиотеки Java считали, что представление даты, например "December 31, 1999, 23:59:59", является совершенно произвольным и должно зависеть от *календаря*. Данное конкретное представление подчиняется григорианскому календарю, самому распространенному календарю в мире. Однако тот же самый момент времени совершенно иначе представляется в китайском или еврейском лунном календаре, не говоря уже о календаре, которым будут пользоваться потенциальные заказчики с Марса.



На заметку! Вся история развития человечества сопровождалась созданием календарей — систем именования различных моментов времени. Как правило, основой для календарей был солнечный или лунный цикл. Если вас интересуют подобные вопросы, обратитесь к книге Наума Дершовица (Nachum Dershowitz) и Эдварда М. Рейнголда (Edward M. Reingold) *Calendrical Calculations* (Cambridge University Press, 2nd ed., 2001). Там вы найдете сведения о календаре французской революции, календаре Майя и других экзотических системах.

Проектировщики библиотеки решили отделить вопросы, связанные с отслеживанием моментов времени, от вопросов их представления. Таким образом, стандартная библиотека Java содержит два отдельных класса: класс `Date`, представляющий момент времени, и класс `GregorianCalendar`, расширяющий более общий класс `Calendar`, описывающий свойства календаря в целом. Теоретически можно расширить класс

Calendar и реализовать китайский лунный или марсианский календари. Однако в стандартной библиотеке, кроме григорианского календаря, пока нет никакой другой реализации.

Отделение измерения времени от календарей представляет собой хорошее решение, вполне соответствующее принципу объектно-ориентированного программирования.

Класс Date содержит лишь небольшое количество методов, позволяющих сравнивать два момента времени. Например, методы before и after определяют, предшествует один момент времени другому или следует за ним.

```
if (today.before(birthday))
   System.out.println("Еще есть время купить подарок.");
```



На заметку! На самом деле класс Date имеет такие методы, как getDay(), getMonth() и getYear(), но использовать их без крайней необходимости *не рекомендуется*. Метод объявляется не рекомендованным к применению (deprecated), когда проектировщики библиотеки решают, что данный метод более не стоит применять в новых программах.

Эти методы были частью класса Date еще до того, как проектировщики библиотеки поняли, что классы, реализующие разные календари, разумнее было бы отделить друг от друга. Сделав это, они пометили методы класса Date как не рекомендованные к применению. Вы можете продолжать использовать их в своих программах, получая при этом предупреждения от компилятора, а еще лучше их не применять совсем, поскольку в будущем они могут быть удалены из библиотеки.

Класс GregorianCalendar содержит гораздо больше методов, чем класс Date. В частности, в нем есть несколько полезных конструкторов. Так, приведенное ниже выражение формирует новый объект, представляющий дату и момент времени, когда он был создан:

```
new GregorianCalendar()
```

Вы можете создать объект, представляющий полночь даты, указанной как год, месяц и день:

```
new GregorianCalendar(1999, 11, 31)
```

Довольно странно, что количество месяцев отсчитывается от нуля. Так, число 11 означает декабрь. Для большей ясности можно использовать константу Calendar.DECEMBER:

```
new GregorianCalendar(1999, Calendar.DECEMBER, 31)
```

Кроме того, посредством конструктора GregorianCalendar можно задать время:

```
new GregorianCalendar(1999, Calendar.DECEMBER, 31, 23, 59, 59)
```

Чаще всего ссылка на созданный объект присваивается переменной:

```
GregorianCalendar deadline = new GregorianCalendar(...);
```

В классе GregorianCalendar инкапсулированы поля экземпляра, в которых записана указанная дата. Не имея доступа к исходному тексту, невозможно определить, какое представление даты и времени использует этот класс. Разумеется, для программиста, применяющего готовый класс, это совершенно неважно. Важно лишь то, какие методы доступны ему.

Модифицирующие методы и методы доступа

После того как вы познакомились с классом GregorianCalendar, у вас, возможно, возникли вопросы: как получить текущий день, месяц или год на базе даты, инкапсулированной в объекте данного класса? Каким образом можно изменить эти значения? Ответы на эти вопросы вы найдете в данном разделе, а дополнительную информацию — в документации по API. Рассмотрим наиболее важные методы.

Календарь должен вычислять атрибуты, соответствующие указанному моменту времени, например, дату, день недели, месяц или год. Получить одно из этих значений можно, используя метод get() класса GregorianCalendar. Чтобы выбрать желаемый атрибут, нужно передать методу константу, определенную в классе Calendar, например Calendar.MONTH или Calendar.DAY_OF_WEEK:

```
GregorianCalendar now = new GregorianCalendar();
int month = now.get(Calendar.MONTH);
int weekday = now.get(Calendar.DAY_OF_WEEK);
```

В конце данного раздела во врезке API перечислены все константы, которые можно использовать для получения требуемой информации из объекта GregorianCalendar.

Состояние объекта можно изменить с помощью метода set():

```
deadline.set(Calendar.YEAR, 2001);
deadline.set(Calendar.MONTH, Calendar.APRIL);
deadline.set(Calendar.DAY_OF_WEEK, 15);
```

Существует способ установить год, месяц и день с помощью одного вызова:

```
deadline.set(2001, Calendar.APRIL, 15);
```

При желании вы можете добавить к заданной дате определенное количество дней, недель, месяцев и т.д.

```
deadline.add(Calendar.MONTH, 3); // Сдвинуть момент deadline на 3 месяца.
```

Добавив отрицательное число, вы сдвинете момент времени, описываемый объектом, назад.

Между методом get(), с одной стороны, и методами set() и add(), с другой, есть принципиальная разница. Метод get() только просматривает состояние объекта и сообщает о нем, в то время как методы set() и add() модифицируют состояние объекта. Методы, которые могут изменять поля экземпляра, называются *модифицирующими* (mutator), а методы, которые могут лишь просматривать поля экземпляра, не изменяя их, называются *методами доступа* (accessor).



На заметку! В языке C++ для формирования метода доступа используется суффикс const. Метод, не объявленный с помощью ключевого слова const, считается модифицирующим. Однако в языке Java нет специальных синтаксических средств, позволяющих различать модифицирующие методы и методы доступа.

В именах методов доступа принято использовать префикс get(), а в именах модифицирующих методов — префикс set(). Например, класс GregorianCalendar обеспечивает получение и установку момента времени, представленного объектом, посредством методов getTime() и setTime().

```
Date time = calendar.getTime();
calendar.setTime(time);
```

Эти методы очень полезны для преобразований объектов GregorianCalendar в объекты Date и наоборот. Предположим, что нам известны год, месяц и день и мы хотим создать объект Date, поля которого были бы заполнены этими значениями. Поскольку класс Date ничего не знает о календарях, создадим сначала объект GregorianCalendar, а затем вызовем метод getTime() для получения даты.

```
GregorianCalendar calendar = new GregorianCalendar(year, month, day);
Date hireDay = calendar.getTime();
```

И наоборот, если нужно определить год, месяц или день, записанные в объекте Date, создадим объект GregorianCalendar, установим время, а затем вызовем метод get().

```
GregorianCalendar calendar = new GregorianCalendar();
calendar.setTime(hireDay);
int year = calendar.get(Calendar.YEAR);
```

В конце этого раздела приведена программа, иллюстрирующая работу класса GregorianCalendar. Программа выводит на экран календарь текущего месяца в следующем формате:

```
Sun Mon Tue Wed Thu Fri Sat
                        1
 2   3   4   5   6   7   8
 9  10  11  12  13  14  15
16  17  18  19* 20  21  22
23  24  25  26  27  28  29
30  31
```

Текущий день помечен звездочкой. Как видим, программа должна знать, как вычисляется длина месяца и текущий день недели.

Рассмотрим ключевые шаги, связанные с работой этой программы. Сначала создадим объект GregorianCalendar, инициализированный текущими датой.

```
GregorianCalendar d = new GregorianCalendar();
```

Затем определим текущий день и месяц, дважды вызвав метод get().

```
int today = d.get(Calendar.DAY_OF_MONTH);
int month = d.get(Calendar.MONTH);
```

После этого передадим методу set() объекта d параметр, задающий первое число месяца, и получим день недели, соответствующий этой дате.

```
d.set(Calendar.DAY_OF_MONTH, 1);
int weekday = d.get(Calendar.DAY_OF_WEEK);
```

Переменная weekday будет содержать значение Calendar.SUNDAY, если первый день месяца — воскресенье, Calendar.MONDAY — если понедельник, и т.д. (Эти значения на самом деле являются целыми числами 1, 2, ..., 7, но лучше не писать код, основываясь на знании этого.)

Обратите внимание, что первая строка календаря выведена с отступом, так что первый день месяца выпадает на соответствующий день недели. Это не совсем просто, поскольку существуют разные соглашения относительно первого дня недели. В Соединенных Штатах неделя начинается с воскресенья и заканчивается субботой, в то время как в Европе она начинается с понедельника и оканчивается воскресеньем.

Виртуальная машина Java знает о *локали* (locale) текущего пользователя. Локаль описывает местные соглашения относительно форматирования, включая начало недели и имена дней недели.



Совет. Если вы хотите видеть вывод программы в другой локали, добавьте в начало метода main строку, подобную следующей:

```
Locale.setDefault(Locale.ITALY);.
```

Метод `getFirstDayOfWeek()` получает начальный день недели в текущей локали. Чтобы определить размер необходимого отступа, мы отнимаем 1 от дня, описанного объектом календаря, до тех пор, пока не достигнем первого дня недели.

```
int firstDayOfWeek = d.getFirstDayOfWeek();
int indent = 0;
while (weekday != firstDayOfWeek)
{
   indent++;
   d.add(Calendar.DAY_OF_MONTH, -1);
   weekday = d.get(Calendar.DAY_OF_WEEK);
}
```

Затем мы печатаем заголовок с наименованиями дней недели. Их мы получаем от класса `DateFormatSymbols`.

```
String [] weekdayNames = new DateFormatSymbols().getShortWeekdays();
```

Метод `getShortWeekdays()` возвращает строку сокращенных наименований дней недели на языке пользователя (на английском — `"Sun"`, `"Mon"` и т.д.). Массив проиндексирован значениями дней недели. Так будет выглядеть цикл, печатающий заголовок:

```
do
{
   System.out.printf("%4s", weekdayNames[weekday]);
   d.add(Calendar.DAY_OF_MONTH, 1);
   weekday = d.get(Calendar.DAY_OF_WEEK);
}
while (weekday != firstDayOfWeek);
System.out.println();
```

Теперь мы готовы печатать само тело календаря. Выравниваем первую строку и устанавливаем объект даты в начало месяца. Входим в цикл, где `d` проходит по всем дням месяца.

На каждой итерации печатаем значение даты. Если `d` приходится на сегодня, помечаем этот день символом `*`. Когда в цикле достигаем начала новой недели, выводим перевод строки.

Затем установим объект `d` на следующий день:

```
d.add(Calendar.DAY_OF_MONTH, 1);
```

Когда нам следует остановиться? Нам неизвестно, сколько в месяце дней: 31, 30, 29 или 28. Мы продолжаем итерации, пока объект `d` остается в пределах текущего месяца.

```
do
{
   ...
}
while (d.get(Calendar.MONTH) == month);
```

Как только объект `d` перейдет на следующий месяц, программа завершит свою работу.

Полный текст программы приведен в листинге 4.1.

Как видим, класс `GregorianCalendar` позволяет легко создавать программы для работы с календарем, выполняя такие сложные действия, как отслеживание дней недели и учет продолжительности месяцев. Программист не обязан ничего знать о том, *как именно* класс `GregorianCalendar` вычисляет месяцы и дни недели. Нужно просто использовать *интерфейс* класса — методы `get()`, `set()` и `add()`.

Основное назначение этой программы — показать, как можно использовать интерфейс класса для выполнения действительно сложных задач, не вникая в детали реализации.

Листинг 4.1. Содержимое файла `CalendarTest.java`

```
1. import java.text.DateFormatSymbols;
2. import java.util.*;
3.
4. /**
5. * @version 1.4 2007-04-07
6. * @author Cay Horstmann
7. */
8.
9. public class CalendarTest
10.{
11.   public static void main(String[] args)
12.   {
13.      // Создание объекта d, представляющего текущую дату.
14.      GregorianCalendar d = new GregorianCalendar();
15.
16.      int today = d.get(Calendar.DAY_OF_MONTH);
17.      int month = d.get(Calendar.MONTH);
18.
19.      // Установка объекта d на первое число месяца.
20.      d.set(Calendar.DAY_OF_MONTH, 1);
21.
22.      int weekday = d.get(Calendar.DAY_OF_WEEK);
23.
24.      // Получить первый день недели (воскресенье в США).
25.      int firstDayOfWeek = d.getFirstDayOfWeek();
26.
27.      // Определить необходимый отступ для первой строки.
28.      int indent = 0;
29.      while (weekday != firstDayOfWeek)
30.      {
31.         indent++;
32.         d.add(Calendar.DAY_OF_MONTH, -1);
33.         weekday = d.get(Calendar.DAY_OF_WEEK);
34.      }
35.
36.      // Напечатать названия дней недели.
37.      String[] weekdayNames = new DateFormatSymbols().getShortWeekdays();
38.      do
39.      {
40.        System.out.printf("%4s", weekdayNames[weekday]);
41.        d.add(Calendar.DAY_OF_MONTH, 1);
42.        weekday = d.get(Calendar.DAY_OF_WEEK);
43.      }
```

```
44.      while (weekday != firstDayOfWeek);
45.      System.out.println();
46.
47.      for (int i = 1; i <= indent; i++)
48.      System.out.print(" ");
49.
50.      d.set(Calendar.DAY_OF_MONTH, 1);
51.      do
52.      {
53.         // Вывод дня месяца.
54.         int day = d.get(Calendar.DAY_OF_MONTH);
55.         System.out.printf("%3d", day);
56.
57.         // Символом * помечается текущий день.
58.         if (day == today)
59.            System.out.print("*");
60.         else
61.            System.out.print(" ");
62.
63.         // Перевод d на следующий день.
64.         d.add(Calendar.DAY_OF_MONTH, 1);
65.         weekday = d.get(Calendar.DAY_OF_WEEK);
66.
67.         // Перевод строки в начале недели.
68.         if (weekday == firstDayOfWeek)
69.         System.out.println();
70.      }
71.      while (d.get(Calendar.MONTH) == month);
72.      // Цикл завершается, когда объект d установлен
73.      // на первый день следующего месяца.
74.
75.      // При необходимости переводится строка.
76.      if (weekday != firstDayOfWeek)
77.         System.out.println();
78.   }
79. }
```

API java.util.GregorianCalendar 1.1

- `GregorianCalendar()`

  Создает объект `GregorianCalendar`, представляющий текущее время для часового пояса и страны, заданных по умолчанию.

- `GregorianCalendar(int year, int month, int date)`
- `GregorianCalendar(int year, int month, int date, int hour, int minutes, int seconds)`

  Создает объект `GregorianCalendar`, представляющий указанную дату и время.

Параметры		
	`year`	Год.
	`month`	Месяц. Отсчет начинается с нуля. Например, 0 — это январь.
	`date`	Число месяца.
	`hour`	Час (от 0 до 23).
	`minutes`	Минуты (от 0 до 59).
	`seconds`	Секунды (от 0 до 59).

- `int get(int field)`

  Возвращает значение заданного поля.

  Параметр `field` Одна из констант: `Calendar.ERA`, `Calendar.YEAR`, `Calendar.MONTH`, `Calendar.WEEK_OF_YEAR`, `Calendar.WEEK_OF_MONTH`, `Calendar.DAY_OF_MONTH`, `Calendar.DAY_OF_YEAR`, `Calendar.DAY_OF_WEEK`, `Calendar.DAY_OF_WEEK_IN_MONTH`, `Calendar.AM_PM`, `Calendar.HOUR`, `Calendar.HOUR_OF_DAY`, `Calendar.MINUTE`, `Calendar.SECOND`, `Calendar.MILLISECOND`, `Calendar.ZONE_OFFSET` или `Calendar.DST_OFFSET`

- `void set(int field, int value)`

  Устанавливает значения заданных полей.

  Параметры `field` Одна из констант, допустимых для метода `get`.

  `value` Новое значение.

- `void set(int year, int month, int day)`

  Устанавливает новую дату в объекте.

  Параметры `year` Год.

  `month` Месяц. Отсчет начинается с нуля. Например, 0 — это январь.

  `day` Число месяца.

- ```
  void set(int year, int month, int day, int hour,
           int minutes, int seconds)
  ```

  Устанавливает новые значения полей.

  Параметры `year` Год.

  `month` Месяц. Отсчет начинается с нуля. Например, 0 — это январь.

  `day` Число месяца.

  `hour` Час (от 0 до 23).

  `minutes` Минуты (от 0 до 59).

  `seconds` Секунды (от 0 до 59).

- `void add(int field, int amount)`

  Арифметический метод для работы с датой и временем. Добавляет к полям заданную величину. Например, чтобы прибавить 7 дней к текущей дате, нужно вызвать `c.add(Calendar.DAY_OF_MONTH, 7)`.

  Параметры `field` Изменяемое поле (должно содержать значение одной из констант, предусмотренных для метода `get`).

  `amount` Добавляемое время (может быть отрицательным числом).

- `int getFirstDayOfWeek()`

  Получает первый день недели в локали текущего пользователя. Например, для США это будет `Calendar.SUNDAY`.
  ```

- `void setTime(Date time)`

  Устанавливает заданный момент времени.

  Параметр `time` Момент времени.

- `Date getTime()`

  Возвращает момент времени, представленный текущим значением объекта.

API **java.util.DateFormatSymbols 1.1**

- `String getShortWeekDays()`
- `String getShortMonths()`
- `String getWeekDays()`
- `String getMonths()`

  Получают наименования дней недели и месяцев в текущей локали. Используют константы дней недели и месяцев `Calendar` как индексы массива.

Определение собственных классов

В главе 3 мы уже попробовали создавать простые классы. Однако все они состояли из одного-единственного метода `main()`. Теперь пришло время научиться создавать "рабочие" классы для более сложных приложений. Обычно в этих классах нет метода `main()`. Вместо этого у них есть другие методы и поля. Чтобы написать полностью законченную программу, нужно объединить несколько классов, один из которых содержит метод `main()`.

Класс `Employee`

Простейшее определение класса в языке Java имеет следующий вид:

```
class ИмяКласса
{
   конструктор_1
   конструктор_2
   ...
   метод_1
   метод_2
   ...
   поле_1
   поле_2
   ...
}
```



На заметку! Мы придерживаемся стиля, согласно которому сначала указываются методы класса, а в конце — его поля. Это сделано для того, чтобы привлечь основное внимание к интерфейсу, а не к реализации класса.

Рассмотрим следующую, весьма упрощенную версию класса `Employee`, который можно использовать для создания платежной ведомости.

```
class Employee
{
   // Конструктор
```

```
   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   // Метод 1
   public String getName()
   {
      return name;
   }

   // Другие методы
   ...
   // Поля экземпляра

   private String name;
   private double salary;
   private Date hireDay;
}
```

Детали реализации этого класса мы проанализируем в следующих разделах. А сейчас рассмотрим код, представленный в листинге 4.2 и иллюстрирующий работу класса `Employee`.

В данной программе создается массив `Employee`, в который заносятся три объекта `Employee`.

```
Employee[] staff = new Employee[3];

staff[0] = new Employee("Carl Cracker", . . .);
staff[1] = new Employee("Harry Hacker", . . .);
staff[2] = new Employee("Tony Tester", . . .);
```

Затем, для того чтобы поднять зарплату каждого сотрудника на 5%, вызывается метод `raiseSalary()` класса `Employee`.

```
for (Employee e : staff)
  e.raiseSalary(5);
```

В заключение с помощью методов `getName()`, `getSalary()` и `getHireDay()` выводится информация о каждом сотруднике.

```
for (Employee e : staff)
  System.out.println("name=" + e.getName()
    + ", зарплата=" + e.getSalary()
    + ", дата найма на работу=" + e.getHireDay());
```

Отметим, что этот пример программы состоит из *двух* классов: `Employee` и `EmployeeTest`, объявленного со спецификатором доступа `public`. Метод `main()` с выражениями, которые мы только что описали, содержится в классе `EmployeeTest`.

Исходный текст программы содержится в файле `EmployeeTest.java`, поскольку его имя должно совпадать с именем общедоступного класса. В исходном файле может быть только один класс, объявленный как `public`, и любое количество классов, в объявлении которых данное ключевое слово отсутствует.

Далее при компиляции исходного кода компилятор создает два файла классов: `EmployeeTest.class` и `Employee.class`.

Выполнение программы начинается, когда интерпретатор получает имя класса, содержащего метод `main()`.

```
java EmployeeTest
```

При этом интерпретатор начинает обрабатывать метод `main()` из класса `EmployeeTest`. В результате выполнения кода создаются три новых объекта `Employee` и отображается их состояние.

Листинг 4.2. Содержимое файла `EmployeeTest.java`

```
1. import java.util.*;
2.
3. /**
4.    * Эта программа тестирует класс Employee.
5.    * @version 1.11 2004-02-19
6.    * @author Cay Horstmann
7.    */
8. public class EmployeeTest
9. {
10.   public static void main(String[] args)
11.   {
12.      // Заполнение массива staff тремя объектами.
13.      Employee[] staff = new Employee[3];
14.
15.      staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
16.      staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
17.      staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
18.
19.      // Увеличение зарплаты на 5%
20.      for (Employee e : staff)
21.         e.raiseSalary(5);
22.
23.      // Вывод информации о всех объектах Employee
24.      for (Employee e : staff)
25.         System.out.println("name=" + e.getName()
26.            + ",salary=" + e.getSalary()
27.            + ",hireDay=" + e.getHireDay());
28.   }
29.}
30.
31.class Employee
32.{
33.   public Employee(String n, double s, int year, int month, int day)
34.   {
35.      name = n;
36.      salary = s;
37.      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
38.      // В классе GregorianCalendar январю соответствует 0.
39.      hireDay = calendar.getTime();
40.   }
41.
42.   public String getName()
43.   {
44.      return name;
45.   }
46.
```

```
47.   public double getSalary()
48.   {
49.      return salary;
50.   }
51.
52.   public Date getHireDay()
53.   {
54.      return hireDay;
55.   }
56.
57.   public void raiseSalary(double byPercent)
58.   {
59.      double raise = salary * byPercent / 100;
60.      salary += raise;
61.   }
62.
63.   private String name;
64.   private double salary;
65.   private Date hireDay;
66. }
```

Использование нескольких исходных файлов

Программа, приведенная в листинге 4.2, содержит два класса в одном исходном файле. Многие программисты предпочитают помещать каждый класс в отдельный файл. Например, класс `Employee` можно разместить в файле `Employee.java`, а класс `EmployeeTest` — в `EmployeeTest.java`.

Существуют различные способы компиляции программы, код которой содержится в двух исходных файлах. Вы можете использовать шаблонный символ при вызове компилятора, например:

```
javac Employee*.java
```

В результате все исходные файлы, имена которых соответствуют указанному вами шаблону, будут скомпилированы в файлы классов. Можно также ограничиться приведенной ниже командой:

```
javac EmployeeTest.java
```

Как ни странно, файл `Employee.java` также будет скомпилирован. Обнаружив, что в файле `EmployeeTest.java` используется класс `Employee`, компилятор языка Java станет искать файл `Employee.class`. Если он его не найдет, то автоматически будет скомпилирован файл `Employee.java`. Более того, если файл `Employee.java` создан позже, чем существующий файл `Employee.class`, компилятор языка Java также *автоматически* выполнит повторную компиляцию и создаст файл класса.



На заметку! Если вы знакомы с утилитой `make`, имеющейся в Unix и других операционных системах, то поведение компилятора не вызовет вопросов. Дело в том, что в компиляторе языка Java реализованы возможности этой утилиты.

Анализ класса Employee

Проанализируем класс Employee. Начнем с методов этого класса. Изучая исходный текст программы, легко увидеть, что в классе Employee реализованы один конструктор и четыре метода.

```
public Employee(String n, double s,int year, int month, int day)
public String getName()
public double getSalary()
public Date getHireDay()
public void raiseSalary(double byPercent)
```

Все методы этого класса объявлены как public, т.е. обращение к этим методам может осуществляться из любого класса. (Существуют четыре возможных уровня доступа; они рассматриваются в этой и следующей главах.)

Далее, отметим, что в классе есть три поля экземпляра для хранения данных, обрабатываемых внутри объекта Employee.

```
private String name;
private double salary;
private Date hireDay;
```

Ключевое слово private означает, что к данным полям имеют доступ только методы самого класса Employee. Ни один внешний метод не может читать эти поля или изменять их.



На заметку! Поля экземпляра могут быть объявлены как public, однако делать этого не следует. Любые компоненты программы могут обращаться к общедоступным полям и модифицировать их содержимое. Любой метод любого класса может модифицировать общедоступные поля, и по нашему опыту, некоторый код обычно непременно воспользуется этой привилегией доступа, когда вы менее всего этого ожидаете. Мы настоятельно рекомендуем всегда закрывать доступ к полям экземпляра с помощью ключевого слова private.

В заключение отметим, что два из трех полей экземпляра представляют собой объекты: поля name и hireDay являются ссылками на экземпляры классов String и Date. Это довольно распространенное явление: классы часто содержат экземпляры других классов.

Первые шаги с конструкторами

Рассмотрим конструктор класса Employee.

```
public Employee(String n, double s, int year, int month, int day)
{
   name = n;
   salary = s;
   GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
   hireDay = calendar.getTime();
}
```

Как видите, имя конструктора совпадает с именем класса. Этот конструктор выполняется при создании объекта Employee, заполняя поля экземпляра заданными значениями.

Например, при создании экземпляра класса Employee с помощью оператора:

```
new Employee("James Bond", 100000, 1950, 1, 1);
```

поля экземпляра будут заполнены такими значениями:

```
name = "James Bond";
salary = 100000;
hireDay = January 1, 1950;
```

Между конструкторами и другими методами есть существенная разница: конструктор можно вызывать только в сочетании с операцией `new`. Конструктор нельзя применить к существующему объекту, чтобы изменить информацию в его полях. Например, вызов:

```
james.Employee("James Bond", 250000, 1950, 1, 1); // ОШИБКА
```

при компиляции приведет к ошибке.

Позднее в этой главе мы вернемся к конструкторам. Пока запомните следующее.

- Имя конструктора совпадает с именем класса.
- Класс может иметь несколько конструкторов.
- Конструктор может иметь ноль, один или более параметров.
- Конструктор не возвращает никакого значения.
- Конструктор всегда вызывается совместно с операцией `new`.



На заметку! Конструкторы в языках Java и C++ работают одинаково. Однако учтите, что все объекты в языке Java размещаются в динамической памяти и конструкторы вызываются только вместе с операцией `new`. Программисты, имеющие опыт работы на языке C++, часто допускают такую ошибку:

```
Employee number007("James Bond", 10000, 1950, 1, 1);
  // C++, а не Java
```

Это выражение в C++ работает, а в Java — нет.



Внимание! Будьте осторожны и не называйте локальные переменные так же, как и поля экземпляра. Например, приведенный ниже конструктор не сможет установить зарплату сотрудника.

```
public Employee(String n, double s, ...)
  {
       String name = n;          // ОШИБКА
       double salary = s;        // ОШИБКА
  }
```

В конструкторе объявляются *локальные* переменные `name` и `salary`. Доступ к этим переменным возможен только внутри конструктора. Они *маскируют* поля экземпляра с теми же именами. Некоторые программисты — например, авторы этой книги — могут написать подобный код чисто автоматически. Подобные ошибки крайне трудно обнаружить. Нужно быть внимательными и не называть переменные именами полей.

Явные и неявные параметры

Методы объекта имеют доступ ко всем его полям. Рассмотрим следующий метод:

```
public void raiseSalary(double byPercent)
{
  double raise = salary * byPercent / 100;
  salary += raise;
}
```

Он устанавливает новое значение поля salary в объекте, с которым этот метод вызывается. Например, вызов метода

```
number007.raiseSalary(5);
```

увеличивает значение поля number007.salary в объекте number007 на 5%. Строго говоря, вызов данного метода приводит к выполнению двух операторов:

```
double raise = number007.salary * 5 /100;
number007.salary += raise;
```

Метод raiseSalary имеет два параметра. Первый параметр, называемый *неявным*, представляет собой объект типа Employee, который указывается перед именем метода. Второй параметр — число, указанное в скобках после имени метода, — называется *явным*.

Легко видеть, что явные параметры перечисляются в объявлении метода, например double byPercent. Неявный параметр в объявлении метода не приводится.

В каждом методе ключевое слово this ссылается на неявный параметр. При желании можно переписать метод raiseSalary следующим образом:

```
public void raiseSalary(double byPercent)
{
   double raise = this.salary * byPercent / 100;
   this.salary += raise;
}
```

Некоторые программисты предпочитают такой стиль, поскольку в нем более четко различаются поля экземпляра и локальные переменные.



На заметку! В языке C++ методы обычно определяются вне класса. Например:

```
void Employee::raiseSalary(double byPercent) // C++, а не Java
{
    ...
}
```

Если определить метод внутри класса, он автоматически станет встроенным (inline).

```
class Employee
{
 ...
 int getName() { return name; } // Встраиваемая функция в C++
}
```

В языке Java все методы определяются внутри класса. Это не делает их встраиваемыми. Виртуальная машина Java анализирует, как часто производится обращение к методу, и принимает решение о том, должен ли метод быть встраиваемым. JIT-компилятор находит краткие, частые вызовы методов, которые не являются переопределенными, и оптимизирует их соответствующим образом.

Преимущества инкапсуляции

Рассмотрим довольно простые методы getName(), getSalary() и getHireDay().

```
public String getName()
{
   return name;
}

public double getSalary()
{
  return salary;
}

public Date getHireDay()
{
   return hireDay;
}
```

Они представляют собой яркий пример методов доступа (accessor). Поскольку они лишь возвращают значения полей экземпляра, иногда их называют *методами доступа к полю* (field accessor).

Но не проще ли было сделать поля `name`, `salary` и `hireDay` общедоступными (т.е. объявить их как `public`) и не создавать отдельные методы доступа к ним?

Дело в том, что поле `name` доступно лишь для чтения. После того как значение этого поля будет установлено конструктором, ни один метод не сможет его изменить. Следовательно, у нас есть гарантия, что информация, хранящаяся в этом поле, не будет искажена.

Поле `salary` допускает запись, однако изменить его значение может только метод `raiseSalary()`. В частности, если окажется, что поле содержит неверное значение, нужно будет отладить только один метод. Если бы поле `salary` было общедоступным, причина ошибки могла бы находиться где угодно.

Иногда нужно иметь возможность читать и модифицировать содержимое поля. Для этого необходимо реализовать в составе класса следующие три элемента.

- Приватное (`private`) поле данных.
- Общедоступный (`public`) метод доступа.
- Общедоступный модифицирующий метод.

Это намного утомительнее, чем просто сделать общедоступным одно-единственное поле данных, однако при этом программист получает существенные преимущества.

Во-первых, внутреннюю реализацию класса можно изменять совершенно независимо от других классов.

Предположим, например, что имя и фамилия сотрудника хранятся отдельно:

```
String firstName;
String lastName;
```

Тогда метод `getName` должен формировать возвращаемое значение следующим образом:

```
firstName + " " + lastName
```

Остальные части программы остаются неизменными. Разумеется, и методы доступа, и модифицирующие методы должны быть переработаны, чтобы учесть новое представление данных. Но это дает нам еще одно преимущество. Модифицирующие методы могут выполнять проверку ошибок, в то время как при присвоении полю некоторого значения ошибки не выявляются. Например, метод `setSalary()` может проверить, не стала ли зарплата отрицательной величиной.



Внимание! Будьте осторожны при создании методов доступа, возвращающих ссылки на изменяемый объект. Создавая класс `Emloyee`, мы нарушили это правило: метод `getHireDay()` возвращает объект `Date`.

```
class Employee
{
    ...
    public Date getHire();
    {
         return hireDay;
    }
    ...
    private Date hireDay;
}
```

Это нарушает принцип инкапсуляции! Рассмотрим пример неверного кода:

```
Employee harry = ...;
Date d = harry.getHireDay();
double tenYearsInMilliSeconds = 10 * 365.25 * 24 * 60 * 60 * 1000;
d.setTime(d.getTime() - (long) tenYearsInMilliSeconds);
  // Значение объекта изменено.
```

Причину ошибки трудно уловить. Обе ссылки `d` и `harry.hireDay` относятся к одному и тому же объекту (рис. 4.5). Применение модифицирующего метода к объекту `d` автоматически изменяет состояние объекта, содержащегося в классе `Employee`!

Чтобы вернуть ссылку на изменяемый объект, его нужно сначала *клонировать*. Клон — это точная копия объекта, расположенная в другом месте памяти. Детали клонирования рассмотрены в главе 6. Ниже приведен исправленный код.

```
class Employee
{
    ...
    public Date hetHireDay()
    {
         return (Date) hireDay.clone();
    }
    ...
}
```

В качестве эмпирического правила: используйте метод `clone()`, если вам нужно скопировать изменяемое поле данных.

Доступ к данным из различных экземпляров класса

Как вы уже знаете, метод имеет доступ к любым данным объекта, которому он принадлежит. Но он также может обращаться к приватным данным *всех экземпляров своего класса*! Рассмотрим метод `equals`, сравнивающий между собой два экземпляра класса `Employee`.

```
class Employee
{
   ...
   boolean equals(Employee other)
   {
      return name.equals(other.name);
   }
}
```

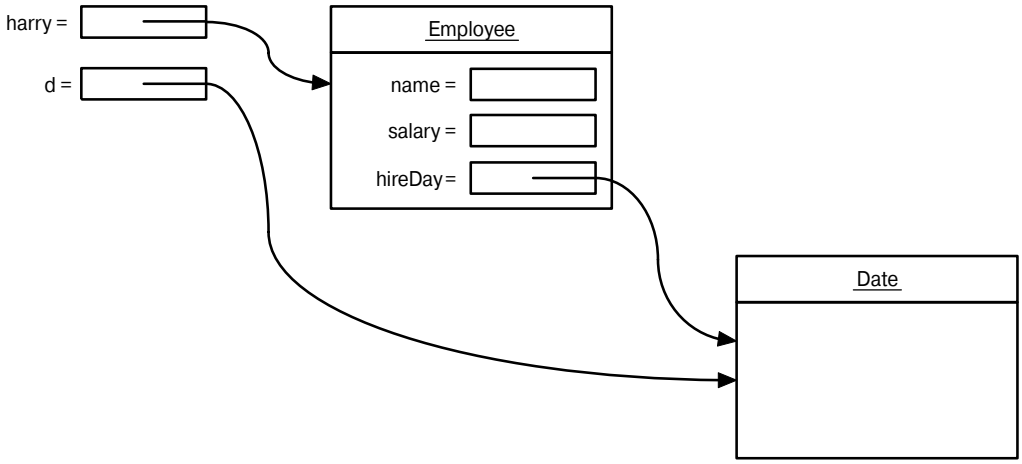


Рис. 4.5. Ссылки на изменяемый объект

Вызов этого метода выглядит следующим образом:

```
if (harry.equals(boss)) ...
```

Этот метод имеет доступ к приватным полям объекта `harry`, что вовсе не удивительно. Но он также имеет доступ к полям объекта `boss`. Это вполне объяснимо, поскольку `boss` — объект `Employee`, а методы, принадлежащие классу `Employee`, могут обращаться к приватным полям любого объекта этого типа.



На заметку! В языке C++ действует такое же правило. Метод имеет доступ к переменным и функциям любого объекта своего класса.

Приватные методы

При реализации класса мы сделали все поля данных приватными, поскольку предоставлять к ним доступ из других классов весьма рискованно. А как поступать с методами? Для взаимодействия с другими объектами необходимы общедоступные методы. Однако в ряде случаев для вычислений необходимы вспомогательные методы. Как правило, эти вспомогательные методы не являются частью интерфейса, поэтому указывать при их объявлении ключевое слово `public` нет необходимости; чаще всего они объявляются как `private`.

Чтобы сделать метод приватным, измените ключевое слово `public` на `private`.

Сделав метод приватным, вы не обязаны его сохранять при переходе к другой реализации. При этом метод может быть труднее реализовать, или необходимость в нем отпадет, если изменится представление данных; это не существенно. Суть в том, что до тех пор, пока метод является приватным (`private`), проектировщики класса могут быть уверены в том, что он никогда не будет использован вне операций класса, и могут просто удалить его. Если же метод общедоступный (`public`), вы не можете просто так отбросить его, поскольку другой код может от него зависеть.

Неизменяемые поля экземпляра

Поля экземпляра можно объявить с помощью ключевого слова `final`. Такое поле должно инициализироваться при создании объекта. То есть следует гарантировать, что значение поля будет установлено после конца каждого конструктора. После этого его значение изменить уже нельзя. Например, поле `name` класса `Employee` можно объявить неизменяемым, поскольку после создания объекта оно никогда не изменяется — метода `setName` не существует.

```
class Employee
{
   ...
   private final String name;
}
```

Модификатор `final` удобно применять при объявлении полей простых типов либо полей, типы которых задаются *неизменяемыми классами*. Неизменяемым называется класс, методы которого не позволяют изменить состояние объекта. Так, например, неизменяемым является класс `String`. Если класс допускает изменения, то ключевое слово `final` может стать источником недоразумений. Рассмотрим следующий оператор:

```
private final Date hiredate;
```

Он означает, что переменная `hiredate` не изменяется после создания объекта. Однако это не означает, что состояние объекта, на который ссылается переменная, остается неизменным. В любой момент можно вызвать метод `setTime()`.

Статические поля и методы

Во всех программах, которые мы до сих пор обсуждали, при объявлении метода `main()` использовался модификатор `static`. Рассмотрим действие этого модификатора.

Статические поля

Поле, имеющее модификатор `static`, существует в одном экземпляре для всего класса. В то же время, если поле не статическое, то каждый объект содержит его копию. Предположим, что нам требуется присвоить уникальный идентификационный номер каждому сотруднику. Добавим в класс `Employee` поле `id` и статическое поле `nextId`.

```
class Employee
{
   ...
   private int id;
   private static int nextId = 1;
}
```

Теперь каждый объект `Employee` имеет свое поле `id`, кроме того, есть поле `nextId`, которое одновременно принадлежит всем экземплярам класса.

Попробуем объяснить это несколько иначе. Если существует тысяча объектов `Employee`, то в них есть тысяча полей `id`, по одному в каждом объекте. В то же время существует только один экземпляр статического поля `nextId`. Даже если не создано ни одного объекта `Employee`, статическое поле `nextId` существует. Оно принадлежит классу, а не конкретному объекту.



На заметку! В большинстве объектно-ориентированных языков статические поля называются *полями класса* (class field). Термин “статический” унаследован от языка C++.

Реализуем простой метод.

```
public void setId()
{
   id = nextId;
   nextId++;
}
```

Предположим, нам нужно задать идентификационный номер объекта `harry`.

```
harry.setId();
```

Теперь значение поля `id` объекта `harry` задано, а значение статического поля `nextId` увеличено на единицу.

```
harry.id = Employee.nextId;
Employee.nextId++;
```

Статические константы

Статические переменные используются довольно редко. В то же время статические константы используются гораздо чаще. Например, класс `Math` определяет статическую константу.

```
public class Math
{
   ...
   public static final double PI = 3.14159265358979323846;
   ...
}
```

Обратиться к этой константе в своей программе можно с помощью выражения `Math.PI`.

Если бы ключевое слово `static` было пропущено, константа `PI` была бы обычным полем экземпляра класса `Math`. Это значит, что для доступа к такой константе нужно было бы создать объект `Math`, причем каждый подобной объект имел бы свою копию константы `PI`.

Еще одна часто используемая статическая переменная — `System.out`. Она объявлена в классе `System`.

```
public class System
{
   ...
   public static final PrintStream out = ...;
   ...
}
```

Как мы уже несколько раз упоминали, применять общедоступные поля не следует никогда, поскольку любой объект сможет изменить их значения. Однако общедоступные константы (т.е. поля, объявленные с ключевым словом `final`) можно использовать смело. Поскольку поле `out` объявлено как `final`, ему нельзя присвоить другой поток вывода.

```
System.out = new PrintStream(...); // ОШИБКА -- поле out изменить нельзя.
```



На заметку! Просмотрев код класса `System`, вы увидите метод `setOut()`, позволяющий присвоить полю `System.out` другой поток. Как же этот метод может изменить переменную, описанную как `final`? Дело в том, что метод `setOut` — *платформенно-ориентированный*, он реализован средствами, отличными от Java. Платформенно-ориентированные методы могут обходить механизмы контроля, предусмотренные в языке Java. Это очень специфическое решение, которое ни в коем случае не следует повторять в своих программах.

Статические методы

Статические методы — это методы, которые не оперируют объектами. Например, метод `pow()` из класса `Math` — статический. Выражение `Math.pow(x, y)` вычисляет x^y. При выполнении своей задачи этот метод не использует ни одного экземпляра класса `Math`. Иными словами, он не имеет неявного параметра. Это значит, что статические методы — это методы, в которых не используется объект `this`. (В нестатических методах параметр `this` ссылается на неявный параметр метода; см. раздел "Явные и неявные параметры").

Поскольку статические методы не работают с объектами, из них невозможно получить доступ к полям экземпляра. Однако статические методы имеют доступ к статическим полям класса. Ниже приведен пример статического метода.

```
public static int getNextId()
{
   return nextId; // Возвращает статическое поле
}
```

Чтобы вызвать этот метод, нужно указать имя класса:

```
int n = Employee.getNextId();
```

Можно ли пропустить ключевое слово `static` при описании этого метода? Да, но при этом для его вызова потребуется ссылка на объект типа `Employee`.



На заметку! Для вызова статического метода можно использовать и объекты. Например, если `harry` — это объект `Employee`, то можно вместо `Employee.getNextId()` использовать вызов `harry.getNextId()`. Однако такое обозначение усложняет восприятие программы, поскольку для вычисления результата метод `getNextId()` не обращается к объекту `harry`. Мы рекомендуем для вызова статических методов использовать имена классов, а не объекты.

Статические методы следует применять в двух случаях.

- Когда методу не нужен доступ к информации о состоянии объекта, поскольку все необходимые параметры задаются явно (например, в методе `Math.pow`).
- Когда методу нужен доступ лишь к статическим полям класса (в качестве примера можно привести метод `Employee.getNextId()`).



На заметку! Статические поля и методы в языках Java и C++, по существу, отличаются только синтаксически. В C++ для доступа к статическому полю или методу, находящемуся вне области видимости, можно использовать операцию `::`, например `Math::PI`.

Термин "статический" — исторический курьез. Сначала ключевое слово `static` было введено в языке C для обозначения локальных переменных, которые не уничтожались при выходе из блока. В этом контексте слово "статический" имеет смысл: переменная продолжает существовать после выхода из блока, а также при повторном входе в него. Затем слово "статический" в C приобрело второе значение — глобальные переменные и функции, к которым нельзя получить доступ из других файлов. Ключевое слово `static` было просто использовано повторно, чтобы не вводить новое. В итоге в языке C++ это ключевое слово было применено в третий раз, получив совершенно новую интерпретацию. Оно обозначает переменные и функции, принадлежащие классу, но не принадлежащие ни одному объекту этого класса. Именно этот смысл имеет ключевое слово `static` в языке Java.

Методы-фабрики

Рассмотрим еще одно применение статических методов. Класс `NumberFormat` использует *методы-фабрики* (factory methods) для создания объектов, соответствующих различным стилям форматирования.

```
NumberFormat currencyFormatter = NumberFormat.getCurrencyInstance();
NumberFormat percentFormatter = NumberFormat.getPercentInstance();
double x = 0.1;
System.out.println(currencyFormatter.format(x)); // Выводит $0.10
System.out.println(percentFormatter.format(x));  // Выводит 10%
```

Почему же не использовать для этой цели конструктор? На то есть две причины.

- Конструктору нельзя присвоить произвольное имя. Его имя всегда должно совпадать с именем класса. В примере с классом `NumberFormat` имеет смысл применять разные имена для разных типов форматирования.
- При использовании конструктора тип объекта фиксирован. Если же применяются методы-фабрики, они возвращают объект `DecimalFormat`, который наследует свойства `NumberFormat`. (Подробно вопросы наследования будут обсуждаться в главе 5.)

Метод `main()`

Отметим, что статические методы можно вызывать, не имея ни одного объекта данного класса. Например, для того чтобы вызвать метод `Math.pow()`, объекты `Math` не нужны. По той же причине метод `main()` объявляется как статический.

```
public class Application
{
   public static void main(String[] args)
   {
     // Здесь создаются объекты.
   }
}
```

Метод `main()` не оперирует никакими объектами. Фактически при запуске программы еще и нет никаких объектов. Статический метод `main()` исполняется и конструирует объекты, необходимые программе.



Совет. Каждый класс может содержать метод `main()`. Используя его, можно тестировать классы независимо друг от друга. Например, метод `main()` можно добавить в класс `Employee`.

```
class Employee
{
   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar =
         new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }
   . . .
```

```
   public static void main(String[] args) // Отладочный модуль.
   {
      Employee e = new Employee ("Romeo", 50000, 2003, 3, 31);
      e.raiseSalary(10);
      System.out.println(e.getName() + " " + e.getSalary());
   }
   . . .
}
```

Если вы хотите протестировать только класс `Employee`, выполните следующую команду:

```
java Employee
```

Если класс `Employee` представляет собой часть большого приложения, запустите это приложение с помощью команды

```
java Application
```

и метод `main` класса `Employee` никогда не будет выполнен.

Программа, показанная в листинге 4.3, содержит версию класса `Employee` со статическим полем `nextId` и статическим методом `getNextId()`. Массив заполняется тремя объектами `Employee`, а затем выводится информация о сотрудниках. В заключение на экране отображается очередной доступный идентификационный номер для демонстрации статического метода.

Отметим, что класс `Employee` имеет статический метод `main()` для модульного тестирования. Попробуйте выполнить метод `main()` с помощью приведенных ниже команд:

```
java Employee
```

и

```
java StaticTest
```

Листинг 4.3. Содержимое файла `StaticTest.java`

```
1. /**
2.    * Эта программа демонстрирует статические методы.
3.    * @version 1.01 2004-02-19
4.    * @author Cay Horstmann
5.    */
6. public class StaticTest
7. {
8.    public static void main(String[] args)
9.    {
10.      // Заполнение массива staff объектами Employee.
11.      Employee[] staff = new Employee[3];
12.
13.      staff[0] = new Employee("Tom", 40000);
14.      staff[1] = new Employee("Dick", 60000);
15.      staff[2] = new Employee("Harry", 65000);
16.
17.      // Вывод информации об объектах Employee.
18.      for (Employee e : staff)
19.      {
20.         e.setId();
21.         System.out.println("name=" + e.getName() + ",id=" + e.getId()
22.            + ",salary=" + e.getSalary());
23.      }
```

```
24.
25.      int n = Employee.getNextId(); // Вызов статического метода.
26.      System.out.println("Next available id=" + n);
27.   }
28. }
29.
30. class Employee
31. {
32.   public Employee(String n, double s)
33.   {
34.      name = n;
35.      salary = s;
36.      id = 0;
37.   }
38.
39.   public String getName()
40.   {
41.      return name;
42.   }
43.
44.   public double getSalary()
45.   {
46.      return salary;
47.   }
48.
49.   public int getId()
50.   {
51.      return id;
52.   }
53.
54.   public void setId()
55.   {
56.      id = nextId; // Установка идентификатора в следующий доступный.
57.      nextId++;
58.   }
59.
60.   public static int getNextId()
61.   {
62.      return nextId; // Возврат статического поля.
63.   }
64.
65.   public static void main(String[] args) // Тестирование.
66.   {
67.      Employee e = new Employee("Harry", 50000);
68.      System.out.println(e.getName() + " " + e.getSalary());
69.   }
70.
71.   private String name;
72.   private double salary;
73.   private int id;
74.   private static int nextId = 1;
75. }
```

Параметры методов

Рассмотрим термины, которые используются для описания способа передачи параметров методам (или функциям). Термин *вызов по значению* (call by value) означает, что метод получает значение, переданное ему вызывающим модулем. В противоположность этому *вызов по ссылке* (call by reference) означает, что метод получает от вызывающего модуля *местоположение* (location) переменной. Таким образом, метод может *модифицировать* значение, хранящееся в переменной, переданной по ссылке, но не может изменять значение переменной, переданной по значению. Термин "вызов по..." относится к стандартной компьютерной терминологии, описывающей способ передачи параметров в различных языках программирования, а не только в языке Java. (На самом деле, существует еще и третий способ передачи параметров — *вызов по имени* (call by name), представляющий в основном исторический интерес, поскольку он был применен в языке Algol, одном из наиболее старых языков программирования высокого уровня.)

В языке Java всегда используется только вызов по значению. Это значит, что метод получает копии значений всех параметров. По этой причине метод не может модифицировать содержимое ни одного параметра, переданного ему.

Рассмотрим следующий вызов:

```
double percent = 10;
harry.raiseSalary(percent);
```

Не имеет значения, как именно реализован метод, мы знаем, что после его вызова значение переменной `percent` останется равным `10`.

Изучим эту ситуацию подробнее. Допустим, что метод пытается утроить значение параметра:

```
public static void tripleValue(double x); // Не работает.
{
   x = 3 * x;
}
```

Вызовем этот метод:

```
double percent = 10;
tripleValue(percent);
```

Однако такой способ не работает. После вызова метода значение переменной `percent` остается равным 10. Происходит следующее.

1. Переменная `x` инициализируется копией значения параметра `percent` (т.е. числом 10).
2. Значение переменной `x` утраивается — теперь оно равно 30. Однако значение переменной `percent` остается равным 10 (рис. 4.6).
3. Метод завершает свою работу, и параметр `x` больше не используется.

Однако существуют два типа параметров методов:

- простые типы (числа, логические значения);
- ссылки на объекты.

Как мы убедились, методы не могут модифицировать параметры простых типов. С объектами ситуация иная. Вы можете без труда реализовать метод, утраивающий зарплату сотрудников:

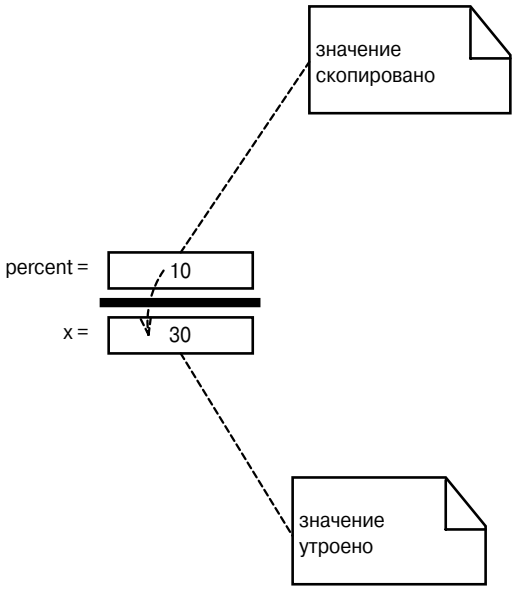


Рис. 4.6. Модификация значения внутри метода не оказывает действия на параметр, передаваемый из вызывающего метода

```
public static void tripleSalary(Employee x) // Работает
{
   x.raiseSalary(200);
}
```

При обработке следующего фрагмента кода будут выполнены перечисленные ниже действия.

```
harry = new Employee(...);
tripleSalary(harry);
```

1. Переменная `x` инициализируется копией значения переменной `harry`, т.е. ссылкой на объект.
2. Метод `raiseSalary()` применяется к этой ссылке на объект. Метод объекта `Employee`, на который указывают ссылки `x` и `harry`, увеличивает зарплату сотрудников на 200%.
3. Метод завершает свою работу, и параметр `x` больше не используется. Разумеется, переменная `harry` продолжает ссылаться на объект, в котором зарплата увеличена втрое (рис. 4.7).

Как видите, реализовать метод, изменяющий состояние объекта, передаваемого как параметр, довольно легко. Фактически такие изменения вносятся очень часто. Причина этого проста — метод получает копию ссылки на объект, поэтому и копия, и оригинал ссылки указывают на один и тот же объект.

Во многих языках программирования (в частности, в C++ и Pascal) есть два способа передачи параметров: вызов по значению и вызов по ссылке. Некоторые программисты (и, к сожалению, даже авторы некоторых книг) утверждают, что язык Java использует при передаче объектов только вызов по ссылке. Однако это не так.

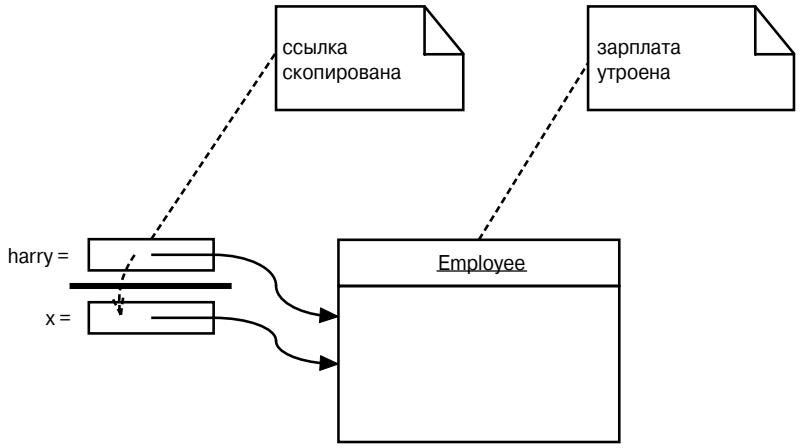


Рис. 4.7. Если параметр ссылается на объект, этот объект может быть модифицирован

Попробуем написать метод, выполняющий обмен двух объектов `Employee`.

```
public static void swap(Employee x, Employee y) // Не работает.
{
   Employee temp = x;
   x = y;
   y = temp;
}
```

Если бы в языке Java для передачи объектов в качестве параметров использовался вызов по ссылке, этот метод работал бы так:

```
Employee a = new Employee("Alice", . . .);
Employee b = new Employee("Bob", . . .);
swap(a, b);
// Ссылается ли теперь a на Bob, а b — на Alice?
```

Однако на самом деле этот метод не меняет местами объектные ссылки, хранящиеся в переменных `a` и `b`. Параметры `x` и `y` метода `swap()` инициализируются *копиями* этих ссылок. Затем метод меняет местами эти копии.

```
// Ссылка x соответствует Alice, а ссылка y — Bob.
Employee temp = x;
x = y;
y = temp;
// Теперь x ссылается на Bob, а y — на Alice.
```

В конце концов, следует признать, что все было напрасно. По завершении работы метода переменные `x` и `y` уничтожаются, а исходные переменные `a` и `b` продолжают ссылаться на прежние объекты (рис. 4.8).

Таким образом, в языке Java для передачи объектов не используется вызов по ссылке. Вместо этого *ссылки на объекты передаются по значению.*

Ниже перечисляется, что может делать метод со своими параметрами, а что — нет.

- Метод не может изменять параметры простых типов (т.е. числа и логические значения).
- Метод может изменять *состояние* объекта, передаваемого как параметр.
- Метод не может изменять ссылки и переназначать их на новые объекты.

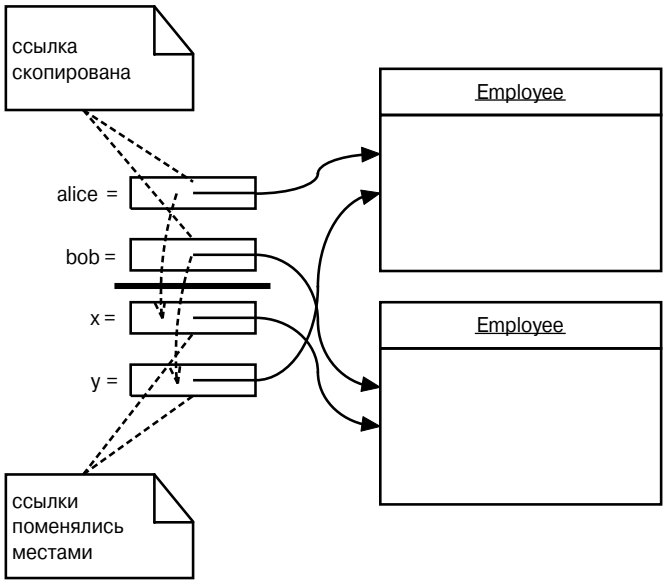


Рис. 4.8. Перемещение ссылок не влияет на значения в вызывающем методе

Программа, приведенная в листинге 4.4, иллюстрирует эти утверждения. Сначала программа безуспешно пытается утроить значение числового параметра.

```
Тестирование tripleValue:
До: percent=10.0
В конце метода: x=30.0
После: percent=10.0
```

Затем программа успешно утраивает зарплату сотрудника.

```
Тестирование tripleSalary:
До: salary=50000.0
В конце метода: salary=150000.0
После: salary=15000.0
```

После выполнения метода состояние объекта, на который ссылается переменная `harry`, изменяется. Ничего невероятного здесь нет, поскольку метод модифицировал состояние объекта с помощью копии ссылки на него.

В заключение программа демонстрирует безрезультатную работу метода `swap()`.

```
Тестирование swap:
До: a=Alice
До: b=Bob
В конце метода: x=Bob
В конце метода: y=Alice
После: a=Alice
После: b=Bob
```

Как видим, переменные `x` и `y` меняются местами, однако переменные `a` и `b` остаются прежними.



На заметку! В C++ применяется как вызов по значению, так и вызов по ссылке. Например, легко можно реализовать методы `void tripleValue(double& x)` или `void swap(Employee& x, Emloyee& y)`. Эти методы модифицируют свои параметры, являющиеся ссылками.

Листинг 4.4. Содержимое файла `ParamTest.java`

```
1. /**
2.    * Эта программа демонстрирует передачу параметров.
3.    * @version 1.00 2000-01-27
4.    * @author Cay Horstmann
5.    */
6. public class ParamTest
7. {
8.    public static void main(String[] args)
9.    {
10.      /*
11.       Тест 1: методы не могут изменять числовые параметры.
12.      */
13.      System.out.println("Тестирование tripleValue:");
14.      double percent = 10;
15.      System.out.println("До: percent=" + percent);
16.      tripleValue(percent);
17.      System.out.println("После: percent=" + percent);
18.
19.      /*
20.         Тест 2: методы могут изменять состояние объектов,
21.         передаваемых в качестве параметров
22.      */
23.      System.out.println("\nТестирование tripleSalary:");
24.      Employee harry = new Employee("Harry", 50000);
25.      System.out.println("До: salary=" + harry.getSalary());
26.      tripleSalary(harry);
27.      System.out.println("После: salary=" + harry.getSalary());
28.
29.      /*
30.         Тест 3: методы не могут приписывать новые объекты
31.         параметрам, являющимся ссылками на объекты.
32.      */
33.      System.out.println("\nТестирование swap:");
34.      Employee a = new Employee("Alice", 70000);
35.      Employee b = new Employee("Bob", 60000);
36.      System.out.println("До: a=" + a.getName());
37.      System.out.println("До: b=" + b.getName());
38.      swap(a, b);
39.      System.out.println("После: a=" + a.getName());
40.      System.out.println("После: b=" + b.getName());
41.   }
42.
43.   public static void tripleValue(double x) // Не работает.
44.   {
45.      x = 3 * x;
46.      System.out.println("В конце метода: x=" + x);
47.   }
48.
```

```
49.   public static void tripleSalary(Employee x) // Работает.
50.   {
51.      x.raiseSalary(200);
52.      System.out.println("В конце метода: salary=" + x.getSalary());
53.   }
54.
55.   public static void swap(Employee x, Employee y)
56.   {
57.      Employee temp = x;
58.      x = y;
59.      y = temp;
60.      System.out.println("В конце метода: x=" + x.getName());
61.      System.out.println("В конце метода: y=" + y.getName());
62.   }
63. }
64.
65. class Employee // Упрощенный класс Employee
66. {
67.   public Employee(String n, double s)
68.   {
69.      name = n;
70.      salary = s;
71.   }
72.
73.   public String getName()
74.   {
75.      return name;
76.   }
77.
78.   public double getSalary()
79.   {
80.      return salary;
81.   }
82.
83.   public void raiseSalary(double byPercent)
84.   {
85.      double raise = salary * byPercent / 100;
86.      salary += raise;
87.   }
88.
89.   private String name;
90.   private double salary;
91. }
```

Конструирование объектов

Мы уже знаем, как можно писать простые конструкторы, определяющие начальные состояния объектов. Однако создание объектов — очень важная задача, поэтому в языке Java предусмотрено много разнообразных механизмов написания конструкторов. Эти механизмы рассматриваются ниже.

Перегрузка

Как вы помните, в классе `GregorianCalendar` предусмотрено несколько конструкторов. Создать объект этого класса позволяет оператор:

```
GregorianCalendar today = new GregorianCalendar();
```

То же можно сделать с помощью следующей строки кода:

```
GregorianCalendar deadline = new GregorianCalendar(2099, Calendar, DECEMBER, 31);
```

Такая возможность называется *перегрузкой* (overloading). О перегрузке говорят в том случае, если несколько методов (в нашем случае несколько конструкторов) имеют одинаковое имя, но разные параметры. Компилятор должен сам определить, какой метод вызвать, сравнивая типы параметров, описанных в заголовках методов, с типами значений, указанных в вызове. Если ни один метод не соответствует вызову либо если одному вызову одновременно соответствует несколько вариантов, возникает ошибка компиляции. (Этот процесс называется *разрешением перегрузки* (overloading resolution).)



На заметку! В языке Java можно перегрузить любой метод, а не только конструкторы. Таким образом, чтобы полностью описать метод, нужно указать его имя и типы параметров. Подобное написание называется *сигнатурой* метода. Например, в классе `String` есть четыре общедоступных метода `indexOf()`. Они имеют следующие сигнатуры:

```
indexOf(int)
indexOf(int, int)
indexOf(String)
indexOf(String, int)
```

Тип возвращаемого методом значения не входит в его сигнатуру. Следовательно, нельзя создать два метода, имеющие одинаковые имена и типы параметров и отличающиеся лишь типом возвращаемого значения.

Инициализация полей по умолчанию

Если значение поля в конструкторе явно не задано, ему автоматически присваивается значение по умолчанию: числам — нули, логическим переменным — значения `false`, а ссылкам на объект — `null`. Однако полагаться на действия по умолчанию считается плохим стилем. Если поля инициализируются неявно, программа становится менее понятной.



На заметку! Между полями и локальными переменными есть существенная разница. Локальные переменные всегда должны явно инициализироваться в методе. Однако если поле в составе класса не инициализируется явно, ему автоматически присваивается значение, задаваемое по умолчанию (`0`, `false` или `null`).

Рассмотрим в качестве примера класс `Employee`. Допустим, что в конструкторе инициализация значений некоторых полей не задана. По умолчанию поле `salary` должно инициализироваться нулем, а поля `name` и `hireDay` — `null`. Однако при вызове методов `getName()` или `getHireDay()` ссылка `null` может оказаться совершенно нежелательной.

```
Date h = harry.getHireDay();
calendar.setTime(h); // Если h = null, генерируется исключение.
```

Конструктор по умолчанию

Конструктор по умолчанию не имеет параметров. Например, конструктор по умолчанию класса `Employee` имеет следующий вид:

```
public Employee()
{
   name = "";
   salary = 0;
   hireDay = new Date();
}
```

Если в классе вовсе не определены конструкторы, то для вас создается конструктор по умолчанию. Этот конструктор присваивает *всем* полям экземпляра их значения, предусмотренные по умолчанию. Так, все числовые значения, содержащиеся в полях экземпляра, окажутся равными *нулю*, логические значения — `false`, а ссылки на объекты — `null`.

Если в классе есть хотя бы один конструктор и явно не определен конструктор без параметров, то при создании объектов необходимо задавать параметры конструктора. Например, класс `Employee` в листинге 4.2 имеет один конструктор:

```
Employee(String name, double salary, int y, int m, int d)
```

В этом случае нельзя создать объект, поля которого принимали бы значения по умолчанию. Другими словами, представленный ниже вызов приведет к возникновению ошибки:

```
e = new Employee();
```



Внимание! Учтите, что конструктор по умолчанию вызывается, *только* если в классе не определены другие конструкторы. Если есть хотя бы один конструктор с параметрами и необходимо создавать экземпляр класса с помощью приведенного ниже выражения, следует явно определить конструктор без параметров.

```
new ИмяКласса()
```

Разумеется, если значения по умолчанию вас устраивают, можно создать следующий конструктор:

```
public ИмяКласса()
{
}
```

Явная инициализация полей

Поскольку конструкторы в классе можно перегружать, есть несколько способов задать начальное состояние полей его экземпляров. Всегда следует быть уверенным, что, независимо от вызова конструктора, все поля экземпляра имеют некие разумные значения.

Имеется возможность присвоить каждому полю соответствующее значение в определении класса. Например:

```
class Employee
{
   ...
   private String name = "";
}
```

Это присваивание выполняется до вызова конструктора. Такой подход особенно полезен тогда, когда надо, чтобы, независимо от вызова конструктора класса, поле имело конкретное значение.

При инициализации поля не обязательно использовать константу. Ниже приведен пример, в котором поле инициализируется с помощью вызова метода.

```
class Employee
{
   static int assignId()
   { int r = nextId;
     nextId++;
     return r;
   }
   ...
   private int id = assignId();
}
```



На заметку! В языке C++ нельзя инициализировать поля экземпляра непосредственно в описании класса. Значения всех полей должны задаваться в конструкторе. Однако в C++ есть синтаксическая конструкция, называемая *списком инициализации* (initializer list).

```
Employee::Employee(String n, double s, int y, int m, int d) // C++
:   name(n),
    salary(s),
    hireDay(y, m, d)
{
}
```

В C++ эта синтаксическая конструкция используется для вызова конструкторов полей экземпляра. В языке Java поступать так нет никакой необходимости, поскольку объекты не могут содержать другие объекты; разрешается иметь только ссылки на них.

Имена параметров

Создавая даже самый тривиальный конструктор (а большинство из них таковыми и являются), трудно выбрать подходящие имена для его параметров.

Обычно в качестве имен параметров используются отдельные буквы:

```
public Employee(String n, double s)
{
   name = n;
   salary = s;
}
```

Однако недостаток такого подхода заключается в том, что, читая программу, невозможно понять, что означают параметры `n` и `s`.

Некоторые программисты добавляют к осмысленным именам параметров префикс `"a"`.

```
public Employee(String aName, double aSalary)
{
   name = aName;
   salary = aSalary
}
```

Такая программа вполне понятна. Любой читатель может сразу определить, в чем заключается смысл параметра. Есть еще один широко распространенный прием. Чтобы использовать его, надо знать, что параметры *маскируют* поля экземпляра с одинаковыми именами. Например, если вызвать метод с параметром `salary`, то эта переменная будет ссылаться на параметр, а не на поле экземпляра. Доступ к полю экземпляра осуществляется с помощью выражения `this.salary`. Напомним, что ключевое слово `this` означает неявный параметр, т.е. создаваемый объект. Следующий пример иллюстрирует данный подход:

```
public Employee(String name, double salary)
{
   this.name = name;
   this.salary = salary;
}
```



На заметку! В языке C++ к именам полей экземпляра обычно добавляют префиксы, представляющие собой либо символ подчеркивания, либо букву (нередко для этой цели используется `m` или `x`). Например, поле, в котором записан размер зарплаты, может называться `_salary`, `mSalary` или `xSalary`. Разработчики, создающие программы на языке Java, как правило, так не поступают.

Вызов одного конструктора из другого

Ключевое слово `this` ссылается на неявный параметр метода. Однако у этого слова есть еще одно значение.

Если первый оператор конструктора имеет вид `this(...)`, то вызывается другой конструктор этого же класса. Ниже приведен типичный пример.

```
public Employee(double s)
{
   // Вызывает конструктор Employee(String, double).
   this("Employee " + nextId, s);
   nextId++;
}
```

Если вы используете выражение `new Employee(60000)`, то конструктор `Employee(double)` вызывает конструктор `Employee(String, double)`.

Применять ключевое слово `this` для вызова другого конструктора очень полезно — при этом достаточно лишь один раз написать общий код, создающий объект.



На заметку! Объект `this` в языке Java идентичен указателю `this` в языке C++. Однако в C++ невозможно вызвать один конструктор с помощью другого. Для того чтобы реализовать общий код инициализации объекта в C++, нужно создать отдельный метод.

Инициализационные блоки

Ранее мы рассмотрели два способа инициализации поля:

- задать его значение в конструкторе;
- присвоить значение при объявлении.

На самом деле в языке Java существует и *третий* механизм — использование *инициализационного блока* (initialization block). Такой блок выполняется каждый раз, когда создается объект данного класса. Рассмотрим следующий пример кода:

```
class Employee
{
   public Employee(String n, double s)
   {
      name = n;
      salary = s;
   }

   public Employee()
   {
      name = "";
      salary = 0;
   }
   ...
   private static int nextId;

   private int id;
   private String name;
   private double salary;
   ...

   // Инициализационный блок
   {
     id = nextId;
     nextId++;
   }
}
```

В этом примере начальное значение поля `id` задается в инициализационном блоке объекта, причем неважно, какой именно конструктор используется для создания экземпляра класса. Инициализационный блок выполняется первым, а тело конструктора — после него.

Этот механизм совершенно не обязателен и обычно не используется. Гораздо чаще применяются более понятные способы задания начальных значений полей.



На заметку! В инициализационном блоке допустимо обращение к полям, определения которых находятся после данного блока. Некоторые версии Java-компилятора разработки Sun считают подобное обращение некорректным. Эта ошибка была зарегистрирована под номером # 4459133 и устранена в Java SE 1.4.1. Несмотря на то что инициализация полей, определяемых после блока, формально допустима, поступать так не рекомендуется во избежание возникновения циклических определений. Конкретные правила изложены в разделе 8.3.2.3 спецификации Java (`http://java.sun.com/docs/books/jls`). Поскольку правила достаточно сложны, и учесть их в реализации компилятора крайне трудно, разработчикам рекомендуется помещать инициализационные блоки после определения полей.

При таком многообразии способов инициализации полей данных довольно трудно отследить все возможные пути процесса конструирования объектов. Поэтому мы рассмотрим, какие действия выполняются при вызове конструктора.

1. Все поля инициализируются значениями, предусмотренными по умолчанию (`0`, `false` или `null`).

2. Инициализаторы всех полей и инициализационные блоки выполняются в порядке их следования в объявлении класса.
3. Если в первой строке конструктора вызывается другой конструктор, то выполняется вызванный конструктор.
4. Выполняется тело конструктора.

Каждый согласится, что код, отвечающий за инициализацию полей, нужно организовать так, чтобы в нем было легко разобраться. Например, было бы странным, если бы вызов конструкторов класса зависел от порядка объявления полей. Такой подход чреват ошибками.

Инициализировать статическое поле следует, либо задавая его начальное значение, либо используя *статический инициализационный блок*. Первый механизм мы уже рассматривали:

```
static int nextId = 1;
```

Если для инициализации статических полей класса нужен сложный код, удобно использовать статический инициализационный блок.

Поместите код внутрь блока и пометьте его ключевым словом `static`. Пусть, например, нам нужно, чтобы идентификационные номера сотрудников начинались со случайного числа, не превышающего 10000.

```
// Статический инициализационный блок.
static
{
   Random generator = new Random();
   nextId = generator.nextInt(10000);
}
```

Статическая инициализация выполняется, когда класс загружается впервые. Аналогично полям экземпляра статические поля принимают значения `0`, `false` или `null`, если программист не задал другие значения явным образом. Все операторы, задающие начальные значения статических полей, и статические блоки инициализации выполняются в порядке их перечисления в объявлении класса.



На заметку! Оказывается, вполне возможно создать программу “Hello, World”, не использующую метод `main()`.

```
public class Hello
{
    static
    {
         System.out.println("Hello, World!");
    }
}
```

При выполнении команды `java Hello` класс загружается, статический блок инициализации выводит строку `Hello, World!`, и лишь затем появляется сообщение о том, что метод `main()` не определен. Это сообщение можно подавить с помощью вызова метода `System.exit(0)` в конце статического инициализационного блока.

Программа, приведенная в листинге 4.5, иллюстрирует многие средства, обсуждавшиеся в этой главе:

- перегрузка конструкторов;
- вызов конструктора с помощью выражения `this(...)`;
- использование конструктора по умолчанию;
- использование инициализационного блока;
- использование статического инициализационного блока;
- инициализация поля экземпляра.

Листинг 4.5. Содержимое файла `ConstructorTest.java`

```
1. import java.util.*;
2.
3. /**
4.    * Эта программа демонстрирует конструирование объектов.
5.    * @version 1.01 2004-02-19
6.    * @author Cay Horstmann
7.    */
8. public class ConstructorTest
9. {
10.   public static void main(String[] args)
11.   {
12.    // Заполняет массив staff тремя объектами Employee.
13.      Employee[] staff = new Employee[3];
14.
15.      staff[0] = new Employee("Harry", 40000);
16.      staff[1] = new Employee(60000);
17.      staff[2] = new Employee();
18.
19.      // Вывод информации о всех сотрудниках.
20.      for (Employee e : staff)
21.         System.out.println("name=" + e.getName() + ",id=" + e.getId()
22.            + ",salary=" + e.getSalary());
23.   }
24. }
25.
26. class Employee
27. {
28.   // Три перегруженных конструктора.
29.   public Employee(String n, double s)
30.   {
31.      name = n;
32.      salary = s;
33.   }
34.
35.   public Employee(double s)
36.   {
37.      // Вызов конструктора Employee(String, double).
38.      this("Employee #" + nextId, s);
39.   }
40.
```

```
41.   // Конструктор по умолчанию.
42.   public Employee()
43.   {
44.      // Поле name инициализируется как ""-- см. ниже.
45.      // Поле salary задается неявно -- инициализируется нулем.
46.      // Поле id задается в блоке инициализации.
47.   }
48.
49.   public String getName()
50.   {
51.      return name;
52.   }
53.
54.   public double getSalary()
55.   {
56.      return salary;
57.   }
58.
59.   public int getId()
60.   {
61.      return id;
62.   }
63.
64.   private static int nextId;
65.
66.   private int id;
67.   private String name = ""; // Инициализация поля name.
68.   private double salary;
69.
70.   // Статический блок инициализации.
71.   static
72.   {
73.      Random generator = new Random();
74.      // Задаем nextId как случайное число от 0 до 9999.
75.      nextId = generator.nextInt(10000);
76.   }
77.
78.   // Инициализационный блок объекта
79.   {
80.      id = nextId;
81.      nextId++;
82.   }
83.
84. }
```

API java.util.Random 1.0

- `Random()`

  Создает новый генератор случайных чисел.

- `int nextInt(int n)` 1.2

  Возвращает случайное число в интервале от `0` до `n-1`.

Уничтожение объекта и метод `finalize()`

В некоторых объектно-ориентированных языках, в частности в C++, есть явные деструкторы, предназначенные для уничтожения объектов. Основная их задача — освобождение памяти, занятой объектами. Поскольку в языке Java реализован механизм автоматической сборки мусора, освобождать память вручную нет никакой необходимости, поэтому в языке Java деструкторы отсутствуют.

Разумеется, некоторые объекты используют кроме памяти и другие ресурсы, например файлы, или обрабатывают другие объекты, которые, в свою очередь, обращаются к системным ресурсам. В этом случае очень важно, чтобы ресурсы вовремя освобождались.

В любой класс можно добавить метод `finalize()`. Этот метод будет вызван перед тем, как система "сборки мусора" уничтожит объект. На практике, если вам нужно возобновить ресурсы и сразу использовать их повторно, *нельзя полагаться на метод* `finalize()`, — вы не знаете, когда именно этот метод будет вызван.



На заметку! Существует метод `System.runFinalizerOnExit(true)`, гарантирующий, что метод `finalize()` будет вызван до того, как программа завершит свою работу. Однако этот метод крайне ненадежен и не рекомендован к использованию. В качестве альтернативы можно применить метод `Runtime.addShutdownHook()`. Дополнительную информацию о нем можно найти в документации по API.

Если ресурс должен быть освобожден сразу после его использования, нужно самостоятельно написать соответствующий код. Добавьте метод `dispose()` или `close()`, который нужно *явно* вызвать для очистки памяти. Если класс имеет такой метод, вы должны вызвать его по завершении работы с объектом этого класса.

Пакеты

Язык Java позволяет объединять классы в наборы, называемые *пакетами* (package). Пакеты облегчают организацию работы и позволяют отделить классы, созданные одним разработчиком, от классов, разработанных другими.

Стандартная библиотека языка Java содержит большое количество пакетов; в качестве примеров можно привести `java.lang`, `java.util`, `java.net` и т.д. Стандартные пакеты языка Java представляют собой иерархические структуры. Подобно каталогам на диске компьютера, пакеты могут быть вложены один в другой. Все стандартные пакеты принадлежат иерархиям `java` и `javax`.

В основном пакеты используются для обеспечения уникальности имен классов. Предположим, двух программистов осенила блестящая идея создать класс `Employee`. Если оба класса будут находиться в разных пакетах, конфликт не возникнет. Чтобы обеспечить абсолютную уникальность имени пакета, компания Sun рекомендует использовать доменное имя компании в Internet (которое по определению уникально), записанное в обратном порядке. В составе пакета можно создавать подпакеты и использовать их в разных проектах. Например, один из авторов этой книги имеет зарегистрированный домен `horstmann.com`. Записав это имя в обратном порядке, можно использовать его как название пакета — `com.horstmann`. В дальнейшем в этом пакете можно создать подпакет, например `com.horstmann.corejava`.

Единственная цель вложенных пакетов — гарантия уникальности имен. С точки зрения компилятора между вложенными пакетами нет никакой связи. Например, пакеты `java.util` и `java.util.jar` никак не связаны друг с другом. Каждый из них представляет собой независимую коллекцию классов.

Импортирование классов

Класс может использовать все классы из собственного пакета и все *общедоступные* классы из других пакетов.

Доступ к классам из других пакетов можно получить двумя путями. Во-первых, можно указывать полное имя пакета перед именем *каждого* класса. Например:

```
java.util.Date today = new java.util.Date();
```

Очевидно, что этот способ слишком утомителен. Более простой и распространенный способ предусматривает применение ключевого слова `import`. В этом случае имя пакета перед именем класса записывать не обязательно.

Импортировать можно как один конкретный класс, так и весь пакет. Операторы `import` следует поместить в начало исходного файла (после всех операторов `package`). Например, все классы из пакета `java.util` можно импортировать следующим образом:

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

После этого в строке кода, указанной ниже, имя пакета не указывается.

```
Date today = new Date();
```

Можно также импортировать отдельный класс из пакета:

```
import java.util.Date;
```

Импортировать все классы проще. На размер кода это не влияет. Однако если явным образом указать импортируемый класс, становится ясно, какие именно классы будут использованы в программе.



Совет. Работая с Eclipse, можно использовать пункт меню Source⇨Organize Imports (Исходный код⇨Импортирование). В результате выражения типа `import java.util.*;` будут автоматически преобразованы в последовательности строк, предназначенных для импортирования отдельных классов:

```
import java.util.ArrayList;
import java.util.Date;
```

Эта возможность несколько упрощает работу программиста.

Однако следует заметить, что оператор `import` со звездочкой можно применять для импортирования только одного пакета. Нельзя использовать обозначение `import java.*` или `import java.*.*`, чтобы импортировать все пакеты, имена которых содержат префикс `java`.

В большинстве случаев импортируется весь пакет, независимо от его размера. Единственный вариант, при котором на пакет следует обратить особое внимание, — конфликт имен. Например, и пакет `java.util`, и пакет `java.sql` содержат класс `Date`. Допустим, вы пишете программу, импортирующую оба этих пакета:

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

Если теперь попытаться использовать класс `Date`, возникнет ошибка компиляции:

```
Date today; // ОШИБКА -- java.util.Date или java.sql.date?
```

Компилятор не может определить, какой класс `Date` вам нужен. Решить эту проблему можно, добавив конкретный оператор `import`:

```
import java.util.*;
import java.sql.*;
import java.util.Date;
```

А если вам на самом деле нужны оба класса `Date`? В этом случае нужно указывать полное имя пакета для каждого имени класса:

```
java.util.Date deadline = new java.util.Date();
java.sql.Date today = new java.sql.Date();
```

Обнаружение классов в пакетах является задачей *компилятора*. В ссылках, находящихся в файлах классов, всегда содержатся полные имена пакетов.



На заметку! Программисты, использующие язык C++, считают, что оператор `import` является аналогом директивы `#include`. Однако между ними нет ничего общего. В языке C++ директиву `#include` нужно применять, поскольку компилятор C++ не просматривает файлы, за исключением того, который он компилирует, и файлов, указанных в директиве `#include`. Компилятор языка Java "видит" содержимое всех файлов при условии, если он "знает", где их искать.

В языке Java можно вовсе не применять механизм импортирования, явно назвав все пакеты, например `java.util.Date`. В языке C++ избежать использования директивы `#include` нельзя.

Единственное преимущество оператора `import` — удобство. Он позволяет использовать более короткие имена классов, не указывая полное имя пакета. Например, после оператора `import java.util.*` (или `import java.util.Date`) на класс `java.util.Date` можно ссылаться по имени `Date`.

Аналогичный механизм работы с пакетами в языке C++ реализован в виде директивы `namespace`. Ключевые слова `package` и `import` в языке Java можно считать аналогами директив `namespace` и `using` в C++.

Статический импорт

Начиная с Java SE 5.0, оператор `import` был расширен. Теперь вы можете импортировать не только классы, но также статические методы и поля. Предположим, например, что в начало файла с исходным текстом вы поместили следующую строку кода:

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

Это позволит использовать статические методы и поля, определенные в классе `System`, не указывая имени класса:

```
out.println("Goodbye, World!");   // вместо System.out
exit(0);                          // вместо System.exit
```

Вы также можете явным образом импортировать статические методы или поля:

```
import static java.lang.System.out;
```

Вряд ли можно встретить программиста, который захочет сокращать такие выражения, как `System.out` или `System.exit`. Полученный в результате код станет более трудным для восприятия. Однако новая возможность окажется полезной как минимум в двух случаях.

1. При использовании математических функций. Если вы импортируете статические методы класса `Math`, то сможете непосредственно задавать в программе имена математических функций. Согласитесь, что выражение

   ```
   sqrt(pow(x, 2) + pow(y, 2))
   ```

выглядит гораздо естественнее, чем следующая строка кода:

```
Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2))
```

2. При использовании констант. Если в программе применяется большое количество констант, возможность импортирования статических полей придется как нельзя более кстати. Выражение

```
if (d.get(DAY_OF_WEEK) == MONDAY)
```

проще для восприятия, чем строка

```
if (d.get(Calendar.DAY_OF_WEEK) == Calendar.MONDAY)
```

Добавление классов в пакеты

Чтобы поместить класс в пакет, нужно указать имя пакета в начале исходного файла *перед* определением класса. Например, файл `Employee.java` в листинге 4.7 начинается следующими строками:

```
package com.horstmann.corejava;

public class Employee
{
   ...
}
```

Если оператор `package` в исходном файле не указан, то классы, описанные в этом файле, помещаются в *пакет по умолчанию*. Пакет по умолчанию не имеет имени. До сих пор все рассмотренные нами классы принадлежали пакету по умолчанию.

Пакеты следует помещать в подкаталог, путь к которому соответствует полному имени пакета. Например, все файлы классов в пакете `com.horstmann.corejava` должны находиться в подкаталоге `com/horstmann/corejava` (соответственно `com\horstmann\corejava` в системе Windows). Компилятор помещает файлы классов в ту же структуру каталогов.

Программы, приведенные в листингах 4.6 и 4.7, распределены по двум пакетам: класс `PackageTest` принадлежит пакету по умолчанию, а класс `Employee` — пакету `com.horstmann.corejava`. Следовательно, файл `Employee.class` должен содержаться в подкаталоге `com/horstmann/corejava`. Другими словами, структура каталогов должна выглядеть следующим образом:

```
.(базовый каталог)
  !-- PackageTest.java
  !-- PackageTest.class
  !-- com/
      !-- horstmann/
          !-- corejava/
              !-- Employee.java
              !-- Employee.class
```

Чтобы скомпилировать эту программу, перейдите в каталог, содержащий файл `PackageTest.java`, и выполните следующую команду:

```
javac Package.java
```

Компилятор автоматически найдет файл `com/horstmann/corejava/Employee.java` и скомпилирует его.

Рассмотрим более реалистичный пример. В данном примере пакет по умолчанию не используется, вместо этого классы распределены по разным пакетам (`com.horstmann.corejava` и `com.mycompany`).

```
. (базовый каталог)
  !-- com/
        !--horstmann/
        !   !-- corejava/
        !        !--Employee.java
        !        !--Employee.class
        !-- mycompany/
            !-- PayrollApp.java
            !-- PayrollApp.class
```

В этой ситуации вы также должны компилировать и запускать классы из *базового каталога*, т.е. того каталога, в котором содержится подкаталог `com`.

```
javac com/mycompany/PayrollApp.java
java com.mycompany.PayrollApp
```

Помните, что компилятор работает с *файлами* (при указании файла задается путь и расширение `.java`), а интерпретатор оперирует с *классами* (для класса указывается пакет).



Внимание! Компилятор не проверяет структуру каталогов. Предположим, например, что исходный файл начинается со следующей директивы:

```
package com.mycompany;
```

Этот файл можно скомпилировать, даже если он не находится в каталоге `com/mycompany`. Исходный файл будет скомпилирован без ошибок, если он не зависит от других пакетов, однако при попытке выполнить эту программу виртуальная машина не найдет классы.

Листинг 4.6. Содержимое файла `PackageTest.java`

```
1. import com.horstmann.corejava.*;
2.
3. // В этом пакете определен класс Employee.
4.
5. import static java.lang.System.*;
6.
7. /**
8.    * Эта программа демонстрирует использование пакетов.
9.    * @version 1.11 2004-02-19
10.   * @author Cay Horstmann
11.   */
12.
13.public class PackageTest
14.{
15.   public static void main(String[] args)
16.   {
17.      // Поскольку использован оператор import, имя
18.      // com.horstmann.corejava.Employee применять не обязательно.
19.      Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
20.
21.      harry.raiseSalary(5);
22.
23.      // Поскольку использован оператор import, имя
24.      // System.out применять не обязательно.
25.      out.println("name=" + harry.getName() + ",salary=" + harry.getSalary());
26.   }
27. }
```

Листинг 4.7. Содержимое файла `Employee.java`

```
1. package com.horstmann.corejava;
2.
3. // Классы в этом файле являются частью пакета.
4.
5. import java.util.*;
6.
7. // Операторы import помещаются после оператора package.
8.
9. public class Employee
10. {
11.   public Employee(String n, double s,
12.      int year, int month, int day)
13.   {
14.      name = n;
15.      salary = s;
16.      GregorianCalendar calendar
17.         = new GregorianCalendar(year, month - 1, day);
18.      // В классе GregorianCalendar январю соответствует 0.
19.      hireDay = calendar.getTime();
20.   }
21.
22.   public String getName()
23.   {
24.      return name;
25.   }
26.
27.   public double getSalary()
28.   {
29.      return salary;
30.   }
31.
32.   public Date getHireDay()
33.   {
34.      return hireDay;
35.   }
36.
37.   public void raiseSalary(double byPercent)
38.   {
39.      double raise = salary * byPercent / 100;
40.      salary += raise;
41.   }
42.
43.   private String name;
44.   private double salary;
45.   private Date hireDay;
46. }
```

Область видимости пакета

Мы уже встречались с модификаторами доступа `public` и `private`. Элементы, помеченные ключевым словом `public`, могут использоваться любым классом. Приватные элементы, т.е. те, при определении которых использовалось ключевое слово `private`, могут использоваться только тем классом, в котором они были определены. Если ни

один модификатор доступа не указан, то элемент (класс, метод или переменная) доступен всем методам в том же самом *пакете*.

Вспомним программу, приведенную в листинге 4.2. Класс `Employee` не определен как общедоступный. Следовательно, любой другой класс из того же самого пакета — в данном случае пакета, заданного по умолчанию, — например класс `EmployeeTest`, может получить доступ к нему. Для классов такой подход надо признать вполне разумным. Однако для переменных этот способ доступа неудачен. Переменные должны быть явно помечены как `private`, или по умолчанию их область видимости будет расширена до пределов пакета. Это, конечно, нарушает принцип инкапсуляции. В процессе работы над программой разработчики часто забывают указать ключевое слово `private`. Вот пример из класса `Window`, принадлежащего пакету `java.awt`, который поставляется с пакетом JDK:

```
public class Window extends Container
{
   String warningString;
   ...
}
```

Обратите внимание, что переменная `warningString` не имеет модификатора доступа `private`! Это значит, что методы всех классов в пакете `java.awt` могут обращаться к ней и изменить ее значение (например, присвоить ей строку `Trust me!`). Фактически все методы, имеющие доступ к `warningString`, принадлежат классу `Window`, поэтому эту переменную смело можно было бы объявить как приватную. Мы подозреваем, что программисты, писавшие этот код, торопились и просто забыли указать ключевое слово `private`. (Мы не станем упоминать имени программиста, чтобы найти виновного — вы сами можете заглянуть в исходный код.)



На заметку! Как ни странно, этот недостаток не был устранен даже после того, как мы указали на него в восьми изданиях этой книги, — очевидно, что программисты, создававшие эту библиотеку, не читают нашей книги. Более того, со временем в класс были добавлены новые поля, и снова половина из них не была объявлена как `private`.

А может, это и вовсе не проблема? Однозначно ответить на этот вопрос нельзя. По умолчанию пакеты не являются приватными. Это значит, что каждый может добавлять в пакеты свои классы. Разумеется, злонамеренные или невежественные программисты могут написать код, модифицирующий переменные, область видимости которых ограничена пакетом. Например, в ранних версиях языка Java можно было легко проникнуть в другой класс пакета `java.awt`.

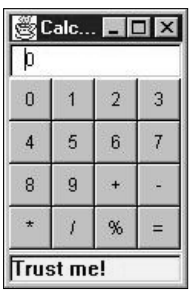


Рис. 4.9. Изменение строки в окне аплета

Для этого было достаточно начать свой класс со строки

```
package java.awt;
```

Затем нужно было поместить полученный файл класса в подкаталог `java\awt`, и доступ к содержимому пакета `java.awt` открыт. Таким образом, можно было изменить строку предупреждения, отображаемую на экране (рис. 4.9).

В версии 1.2 создатели пакета JDK запретили загрузку классов, определенных пользователем, если их имя начиналось с `java.`! Разумеется, эта защита не распространяется на ваши собственные классы. Вместо этого можно использовать другой механизм, называемый *герметизацией пакета* (package sealing), который ограничивает доступ к пакету. Выполнив герметизацию пакета, вы запре-

щаете добавлять к нему классы. В главе 10 показано, как создать JAR-файл, содержащий герметизованные пакеты.

Путь поиска классов

Как вы уже знаете, классы хранятся в подкаталогах файловой системы. Путь к классу должен совпадать с именем пакета. Кроме того, чтобы поместить классы в JAR (Java архив), можно использовать утилиту `jar`. В одном архивном файле JAR содержится много файлов классов и подкаталогов в сжатом виде, что позволяет экономить память и сокращать время доступа к ним. Когда вы используете библиотеку от независимых разработчиков, обычно вы получаете один или более файлов JAR. Также и JDK включает множество файлов JAR, таких как `jre/lib/rt.jar`, содержащий тысячи библиотечных классов. В главе 10 вы прочтете о том, как создавать собственные файлы JAR.



Совет. Для организации файлов и подкаталогов в файлах JAR используется формат ZIP. Обрабатывать `rt.jar` и другие JAR-файлы можно с помощью любой утилиты, поддерживающей ZIP.

Чтобы обеспечить совместный доступ программ к классам, необходимо выполнить следующие шаги.

1. Поместить файлы классов в один или несколько специальных каталогов, скажем `/home/user/classdir`. Отметим, что этот каталог является *базовым* по отношению к дереву пакета. Если потребуется добавить класс `com.horstmann.corejava.Employee`, то файл класса следует поместить в подкаталог `/home/user/classdir/com/horstmann/corejava`.
2. Поместить все файлы JAR в каталог, например `/home/user/archives`.
3. Установить *путь к классу*. Путь к классу — это набор всех базовых каталогов, которые могут содержать файлы классов.

В среде Unix элементы пути к классу отделяются друг от друга двоеточиями:

```
/home/user/classdir:.:/home/user/archives/archive.jar
```

В среде Windows они разделяются точками с запятой:

```
c:\clasdir;.;c:\archives\archive.jar
```

В обоих случаях точка означает текущий каталог.
Путь к классу содержит:

- имя базового каталога `/home/user/classdir` или `c:\classes`;
- обозначение текущего каталога (`.`);
- имя файла JAR `/home/user/archives/archive.jar` или `c:\archives\archive.jar`.

Начиная с Java SE 6, вы можете специфицировать шаблонные символы для каталога файлов JAR:

```
/home/user/classdir:.:/home/user/archives/'*'
```

или

```
c:\classdir;.;c:\archives\*
```

В UNIX символ `*` должен быть защищен (escaped) для предотвращения интерпретации его командной оболочкой.

Все JAR-файлы (но не файлы .class) в каталоге archives включаются в путь поиска классов.

Файлы исполняющей системы (rt.jar и другие JAR-файлы в каталогах jre/lib и jre/lib/ext) всегда участвуют в поиске классов; вам не нужно включать их явно в путь поиска.



Внимание! Компилятор javac всегда ищет файлы в текущем каталоге, а интерпретатор java обращается к текущему каталогу, только если в пути к классам указана точка ".". Если путь к классам не указан, никаких проблем не возникает — по умолчанию в него включается текущий каталог ".". Но если вы задали путь и забыли указать точку, то ваша программа будет скомпилирована без ошибок, однако выполниться не сможет.

Путь поиска классов перечисляет все каталоги и архивные файлы, которые являются *начальными точками* при поиске классов. Рассмотрим наш простой путь классов:

```
/home/user/classdir:.:/home/user/archives/archive.jar
```

Предположим, что интерпретатор осуществляет поиск файла, содержащего класс com.horstmann.corejava.Employee. Сначала он ищет его в системных файлах, которые хранятся в архивах, находящихся в каталогах jre/lib и jre/lib/ext. В этих файлах искомого класса нет, поэтому интерпретатор анализирует пути. Он ищет следующие файлы:

- /home/user/classdir/com/horstmann/corejava/Employee.class;
- com.horstmann/corejava/Employee.class, начиная с текущего каталога;
- com.horstmann/corejava/Employee.class в каталоге /home/user/archives/archive.jar.

На поиск файлов компилятор затрачивает больше времени, чем виртуальная машина. Если вы указали класс, не назвав пакета, которому он принадлежит, компилятор сначала должен сам определить, какой именно пакет содержит данный класс. Как возможные источники классов рассматриваются пакеты, указанные в директивах import. Предположим, что исходный файл содержит приведенные ниже директивы и содержащийся в нем код ссылается на класс Employee.

```
import java.util.*
import com.horstmann.corejava.*;
```

Компилятор пытается найти классы java.lang.Employee (поскольку пакет java.lang всегда импортируется по умолчанию), java.util.Employee, com.horstmann.corejava.Employee и Employee в текущем пакете. Он ищет *каждый* из этих файлов во всех каталогах, указанных в пути к классам. Если найдено несколько таких классов, возникает ошибка компиляции. (Поскольку классы должны быть уникальными, порядок следования директив import значения не имеет.)

Компилятор делает еще один шаг. Он просматривает *исходные файлы*, чтобы проверить, были ли они созданы позже, чем файлы классов. Если проверка дает положительный результат, исходный файл автоматически перекомпилируется. Напомним, что из других пакетов можно импортировать лишь общедоступные классы. Исходный файл может содержать только один общедоступный класс, кроме того, имена файла и класса должны совпадать. Следовательно, компилятор может легко определить, где находятся исходные файлы, содержащие общедоступные классы. Из текущего пакета можно импортировать также классы, не определенные как public. Исходные коды классов могут

содержаться в файлах с разными именами. При импортировании класса из текущего пакета компилятор проверяет *все* исходные файлы, чтобы выяснить, в каком из них определен требуемый класс.

Установка пути к классам

Лучше всего указать путь к классам с помощью опции `-classpath` (или `-cp`):

```
java -classpath /home/user/classdir:.:/home/user/archives/archive.jar MyProg.java
```

или

```
java -classpath c:\classdir;.;c:\archives\archive.jar MyProg.java
```

Вся команда должна быть набрана в одной строке. Лучше всего поместить такую длинную команду в сценарий оболочки или командный файл.

Применение опции `-classpath` — предпочтительный способ установки путей к классам. Альтернативный вариант — переменная окружения `CLASSPATH`. Детали зависят от вашей командной оболочки.

В оболочке Bourne Again (bash) используйте команду

```
export CLASSPATH=/home/user/classdir:.:/home/user/archives/archive.jar
```

В оболочке C shell применяйте команду

```
setenv CLASSPATH /home/user/classdir:.:/home/user/archives/archive.jar
```

В командной строке Windows используйте:

```
set CLASSPATH=c:\classdir;.;c:\archives\archive.jar
```

Путь к классам действителен до завершения работы оболочки.



Внимание! Некоторые советуют устанавливать переменную окружения `CLASSPATH` на постоянной основе. В общем случае это плохая идея. Люди забывают о глобальных установках, а потом удивляются, когда классы не загружаются правильно. В частности, показательным примером может служить программа установки QuickTime от Apple в Windows. Он устанавливает глобально `CLASSPATH` так, чтобы она указывала на нужный ему JAR-файл, при этом не включая в путь текущий каталог. В результате многочисленные программисты Java попадают в тупик, когда их программы компилируются, а потом не запускаются.



Внимание! Некоторые советуют вообще не указывать путь к классам, сбрасывая все JAR-файлы в каталог `jre/lib/ext`. Это очень плохое решение по двум причинам. Архивы, которые вручную загружают другие классы, не работают правильно, когда они помещены в каталог расширений (подробнее о загрузчиках классов написано во втором томе). Более того, программистам свойственно забывать о файлах, которые они поместили туда три месяца назад. Затем они ломают голову, когда загрузчик файлов игнорирует их тщательно продуманный путь классов, на самом деле загружая давно забытые классы из каталога расширений.

Комментарии и документирование

Пакет JDK содержит полезный инструмент под названием `javadoc`, генерирующий на базе исходных файлов документацию в формате HTML. Фактически интерактивная документация по API является результатом применения утилиты `javadoc` к исходному коду стандартной библиотеки языка Java.

Добавив к своему исходному коду комментарии, начинающиеся последовательностью символов /**, легко создать документацию, имеющую профессиональный вид. Это очень удобный способ, поскольку он позволяет совместно хранить как код, так и документацию к нему. Если документацию поместить в отдельный файл, то со временем она перестанет соответствовать коду. В то же время, поскольку комментарии являются неотъемлемой частью исходного файла, документацию легко обновить, повторно запустив утилиту javadoc.

Включение комментариев

Утилита javadoc извлекает информацию о следующих элементах.

- пакетах;
- классах и интерфейсах, объявленных как public;
- методах, объявленных как public или protected;
- полях, объявленных как public или protected.

Защищенные элементы, для объявления которых используется ключевое слово protected, будут рассмотрены в главе 5, а интерфейсы — в главе 6.

Разрабатывая программу, можно (и даже необходимо) комментировать каждый из перечисленных элементов. Комментарии помещаются непосредственно *перед* элементом, к которому они относятся. Комментарии начинаются с разделителя /** и заканчиваются символами */.

Комментарии, имеющие вид /** ... */, содержат произвольный текст, за которым следует *дескриптор* (tag). Дескриптор начинается символом @, например @author или @param.

Первое предложение текста комментариев должно представлять собой *краткое описание*. Утилита javadoc автоматически генерирует страницы, состоящие из кратких описаний.

В самом тексте можно использовать элементы языка HTML, например, ... — для выделения текста курсивом, <code>...</code> для установки моноширинного шрифта, ... — для выделения текста полужирным шрифтом и даже <img ...> — для вставки рисунков. Следует, однако, избегать заголовков (<h1> – <h6>) и горизонтальных линий (<hr>), поскольку они могут повлиять на форматирование документа.



На заметку! Если комментарии содержат ссылки на другие файлы, например рисунки (например, диаграммы или образы компонентов пользовательского интерфейса), поместите эти файлы в каталог, содержащий исходный файл по имени doc-files. Утилита javadoc скопирует эти каталоги и файлы, содержащиеся в них, из исходного каталога в каталог для документации. Вы должны использовать каталог doc-files в ваших ссылках, например .

Комментарии к классу

Комментарии к классу должны помещаться *после* директив import, непосредственно перед определением класса. Ниже приведен пример подобных комментариев.

```
/**
   Объект класса <code>Card</code> имитирует игральную карту,
   например червонную даму. Карта имеет масть и ранг
   (1=туз, 2...10, 11=валет, 12=дама, 13=король).
*/
```

```
public class Card
{
   ...
}
```



На заметку! Нет необходимости начинать каждую строку документации звездочкой. Например, следующий комментарий вполне корректен:

```
/**
    Объект класса <code>Card</code> имитирует игральную карту,
    например, червонную даму. Карта имеет масть и ранг
    (1=туз, 2...10, 11=валет,     12=дама, 13=король).
*/
```

Однако большинство интегрированных сред разработки применяют звездочки в начале строки комментария автоматически и перестраивают их при изменении расположения переносов строк.

Комментарии к методам

Комментарии должны непосредственно предшествовать методу, который они описывают. Кроме дескрипторов общего назначения можно использовать специальные дескрипторы.

- `@param` *переменная описание*

  Этот дескриптор добавляет в описание метода раздел "parameters" ("параметры"). Раздел параметров можно "растянуть" на несколько строк. Кроме того, можно использовать элементы языка HTML. Все дескрипторы `@param`, относящиеся к одному методу, должны быть сгруппированы вместе.

- `@return` *описание*

  Данный дескриптор добавляет в описание метода раздел "returns" ("возвращаемое значение"). Этот раздел также может занимать несколько строк. Допустимы также дескрипторы языка HTML.

- `@throws` *описание_класса*

  Этот дескриптор означает, что метод может генерировать исключение. Исключения будут рассмотрены в главе 11.

Рассмотрим пример комментариев к методу.

```
/**
   Увеличивает зарплату сотрудников.
   @param Переменная byPercent содержит величину
     в процентах, на которую повышается зарплата
     (например, 10 = 10%).
   @return Размер повышения
*/

public double raiseSalary(double byPercent)
{
   double raise = salary &* byPercent / 100;
   salary += raise;
   return raise;
}
```

Комментарии к полям

Документировать нужно лишь общедоступные поля — обычно они представляют собой статические константы. Например:

```
/**
   Масть черви.
*/
public static final int HEARTS = 1;
```

Комментарии общего характера

В комментариях к классу можно использовать следующие дескрипторы:

- `@author` *имя*

  Этот дескриптор создает раздел "author" ("автор"). В комментариях может быть несколько таких дескрипторов — по одному на каждого автора.

- `@version` *текст*

  Этот дескриптор создает раздел "version" ("версия"). В данном случае "*текст*" означает произвольное описание текущей версии.

При создании любых комментариев, предназначенных для создания документации, можно использовать также следующие дескрипторы:

- `@since` *текст*

  Этот дескриптор создает раздел "since" ("начиная с..."). Слово "текст" означает описание версии, в которой впервые было реализовано данное свойство. Например, `@since version 1.7.1`.

- `@deprecated` *текст*

  Этот дескриптор добавляет сообщение о том, что класс, метод или переменная не рекомендуется к применению. Подразумевается, что после дескриптора `@deprecated` следует некоторое выражение. Например:

  ```
  @deprecated Use <code>setVisible(true)</code> instead
  ```

С помощью дескрипторов `@see` и `@link` можно использовать гипертекстовые ссылки на соответствующие внешние документы или на позицию в составе того же документа.

- `@see` *ссылка*

  Этот дескриптор добавляет ссылку в раздел "See also" ("См. также"). Его можно использовать в комментариях как к классам, так и к методам. Здесь под словом "*ссылка*" подразумевается одна из следующих конструкций:

  ```
  пакет.класс#элемент метка
  <a href="...">метка</a>
  "текст"
  ```

  Первый вариант встречается чаще всего. Здесь нужно указать имена класса, метода или переменной, а утилита `javadoc` вставит в документацию соответствующую гипертекстовую ссылку. Например, приведенная ниже команда создает ссылку на метод `raiseSalary(double)` класса `com.horstmann.corejava.Employee`:

  ```
  @see com.horstmann.corejava.Employee#raiseSalary(double)
  ```

Обратите внимание, что здесь для отделения имени класса от имени метода или переменной нужно использовать символ #, а не точку. Компилятор javac корректно обрабатывает точки, выступающие в качестве разделителей между именами пакетов, подпакетов, классов, внутренних классов, методов и переменных. Однако утилита javadoc не настолько интеллектуальна, поэтому возникает необходимость в специальном синтаксисе.

Если вслед за дескриптором @see стоит символ <, то программист должен указать гипертекстовую ссылку. Ссылаться можно на любой URL-адрес. Например:

```
@see <a href="www.horstmann.com/corejava.html">Web-страница книги Core Java</a>
```

В каждом их этих случаев можно указать необязательную *метку*, которая играет роль якоря для ссылки. Если метку не указать, то якорем ссылки будет считаться имя кода или URL.

Если вслед за дескриптором @see стоит символ ", то текст отображается в разделе "see also". Например:

```
@see "Core Java, volume 2"
```

Для одного и того же элемента можно использовать несколько дескрипторов @see, однако их следует сгруппировать вместе.

- При желании можно поместить гипертекстовые ссылки на другие классы или методы в любом комментарии. Для этого нужно включить в требуемую позицию специальный дескриптор вида {link *пакет.класс#элемент метка*}. Описание элемента подчиняется тем же правилам, что и дескриптор @see.

Пакетные и обзорные комментарии

Комментарии к классам, методам и переменным помещаются обычно непосредственно в исходные файлы. При этом комментарии выделяются символами /** ... */. Однако для генерации комментариев к пакету в каждый каталог, где находится пакет, нужно добавить отдельный файл в каждый каталог пакета. Здесь у вас есть два выбора.

1. Применить HTML-файл по имени package.html. Весь текст, содержащийся между дескрипторами <body> ... </body>, извлекается утилитой javadoc.
2. Подготовить Java-файл по имени package-info.java. Этот файл должен содержать начальный комментарий Javadoc, отделенный /** и */, за которым следует предложение package. Оно не должно содержать никакого кода или комментариев.

Кроме того, все исходные файлы можно сопроводить обзорными комментариями. Они размещаются в файле под названием overview.html, размещенном в родительском каталоге, содержащем все исходные файлы. Весь текст, находящийся между дескрипторами <body> ... </body>, извлекается утилитой javadoc. Эти комментарии отображаются на экране, когда пользователь выбирает Overview (Обзор) в панели навигации.

Извлечение комментариев

Будем считать, что docDirectory — это имя каталога, в котором должны храниться HTML-файлы. Выполним описанные ниже действия.

1. Перейдем в каталог, содержащий исходные файлы, подлежащие документированию. Если документированию подлежат вложенные пакеты, например com.horstmann.corejava, нужно перейти в каталог, содержащий подкаталог com. (Именно в этом каталоге должен храниться файл overview.html.)

2. Для документирования одного пакета выполним команду

   ```
   javadoc -d каталог_doc имя_пакета
   ```

   Для документирования нескольких пакетов выполним команду

   ```
   javadoc -d каталог_doc имя_пакета_1 имя_пакета_2 ...
   ```

   Если файлы находятся в пакете, заданном по умолчанию, вместо приведенных выше команд нужно выполнить следующую:

   ```
   javadoc -d каталог_doc *.java
   ```

Если пропустить опцию `-d каталог_doc`, HTML-файлы будут извлечены в текущий каталог, что вызовет путаницу; мы не рекомендуем делать это.

При вызове `javadoc` можно указывать различные опции. Например, чтобы включить в документацию дескрипторы `@author` и `@version`, можно использовать опции `-author` и `-version` (по умолчанию они опущены). Опция `-link` позволяет включать ссылки на стандартные классы. Например, приведенная ниже команда автоматически создаст ссылку на документацию, находящуюся на Web-сайте Sun.

```
javadoc -link http://java.sun.com/javase/6/docs/api *.java
```

Если вы используете опцию `-linksource`, то каждый исходный файл будет преобразован в HTML (без цветового кодирования, но с номерами строк), и каждое имя класса и метода превратится в гиперссылку на исходный код.

Другие опции описаны в документации на `javadoc`, которую можно найти по адресу `http://java.sun.com/javase/javadoc`.



На заметку! Если вам надо выполнить более тонкую настройку, например, создать документацию в формате, отличающемся от HTML, можно разработать свой собственный *доклет* (doclet), позволяющий генерировать документацию произвольного вида. Более подробную информацию о данном подходе можно получить на Web-странице `http://java.sun.com/j2se/javadoc`.



Совет. При работе над программами может оказаться полезным доклет DocCheck (`http://java.sun.com/j2se/javadoc/doccheck/`). Он просматривает исходные файлы и отмечает пропущенные комментарии.

Рекомендации по проектированию классов

Завершим мы эту главу некоторыми советами, которые облегчат вам объектно-ориентированную разработку классов.

1. *Всегда храните данные в переменных, объявленных как* `private`.

   Первое и главное требование: всеми средствами избегайте нарушения инкапсуляции. Иногда приходится писать методы доступа к полю или модифицирующие методы, но предоставлять доступ к полям не следует. Горький опыт показал, что способ представления данных может изменяться, однако порядок их использования изменяется гораздо реже. Если данные приватны, их представление не влияет на использующий класс, поэтому ошибки искать легче.

2. *Всегда инициализируйте данные.*

   Язык Java не инициализирует локальные переменные, однако поля в объектах инициализируются. Не полагайтесь на действия по умолчанию, инициализируйте переменные явным образом с помощью конструкторов.

3. *Не используйте в классе слишком много простых типов.*

   Несколько *связанных* между собой полей простых типов следует объединять в новый класс. Такие классы проще для понимания, и их легче модифицировать. Например, следующие четыре поля в классе `Customer` надо объединить в новый класс `Address`:

   ```
   private String street;
   private String city;
   private String state;
   private int zip;
   ```

   Представленный таким образом адрес легче изменять, как, например, в случае необходимости указания интернациональных адресов.

4. *Не для всех полей надо создавать методы доступа и модификации.*

   Очевидно, что при выполнении программы надо получать сведения о зарплате сотрудника, кроме того, ее приходится время от времени изменять. Но вряд ли придется менять дату приема на работу после того, как объект сконструирован. То есть существуют поля, которые после создания объекта вовсе не изменяются. К ним относится, например, массив аббревиатур названий штатов США в классе `Address`.

5. *Используйте стандартную форму определения класса.*

   Содержимое класса всегда перечисляется в следующем порядке:

   - общедоступные элементы;
   - элементы, область видимости которых ограничена пакетом;
   - приватные элементы.

   Внутри каждого раздела перечисляются:

   - методы экземпляра;
   - статические методы;
   - поля экземпляра;
   - статические поля.

   Такой порядок вполне оправдан. Пользователей вашего класса больше интересует общедоступный интерфейс, чем детали приватной реализации. Кроме того, их больше интересуют методы, нежели данные.

   Однако в настоящее время общего мнения об оптимальном стиле программирования еще нет. Согласно стилю программирования на языке Java, принятом в компании Sun, рекомендуется сначала перечислять поля, а затем — методы. Независимо от того, какого стиля вы придерживаетесь, главное — не отступать от него.

6. *Разбивайте на части слишком большие классы.*

   Этот совет, конечно, слишком общий: то, что кажется “слишком большим” одному программисту, кажется нормальным другому. Однако если есть очевидная возможность разделить один сложный класс на два класса попроще, воспользуйтесь ею. (Только опасайтесь другой крайности. Вряд ли оправданы десять классов, в каждом из которых имеется только один метод.)

   Ниже приведен пример плохого стиля.

```
class CardDeck // Плохой стиль.
{
  public CardDeck() { ... }
  public void shuffle() { ... }
  public int getTopValue() { ... }
  public int getTopSuit() { ... }
  public void draw() { ... }

  private int[] value;
  private int[] suit;
```

На самом деле в этом классе реализованы две разные концепции: во-первых, *карточный стол* с методами `shuffle()` (тасование) и `draw()` (раздача) и, во-вторых, *игральная карта* с методами для проверки ранга и масти карты. Разумнее было бы ввести класс `Card`, представляющий собой отдельную игральную карту. Тогда мы имели бы два класса, каждый со своей задачей.

```
class CardDeck
{
  public CardDeck() { ... }
  public void shuffle() { ... }
  public int getTopValue() { ... }
  public int getTopSuit() { ... }
  public void draw() { ... }

  private Card[] cards;
}

class Card
{
  public Card (int aValue, int aSuit) { ... }
  public int getValue() { ... }
  public int getSuit() { ... }

  private int[] value;
  private int[] suit;
}
```

7. *Выбирайте для классов и методов осмысленные имена.*

   Классы, как и переменные, следует называть именами, отражающими их смысл. (В стандартной библиотеке есть примеры, для которых данное правило нарушается, скажем, класс `Date` описывает время, а не дату.)

   Удобно принять следующее соглашение: имя класса должно быть именем существительным (`Order`) или именем существительным, которому предшествует имя прилагательное (`RushOrder`), или герундий (слово, заканчивающееся на "-ing", например `BillingAddress`). Как правило, методы доступа должны начинаться словом `get`, представленным строчными буквами (`getSalary`), а модифицирующие методы — словом `set`, также представленным строчными буквами (`setSalary`).

В этой главе мы рассмотрели основы объектов и классов, которые делают Java объектным языком. Для того чтобы быть действительно объектно-ориентированным, язык программирования также должен поддерживать наследование и полиморфизм. Поддержка этих средств в Java — тема следующей главы.

Глава 5

НАСЛЕДОВАНИЕ

В ЭТОЙ ГЛАВЕ...

- Классы, суперклассы и подклассы
- `Object`: глобальный суперкласс
- Обобщенные массивы-списки
- Объектные оболочки и автоупаковка
- Методы с переменным числом параметров
- Классы-перечисления
- Рефлексия
- Рекомендации по применению наследования

В главе 4 мы изучали классы и объекты. Данная глава посвящена *наследованию* — фундаментальному понятию объектно-ориентированного программирования. Идея, лежащая в основе наследования, заключается в том, что новые классы можно создавать из уже существующих. При наследовании методы и поля существующего класса используются повторно (наследуются) вновь создаваемым классом, причем для адаптации нового класса к новым ситуациям в него добавляют дополнительные поля и методы. Этот прием играет в языке Java весьма важную роль.

Если вы имеете опыт работы лишь с такими процедурно-ориентированными языками, как C, Visual Basic или COBOL, тщательно изучите эту главу. Для программистов, работавших на языке C++ или других объектно-ориентированных языках, например Smaltalk, сведения, изложенные в этой главе, будут знакомы. Следует, однако, учесть, что механизмы наследования, реализованные в языке Java и в других объектно-ориентированных языках, значительно отличаются друг от друга.

В конце главы мы рассмотрим механизм *рефлексии*, позволяющий исследовать свойства классов в ходе выполнения программы. Рефлексия — мощный, но очень сложный механизм. Поскольку рефлексия больше интересует разработчиков инструментальных средств, чем прикладных программистов, при первом чтении можно ограничиться беглым просмотром этого раздела.

Классы, суперклассы и подклассы

Вернемся к классу `Employee`, рассмотренному в предыдущей главе. Допустим, что вы работаете в компании, в которой работа менеджеров учитывается иначе, чем работа остальных сотрудников. Разумеется, менеджеры во многих отношениях являются обычными наемными сотрудниками. И сотрудникам, и менеджерам выплачивается заработная плата. Однако менеджеры за свои достижения получают еще и премии. В этой ситуации следует применять наследование. Почему? Потому что нужно определить новый класс `Manager`, в который добавить новую функциональность. При этом вы можете сохранить кое-что из того, что уже программировали в классе `Employee`, а также сохраняются все поля, представленные в исходном классе. Говоря более абстрактно, между классами `Manager` и `Employee` существует очевидное отношение "является" ("is-a"). Каждый менеджер *является* сотрудником: это отношение "является" — признак наследования.

Ниже приводится пример определения класса `Manager`, порожденного из класса `Employee`. Для обозначения наследования в языке Java используется ключевое слово `extends`.

```
class Manager extends Employee
{
   Дополнительные методы и поля
}
```



На заметку! Механизмы наследования в языках Java и C++ похожи. В языке Java вместо символов : используется ключевое слово `extends`. Любое наследование в языке Java является общедоступным, причем аналога приватному и защищенному наследованию, существующему в языке C++, в языке Java нет.

Ключевое слово `extends` означает, что на базе существующего создается новый класс. Существующий класс называется *суперклассом*, *базовым* или *родительским*, а вновь

создаваемый — *подклассом*, *производным* или *дочерним*. Среди программистов, использующих язык Java, наиболее широко распространены термины "суперкласс" и "подкласс", хотя некоторые специалисты предпочитают пользоваться понятиями "родительский/дочерний", более тесно связанными со словом "наследование".

Класс `Employee` является суперклассом. Это не значит, что он имеет превосходство над своим подклассом или обладает более широкими функциональными возможностями. *На самом деле все наоборот*: подкласс *шире*, чем суперкласс. Например, анализируя код класса `Manager`, легко увидеть, что он инкапсулирует больше данных и содержит больше методов, чем его суперкласс `Employee`.



На заметку! Префиксы *супер-* и *под-* пришли в программирование из теории множеств. Множество всех сотрудников содержит в себе множество всех менеджеров. В этом случае говорят, что множество сотрудников является *супермножеством* (superset) по отношению к множеству менеджеров. Иначе говоря, множество всех менеджеров является *подмножеством* (subset) множества всех сотрудников.

Класс `Manager` имеет новое поле, в котором хранится размер премии, и новый метод, позволяющий задавать эту величину.

```
class Manager extends Employee
{
   ...
   public void setBonus(double b)
   {
      bonus = b;
   }
   private double bonus;
}
```

В этих методах и полях нет ничего особенного. Имея объект `Manager`, можно просто применять метод `setBonus()`.

```
Manager boss = ...;
boss.setBonus(5000);
```

Разумеется, с помощью объекта `Employee` метод `setBonus()` вызвать невозможно — его нет среди методов, определенных в этом классе. Однако методы `getName()` и `getHireDay()` можно вызывать, пользуясь объектами `Manager`, поскольку даже, несмотря на то, что они не определены в классе `Manager`, они наследуются от суперкласса `Employee`.

Поля `name`, `salary` и `hireDay` также наследуются от суперкласса. Каждый объект `Manager` имеет четыре поля: `name`, `salary`, `hireDay` и `bonus`.

Определяя подкласс посредством расширения суперкласса, нужно только указать лишь отличия между подклассом и суперклассом. Проектируя классы, следует помещать методы общего назначения в суперкласс, а более специальные методы — в подкласс. Такое выделение функциональных свойств общего назначения путем их размещения в суперклассе широко распространено в объектно-ориентированном программировании.

Однако некоторые методы суперкласса не подходят для подкласса `Manager`. В частности, метод `getSalary` должен возвращать сумму базовой зарплаты и премии. Следовательно, нужно *переопределить* метод, т.е. реализовать новый метод, *замещающий* метод суперкласса.

```
class Manager extends Employee
{
   ...
   public void getSalary()
   {
      ...
   }
   ...
}
```

Как реализовать этот метод? На первый взгляд это просто — нужно лишь вернуть сумму полей `salary` и `bonus`.

```
public double getSalary()
{
   return salary + bonus; // Не работает.
}
```

Однако выясняется, что этот способ не работает. Метод `getSalary()` класса `Manager` *не имеет доступа к приватным полям суперкласса.* Другими словами, метод `getSalary()` класса `Manager` не может непосредственно обратиться к `salary`, несмотря на то, что каждый объект `Manager` имеет поле с таким именем. Только методы класса `Employee` имеют доступ к приватным полям суперкласса. Если методам класса `Manager` нужен доступ к приватным полям, они должны сделать то же, что и любой другой метод, — использовать общедоступный интерфейс, в данном случае — метод `getSalary()` класса `Employee`.

Итак, попробуем все сначала. Вместо непосредственного обращения к полю `salary` нужно вызвать метод `getSalary()`.

```
public double getSalary()
{
   double baseSalary = getSalary(); // Все еще не работает.
   return baseSalary + bonus;
}
```

Проблема заключается в том, что метод `getSalary()` вызывает сам себя, поскольку в классе `Manager` есть свой метод с таким именем (именно его мы пытаемся реализовать). Возникает бесконечная цепочка вызовов одного и того же метода, что приводит к аварийному завершению программы.

Нужно явно указать, что мы хотим вызвать метод `getSalary()` из суперкласса `Employee`, а не из текущего класса. Для этой цели используется специальное ключевое слово `super`. Приведенное ниже выражение вызывает метод `getSalary()` именно из класса `Employee`.

```
super.getSalary()
```

Вот как выглядит правильный вариант метода `getSalary()` для класса `Manager`.

```
public double getSalary()
{
   double baseSalary = super.getSalary();
   return baseSalary + bonus;
}
```



На заметку! Некоторые программисты считают ключевое слово `super` аналогом ссылки `this`. Однако эта аналогия не вполне точна — ключевое слово `super` не означает ссылку на объект. Например, его значение нельзя присвоить другой объектной переменной. Это слово лишь сообщает компилятору, что нужно вызвать метод суперкласса.

Как видим, подкласс может *добавлять* поля, а также *добавлять* и *переопределять* методы суперкласса, причем при наследовании ни одно поле или метод из класса не удаляется.



На заметку! Для вызова метода суперкласса в языке Java используется ключевое слово `super`. В языке C++ для этого можно указывать имя суперкласса вместе с операцией `::`. Например, метод `getSalary()` класса `Manager` в C++ можно было бы вызвать с помощью выражения `Employee::getSalary()` вместо `super.getSalary()`.

В заключение напишем конструктор класса:

```
public Manager(String n, double s, int year, int month, int day)
{
   super(n, s, year, month, day);
   bonus = 0;
}
```

Здесь ключевое слово `super` имеет другой смысл. Приведенное ниже выражение означает вызов конструктора суперкласса `Employee` с параметрами `n`, `s`, `year`, `month` и `day`:

```
super(n, s, year, month, day);
```

Поскольку конструктор класса `Manager` не имеет доступа к приватным полям класса `Employee`, он должен инициализировать их, вызывая другой конструктор с помощью ключевого слова `super`. Вызов, содержащий обращение `super`, должен быть первым оператором в конструкторе подкласса.

Если конструктор подкласса не вызывает явно ни одного конструктора суперкласса, то автоматически вызывается конструктор по умолчанию, принадлежащий данному суперклассу. Если в суперклассе нет конструктора по умолчанию, а конструктор подкласса не вызывает явно другой конструктор суперкласса, то компилятор Java отобразит сообщение об ошибке.



На заметку! Напомним, что ключевое слово `this` применяется в двух случаях: для указания ссылки на неявный параметр и для вызова другого конструктора того же класса. Аналогично ключевое слово `super` используется в двух ситуациях: для вызова метода суперкласса и конструктора суперкласса. При вызове конструкторов ключевые слова `this` и `super` имеют почти одинаковый смысл. Вызов конструктора должен быть первым оператором в другом конструкторе. Параметры конструктора передаются либо другому конструктору того же класса (`this`), либо конструктору суперкласса (`super`).



На заметку! В языке C++ вызов конструктора `super` не применяется. Вместо этого для создания суперкласса используется список инициализации. В C++ конструктор класса `Manager` выглядел бы следующим образом:

```
Manager::Manager(String n, double s, int year, int month, int day) // C++
: Employee(n, s, year, month, day)
{
    bonus = 0;
}
```

После переопределения метода `getSalary()` для объектов `Manager` все менеджеры в дополнение к зарплатам получат премии.

Рассмотрим конкретный пример: создадим объект Manager и установим размер премии.

```
Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
boss.setBonus(5000);
```

Образуем массив, в котором должны храниться три объекта Employee:

```
Employee[] staff = new Employee[3];
```

Заполним его объектами Manager и Employee:

```
staff[0] = boss;
staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
```

Выведем зарплату каждого сотрудника:

```
for (Employee e : staff)
  System.out.println(e.getName() + " " + e.getSalary());
```

В результате выполнения цикла отображаются следующие строки:

```
Carl Cracker 85000.0
Harry Hacker 50000.0
Tommy Tester 40000.0
```

Поскольку staff[1] и staff[2] являются объектами Employee, они выводят базовые зарплаты соответствующих сотрудников. Однако объект staff[0] относится к классу Manager, и его метод getSalary() добавляет к базовой зарплате премию.

Обратите особое внимание на вызов

```
e.getSalary()
```

Здесь вызывается именно тот метод getSalary(), который необходим в данном случае. Обратите внимание, что объявленным типом переменной e является Employee, однако *фактическим* типом объекта, на который может ссылаться переменная e, может быть как класс Employee (когда i=1 или 2), так и класс Manager (когда i=0).

Когда переменная e ссылается на объект Employee, e.getSalary() вызывает метод getSalary() класса Employee. Однако если переменная e ссылается на объект Manager, вызывается метод getSalary() класса Manager. Виртуальной машине известен фактический тип объекта, на который ссылается переменная e, поэтому с вызовом метода проблемы не возникают.

Способность переменной (например, e) ссылаться на объекты, имеющие разные фактические типы, называется *полиморфизмом*. Автоматический выбор нужного метода во время выполнения программы называется *динамическим связыванием* (dynamic binding). Соответствующие вопросы будут обсуждаться далее в этой главе.



На заметку! В языке Java нет необходимости объявлять виртуальный метод. Динамическое связывание выполняется по умолчанию. Если вы хотите, чтобы метод невозможно было переопределить, пометьте его ключевым словом final (оно будет рассматриваться ниже).

В листинге 5.1 представлен код, демонстрирующий отличия в подсчете заработной платы для объектов Employee и Manager.

Листинг 5.1. Содержимое файла `ManagerTest.java`

```
1. import java.util.*;
2.
3.  /**
4.   * Эта программа демонстрирует наследование.
5.   * @version 1.21 2004-02-21
6.   * @author Cay Horstmann
7.   */
8.  public class ManagerTest
9.  {
10.   public static void main(String[] args)
11.   {
12.      // Создаем объект Manager.
13.      Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
14.      boss.setBonus(5000);
15.
16.      Employee[] staff = new Employee[3];
17.
18.      // Заполнение массива staff объектами Manager и Employee
19.
20.      staff[0] = boss;
21.      staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
22.      staff[2] = new Employee("Tommy Tester", 40000, 1990, 3, 15);
23.
24.      // Вывод информации обо всех объектах Employee
25.      for (Employee e : staff)
26.        System.out.println("name=" + e.getName() + ",salary=" + e.getSalary());
27.   }
28. }
29.
30. class Employee
31. {
32.   public Employee(String n, double s, int year, int month, int day)
33.   {
34.      name = n;
35.      salary = s;
36.      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
37.      hireDay = calendar.getTime();
38.   }
39.
40.   public String getName()
41.   {
42.      return name;
43.   }
44.
45.   public double getSalary()
46.   {
47.      return salary;
48.   }
49.
50.   public Date getHireDay()
51.   {
52.      return hireDay;
53.   }
54.
```

```
55.    public void raiseSalary(double byPercent)
56.    {
57.       double raise = salary * byPercent / 100;
58.       salary += raise;
59.    }
60.
61.    private String name;
62.    private double salary;
63.    private Date hireDay;
64.  }
65.
66. class Manager extends Employee
67. {
68.   /**
69.      @param n Имя сотрудника
70.      @param s Зарплата
71.      @param year Год найма
72.      @param month Месяц найма
73.      @param day День найма
74.   */
75.    public Manager(String n, double s, int year, int month, int day)
76.    {
77.       super(n, s, year, month, day);
78.       bonus = 0;
79.    }
80.
81.    public double getSalary()
82.    {
83.       double baseSalary = super.getSalary();
84.       return baseSalary + bonus;
85.    }
86.
87.    public void setBonus(double b)
88.    {
89.       bonus = b;
90.    }
91.
92.    private double bonus;
93. }
```

Иерархия наследования

Наследование не обязательно ограничивается одним уровнем классов. Например, на основе класса `Manager` можно создать подкласс `Executive`. Совокупность всех классов, производных от общего суперкласса, называется *иерархией наследования*. Условно иерархия наследования показана на рис. 5.1. Путь от конкретного класса к его потомкам в иерархии называется *цепочкой наследования*.

Обычно для класса существует несколько цепочек наследования. На основе класса `Employee` можно создать подкласс `Programmer` или класс `Secretary`, причем они будут совершенно независимыми как от класса `Manager`, так и друг от друга. Процесс формирования подклассов можно продолжать сколь угодно долго.



На заметку! В языке Java множественное наследование не поддерживается. Задачи, для которых в других языках применяется множественное наследование, в Java решаются с помощью механизма интерфейсов (интерфейсы будут рассматриваться в следующей главе).

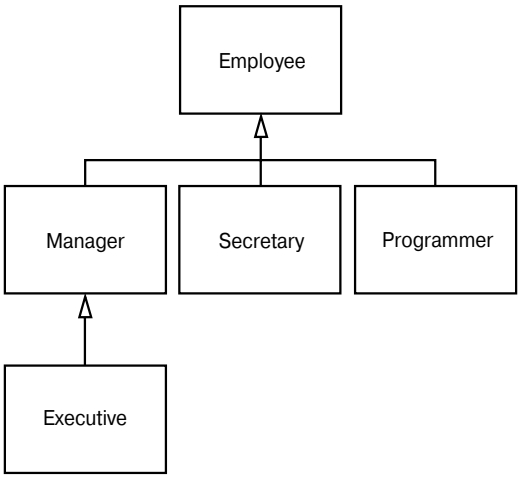


Рис. 5.1. Иерархия наследования для класса `Employee`

Полиморфизм

Существует простое правило, позволяющее определить, стоит ли в конкретной ситуации применять наследование или нет. Если между объектами существует отношение "является" ("is-a"), то каждый объект подкласса является объектом суперкласса. Например, каждый менеджер является сотрудником. Следовательно, имеет смысл сделать класс `Manager` подклассом класса `Employee`. Естественно, обратное утверждение неверно — не каждый сотрудник является менеджером.

Другой способ — *принцип подстановки*. Этот принцип гласит, что объект подкласса можно использовать вместо любого объекта суперкласса.

Например, объект подкласса можно присвоить переменной суперкласса:

```
Employee e;
e = new Employee(...);    // Объект класса Employee.
e = new Manager(...);     // Можно и так.
```

В языке Java объектные переменные являются *полиморфными*. Переменная типа `Employee` может ссылаться как на объект `Employee`, так и на объект любого подкласса класса `Employee` (например, `Manager`, `Executive`, `Secretary` и т.п.).

Применение этого принципа продемонстрировано в листинге 5.1.

```
Manager boss = new manager(...);
Employee[] staff = new Employee[3];
staff[0] = boss;
```

Здесь переменные `staff[0]` и `boss` ссылаются на один и тот же объект. Однако переменная `staff[0]` рассматривается компилятором только как объект `Employee`. Это значит, что допускается следующий вызов:

```
boss.setBonus(5000);     // OK
```

В то же время выражение, приведенное ниже, некорректно:

```
staff[0].setBonus(5000); // ОШИБКА
```

Дело в том, что переменная `staff[0]` объявлена как объект `Employee`, а метода `setBonus()` в этом классе нет.

Однако присвоить ссылку на объект суперкласса переменной подкласса невозможно. Например, следующий оператор является недопустимым:

```
Manager m = staff[i]; // ОШИБКА
```

Причина очевидна: не все сотрудники являются менеджерами. Если бы это присваивание оказалось выполненным, и переменная `m` могла бы ссылаться на объект `Employee`, который менеджером не является, то впоследствии оказался бы возможным вызов метода `m.setBonus(...)`, что привело бы к ошибке при выполнении программы.



Внимание! В языке Java массив ссылок на объекты подкласса можно преобразовать в массив ссылок на объекты суперкласса. При этом выполнять явным образом приведение типов не требуется. В качестве примера рассмотрим массив ссылок на объекты `Manager`:

```
Manager[] managers = new Manager[10];
```

Вполне допустимо преобразовать его в массив `Employee[]`:

```
Employee[] staff = managers; // OK
```

Действительно, каждый менеджер является сотрудником, аналогично объект `Manager` содержит все поля и методы, присутствующие в объекте `Employee`. Однако тот факт, что переменные `managers` и `staff` ссылаются на один и тот же массив, может вызвать неприятные последствия. Рассмотрим следующее выражение:

```
staff[0] = new Employee("Harry Hacker", ...);
```

Компилятор допускает подобное присваивание. Но `staff[0]` и `manager[0]` — это одна и та же ссылка, поэтому ситуация выглядит так, как будто мы пытаемся присвоить ссылку на объект, описывающий рядового сотрудника, переменной, соответствующей объекту менеджера. В результате формально становится возможным вызов `managers[0].setBonus(1000)`, который повлечет обращение к несуществующему методу и разрушение содержимого памяти.

Для того чтобы предотвратить подобную ситуацию, в созданном массиве запоминается тип элементов и отслеживается допустимость хранящихся ссылок. Например, если массив создан с помощью выражения `new Manager[10]`, то он рассматривается как массив элементов `Manager`, и попытка записать в него ссылку на объект `Employee` приведет к возникновению исключения `ArrayStoreException`.

Динамическое связывание

Важно понимать, что происходит при вызове метода, принадлежащего некоторому объекту. При этом выполняются следующие действия.

1. Компилятор проверяет объявленный тип объекта и имя метода. Предположим, происходит вызов метода `x.f(args)`, причем неявный параметр `x` объявлен как экземпляр класса `C`. Заметим, что может существовать несколько методов с именем `f`, имеющих разные типы параметров (например, `f(int)` и метод `f(String)`). Компилятор нумерует все методы с именем `f` в классе `C` и все общедоступные методы с именем `f` в суперклассах класса `C`.

   Теперь компилятор знает всех возможных кандидатов при вызове метода.

2. Затем компилятор определяет типы параметров, указанных при вызове метода. Если среди всех методов с именем `f` есть только один метод, типы параметров которого совпадают с указанными, происходит его вызов. Этот процесс называется *разрешением перегрузки*. Например, при вызове `x.f("Hello")` компилятор

выберет метод f(String), а не метод f(int). Ситуация может осложниться вследствие преобразования типов (int — в double, Manager — в Employee и т.д.). Если компилятор не находит ни одного метода с подходящим набором параметров или в результате преобразования типов возникает несколько методов, соответствующих данному вызову, выдается сообщение об ошибке.

Теперь компилятор знает имя и типы параметров метода, подлежащего вызову.



На заметку! Напомним, что имя метода и список типов его параметров образуют *сигнатуру* метода. Например, методы f(int) и f(String) имеют одинаковые имена, но разные сигнатуры. Если в подклассе определен метод, сигнатура которого совпадает с сигнатурой некоторого метода суперкласса, то метод подкласса замещает его собой.

Однако возвращаемое значение в сигнатуре не учитывается. В версиях, предшествующих Java SE 5.0, тип возвращаемого значения переопределяемого и переопределяющего методов должны были совпадать. Однако теперь допускается, чтобы в подклассе в качестве возвращаемого значения использовался подтип первоначально выбранного типа. Предположим, например, что в классе Employee определен метод getBuddy().

```
public Employee getBuddy() { ... }
```

В подклассе Manager этот метод переопределен следующим образом:

```
public Manager getBuddy() { ... } // Корректно для JDK 5.0
```

При этом считается, что для методов getBuddy() определены *ковариантные* возвращаемые типы.

3. Если метод является приватным (private), статическим (static), финальным (final) или конструктором, компилятор точно знает, как его вызвать. (Модификатор final описывается в следующем разделе.) Такой процесс называется *статическим связыванием*. В противном случае метод, подлежащий вызову, определяется по фактическому типу неявного параметра и во время выполнения программы используется *динамическое связывание*. В нашем примере компилятор сгенерировал бы вызов метода f(String) с помощью динамического связывания.

4. Если при выполнении программы для вызова метода используется динамическое связывание, виртуальная машина должна вызвать версию метода, соответствующую *фактическому* типу объекта, на который ссылается переменная x. Предположим, что объект имеет фактический тип D, являющийся суперклассом класса C. Если в классе D определен метод f(String), то вызывается именно он. Если нет, то поиск метода f(String), подлежащего вызову, выполняется в суперклассе и т.д.

   На поиск вызываемого метода уходит слишком много времени, поэтому виртуальная машина заранее создает для каждого класса *таблицу методов*, в которой перечисляются сигнатуры всех методов и фактические методы, подлежащие вызову. При вызове метода виртуальная машина просто просматривает таблицу методов. В нашем примере виртуальная машина проверяет таблицу методов класса D и обнаруживает метод f(String), подлежащий вызову. Такими методами могут быть D.f(String) или X.f(String), если X — некоторый суперкласс класса D.

   С этим сценарием связана одна особенность. Если вызывается метод super.f(args), то компилятор просматривает таблицу методов суперкласса неявного параметра.

Рассмотрим подробно вызов метода e.getSalary() в листинге 5.1. Переменная e имеет тип Employee. В этом классе есть только один метод getSalary(), у которого нет параметров. Следовательно, в данном случае можно не беспокоиться о разрешении перегрузки.

Поскольку при объявлении метода getSalary() не использовались ключевые слова private, static или final, он связывается динамически. Виртуальная машина создает таблицу методов классов Employee и Manager. Таблица класса Employee показывает, что все методы определены в самом классе.

```
Employee:
   getName() -> Employee.getName()
   getSalary() -> Employee.getSalary()
   getHireDday() -> Employee.getHireDay()
   raiseSalary(double) -> Employee.raiseSalary(double)
```

На самом деле это не совсем так. Как мы увидим позднее, класс Employee имеет суперкласс Object, от которого он наследует большое количество методов; их мы пока будем игнорировать.

Таблица методов класса Manager имеет несколько иной вид. В этом классе три метода наследуются, один метод — переопределяется и один — добавляется.

```
Manager:
   getName() -> Employee.getName()
   getSalary() -> Manager.getSalary()
   getHireDday() -> Employee.getHireDay()
   raiseSalary(double) -> Employee.raiseSalary(double)
   setBonus(double) -> Manager.setBonus(double)
```

Во время выполнения программы вызов метода e.getSalary() осуществляется следующим образом.

1. Сначала виртуальная машина загружает таблицу, соответствующую реальному типу переменной e. Это может быть таблица для Employee, Manager или другого подкласса класса Employee.
2. Затем виртуальная машина определяет класс, в котором определен метод getSalary() с соответствующей сигнатурой. Теперь метод, подлежащий вызову, известен.
3. Виртуальная машина вызывает этот метод.

Динамическое связывание обладает одной важной особенностью: оно позволяет модифицировать программы без перекомпиляции их кодов. Это делает программы *динамически расширяемыми*. Предположим, в программу добавлен новый класс Executive, и переменная e может ссылаться на объект этого класса. Код, содержащий вызов метода e.getSalary(), заново компилировать не нужно. Если переменная e ссылается на объект типа Executive, автоматически вызывается метод Executive.getSalary().



Внимание! При переопределении метода область видимости метода подкласса должна быть не меньше области видимости переопределяемого метода суперкласса. В частности, если при объявлении метода суперкласса использовалось ключевое слово public, метод подкласса также должен быть объявлен как public. Программисты часто ошибаются, забывая указать спецификатор public при описании метода подкласса. В этих случаях компилятор сообщает о том, что привилегии доступа к данным ограничены.

Предотвращение наследования: финальные классы и методы

Иногда наследование является нежелательным. Классы, которые нельзя расширить, называются *финальными*. Для того чтобы отметить этот факт, в определении класса используется ключевое слово `final`. Предположим, например, что мы хотим предотвратить создание подклассов класса `Executive`. В этом случае класс определяется следующим образом:

```
final class Executive extends Manager
{
   ...
}
```

Отдельный метод класса также может быть финальным. Такой метод не может быть переопределен в подклассах. (Все методы финального класса автоматически являются финальными.) Пример объявления финального метода приведен ниже.

```
class Employee
{
   ...
   public final String getName()
   {
     return name;
   }
   ...
}
```



На заметку! Напомним, что с помощью модификатора `final` могут также описываться поля, являющиеся константами. После создания объекта значение такого поля нельзя изменить. Однако если класс объявлен как `final`, финальными становятся только методы; поля в константы не превращаются.

Существует единственный аргумент в пользу применения ключевого слова `final` при объявлении метода или класса: это позволяет гарантировать неизменность семантики в подклассе. Так, например, в классе `Calendar` методы `getTime()` и `setTime()` являются финальными. Поступая подобным образом, разработчики берут на себя ответственность за корректность преобразования содержимого объекта `Date` в состояние календаря. В подклассах невозможно изменить принцип преобразования. В качестве другого примера можно привести финальный класс `String`. Создавать подклассы этого класса запрещено. Поэтому если у вас есть переменная типа `String`, то вы можете быть уверены, что она ссылается именно на строку и ни на какой другой объект.

Некоторые программисты считают, что ключевое слово `final` надо применять при объявлении всех методов. Исключением являются только те случаи, когда есть веские основания для использования полиморфизма. Действительно, в методах C++ и C# для применения полиморфизма надо принять специальные меры. Может быть, указанное правило слишком жесткое, но, несомненно, что при формировании иерархии классов необходимо серьезно заботиться об использовании финальных методов.

На заре развития Java некоторые программисты пытались использовать ключевое слово `final` для того, чтобы исключить накладные расходы, вызываемые динамическим связыванием. Если метод не переопределяется и размеры его невелики, компилятор применяет процедуру оптимизации, которая состоит в непосредственном встраивании кода. Например, вызов метода `e.getName()` заменяется обращением к полю `e.name`. Такое преобразование достаточно эффективно, так как ветвление про-

граммы несовместимо с упреждающей загрузкой команд, применяемой в процессорах. Следует заметить, что если метод `getName()` будет переопределен, компилятор не сможет непосредственно включить его код, поскольку ему станет неизвестно, какой из методов должен быть вызван в процессе выполнения программы.

К счастью, компонент виртуальной машины, называемый *оперативным компилятором* (just-in-time — JIT), может выполнять оптимизацию более эффективно, чем обыкновенный компилятор. Ему в точности известно, какие классы расширяют данный класс, и он может проверить, действительно ли метод переопределяется. Если размеры метода невелики, он часто вызывается и не переопределен, динамический компилятор выполнит непосредственное встраивание его кода. Но что произойдет, если виртуальная машина загрузит другой подкласс, который переопределяет встроенный метод? Тогда оптимизатор должен отменить встраивание кода. В таких ситуациях выполнение программы замедляется, но возникают они редко.



На заметку! В языке C++ методы по умолчанию динамически не связываются, поэтому их можно объявлять встроенными (inline) и заменять их вызовы соответствующими кодами. Однако механизма, позволяющего предотвратить замещение метода суперкласса методом подкласса, в C++ нет. В программе на этом языке можно создать класс, из которого нельзя породить дочерние классы, но для этого нужно выполнить ряд сложных действий, да и, кроме того, аргументы в пользу формирования такого класса сомнительны. (При желании вы можете реализовать его в качестве упражнения. Подсказка: воспользуйтесь виртуальным базовым классом.)

Приведение типов

В главе 3 мы рассмотрели процесс принудительного преобразования одного типа в другой, называемый *приведением типов* (casting). Для этого в языке Java предусмотрена специальная нотация. Например, при выполнении следующего фрагмента кода значение переменной `x` преобразуется в целое число путем отбрасывания дробной части:

```
double x = 3.405;
int nx = (int) x;
```

Точно так же, как иногда возникает потребность преобразования числа с плавающей точкой в целое, бывает необходимо конвертировать объектную ссылку из одного класса в другой. Чтобы выполнить приведение объектной ссылки, используются такие же синтаксические конструкции, как и для числовых выражений. Имя требуемого класса надо поместить в скобки и поставить перед объектной ссылкой, которую нужно преобразовать. Пример соответствующего выражения приведен ниже.

```
Manager boss = (Manager) staff[0];
```

Для приведения типов существует только одна причина — необходимость использовать все возможности объекта после того, как его фактический тип был на время забыт. Например, в классе `ManagerTest` массив `staff` представляет собой массив объектов `Employee`. Этот тип выбран, поскольку *некоторые* элементы массива хранят информацию об обычных сотрудниках. Чтобы получить доступ ко всем новым полям класса `Manager`, нам может понадобиться привести тип некоторых элементов массива к типу `Manager`. (В примере, рассмотренном выше, мы предприняли специальные меры, чтобы избежать приведения типов. Мы инициализировали переменную `boss` объектом `Manager`, перед тем как поместить ее в массив. Для того чтобы задать размер премии, нужно знать правильный тип объекта.)

Как известно, в языке Java каждая объектная переменная имеет тип. Тип объектной переменной определяет разновидность объектов, на которые может ссылаться эта переменная и что она может делать. Например, переменная `staff[1]` ссылается на объект `Employee` (поэтому она может ссылаться и на объекты `Manager`).

В процессе работы компилятор проверяет, не обещаете ли вы слишком много при сохранении значения в переменную. Если переменной суперкласса присваивается объект подкласса, возможности класса сужаются, и компилятор без проблем позволяет программисту сделать это. Если, наоборот, объект суперкласса присваивается переменной подкласса, возможности класса расширяются, поэтому программист должен подтвердить это с помощью обозначения, предназначенного для приведения типов, указав в скобках имя подкласса. Таким образом, виртуальная машина получает возможность контролировать действия программиста в процессе выполнения программы.

Что произойдет, если попытаться осуществить приведение типов вниз по цепочке наследования и попробовать обмануть компилятор?

```
Manager boss = (Manager) staff[1]; // ОШИБКА
```

При выполнении программы система обнаружит несоответствие и сгенерирует исключение `ClassCastException`. Если его не перехватить, работа программы будет прервана. Итак, перед приведением типов следует проверить его корректность. Для этого нужно использовать операцию `instanceof`. Например:

```
if (staff[1] instanceof Manager)
{
   boss = (Manager) staff[1];
   ...
}
```

В заключение заметим, что компилятор не позволит выполнить некорректное приведение типов. Например, наличие в тексте программы представленной ниже строки приведет к ошибке на этапе компиляции, поскольку класс `Date` не является подклассом класса `Employee`:

```
Date c = (Date) staff[1];
```

Таким образом, можно сформулировать основные правила приведения типов.

- Приведение типов можно выполнять только в иерархии наследования.
- Чтобы проверить корректность приведения суперкласса к подклассу, используйте операцию `instanceof`.



На заметку! Если в приведенном ниже выражении `x` содержит значение `null`, исключение не будет сгенерировано:

```
x instanceof C
```

Выражение лишь вернет значение `false`. Это вполне разумно. Поскольку `null`-ссылка не относится ни к одному объекту, она, очевидно, не относится ни к одному объекту `C`.

Часто приведение типа объекта — не самое лучшее решение. В нашем примере выполнять преобразование объекта `Employee` в объект `Manager` не обязательно. Метод `getSalary()` прекрасно работает с обоими типами, поскольку динамическое связывание автоматически определяет правильный метод.

Приведение типов необходимо лишь тогда, когда надо использовать уникальный метод, который есть только в классе `Manager`, например `setBonus()`. Если вы, работая с объектом `Employee`, вдруг захотите вызвать метод `setBonus()`, задайте себе вопрос:

не свидетельствует ли это о недостатках суперкласса? Возможно, следует пересмотреть структуру суперкласса и добавить в него метод `setBonus()`. Помните, чтобы программа завершила свою работу, достаточно одного-единственного необрабатываемого исключения `ClassCastException`. В общем, лучше минимизировать приведение типов и применение операции `instanceof`.



На заметку! В языке Java для приведения типов используется устаревший синтаксис языка C, однако работает он как безопасная операция динамического приведения типов `dynamic_cast` языка C++. Например, приведенные ниже две строки кода, написанные на разных языках, почти эквивалентны.

```
Manager boss = (Manager)staff[1]; //Язык Java
Manager* boss = dynamic_cast<Manager*>(staff[1]); // Язык C++
```

Между ними существует одно различие. Если приведение типов выполняется неверно, оно не порождает нулевой объект, а генерирует исключение. В этом смысле приведение типов в языке Java напоминает приведение *ссылок* в C++. Это очень значительный недостаток. В C++ и контроль, и преобразование типов можно выполнить в одной операции:

```
Manager* boss = dynamic_cast<Manager*>(staff[1]); // Язык C++
if (boss != NULL) ...
```

В языке Java используется комбинация операции `instanceof` и операции приведения типов:

```
if (staff[1] instanceof Manager) //Язык Java
{
   Manager boss = (Manager)staff[1];
   ...
}
```

Абстрактные классы

Поднимаясь по иерархии наследования, классы становятся более универсальными и более абстрактными. В некотором смысле родительские классы, находящиеся в верхней части иерархии, становятся настолько абстрактными, что их рассматривают как основу для разработки других классов, а не как класс, позволяющий создавать конкретные объекты. Расширим, например, иерархию, в которую входит класс `Employee`. Сотрудник — это человек, а человек может быть студентом. Расширим нашу иерархию классов, добавив в нее классы `Person` и `Student`. Отношения между всеми этими классами показаны на рис. 5.2.

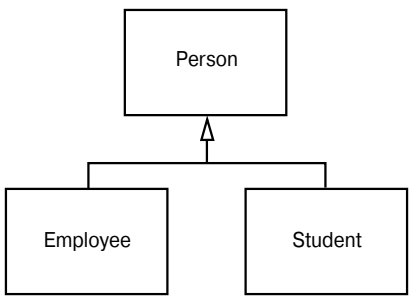


Рис. 5.2. Диаграмма наследования для класса `Person` и его подклассов

Какие проблемы возникают при таком высоком уровне абстракции? Существуют определенные атрибуты, характерные для каждого человека, например имя. И студенты, и сотрудники имеют имена, и при создании общего суперкласса понадобится перенести метод `getName()` на более высокий уровень в иерархии наследования.

Добавим теперь новый метод `getDescription()`, предназначенный для создания краткой характеристики человека, например:

```
сотрудник с зарплатой $50,000.00
студент, изучающий вычислительную технику
```

Для классов `Employee` и `Student` этот метод реализуется довольно просто. Однако какую информацию можно поместить в класс `Person`? В нем ничего нет, кроме имени. Разумеется, можно было бы реализовать метод `Person.getDescription()`, возвращающий пустую строку. Однако есть способ получше. Используя ключевое слово `abstract`, можно вовсе не реализовывать этот метод:

```
public abstract String getDescription();
   // Реализация не требуется.
```

Для большей ясности класс, содержащий один или несколько абстрактных методов, можно объявить абстрактным:

```
abstract class Person
{
   ...
   public abstract String getDescription();
}
```

Кроме абстрактных методов, абстрактные классы могут содержать конкретные поля и методы. Например, класс `Person` хранит имя человека и содержит конкретный метод, возвращающий это имя.

```
abstract class Person
{
   public Person(String n)
   {
      name = n;
   }

   public abstract String getDescription();

   public String getName()
   {
      return name;
   }

   private String name;
}
```



Совет. Многие программисты не осознают, что абстрактные классы могут содержать в себе конкретные методы. Всегда следует стремиться перемещать общие поля и методы (будь они абстрактные или нет) в суперкласс (абстрактный или нет).

Абстрактные методы представляют собой прототипы методов, реализованных в подклассах. Расширяя абстрактный класс, можно оставить некоторые или все абстрактные методы неопределенными. При этом подкласс также станет абстрактным. Кроме того, можно определить все методы. Тогда подкласс не будет абстрактным.

Определим класс Student, расширяющий абстрактный класс Person и реализующий метод getDescription(). Поскольку ни один из методов в классе Student не является абстрактным, нет никакой необходимости объявлять сам класс абстрактным.

Класс может быть объявлен абстрактным, даже если он не содержит ни одного абстрактного метода.

Создать объекты абстрактного класса невозможно. Например, приведенное ниже выражение ошибочно:

```
new Person("Vince Vu");
```

Однако можно создать объекты конкретного подкласса.

Отметим, что можно создавать *объектные переменные* абстрактных классов, однако такие переменные должны ссылаться на объект неабстрактного класса. Рассмотрим следующую строку кода:

```
Person p = new Student("Vince Vu", "Economics");
```

Здесь p — переменная абстрактного типа Person, ссылающаяся на экземпляр неабстрактного подкласса Student.



На заметку! В языке C++ абстрактный метод называется *чистой виртуальной функцией*. Его обозначение оканчивается символами =0.

```
class Person // Язык C++
{
  public:
    virtual string getDescription() = 0;
    ...
};
```

Класс в C++ является абстрактным, если он содержит хотя бы одну чистую виртуальную функцию. В C++ нет специального ключевого слова, обозначающего абстрактные классы.

Определим конкретный подкласс Student, расширяющий абстрактный класс Person.

```
class Student extends Person
{
  public Student(String n, String m)
  {
    super(n);
    major = m;
  }

  public String getDescription()
  {
    return "студент, изучающий " + major;
  }

  private String major;
}
```

В этом подклассе определяется метод getDescription(). Следовательно, все методы в классе Student являются конкретными, и класс больше не является абстрактным.

В программе, представленной в листинге 5.2, определен абстрактный суперкласс Person и два конкретных подкласса Employee и Student. Заполним массив типа Person ссылками на экземпляры классов Employee и Student.

```
Person[] people = new Person[2];
people[0] = new Employee(...);
people[1] = new Student(...);
```

Затем выведем имена и характеристики этих объектов.

```
for (Person p : people){
  System.out.println(p.getName() + ", " + p.getDescription());
```

Некоторых читателей присутствие вызова `p.getDescription()` может озадачить. Не относится ли он к неопределенному методу? Учтите, что переменная `p` никогда не ссылается ни на один объект абстрактного класса `Person`, поскольку создать такой объект попросту невозможно. Переменная `p` всегда ссылается на объект конкретного подкласса, например `Employee` или `Student`. Для этих объектов метод `getDescriprion()` определен.

Можно ли пропустить в классе `Person` все абстрактные методы и определить метод `getDescription()` в подклассах `Employee` и `Student`? Это не будет ошибкой, но тогда метод `getDescription()` нельзя будет вызвать с помощью переменной `p`. Компилятор гарантирует, что вызываются только методы, определенные в классе.

Абстрактные методы представляют собой важное понятие языка Java. В основном они применяются при создании *интерфейсов*, которые будут детально рассмотрены в главе 6.

Листинг 5.2. Содержимое файла `PersonTest.java`

```
1. import java.text.*;
2. import java.util.*;
3.
4. /**
5.    * Эта программа демонстрирует абстрактные классы.
6.    * @version 1.01 2004-02-21
7.    * @author Cay Horstmann
8.    */
9. public class PersonTest
10.{
11.   public static void main(String[] args)
12.   {
13.      Person[] people = new Person[2];
14.
15.      // Заполняем массив объектами Student и Employee.
16.      people[0] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
17.      people[1] = new Student("Maria Morris", "computer science");
18.
19.      // Вывод имен и описаний всех объектов Person
20.      for (Person p : people)
21.         System.out.println(p.getName() + ", " + p.getDescription());
22.   }
23.}
24.
25.abstract class Person
26.{
27.   public Person(String n)
28.   {
29.      name = n;
30.   }
31.
```

```
32.   public abstract String getDescription();
33.
34.   public String getName()
35.   {
36.      return name;
37.   }
38.
39.   private String name;
40. }
41.
42. class Employee extends Person
43. {
44.   public Employee(String n, double s, int year, int month, int day)
45.   {
46.      super(n);
47.      salary = s;
48.      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
49.      hireDay = calendar.getTime();
50.   }
51.
52.   public double getSalary()
53.   {
54.      return salary;
55.   }
56.
57.   public Date getHireDay()
58.   {
59.      return hireDay;
60.   }
61.
62.   public String getDescription()
63.   {
64.      return String.format("сотрудник с зарплатой $%.2f", salary);
65.   }
66.
67.   public void raiseSalary(double byPercent)
68.   {
69.      double raise = salary * byPercent / 100;
70.      salary += raise;
71.   }
72.
73.   private double salary;
74.   private Date hireDay;
75. }
76.
77. class Student extends Person
78. {
79.   /**
80.      @param n имя студента
81.      @param m специализация студента
82.   */
83.   public Student(String n, String m)
84.   {
85.      // Строка n передается конструктору суперкласса
86.      super(n);
87.      major = m;
88.   }
```

```
89.   public String getDescription()
90.   {
91.      return "студент, изучающий " + major;
92.   }
93.
94.   private String major;
95. }
```

Защищенный доступ

Как известно, поля в классе желательно объявлять как `private`, а некоторые методы — как `public`. Любые приватные (`private`) элементы невидимы из других классов. Это утверждение справедливо и для подклассов: подкласс не имеет доступа к приватным полям суперкласса.

Иногда нужно ограничивать доступ к некоторому методу и открывать его лишь для подклассов. Реже возникает необходимость предоставлять методам подкласса доступ к полям суперкласса. В этом случае элемент класса объявляется защищенным с помощью ключевого слова `protected`. Например, если в суперклассе `Employee` поле `hireDay` объявлено защищенным, а не приватным, методы подкласса `Manager` смогут обращаться к нему непосредственно.

Однако методы класса `Manager` имеют доступ лишь к полям `hireDay`, принадлежащим объектам самого класса `Manager`, но не класса `Employee`. Это ограничение введено для того, чтобы программисты не могли злоупотреблять механизмом защищенного доступа и создавать подклассы лишь для получения доступа к защищенным полям.

На практике пользоваться атрибутом `protected` нужно очень осторожно. Предположим, что ваш класс, в котором есть защищенные поля, используется другими разработчиками. Без вашего ведома другие программисты могут создавать подклассы вашего класса и тем самым получать доступ к защищенным полям. Теперь вы уже не сможете изменять реализацию своего класса, не уведомив других программистов. Это противоречит духу объектно-ориентированного программирования, требующего инкапсуляции данных.

Защищенные методы более оправданы. Метод класса можно объявить защищенным, чтобы ограничить его использование. Это значит, что методы подклассов (предки которых им известны по определению) могут вызывать этот метод, а методы других классов — нет.

В качестве примера защищенного метода можно привести `clone()` класса `Object` (подробно этот вопрос будет рассмотрен в главе 6).



На заметку! В языке Java защищенные элементы доступны из всех подклассов, а также из других классов того же пакета. Этим Java отличается от языка C++, в котором ключевое слово `protected` имеет несколько иной смысл. Таким образом, в Java ограничения на доступ к защищенным элементам даже менее строг, чем в языке C++.

Итак, в языке Java есть четыре модификатора доступа, управляющих областью видимости.

1. Область видимости ограничена классом (`private`).
2. Область видимости не ограничена (`public`).
3. Область видимости ограничена пакетом и всеми подклассами (`protected`).
4. Область видимости ограничена пакетом (к сожалению, по умолчанию). Никакого модификатора указывать не нужно.

Object: глобальный суперкласс

Класс Object является предком всех классов — каждый класс в языке Java расширяет класс Object. Однако явно отражать этот факт, как в приведенной ниже строке кода, не нужно:

```
class Employee extends Oblects
```

Если суперкласс явно не указан, им считается класс Object. Поскольку в языке Java *каждый* класс расширяет Object, очень важно знать, какими возможностями обладает сам класс Object. В настоящей главе мы пройдемся по самым основным из них, прочие аспекты класса Object рассмотрим в других главах, а остальную информацию читатели могут найти в документации. (Некоторые методы класса Object можно понять лишь после изучения потоков — см. второй том.)

Переменную типа Object можно использовать в качестве ссылки на объект любого типа:

```
Object obj = new Employee("Harry Hacker", 35000);
```

Разумеется, переменная этого класса полезна лишь как средство для хранения значений произвольного типа. Чтобы сделать с этим значением что-то конкретное, нужно знать его исходный тип, а затем выполнить приведение типов:

```
Employee e = (Employee) obj;
```

В языке Java объектами не являются лишь *простые типы*: числа, символы и логические значения.

Все массивы, независимо от того, содержатся ли в его элементах объекты или простые типы, являются объектами, расширяющими класс Object.

```
Employee[] staff = new Employee[10];
obj = staff; // OK
obj = new int[10]; // OK
```



На заметку! В языке C++ аналогичного глобального базового класса нет, хотя, разумеется, любой указатель можно преобразовать в указатель типа void*.

Метод equals()

Метод equals() класса Object проверяет, эквивалентны ли два объекта. Поскольку метод equals() реализован в классе Object, он определяет лишь, ссылаются ли переменные на один и тот же объект. В качестве проверки по умолчанию эти действия вполне оправданы: всякий объект эквивалентен самому себе. Для некоторых классов большего и не требуется. Например, вряд ли кому-то потребуется анализировать два объекта PrintStream и выяснять, отличаются ли они друг от друга. Однако в ряде случаев эквивалентными должны считаться объекты одного типа, имеющие одинаковые состояния.

Рассмотрим в качестве примера объекты, описывающие сотрудников. Очевидно, что они одинаковы, если совпадают имена, размеры заработной платы и даты приема на работу. (Строго говоря, в реальных системах учета информации о сотрудниках более оправдано сравнение идентификационных номеров. Здесь же мы лишь демонстрируем принципы реализации метода equals().)

```
class Employee
{ // ...
  public boolean equals(Object otherObject)
  {
    // Быстрая проверка идентичности объектов.
    if (this == otherObject) return true;

    // Если явный параметр — null, возвращается значение false.
    if (otherObject == null) return false;

    // Если классы не совпадают, они не эквивалентны.
    if (getClass() != otherObject.getClass())
       return false;

    // Теперь мы знаем, что объект otherObject
    // имеет тип Employee и не является null.
    Employee other = (Employee) otherObject;

    // Проверим, хранятся ли в полях объектов идентичные значения.
    return name.equals(other.name)
       && salary = other.salary
       && hireDay.equals(other.hireDay);
  }
}
```

Метод `getClass()` возвращает класс объекта. Этот метод будет подробно обсуждаться далее в этой главе. Чтобы объекты были эквивалентны, они как минимум должны быть объектами одного и того же класса.

Определяя метод `equals()` для подкласса, надо сначала вызывать тот же метод суперкласса. Если проверка даст отрицательный результат, объекты не могут быть идентичными. Если же поля суперкласса совпадают, можно приступать к сравнению полей подкласса.

```
class Manager extends Employee
{
   . . .
   public boolean equals(Object otherObject)
   {
      if (!super.equals(otherObject)) return false;
      // При выполнении super.equals() проверяется,
      // принадлежит ли otherObject тому же классу.
      Manager other = (Manager) otherObject;
      return bonus == other.bonus;
   }
}
```

Проверка эквивалентности объектов и наследование

Как должен работать метод `equals()`, если неявные и явные параметры не принадлежат одному и тому же классу? Это предмет дискуссий. В предыдущем примере метод `equals()` возвращал значение `false`, если классы не совпадают в точности. Однако многие программисты используют следующую проверку:

```
if (!(otherObject instanceof Employee)) return false;
```

При этом остается возможность, что otherObject принадлежит подклассу. Данный подход может привести к возникновению проблем. Спецификация языка Java требует, чтобы метод equals() обладал следующими характеристиками.

1. *Рефлексивность.* Для любой ненулевой ссылки x вызов x.equals(x) должен возвращать значение true.
2. *Симметричность.* Для любых ссылок x и y вызов x.equals(y) должен возвращать значение true тогда и только тогда, когда y.equals(x) возвращает true.
3. *Транзитивность.* Для любых ссылок x, y и z, если вызовы x.equals(y) и y.equals(z) возвращают значение true, вызов x.equals(z) возвращает значение true.
4. *Согласованность.* Если объекты, на которые ссылались переменные x и y не изменяются, то повторный вызов x.equals(y) должен возвращать то же значение.
5. Для любой null-ссылки x вызов x.equals(null) должен возвращать значение false.

Целесообразность данных правил очевидна. Так, например, очевидно, что результаты проверки не должны зависеть от того, вызывается ли в программе x.equals(y) или y.equals(x).

Однако требование симметрии имеет свои особенности в случае, если явный и неявный параметры принадлежат разным классам. Рассмотрим следующий вызов:

```
e.equals(m)
```

Здесь объект e принадлежит классу Employee, а объект m — классу Manager, причем каждый из них содержит одинаковые имена, зарплату и дату приема на работу. Если не выполнить проверку эквивалентности классов, которым принадлежат объекты m и e, метод вернет значение true. Однако это значит, что и обратный вызов, m.equals(e), также должен возвращать значение true, — правило 2 не позволяет ему возвращать значение false или генерировать исключение.

В результате оказывается, что класс Manager неоправданно тесно связан с другими классами. Если метод equals() используется для сравнения с объектом Employee, он не должен принимать во внимание информацию, отличающую менеджера от обычного сотрудника. В данной ситуации операция instanceof выглядит достаточно привлекательно.

Некоторые специалисты считают, что проверка с использованием метода getClass() некорректна, поскольку при этом нарушается принцип подстановки. В пользу данного мнения приводят метод equals() класса AbstractSet, который проверяет, содержат ли два множества одинаковые элементы и расположены ли они в одинаковом порядке. Класс AbstractSet выступает в роли суперкласса для классов TreeSet и HashSet, которые не являются абстрактными. Эти классы используют различные алгоритмы для работы с элементами множества. На практике разработчику необходимо иметь возможность сравнивать любые два множества, независимо от того, как они реализованы.

Следует признать, что данный пример слишком специфичен. В данном случае имело бы смысл объявить метод AbstractSet.equals() финальным, чтобы невозможно было изменить семантику проверки на эквивалентность множеств. (На самом деле при объявлении метода ключевое слово final не указано. Это дает возможность подклассам реализовать более эффективный алгоритм проверки на эквивалентность.)

Таким образом, вырисовываются два сценария.

- Если проверка эквивалентности реализована в подклассе, правило симметричности требует использования метода `getClass()`.
- Если проверка производится средствами суперкласса, можно применять операцию `instanceof`; при этом становится возможной ситуация, когда два объекта разных классов будут признаны эквивалентными.

В примере с сотрудниками и менеджерами мы считаем два объекта эквивалентными, если их поля совпадают. Если в нашем распоряжении есть два объекта `Manager` с одинаковыми именами, заработной платой и датой приема на работу, но с различными суммами премии, мы должны признать объекты разными. Следовательно, мы должны использовать проверку с помощью метода `getClass()`.

Теперь предположим, что для проверки эквивалентности используется идентификационный номер сотрудника. Такая проверка имеет смысл для всех подклассов. В этом случае мы можем применять операцию `instanceof` и объявить метод `Employee.equals()` как `final`.



На заметку! Стандартная библиотека Java содержит более 150 реализаций метода `equals()`. Некоторые из них включают множество операций `instanceof`, вызовов метода `getClass()` и фрагментов кода, предназначенных для обработки исключения `ClassCastException`, другие не выполняют практически никаких действий.

Ниже приведены рекомендации для создания метода `equals()`.

1. Предположим, что явный параметр называется `otherObject`, — впоследствии его тип нужно будет привести к типу другой переменной, которую назовем `other`.
2. Проверить, идентичны ли ссылки `this` и `otheObject`.

   ```
   if (this == otherObject) return true;
   ```

   Это выражение используется для оптимизации проверки. Гораздо быстрее проверить идентичность ссылок, чем сравнивать поля объектов.
3. Выяснить, является ли ссылка `otherObject` нулевой (`null`). Если да, вернуть значение `false`. Эту проверку нужно делать обязательно.

   ```
   if (otherObject == null) return false;
   ```
4. Сравнить классы `this` и `otheObject`. Если семантика проверки может измениться в подклассе, использовать метод `getClass()`.

   ```
   if (getClass() != otherObject.getClass()) return false;
   ```

   Если принцип проверки остается справедливым для всех подклассов, использовать операцию `instanceof`.

   ```
   if (!(otherObject instanceof ClassName)) return false;
   ```
5. Преобразовать объект `otherObject` в переменную требуемого класса.

   ```
   Имя_класса other = (Имя_класса)otherObject;
   ```
6. Теперь нужно сравнить между собой все поля. Для полей простых типов используется операция ==, для объектных полей — метод `equals()`. Если все поля двух объектов совпадают друг с другом, возвращается значение `true`, в противном случае — значение `false`.

   ```
   return поле_1 == other.поле_1
     && поле_2.equals(other.поле_2)
     && ...;
   ```

Если в подклассе вы переопределяете метод `equals()`, в него надо включить вызов `super.equals(other)`.



Совет. Если у вас есть поля типа массива, вы можете использовать статический метод `Arrays.equals()` для проверки эквивалентности соответствующих элементов массивов.



Внимание! Реализуя метод `equals()`, многие программисты допускают стандартную ошибку. Сможете ли вы сказать, какая проблема возникнет при выполнении следующего фрагмента кода?

```
public class Employee
{
    public boolean equals(Employee other)
    {
        return name.equals(other.name)
            && salary == other.salary
            && hireDay.equals(other.hireDay);
    }
    ...
}
```

Этот метод определяет явный тип параметра как `Employee`. В результате он не переопределяет метод `equals()` класса `Object`, а определяет совершенно независимый метод.

Начиная с Java SE 5.0, появилась возможность застраховаться от возникновения подобной ошибки, специальным образом указывая, что разрабатываемый метод призван заместить соответствующий метод суперкласса. Для этой цели используется дескриптор `@Override`.

```
@Override public boolean equals(Object other)
```

Если при этом вы случайно определите новый метод, компилятор вернет сообщение об ошибке. Предположим, что в классе `Employee` присутствует приведенная ниже строка кода:

```
@Override public boolean equals(Employee other)
```

Поскольку данный метод не переопределяет никакого метода, определенного в суперклассе `Object`, будет обнаружена ошибка.



java.util.Arrays 1.2

- `static boolean equals(type[] a, type[] b)` **5.0**

  Возвращает `true`, если массивы имеют одинаковую длину и эквивалентные элементы в соответствующих позициях. Массивы могут иметь компоненты типов `Object`, `int`, `long`, `short`, `char`, `byte`, `boolean`, `float` или `double`.

Метод hashCode()

Хеш-код — это целое число, генерируемое на основе конкретного объекта. Хеш-код можно рассматривать как некоторый шифр: если `x` и `y` — разные объекты, то с высокой степенью вероятности должны различаться результаты вызовов `x.hashCode()` и `y.hashCode()`. В табл. 5.1 приведено несколько примеров хеш-кодов, полученных в результате вызова метода `hashCode()` класса `String`.

В классе `String` для вычисления хеш-кода используется следующий алгоритм:

```
int hash = 0;
for (int i = 0; i < length(); i++)
  hash = 31 * hash + charAt(i);
```

Таблица 5.1. Хеш-коды, полученные с помощью метода `hashCode()`

Строка	Хеш-код
Hello	69609650
Harry	69496448
Hacker	–2141031506

Метод `hashCode()` определен в классе `Object`. Поэтому каждый объект имеет хеш-код, определяемый по умолчанию. В классе `Object` хеш-код вычисляется на основе адреса памяти, занимаемой объектом. Рассмотрим следующий пример:

```
String s = "Ok";
StringBuilder sb = new StringBuilder(s);
System.out.println(s.hashCode() + " " + sb.hashCode());
String t = new String("Ok");
StringBuilder tb = new StringBuilder(t);
System.out.println(t.hashCode() + " " + tb.hashCode());
```

Результаты выполнения данного фрагмента кода приведены в табл. 5.2.

Таблица 5.2. Хеш-коды для объектов `String` и `StringBuffer`

Объект	Хеш-код
`s`	2556
`sb`	20526976
`t`	2556
`tb`	20527144

Обратите внимание, что строкам `s` и `t` соответствуют одинаковые хеш-коды, так как они вычисляются на основе *содержимого* объекта. Для построителей строк `sb` и `tb` хеш-коды различаются. Причина в том, что в классе `StringBuilder` метод `hashCode()` не определен и используется метод `hashCode()` класса `Object`, который определяет хеш-код по адресу занимаемой памяти.

Если вы переопределяете метод `equals()`, вам также следует переопределить и метод `hashCode()` для объектов, которые пользователь может вставлять в хеш-таблицу. Он может быть использован при включении объектов в хеш-таблицу. (Хеш-таблицы будут обсуждаться во втором томе.)

Метод `hashCode()` должен возвращать целое число (которое может быть отрицательным). Для того чтобы обеспечить различие хеш-кодов для разных объектов, достаточно объединить хеш-коды полей экземпляра.

Ниже приведен пример метода `hashCode()` для класса `Employee`.

```
class Employee
{
   public int hashCode()
   {
      return 7 * name.hashCode()
         + 11 * new Double(salary).hashCode()
         + 13 * hireDay.hashCode();
   }
   . . .
}
```

Методы equals() и hashCode() должны быть совместимы: если x.equals(y) возвращает значение true, то результаты выполнения x.hashCode() и y.hashCode() также должны совпадать. Например, если в методе Employee.equals() для определения эквивалентности применяется табельный номер сотрудника, то при вычислении хеш-кода также должен использоваться этот номер, а не имя и не адрес памяти, занимаемый объектом.



Совет. Если у вас есть поля типа массива, вы можете использовать статический метод Arrays.equals() для проверки эквивалентности соответствующих элементов массивов.



java.lang.Object 1.0

- int hashCode()

  Возвращает хеш-код объекта. Хеш-код представляет собой положительное или отрицательное целое число. Эквивалентным объектам должны соответствовать одинаковые хеш-коды.



java.lang.Arrays 1.2

- int hashCode(type[] a)

  Вычисляет хеш-код массива a, который может содержать компоненты типов Object, int, long, short, char, byte, boolean, float или double.

Метод toString()

Еще одним важным методом класса Object является toString(), возвращающий значение объекта в виде строки. В качестве примера можно привести метод toString() класса Point. Он возвращает строку, подобную приведенной ниже.

```
java.awt.Point[x=10,y=20]
```

Большинство (но не все) методы toString() возвращают строку, состоящую из имени класса, за которым указываются значения его полей в квадратных скобках. Ниже приведена реализация метода toString() для класса Employee.

```
public String toString()
{
   return "Employee[name=" + name
      + ",salary=" + salary
      + ",hireDay=" + hireDay
      + "]";
}
```

На самом деле этот метод можно усовершенствовать. Не будем жестко кодировать имя класса в методе toString(), а лишь вызовем метод getClass().getName() и получим строку, содержащую имя класса:

```
public String toString()
{
   return getClass().getName()
      + "[name=" + name
      + ",salary=" + salary
      + ",hireDay=" + hireDay
      + "]";
}
```

Теперь метод `toString()` работает и с подклассами.

Разумеется, программист, создающий подкласс, должен определить свой собственный метод `toString()` и добавить поля подкласса. Если в суперклассе используется вызов `getClass().getName()`, подкласс просто вызывает метод `super.ToString()`. Ниже приведен пример метода `toString()` для класса `Manager`.

```
class Manager extends Employee
{
   . . .
   public String toString()
   {
      return super.toString()
        + "[bonus=" + bonus
        + "]";
   }
}
```

Теперь состояние объекта `Manager` выводится следующим образом:

```
Manager[name=...,salary=...,hireDay=...][bonus=...]
```

Метод `toString()` универсален. Есть важная причина для реализации его в каждом классе: если объект объединяется со строкой с помощью операции +, компилятор языка Java автоматически вызывает метод `toString()`, чтобы получить представление о его текущем состоянии.

```
Point p = new Point(10, 20);
String message = "The current position is = " + p;
   // Автоматически вызывает метод p.String().
```



Совет. Вместо `x.toString()` можно использовать выражение `"" + x`. Конкатенация пустой строки с представлением объекта `x` в виде строки эквивалентна вызову метода `x.toString()`. Такое выражение будет корректным, даже если переменная принадлежит к одному из простых типов.

Предположим, что `x` — произвольный объект и в программе имеется следующая строка:

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

В этом случае метод `println()` вызовет метод `x.toString()` и выведет строку результата.

Метод `toString()`, определенный в классе `Object`, выводит имя класса и адрес объекта. Рассмотрим следующий вызов:

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

После выполнения метода `println()` отображается следующая строка:

```
java.io.PrintStream@2f6684
```

Как видите, разработчики класса `PrintStream` не позаботились о переопределении метода `toString()`.



Внимание! Это неприятно, но массивы наследуют метод `toString()` от `Object`, в результате чего тип массива печатается в архаичном формате. Например:

```
int[] luckyNumbers = { 2, 3, 5, 7, 11, 13 };
String s = "" + luckyNumbers;
```

дает строку `"[I@1a46e30"`. (Префикс `[I` означает массив целых чисел.) Спасением может быть вызов вместо этого статического метода `Arrays.toString()`. Следующий код выдаст строку `"[2, 3, 5, 7, 11, 13]"`:

```
String s = Arrays.toString(luckyNumbers);
```

Чтобы корректно печатать многомерные массивы, используйте `Arrays.deepToString()`.

Метод `toString()` — отличное средство протоколирования. Многие классы из стандартной библиотеки определяют метод `toString()`, так что вы можете получать полезную информацию о состоянии объекта. Так, например, для протоколирования можно применить следующее выражение:

```
System.out.println("Текущая позиция = " + position);
```

Прочитав главу 11, вы узнаете, что существует еще лучшее решение.

```
Logger.global.info("Текущая позиция = " + position);
```



Совет. Мы настоятельно рекомендуем переопределять метод `toString()` в каждом создаваемом вами классе. Это будет полезно как вам, так и программистам, использующим результаты вашей работы.

В листинге 5.3 содержится код методов `equals()`, `hashCode()` и `toString()`, реализованных для классов `Employee` и `Manager`.

Листинг 5.3. Содержимое файла `EqualsTest.java`

```
1. import java.util.*;
2.
3. /**
4.    * Эта программа демонстрирует использование метода equals.
5.    * @version 1.11 2004-02-21
6.    * @author Cay Horstmann
7.    */
8. public class EqualsTest
9. {
10.    public static void main(String[] args)
11.    {
12.       Employee alice1 = new Employee("Alice Adams", 75000, 1987, 12, 15);
13.       Employee alice2 = alice1;
14.       Employee alice3 = new Employee("Alice Adams", 75000, 1987, 12, 15);
15.       Employee bob = new Employee("Bob Brandson", 50000, 1989, 10, 1);
16.
17.       System.out.println("alice1 == alice2: " + (alice1 == alice2));
18.
19.       System.out.println("alice1 == alice3: " + (alice1 == alice3));
20.
21.       System.out.println("alice1.equals(alice3): " + alice1.equals(alice3));
22.
23.       System.out.println("alice1.equals(bob): " + alice1.equals(bob));
```

```
24.
25.      System.out.println("bob.toString(): " + bob);
26.
27.      Manager carl = new Manager("Carl Cracker", 80000, 1987, 12, 15);
28.      Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
29.      boss.setBonus(5000);
30.      System.out.println("boss.toString(): " + boss);
31.      System.out.println("carl.equals(boss): " + carl.equals(boss));
32.      System.out.println("alice1.hashCode(): " + alice1.hashCode());
33.      System.out.println("alice3.hashCode(): " + alice3.hashCode());
34.      System.out.println("bob.hashCode(): " + bob.hashCode());
35.      System.out.println("carl.hashCode(): " + carl.hashCode());
36.   }
37. }
38.
39. class Employee
40. {
41.   public Employee(String n, double s, int year, int month, int day)
42.   {
43.      name = n;
44.      salary = s;
45.      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
46.      hireDay = calendar.getTime();
47.   }
48.
49.   public String getName()
50.   {
51.      return name;
52.   }
53.
54.   public double getSalary()
55.   {
56.      return salary;
57.   }
58.
59.   public Date getHireDay()
60.   {
61.      return hireDay;
62.   }
63.
64.   public void raiseSalary(double byPercent)
65.   {
66.      double raise = salary * byPercent / 100;
67.      salary += raise;
68.   }
69.
70.   public boolean equals(Object otherObject)
71.   {
72.      // Быстрая проверка идентичности объектов.
73.      if (this == otherObject) return true;
74.
75.      // Если явный параметр — null, возвращается значение false.
76.      if (otherObject == null) return false;
77.
```

```
78.     // Если классы не совпадают, они не эквивалентны.
79.     if (getClass() != otherObject.getClass())
80.        return false;
81.
82.     // Теперь мы знаем, что otherObject —
83.     // ненулевой объект Employee.
84.     Employee other = (Employee)otherObject;
85.
86.     // Проверка идентичности значений, записанных в полях.
87.     return name.equals(other.name)
88.        && salary == other.salary
89.        && hireDay.equals(other.hireDay);
90.  }
91.
92.  public int hashCode()
93.  {
94.    return 7 * name.hashCode()
95.      + 11 * new Double(salary).hashCode()
96.      + 13 * hireDay.hashCode();
97.  }
98.
99.  public String toString()
100. {
101.    return getClass().getName()
102.       + "[name=" + name
103.       + ",salary=" + salary
104.       + ",hireDay=" + hireDay
105.       + "]";
106. }
107.
108.  private String name;
109.  private double salary;
110.  private Date hireDay;
111. }
112.
113.class Manager extends Employee
114.{
115.  public Manager(String n, double s,
116.     int year, int month, int day)
117.  {
118.     super(n, s, year, month, day);
119.     bonus = 0;
120.  }
121.
122.  public double getSalary()
123.  {
124.     double baseSalary = super.getSalary();
125.     return baseSalary + bonus;
126.  }
127.
128.  public void setBonus(double b)
129.  {
130.     bonus = b;
131.  }
132.
```

```
133.  public boolean equals(Object otherObject)
134.  {
135.     if (!super.equals(otherObject)) return false;
136.     Manager other = (Manager)otherObject;
137.     // Метод super.equals() проверяет, принадлежат ли объекты
138.     // this и other одному и тому же классу.
139.     return bonus == other.bonus;
140.  }
141.
142.  public int hashCode()
143.  {
144.     return super.hashCode()
145.        + 17 * new Double(bonus).hashCode();
146.  }
147.
148.  public String toString()
149.  {
150.     return super.toString()
151.       + "[bonus=" + bonus
152.       + "]";
153.  }
154.
155.  private double bonus;
156.}
```

API java.lang.Object 1.0

- `Class getClass()`

  Возвращает класс объекта, содержащий информацию об объекте. Как мы увидим далее в этой главе, в языке Java поддерживается представление классов, инкапсулированное в классе `Class`.

- `boolean equals(Object otherObject)`

  Сравнивает два объекта, возвращает значение `true`, если объекты занимают одну и ту же область памяти, в противном случае возвращается значение `false`. Создавая собственные классы, этот метод следует переопределять.

- `String toString()`

  Возвращает строку, представляющую значение объекта. Создавая собственные классы, этот метод следует переопределять.

- `Object clone()`

  Создает копию (клон) объекта. Исполняющая система Java выделяет память для нового экземпляра и копирует туда содержимое области памяти, в которой хранится текущий объект.



На заметку! Клонирование объектов имеет большое значение, однако это довольно сложный процесс, изобилующий ловушками, в которые могут попадать неопытные программисты. В главе 6 метод `clone()` будет рассмотрен подробнее.

API `java.lang.Class 1.0`

- `String getName()`

  Возвращает имя класса.

- `Class getSuperclass()`

  Возвращает имя суперкласса данного класса в виде объекта `Class`.

Обобщенные массивы-списки

Во многих языках программирования, в частности в C, размер всех массивов должен задаваться еще на этапе компиляции программы. Это ограничение существенно затрудняет работу программиста. Сколько сотрудников будет работать в отделе? Однозначно — не больше 100 человек. А что если где-то есть отдел с 150 сотрудниками? А если отдел очень мал и в нем работают только 10 человек? Ведь в этом случае 90% массива не используется!

В языке Java ситуация намного лучше — размер массива можно задавать уже во время выполнения программы.

```
int actualSize = ...;
Employee[] staff = new Employee[actualSize];
```

Разумеется, этот код не решает проблему динамической модификации массивов во время выполнения программы. Задав размер массива, его потом нелегко изменить. Проще всего разрешить эту ситуацию, используя списочные массивы. Для их создания применяются экземпляры класса, который называется `ArrayList`. Данный класс подобен массиву, но может динамически изменять размеры по мере добавления новых элементов или удаления существующих. При этом программисту не приходится писать дополнительный код.

В Java SE 5.0 `ArrayList` был преобразован в обобщенный (generic) класс с *параметром-типом*. Тип элемента массива помещается в угловые скобки и добавляется к имени класса: `ArrayList<Employee>`. В главе 13 будет рассказано о создании собственных обобщенных классов, однако пользоваться объектами `ArrayList` можно, не применяя специальных технологий.

Ниже представлена строка кода, посредством которого создается списочный массив, предназначенный для хранения объектов `Employee`.

```
ArrayList<Employee> staff = new ArrayList<Employee>();
```



На заметку! До появления Java SE 5.0 обобщенные классы отсутствовали. Вместо них применялся единственный класс `ArrayList`, который позволял хранить объекты типа `Object`. Если вы работаете со старыми версиями Java, не добавляйте к имени `ArrayList` суффикс в угловых скобках `<...>`. Вы можете продолжать использовать `ArrayList` без суффикса `<...>` в Java SE 5.0 и более новых. Класс `ArrayList` можно рассматривать как “низкоуровневый тип”.



На заметку! В еще более старых версиях языка Java для создания массивов, размеры которых динамически изменялись, программисты использовали класс `Vector`. Однако класс `ArrayList` эффективнее. С его появлением исчезла необходимость использовать класс `Vector`.

Для добавления новых элементов в списочный массив используется метод `add()`. Ниже показан код, используемый для заполнения такого массива объектами `Employee`.

```
staff.add(new Employee("Harry Hacker", . . .));
staff.add(new Employee("Tony Tester", . . .));
```

Списочный массив управляет внутренним массивом ссылок на объекты. В конце-концов элементы массива могут оказаться исчерпаны. Здесь на помощь приходят специальные средства списочного массива: если метод `add()` вызывается при заполненном внутреннем массиве, `ArrayList` автоматически создает массив большего размера и копирует в него все объекты.

Если вы заранее знаете, сколько элементов нужно хранить, то перед заполнением списка массивов вызовите метод `ensureCapacity()`:

```
staff.ensureCapacity(100);
```

Этот метод выделит память для внутреннего массива, состоящего из 100 объектов. Затем можно вызвать метод `add()`, который не будет иметь проблем с перераспределением памяти.

Начальную емкость можно передать конструктору класса `ArrayList` в качестве параметра:

```
ArrayList<Employee> staff = new ArrayList<Employee>(100);
```



Внимание! Выделение памяти для списочного и обычного массивов происходит по-разному. Так, приведенные ниже выражения не эквивалентны.

```
new ArrayList<Employee>(100)     // Емкость списочного массива равна 100.
new Employee[100]                // Размер массива равен 100.
```

Между емкостью списочного массива и размером обычного массива есть существенное различие. Выделяя память для массива из 100 элементов, вы резервируете их для дальнейшего использования. В то же время емкость списочного массива — это всего лишь *возможная* величина. В ходе работы она может быть увеличена (за счет выделения дополнительной памяти), но сразу после создания массив не содержит ни одного элемента.

Метод `size()` возвращает фактическое количество элементов в списочном массиве. Например, в результате приведенного ниже вызова возвращает текущее количество элементов в массиве `staff`.

```
staff.size()
```

Эквивалентное выражение для определения размера обычного массива `a` выглядит так:

```
a.lenght
```

Если вы уверены, что список массивов будет иметь постоянный размер, можете вызвать метод `trimToSize()`. Этот метод устанавливает размер блока памяти так, чтобы он точно соответствовал количеству элементов, подлежащих хранению. Механизм сборки мусора предотвращает неэффективное использование памяти, освобождая ее излишки.

Если вы использовали метод `trimToSize()`, то в результате добавления элементов блок памяти будет перемещен, что потребует дополнительного времени. Поэтому вызывать данный метод надо лишь в том случае, когда вы уверены, что дополнительные элементы в списочный массив включаться не будут.



На заметку! Класс `ArrayList` ведет себя так же, как и шаблон `vector` в C++. И `ArrayList`, и `vector` являются универсальными типами. Однако в шаблоне `vector` происходит перегрузка операции `[]`, что упрощает доступ к элементам. Поскольку в языке Java перегрузка операций не предусмотрена, следует явным образом вызывать методы. Кроме того, шаблон `vector` в C++ передается по значению. Если `a` и `b` представляют собой два вектора, то выражение `a = b` приведет к созданию нового вектора, длина которого равна `b`. Все элементы `b` будут скопированы в `a`. В результате выполнения того же самого выражения в языке Java переменные `a` и `b` будут ссылаться на один и тот же списочный массив.



java.util.ArrayList 1.2

- `ArrayList<T>()`

  Создает пустой списочный массив.

- `ArrayList<T>(int initialCapacity)`

  Создает пустой списочный массив, имеющий заданную емкость.

  Параметр `initialCapacity` Первоначальная емкость списочного массива.

- `boolean add(T obj)`

  Добавляет элемент в конец массива. Всегда возвращает значение `true`.

  Параметр `obj` Добавляемый элемент.

- `int size()`

  Возвращает количество элементов, хранящихся в списочном массиве. (Количество элементов отличается от емкости массива и не превосходит ее.)

- `void ensureCapacity(int capacity)`

  Гарантирует, что список массивов имеет емкость, достаточную для хранения заданного количества элементов без изменения внутреннего массива.

  Параметр `capacity` Требуемая емкость списочного массива.

- `void trimToSize()`

  Сокращает емкость списочного массива до его текущего размера.

Доступ к элементам списочных массивов

К сожалению, ничто не дается бесплатно. Удобство, предоставляемое автоматическим регулированием размера списочного массива, компенсируется более сложным синтаксисом, который требуется для доступа к его элементам. Причина в том, что класс `ArrayList` не является частью языка Java; это лишь вспомогательный класс, помещенный в стандартную библиотеку.

Вместо квадратных скобок, используемых для доступа к элементу массива, программисты вынуждены применять методы `get()` и `set()`.

Например, чтобы задать `i`-й элемент, используется выражение

```
staff.set(i, harry);
```

При работе с обычным массивом `a` те же действия выполнялись бы следующим образом:

```
a[i] = harry;
```

Как в обычных, так и в списочных массивах индексы отсчитываются от нуля.



Внимание! Не вызывайте `list.set(i, x)` до тех пор, пока размер массива-списка больше `i`. Например, следующий код не верен:

```
ArrayList<Employee> list = new ArrayList<Employee>(100); //емкость 100, размер 0
list.set(0, x); // пока нет элемента 0
```

Используйте для заполнения массива метод `add()` вместо `set()`, и применяйте `set` только для замены ранее добавленного элемента.

Получить элемент списочного массива можно с помощью метода `get()`.

```
Employee e = staff.get(i);
```

Это — эквивалент следующего:

```
Employee e = a[i];
```



На заметку! До появления Java SE 5.0 универсальные классы отсутствовали, и единственным вариантом возвращаемого значения метода `get()` класса `ArrayList` была ссылка `Object`. Очевидно, что в вызывающем методе приходилось выполнять приведение типов:

```
Employee e = (Employee) staff.get(i);
```

При использовании класса `ArrayList` возможны неприятные ситуации, связанные с тем, что его методы `add()` и `set()` допускают передачу любого типа параметра. Приведенный ниже вызов воспринимается компилятором как корректный.

```
staff.set(i, new Date());
```

Проблемы возникнут лишь тогда, когда вы извлечете объект и попытаетесь преобразовать его в тип `Employee`. При использовании `ArrayList<Employee>` ошибка будет выявлена на этапе компиляции.

Иногда удается сочетать гибкость и удобство доступа к элементам, пользуясь следующим приемом. Создайте список массивов и добавьте в него все нужные элементы:

```
ArrayList<X> list = new ArrayList<X>();
while (...)
{
   x = ...;
   list.add(x);
}
```

Затем, используя метод `toArray()`, скопируйте все элементы в массив:

```
X[] a = new X[list.size()];
list.toArray(a);
```

Элементы можно добавлять не только в конец списочного массива, но и в его середину:

```
int n = staff.size() / 2;
staff.add(n, e);
```

Элемент, имеющий индекс `n`, и расположенные за ним элементы сдвигаются, чтобы освободить место для нового элемента. Если новый размер списка массивов после вставки элемента превышает его емкость, происходит копирование массива.

Аналогично можно удалить элемент из середины списочного массива:

```
Employee e = staff.remove(n);
```

Элементы, расположенные после удаленного элемента, сдвигаются влево, а размер списка массивов уменьшается на единицу.

Вставка и удаление элементов не слишком эффективна. Для массивов, размеры которых невелики, проблема не возникает. Однако если при работе с большими объемами данных приходится часто вставлять и удалять элементы, попробуйте вместо списочного массива применить связный список. Способы программирования связных списков описаны в главе 13.

Начиная с Java SE 5.0, вы можете использовать цикл "for each" для перебора содержимого массива-списка:

```
for (Employee e : staff)
   делать что-то с e
```

Этот цикл дает тот же эффект, что и:

```
for (int i = 0; i < staff.size(); i++)
{
   Employee e = staff.get(i);
   делать что-то с e
}
```

В листинге 5.4 показана модифицированная программа `EmployeeTest` из главы 4. Массив `Employee[]` заменен классом `ArrayList<Employee>`. Обратите внимание на следующие особенности.

- Задавать размер массива не нужно.
- С помощью метода `add()` можно добавлять сколько угодно элементов.
- Вместо `length` для подсчета количества элементов применяется метод `size()`.
- Вместо выражения `a[i]` для доступа к элементу массива используется метод `a.get(i)`.

Листинг 5.4. Содержимое файла `ArrayListTest.java`

```
1. import java.util.*;
2.
3. /**
4.    * Эта программа демонстрирует использование класса ArrayList.
5.    * @version 1.1 2004-02-21
6.    * @author Cay Horstmann
7.    */
8. public class ArrayListTest
9. {
10.  public static void main(String[] args)
11.  {
12.    //Заполнение staff тремя объектами Employee.
13.    ArrayList<Employee> staff = new ArrayList<Employee>();
14.
15.    staff.add(new Employee("Carl Cracker", 75000, 1987, 12, 15));
16.    staff.add(new Employee("Harry Hacker", 50000, 1989, 10, 1));
17.    staff.add(new Employee("Tony Tester", 40000, 1990, 3, 15));
18.
19.    // Увеличение заработной платы на 5%.
20.    for (Employee e : staff)
21.      e.raiseSalary(5);
22.
```

```
23.    // Вывод информации обо всех сотрудниках.
24.    for (Employee e : staff)
25.       System.out.println("name=" + e.getName()
26.          + ",salary=" + e.getSalary()
27.          + ",hireDay=" + e.getHireDay());
28.  }
29.}
30.
31.class Employee
32.{
33. public Employee(String n, double s,
34.    int year, int month, int day)
35. {
36.    name = n;
37.    salary = s;
38.    GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
39.    hireDay = calendar.getTime();
40. }
41.
42. public String getName()
43. {
44.    return name;
45.  }
46.
47.  public double getSalary()
48.  {
49.    return salary;
50.  }
51.
52.  public Date getHireDay()
53.  {
54.    return hireDay;
55.  }
56.
57.  public void raiseSalary(double byPercent)
58.  {
59.    double raise = salary * byPercent / 100;
60.    salary += raise;
61.  }
62.
63.  private String name;
64.  private double salary;
65.  private Date hireDay;
66. }
```

API java.util.ArrayList<T> 1.2

- `void set(int index, T obj)`

  Помещает значение в элемент списка массивов с указанным индексом, заменяя его предыдущее содержимое.

  Параметры `index` Позиция (число между `0` и `size()-1`).

  `obj` Новое значение.

- `T get(int index)`

  Извлекает значение, хранящееся в позиции, указанной индексом.

  Параметр `index` Индекс извлекаемого элемента (число между `0` и `size()-1`).

- `void add(int Index, T obj)`

  Сдвигает существующие элементы для вставки нового элемента.

  Параметры `index` Позиция вставляемого элемента (число между 0 и `size()-1`).

  `obj` Новый элемент.

- `T remove(int index)`

  Удаляет указанный элемент и сдвигает элементы, следующие за ним. Метод возвращает удаленный элемент.

  Параметр `index` Позиция удаляемого элемента (число между 0 и `size()-1`).

Совместимость между типизированными и "сырыми" списочными массивами

Если вы собираетесь обрабатывать исходный код вашей программы с помощью Java SE 5.0 или более поздних версий, то можете при создании списочных массивов указывать тип элементов, которые могут храниться в нем. Однако в ряде случаев приходится обеспечивать совместимость с уже существующим кодом, в котором использованы "сырые" списочные массивы, реализуемые с помощью класса `ArrayList`.

Предположим, что в программе присутствует следующий код:

```
public class EmployeeDB
{
   public void update(ArrayList list) { ... }
   public ArrayList find(String query) { ... }
}
```

Вы можете указать в качестве параметра метода `update()` типизированный списочный массив безо всякого приведения:

```
ArrayList<Employee> staff = ...;
employeeDB.update(staff);
```

Объект `staff` просто передается методу `update`.



Внимание! Несмотря на то что компилятор не считает данное выражение ошибкой и даже не выводит предупреждающее сообщение, такой подход нельзя считать полностью безопасным. Метод `update()` может добавлять в списочный массив элементы, типы которых отличаются от `Employee`. Это звучит пугающе, но если подумать, именно такое поведение характерно для версий, предшествовавших Java SE 5.0. Целостность виртуальной машины Java должна быть обеспечена в первую очередь, потому что, хотя уровень безопасности и не снижается, воспользоваться преимуществом проверки типов на этапе компиляции не удается.

Если же вы попытаетесь использовать обычный "сырой" списочный массив вместо типизированного, компилятор отобразит предупреждающее сообщение:

```
// При обработке следующей строки будет выведено
// предупреждающее сообщение
ArrayList<Employee> result = employeeDB.find(query);
```



На заметку! Чтобы увидеть текст предупреждающего сообщения, надо указать при вызове компилятора опцию `-Xlint:unchecked`.

Попытка выполнить приведение типов не исправит ситуацию:

```
// На этот раз появится другое предупреждающее сообщение
ArrayList<Employee> result =
   (ArrayList<Employee>) employeeDB.find(query);
```

Изменится лишь сообщение. На этот раз оно информирует о несогласованном приведении типов.

Данная ситуация возникает из-за некоторых ограничений обобщенных типов языка Java. Для совместимости компилятор после проверки соблюдения правил работы с типами преобразует типизированные списочные массивы в обычные объекты `ArrayList`. В процессе выполнения программы все списочные массивы одинаковы: виртуальная машина не получает информации о типах. Поэтому приведения типов `(ArrayList)` и `(ArrayList<Employee>)` при работе программы обрабатываются одинаково.

В таких условиях от ваших действий мало что зависит. Модифицируя имеющийся код, вам остается следить за сообщениями компилятора и успокаивать себя тем, что они не свидетельствуют о наличии серьезных ошибок.

Объектные оболочки и автоупаковка

Иногда приходится преобразовывать переменные простых типов вроде `int` в объекты. Все простые типы имеют аналоги в виде классов. Например, существует класс `Integer`, соответствующий типу `int`. Такие классы принято называть *объектными оболочками* (object wrapper). Они имеют очевидные имена: `Integer`, `Long`, `Float`, `Double`, `Short`, `Byte`, `Character`, `Void` и `Boolean`. (Первые шесть классов имеют общий суперкласс `Number`.) Классы-оболочки являются финальными. Таким образом, вы не можете, например, переопределить метод `toString()` в классе `Integer`, чтобы отобразить число римскими цифрами. Кроме того, изменить значение, хранящееся в объекте интерфейсного класса, также невозможно.

Предположим, мы хотим, чтобы в списочном массиве хранились целые числа. К сожалению, с помощью параметра в угловых скобках невозможно задать простой тип, например, выражение `ArrayList<int>` недопустимо. Здесь приходит на помощь класс-оболочку. Можно объявить списочный массив, предназначенный для хранения объектов `Integer`:

```
ArrayList<Integer> list = new ArrayList<Integer>();
```



Внимание! При использовании объекта `ArrayList<Integer>` производительность становится меньше, чем при работе с массивом `int[]`. Причина очевидна: каждое значение инкапсулировано внутри объекта, и для его записи или извлечения необходимо предпринимать дополнительные действия. Таким образом, использование классов-оболочек оправдано в небольших наборах данных, когда удобство работы программиста важнее эффективности работы программы.

С появлением Java SE 5.0 стало проще добавлять элементы в массив и извлекать их. Рассмотрим следующую строку кода:

```
list.add(3);
```

Она автоматически преобразуется в выражение

```
list.add(new Integer(3));
```

Подобное автоматическое преобразование называется автоупаковкой (autoboxing).



На заметку! Вы можете подумать, что "автоматическое помещение в оболочку" (autowrapping) здесь больше подходит, но метафора "упаковка" (boxing) была взята из C#.

В противоположном случае, если вы присвоите объект `Integer` переменной `int`, целочисленное значение будет автоматически извлечено из объекта, т.е. распаковано (unboxed). Другими словами, компилятор преобразует строку кода

```
int n = list.get(i);
```

в выражение

```
int n = list.get(i).intValue();
```

Автоматическая упаковка простого типа в объект и извлечение значения может выполняться и при вычислении арифметических выражений. Например, вы можете применить операцию инкрементирования к переменной, содержащей ссылку на объект `Integer`:

```
Integer n = 3;
n++;
```

Компилятор автоматически извлечет целочисленное значение из объекта, увеличит его на единицу и снова поместит в объект.

На первый взгляд может показаться, что примитивные типы и их оболочки — одно и то же. Различие становится ясно видно при использовании операции проверки на идентичность. Как вы знаете, операция ==, будучи примененной к объекту, проверяет, ссылаются ли переменные на один и тот же адрес памяти. Ниже приведен пример, в котором, несмотря на равенство целочисленных значений, проверка, вероятнее всего, даст отрицательный результат.

```
Integer a = 1000;
Integer b = 1000;
if (a == b) ...
```

Однако реализация Java *может*, если пожелает, поместить часто встречающиеся значения в оболочки идентичных объектов, и тогда сравнение даст положительный результат. Такая неоднозначность результатов — не то, что вам нужно. Выход — использовать при сравнении объектов-оболочек метод `equals()`.



На заметку! Спецификация автоупаковки требует, чтобы значения `boolean`, `byte`, `char`, меньшие 127, а также `short` и `int` между –128 и 127, упаковывались в фиксированные объекты. Например, если `a` и `b` инициализированы значением 100 в предыдущем примере, то их сравнение должно дать положительный результат.

Следует отметить, что за преобразование простых типов в объекты-оболочки и извлечение значений из объектов отвечает не виртуальная машина, а *компилятор*. Он включает в программу необходимые вызовы, а виртуальная машина лишь выполняет байт-код.

Вы часто встретите объектные оболочки числовых значений еще и по другой причине. Проектировщики Java решили, что в составе классов-оболочек удобно было реализовать методы для преобразования строк в числа.

Чтобы преобразовать строку в целое число, используется выражение, подобное приведенному ниже:

```
int x = Integer.parseInt(s);
```

Обратите внимание, что при этом создавать объект `Integer` не обязательно; `parseInt()` представляет собой статический метод. Но класс `Integer` — подходящее место для этого.

Ниже описаны наиболее важные методы класса `Integer`. Аналогичные методы есть и в других классах, предназначенных для хранения числовых значений.



На заметку! Некоторые программисты считают, что с помощью интерфейсных классов можно реализовать методы, модифицирующие свои числовые параметры. Однако это неправильно. Как было сказано в главе 4, на языке Java невозможно написать метод, увеличивающий целое число, передаваемое ему в качестве параметра, поскольку все параметры передаются по значению.

```
public static void triple(int x)        // Неправильно.
{
   x++;    // Увеличивает локальную копию.
}
```

Но, может быть, это ограничение можно обойти, используя вместо типа `int` класс `Integer`?

```
public static void triple(Integer x)    // Неправильно.
{
   ...
}
```

Проблема в том, что объект `Integer` не позволяет изменять содержащуюся в нем информацию. Таким образом, изменить числовой параметр, передаваемый методу, невозможно.

Если вам все же необходимо создать метод, изменяющий свои числовые параметры, используйте один из вспомогательных типов, определенных в пакете `org.omg.CORBA`. К этим типам относятся `IntHolder`, `BooleanHolder` и др. Каждый такой тип содержит общедоступное (!) поле `value`, через которое можно обращаться к хранящемуся в нем числу.

```
public static void triple(IntHolder x)
{
   x.value++;
}
```



java.lang.Integer 1.0

- `int intValue()`

  Возвращает значение объекта `Integer` в виде числа типа `int` (данный метод переопределяет метод `intValue()` класса `Number`).

- `static String toString(int i)`

  Возвращает новый объект `String`, представляющий число в десятичном виде.

- `static String toString(int i, int radix)`

  Возвращает новый объект типа `String`, представляющий число в системе исчисления, определяемой параметром `radix`.

- `static int parseInt(String s)`
- `static int parseInt(String s, int radix)`

  Возвращает целое значение. Предполагается, что объект `String` содержит строку, представляющую целое число в десятичной системе (для первого варианта) или системе счисления, которая задается параметром `radix`.

- `static Integer valueOf(String s)`
- `static Integer valueOf(String s, int radix)`

  Возвращает новый объект типа `Integer`, инициализированный целым значением, которое задается с помощью первого параметра. Предполагается, что объект `String` содержит строку, представляющую целое число в десятичной системе (для первого варианта) или в системе счисления, которая задается параметром `radix`.

API **java.text.NumberFormat 1.1**

- `Number parse(String s)`

  Возвращает числовое значение, полученное в результате разбора параметра. Предполагается, что объект типа `String` содержит строку, представляющую целое десятичное число.

Методы с переменным числом параметров

До появления Java SE 5.0 количество параметров любого Java-метода было фиксировано. Теперь есть возможность создавать методы, позволяющие при разных вызовах задавать различное количество параметров.

С одним из таких методов, `printf()`, вы уже знакомы. Ниже представлены два обращения к нему.

```
System.out.printf("%d", n);
System.out.printf("%d %s", n, "widgets");
```

В обеих строках вызывается один и тот же метод, но в первом случае методу передаются два параметра, а во втором случае — три. Определение метода `printf()` выглядит следующим образом:

```
public class PrintStream
{
   public PrintStream printf(String fmt, Object... args)
     { return format(fmt, args); }
}
```

Здесь многоточие (`...`) — это часть Java-кода. Оно указывает на то, что в дополнение к параметру `fmt` можно указывать любое количество объектов. (Если вызывающий метод передает целочисленное значение или значение одного из простых типов, автоупаковка преобразует его в объект.) После этого метод решает непростую задачу разбора строки `fmt` и связывания спецификаторов формата со значениями `args[i]`.

Другими словами, в методе `printf()` тип параметра `Object...` означает то же самое, что и `Object[]`.

Компилятор при обработке исходного кода выявляет каждое обращение к `printf()`, помещает параметры в массив и, если необходимо, выполняет преобразование простых типов в интерфейсные объекты.

```
System.out.printf("%d %d",
   new Object[] { new Integer(d), "widgets" } );
```

При необходимости вы можете создавать собственные методы с переменным числом параметров и указывать любые типы параметров, в том числе простые типы. Ниже приведен пример простого метода, который вычисляет и возвращает максимальное из произвольного количества значений.

```
public static double max(double... values)
{
   double largest = Double.MIN_VALUE;
   for (double v : values) if (v > largest) largest = v;
   return largest;
}
```

Вызов функции может выглядеть следующим образом:

```
double m = max(3.1, 40.4, -5);
```

Компилятор передает функции `max()` выражение `new double[] { 3.1, 40.4, -5 }`.



На заметку! В качестве последнего параметра метода, для которого число параметров может быть переменным, допустимо задавать массив:

```
System.out.printf("%d %s", new Object[] { new Integer(1), "widgets" } );
```

Таким образом, вы можете преобразовать существующую функцию, последний параметр которой является массивом, в метод с переменным числом параметров, не изменяя существующего кода. Подобным образом в Java SE 5.0 был расширен метод `MessageFormat.format()`. При желании вы даже можете объявить метод `main()` так:

```
public static void main(String... args)
```

Классы-перечисления

В главе 3 было сказано, что, начиная с Java SE 5.0, поддерживаются перечислимые (enumerated) типы. Пример соответствующего выражения приведен ниже:

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

Тип, объявленный таким образом, на самом деле представляет собой класс. Возможно существование всего четырех экземпляров указанного класса. Другие объекты создать нельзя.

Таким образом, вам не обязательно использовать метод `equals()`, чтобы проверять эквивалентность перечислимых значений. Для этой цели вполне подойдет операция ==.

При желании вы можете добавлять к перечислимым типам конструкторы, методы и поля. Очевидно, что конструкторы могут вызываться только при создании перечислимых констант. Вот пример:

```
enum Size
{
   SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");

   private Size(String abbreviation) { this.abbreviation = abbreviation; }
   public String getAbbreviation() { return abbreviation; }

   private String abbreviation;
}
```

Все перечислимые типы реализуются с помощью подклассов класса `Enum`. Они наследуют от этого класса ряд методов. Наиболее часто используется метод `toString()`, возвращающий имя перечислимой константы.

Например, метод `Size.SMALL.toString()` вернет строку `"SMALL"`.

Статический метод `valueOf()` выполняет действия, противоположные методу `toString()`. Например, в результате выполнения приведенного ниже выражения переменной `s` будет присвоено значение `Size.SMALL`:

```
Size s = (Size) Enum.valueOf(Size.class, "SMALL");
```

Каждый перечислимый тип содержит статический метод `values()`, который возвращает массив всех перечислимых значений:

```
Size[] values = Size.values();
```

Метод `ordinal()` возвращает позицию перечислимой константы в объявлении `enum`, начиная с нуля. Например, `Size.MEDIUM.ordinal()` вернет 1.

Работу с перечислимыми типами демонстрирует короткая программа, приведенная в листинге 5.5.



На заметку! В отличие от класса `Class`, класс `Enum` допускает указание типа, который для простоты был опущен. Например, перечислимый тип `Size` на самом деле преобразуется в `Enum<Size>`. В данном разделе мы не рассматривали эту возможность. Параметр типа используется в методе `compareTo`. (Метод `compareTo` мы обсудим в главе 6, а параметры типа — в главе 12.)

Листинг 5.5. Содержимое файла `EnumTest.java`

```
1. import java.util.*;
2.
3.
4. /**
5.    * Эта программа демонстрирует перечислимые типы.
6.    * @version 1.0 2004-05-24
7.    * @author Cay Horstmann
8.    */
9. public class EnumTest
10.{
11.   public static void main(String[] args)
12.   {
13.      Scanner in = new Scanner(System.in);
14.      System.out.print("Enter a size: (SMALL, MEDIUM, LARGE, EXTRA_LARGE) ");
15.      String input = in.next().toUpperCase();
16.      Size size = Enum.valueOf(Size.class, input);
17.      System.out.println("size=" + size);
18.      System.out.println("abbreviation=" + size.getAbbreviation());
19.      if (size == Size.EXTRA_LARGE)
20.         System.out.println("Good job--you paid attention to the _.");
21.   }
22.}
23.
24.enum Size
25.{
26.   SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");
27.
```

```
28.   private Size(String abbreviation)
29.      { this.abbreviation = abbreviation; }
30.   public String getAbbreviation()
31.      { return abbreviation; }
32.
33.   private String abbreviation;
34. }
```

API **java.lang.Enum 5.0**

- `static Enum valueOf(Class enumClass, String name)`

  Возвращает перечислимую константу указанного класса с заданным именем.

- `String toString()`

  Возвращает имя перечислимой константы.

- `int ordinal()`

  Возвращает позицию данной перечислимой константы в объявлении `enum`, начиная с нуля.

- `int compareTo(E order)`

  Возвращает отрицательное целое, если перечислимая константа следует перед `other`, 0 — если `this==other` и в противном случае — положительное целое. Порядок следования констант задается объявлением `enum`.

Рефлексия

Библиотека рефлексии (reflection library) предоставляет богатый набор инструментальных средств для динамической работы с Java-кодом. Эта возможность широко используется в JavaBeans при создании компонентов. (Средства JavaBeans будут рассматриваться во втором томе.) Благодаря рефлексии появляется возможность поддерживать инструментальные средства, подобные тем, которыми пользуются программисты, работающие на Visual Basic. Так, например, когда в процессе разработки или выполнения программы добавляется новый класс, можно организовать опрос с целью выяснить возможности нового класса.

Программа, которая способна анализировать возможности классов, называется *рефлективной* (reflective). Рефлексия — очень мощный механизм, который можно использовать для решения перечисленных ниже задач.

- Анализ возможностей классов в процессе выполнения программы.
- Проверка объектов при выполнении программы, например, с помощью рефлексии можно реализовать метод `toString()`, совместимый *со всеми* классами.
- Реализация обобщенного кода для работы с массивами.
- Применение объектов `Method`, которые работают подобно указателями на функции в языках, подобных C++.

Рефлексия — не только мощный, но и сложный механизм. Он интересен в основном разработчикам инструментальных средств; программисты, создающие обычные прикладные программы, в большинстве случаев могут не применять его. Если вы занимаетесь разработкой приложений и еще не готовы воспринимать сложные вопро-

сы, то можете пропустить остаток этой главы и вернуться к соответствующим разделам позже.

Класс Class

Во время выполнения программы исполняющая система Java всегда осуществляет проверку типов объектов любых классов. Эта информация используется виртуальной машиной для выбора правильного метода, подлежащего вызову.

Получить доступ к этой информации можно иначе — с помощью специального класса, который носит довольно странное имя — Class. Метод getClass() класса Object возвращает экземпляр типа Class.

```
Employee e;
...
Class cl = e.getClass();
```

Аналогично тому, как созданный нами объект типа Employee описывает свойства конкретного сотрудника, объект Class описывает свойства конкретного класса. Вероятно, наиболее популярным методом класса Class является метод getName(), возвращающий имя класса. Например, оператор

```
System.out.println(e.getClass().getName() + " " + e.getName());
```

выводит строку

```
Employee Harry Hacker
```

если объект e относится к классу Employee, или

```
Manager Harry Hacker
```

если объект e относится к классу Manager.

Если класс находится в пакете, то имя пакета будет частью имени класса:

```
Date d = new Date();
Class cl = d.getClass();
String name = cl.getName(); // имя устанавливается в "java.util.Date"
```

Используя статический метод forName(), можно также получить объект Class, соответствующий строке:

```
String className = "java.util.Date";
Class cl = Class.forName(className);
```

Этот метод можно применять, если имя класса хранится в строке, содержимое которой во время выполнения программы изменяется. Он работает правильно, если className является именем класса или интерфейса. В противном случае метод forName() генерирует *контролируемое исключение*. Правила применения *обработчика исключения* при работе с этим методом описаны ниже, в разделе "Основы обработки исключений".



Совет. При запуске программы сначала загружается метод main(). Он загружает все необходимые классы. Каждый из классов в свою очередь загружает необходимые ему классы и т.д. Если размеры приложения достаточно велики, этот процесс может занять значительное время, причем пользователь вынужден будет ожидать окончания загрузки. Для того чтобы вызвать у пользователя иллюзию быстрого старта, можно воспользоваться следующим приемом. Убедитесь, что класс, содержащий метод main(), не ссылается явно на другие классы. Начинайте выводить на экран данные и загрузите остальные классы с помощью вызовов метода Class.forName().

Третий способ получения объекта Class прост и удобен. Если T — это некоторый тип, то T.class — объект соответствующего класса. Например:

```
Class cl1 = Date.class; // Если импортирован java.util.*;
Class cl2 = int.class;
Class cl3 = Double[].class;
```

Отметим, что объект Class фактически описывает *тип*, который не обязательно является классом. Например, тип int — это не класс, однако, несмотря на это, int.class — это объект Class.



На заметку! Начиная с Java SE 5.0, класс Class является параметризованным. Например, Employee.class соответствует типу Class<Employee>. Мы не будем подробно обсуждать этот вопрос, так как использование типов еще больше усложнит и без того сложные понятия. В процессе практической работы вы вполне можете игнорировать параметр, определяющий тип, и работать с обычным классом Class. Дополнительную информацию вы найдете в главе 13.



Внимание! Исторически сложилось так, что метод getName() возвращает для массивов довольно странные имена:

- Double[].class.getName() возвращает "[Ljava.lang.Double;";
- int[].class.getName() возвращает "[I".

Виртуальная машина Java поддерживает уникальный объект Class для каждого типа. Следовательно, для сравнения объектов можно использовать операцию ==, как показано ниже:

```
if (e.getClass() == Employee.class) . . .
```

Еще один полезный метод позволяет создавать в процессе работы программы новый экземпляр класса. Имя метода, как и следовало ожидать, — newInstance(). Вот пример его вызова:

```
e.getClass().newInstance();
```

В результате выполнения представленного выражения создается новый экземпляр того же класса, что и e. Для инициализации вновь созданного объекта метод newInstance() использует конструктор по умолчанию (т.е. конструктор без параметров). Если в классе конструктор по умолчанию отсутствует, генерируется исключение.

Используя методы forName() и newInstance(), можно создавать экземпляры классов, имена которых хранятся в строках:

```
String s = "java.util.Date";
Object m = Class.forName(s).newInstance();
```



На заметку! Если при создании объекта на основе имени класса нужно передать конструктору какие-либо параметры, использовать приведенные выше выражения нельзя. Вместо этого нужно применять метод newInstance() класса Constructor.



На заметку! Метод newInstance() является аналогом *виртуального конструктора* в языке C++. Хотя в C++ на самом деле нет виртуальных конструкторов как средства языка, и такая идиома реализуется с помощью специализированных библиотек. Класс Class соответствует классу type_info в языке C++, а метод getClass() — операции typeid. Класс Class в языке Java более гибок, чем его аналог в C++. Класс type_info может только извлекать строку с именем типа, но не способен создавать объекты этого типа.

Основы обработки исключений

Обработка исключений детально рассматривается в главе 11, но в предшествующих ей главах иногда будут встречаться методы, при возникновении которых могут возникать исключения.

Если во время выполнения программы возникает ошибка, программа "*генерирует исключение*". Это более гибкий процесс, чем простое прекращение выполнения программы, поскольку программист может создавать *обработчики* исключений, которые "перехватят" исключение и что-то сделают с ним.

Если обработчик не предусмотрен, система завершает программу и выводит на экран сообщение о типе исключения. Возможно, вы уже сталкивались с подобными сообщениями об исключениях, например, если пытались использовать ссылку `null` или обращались за пределы массива.

Существуют два типа исключений: *неконтролируемые* (uncheked) и *контролируемые* (checked). При возникновении контролируемого исключения компилятор проверяет, предусмотрен ли для него обработчик. Однако большинство исключений являются неконтролируемыми. К ним относится, например, ситуация, возникающая при обращении к ссылке `null`. Компилятор не проверяет, предусмотрен ли программистом обработчик. В принципе надо стараться избегать подобных ошибок, а не программировать их обработку.

Однако не всех ошибок можно избежать. Если, несмотря на все ваши усилия, остался фрагмент кода, при выполнении которого может быть сгенерировано исключение, и вы не предусмотрели его обработку, компилятор будет настаивать на том, чтобы вы создали обработчик. Например, метод `Class.forName()` может сгенерировать контролируемое исключение. В главе 11 мы рассмотрим несколько стратегий обработки исключений, а пока изучим реализацию простейших обработчиков.

Поместим одно или несколько выражений, при выполнении которых могут возникнуть контролируемые исключения, в блок `try`, а код обработчика — в блок `catch`.

```
try
{
   Операторы, которые могут генерировать исключение.
}
catch(Exception e)
{
   Код обработчика.
}
```

Рассмотрим пример:

```
try
{
 String name = ...;                  // Определение имени класса.
 Class cl = Class.forName(name);     // Может сгенерировать исключение.
  ...                                // Действия с классом cl.
}
catch(Exception e)
{
 e.printStackTrace();
}
```

Если имя класса не существует, оставшаяся часть блока, расположенного в блоке `try`, игнорируется и управление передается блоку `catch`. (В данном случае предусмотрен вывод содержимого стека с помощью метода `StackTrace()` класса `Throwable`.

Этот класс является суперклассом класса Exception.) Если в блоке try исключение не генерируется, то код обработчика в разделе catch не выполняется.

От программиста требуется лишь предусмотреть обработчик для проверяемого исключения. Метод, ответственный за возникновение исключения, обнаружить легко. Если в программе вызывается метод, который может сгенерировать исключение, а обработчика для него нет, то компилятор выведет соответствующее сообщение.

API java.lang.Class 1.0

- `static Class forName(String className)`

  Возвращает объект Class, представляющий класс с именем className.

- `Object newInstance()`

  Возвращает новый экземпляр класса.

API java.lang.reflect.Constructor 1.1

- `Object newInstance(Object[] args)`

  Создает новый экземпляр класса.

  Параметр args — Параметры, передаваемые конструктору. Подробнее о применении параметров читайте в разделе, посвященном рефлексии.

API java.lang.Throwable 1.0

- `void printStackTrace()`

  Выводит в стандартный поток ошибок объект Throwable и содержимое стека.

Использование рефлексии для анализа свойств классов

Ниже приводится краткий обзор важных характеристик механизма рефлексии, позволяющего программисту анализировать структуру класса.

Три класса, Field, Method и Constructor, в пакете java.lang.reflect описывают соответственно поля, методы и конструкторы класса. Все три класса содержат метод getName(), возвращающий имя анализируемого класса. В состав класса Field входит метод getType(), который возвращает объект типа Class, описывающий тип поля. Классы Method и Constructor имеют методы, определяющие типы параметров, а класс Method может также определять тип возвращаемого значения. Все три класса содержат метод getModifiers(), возвращающий целое число, которое соответствует использованным модификаторам, например public или static. Для анализа этого числа применяются статические методы класса Modifiers из пакета java.lang.reflect. Например, в этом классе есть методы isPublic(), isPrivate() и isFinal(), определяющие, является тот или иной метод либо конструктор общедоступным, приватным или финальным. Все, что нужно сделать, — это применить соответствующий метод класса Modifier к целому числу, которое возвращается методом getModifiers(). Для вывода самих модификаторов используется метод Modifier.toString().

Методы getFields(), getMethods() и getConstructors() класса Class возвращают массивы *общедоступных* полей, методов и конструкторов, принадлежащих анализируемому классу. К ним относятся и *общедоступные* поля суперклассов. Методы

`getDeclaredFields()`, `getDeclaredMethods()` и `getDeclaredConstructors()` класса `Class` возвращают массивы, состоящие из всех полей, методов и конструкторов, объявленных в классе. К ним относятся приватные и защищенные элементы, но не элементы суперклассов.

В листинге 5.6 показано, как отобразить всю информацию о классе. Программа предлагает пользователю ввести имя класса, а затем выводит сигнатуры всех методов и конструкторов вместе с именами всех полей класса. Предположим, что пользователь задал приведенную ниже строку.

```
java.lang.Double
```

В результате на экране появится следующий текст:

```
public class java.lang.Double extends java.lang.Number
{
  public java.lang.Double(java.lang.String);
  public java.lang.Double(double);

  public int hashCode();
  public int compareTo(java.lang.Object);
  public int compareTo(java.lang.Double);
  public boolean equals(java.lang.Object);
  public java.lang.String toString();
  public static java.lang.String toString(double);
  public static java.lang.Double valueOf(java.lang.String);
  public static boolean isNaN(double);
  public boolean isNaN();
  public static boolean isInfinite(double);
  public boolean isInfinite();
  public byte byteValue();
  public short shortValue();
  public int intValue();
  public long longValue();
  public float floatValue();
  public double doubleValue();
  public static double parseDouble(java.lang.String);
  public static native long doubleToLongBits(double);
  public static native long doubleToRawLongBits(double);
  public static native double longBitsToDouble(long);

  public static final double POSITIVE_INFINITY;
  public static final double NEGATIVE_INFINITY;
  public static final double NaN;
  public static final double MAX_VALUE;
  public static final double MIN_VALUE;
  public static final java.lang.Class TYPE;
  private double value;
  private static final long serialVersionUID;
}
```

Эта программа замечательна тем, что она может анализировать любой класс, загружаемый интерпретатором языка Java, а не только классы, доступные во время компиляции. В следующей главе мы применим ее для анализа внутренних классов, автоматически генерируемых компилятором языка Java.

Листинг 5.6. Содержимое файла `ReflectionTest.java`

```
1. import java.util.*;
2. import java.lang.reflect.*;
3.
4. /**
5.    * Эта программа использует рефлексию для вывода
6.    * всех характеристик класса.
7.    * @version 1.1 2004-02-21
8.    * @author Cay Horstmann
9.    */
10.public class ReflectionTest
11.{
12.   public static void main(String[] args)
13.   {
14.      // Имя класса извлекается из командной строки
15.      // или вводится с клавиатуры.
16.      String name;
17.      if (args.length > 0) name = args[0];
18.      else
19.      {
20.         Scanner in = new Scanner(System.in);
21.         System.out.println("Введите имя класса (например java.util.Date): ");
22.         name = in.next();
23.      }
24.
25.      try
26.      {
27.         // Вывод имени класса и суперкласса (если это не Object).
28.         Class cl = Class.forName(name);
29.         Class supercl = cl.getSuperclass();
30.         String modifiers = Modifier.toString(cl.getModifiers());
31.         if (modifiers.length() > 0) System.out.print(modifiers + " ");
32.         System.out.print("class " + name);
33.         if (supercl != null && supercl != Object.class) System.out.print(" extends "
34.            + supercl.getName());
35.
36.         System.out.print("\n{\n");
37.         printConstructors(cl);
38.         System.out.println();
39.         printMethods(cl);
40.         System.out.println();
41.         printFields(cl);
42.         System.out.println("}");
43.      }
44.      catch (ClassNotFoundException e)
45.      {
46.         e.printStackTrace();
47.      }
48.      System.exit(0);
49.   }
50.
51.   /**
52.     * Вывод сигнатур всех конструкторов класса.
53.     * @param cl Класс
54.     */
```

```
55.   public static void printConstructors(Class cl)
56.   {
57.      Constructor[] constructors = cl.getDeclaredConstructors();
58.
59.      for (Constructor c : constructors)
60.      {
61.         String name = c.getName();
62.         System.out.print("   ");
63.         String modifiers = Modifier.toString(c.getModifiers());
64.         if (modifiers.length() > 0) System.out.print(modifiers + " ");
65.         System.out.print(name + "(");
66.
67.         // Вывод типов параметров.
68.         Class[] paramTypes = c.getParameterTypes();
69.         for (int j = 0; j < paramTypes.length; j++)
70.         {
71.            if (j > 0) System.out.print(", ");
72.            System.out.print(paramTypes[j].getName());
73.         }
74.         System.out.println(");");
75.      }
76.   }
77.
78.   /**
79.     * Вывод сигнатур всех методов класса.
80.     * @param cl Класс
81.     */
82.   public static void printMethods(Class cl)
83.   {
84.      Method[] methods = cl.getDeclaredMethods();
85.
86.      for (Method m : methods)
87.      {
88.         Class retType = m.getReturnType();
89.         String name = m.getName();
90.
91.         System.out.print("   ");
92.         // Вывод модификаторов, типа возвращаемого
93.         // значения и имени метода
94.         String modifiers = Modifier.toString(m.getModifiers());
95.         if (modifiers.length() > 0) System.out.print(modifiers + " ");
96.         System.out.print(retType.getName() + " " + name + "(");
97.
98.         // Вывод типов параметров.
99.         Class[] paramTypes = m.getParameterTypes();
100.        for (int j = 0; j < paramTypes.length; j++)
101.        {
102.           if (j > 0) System.out.print(", ");
103.           System.out.print(paramTypes[j].getName());
104.        }
105.        System.out.println(");");
106.     }
107.  }
108.
```

```
109.  /**
110.    * Вывод всех полей класса.
111.    * @param cl Класс
112.    */
113.  public static void printFields(Class cl)
114.  {
115.     Field[] fields = cl.getDeclaredFields();
116.
117.     for (Field f : fields)
118.     {
119.        Class type = f.getType();
120.        String name = f.getName();
121.        System.out.print(" ");
122.        String modifiers = Modifier.toString(f.getModifiers());
123.        if (modifiers.length() > 0) System.out.print(modifiers + " ");
124.        System.out.println(type.getName() + " " + name + ";");
125.     }
126.  }
127.}
```

API java.lang.Class 1.0

- `Field[] getFields()` **1.1**
- `Field[] getDeclaredFields()` **1.1**

  Метод `getFields()` возвращает массив, который содержит объекты `Field`, соответствующие общедоступным полям анализируемого класса или его суперкласса. Метод `getDeclaredFields()` возвращает массив, содержащий объекты `Field`, соответствующие всем полям анализируемого класса. Метод возвращает массив нулевой длины, если таких полей нет или если объект `Class` представляет собой простой тип или массив.

- `Method[] getMethods()` **1.1**
- `Method[] gteDeclaredMethods()` **1.1**

  Данные методы возвращают массив, содержащий объекты `Method`, соответствующие либо только общедоступным методам, включая унаследованные (`getMethods()`), либо всем методам, за исключением унаследованных (`getDeclaredMethods()`), анализируемого класса или интерфейса.

- `Constructor[] getConstructors()` **1.1**
- `Constructor[] getDeclaredConstructors()` **1.1**

  Каждый из этих методов возвращает массив, содержащий объекты `Constructor`, соответствующие либо только общедоступным (`getConstructors()`), либо всем конструкторам (`getDeclaredMethods()`) класса, представленного объектом типа `Class`.

API java.lang.reflect.Field 1.1

API java.lang.reflect.Method 1.1

API java.lang.reflect.Constructor 1.1

- `Class getDeclaringClass()`

  Возвращает объект `Class`, соответствующий классу, в котором определен заданный конструктор, метод или поле.

- `Class[] getExceptionTypes()` (в классах `Constructor` или `Method`)

  Возвращает массив объектов типа `Class`, которые представляют собой типы исключений, генерируемых заданным методом.

- `int getModifiers()`

  Возвращает целое число, соответствующее модификатору заданного конструктора, метода или поля. Для анализа возвращаемого значения необходимо использовать методы класса `Modifier`.

- `String getName()`

  Возвращает строку, в которой содержится имя конструктора, метода или поля.

- `Class[] getParameterTypes()` (в классах `Constructor` или `Method`)

  Возвращает массив объектов типа `Class`, представляющих типы параметров.

- `Class getReturnType()` (в классе `Method`)

  Возвращает объект `Class`, соответствующий возвращаемому типу.

API java.lang.reflect.Modifier 1.1

- `static String toString(int modifiers)`

  Возвращает строку с модификаторами, соответствующими битам в целом числе `modifiers`.

- `static boolean isAbstract (int modifiers)`
- `static boolean isFinal (int modifiers)`
- `static boolean isInterface (int modifiers)`
- `static boolean isNative (int modifiers)`
- `static boolean isPrivate (int modifiers)`
- `static boolean isProtected (int modifiers)`
- `static boolean isPublic (int modifiers)`
- `static boolean isStatic(int modifiers)`
- `static boolean isStrict (int modifiers)`
- `static boolean isSynchronized (int modifiers)`
- `static boolean isVolatile (int modifiers)`

  Данные методы проверяют разряды числа `modifiers`, которые соответствуют модификаторам, определяемым именами методов.

Использование рефлексии для анализа объектов во время выполнения

В предыдущем разделе вы узнали, что для определения *имен* и *типов* полей любого объекта надо выполнить следующие действия:

- получить соответствующий объект типа `Class`;
- вызвать его метод `getDeclaredFields()`.

В этом разделе мы попробуем определить *содержимое* полей данных. Разумеется, если имя и тип объекта известны при написании программы, это не составляет никакого труда. Однако рефлексия позволяет сделать это и для объектов, которые не известны на этапе компиляции.

Ключевым в этом процессе является метод `get()` класса `Field`. Если объект `f` имеет тип `Field` (например, если он получен в результате применения метода `getDeclaredFields()`), а объект `obj` относится к тому же классу, что и объект `f`, то вызов `f.get(obj)` вернет объект, значение которого будет текущим значением поля в объекте `obj`. Рассмотрим конкретный пример:

```
Employee harry = new Employee ("Harry Hacker", 3500, 10, 1, 1989);
Class cl = harry.getClass(0;
  //  Объект типа Class, представляющий класс Employee.
Field f = c1.getDeclaredField("name");
  // Имя поля для класса Employee.
Object V = f.get(harry);
  // Значение поля name объекта harry,
  // т.е. объект типа String "Harry Hacker".
```

Несмотря на то что код кажется несложным, при его выполнении возникнет проблема. Поскольку поле `name` объявлено как `private`, метод `get()` сгенерирует исключение `IllegalAccessException`. Метод `get()` можно применять только для общедоступных полей. Механизм безопасности языка Java позволяет определить, какие поля содержит объект, но не даст прочитать их содержимое, если эти поля недоступны.

По умолчанию механизм рефлексии выполняет правила доступа, установленные в языке Java. Однако если программа не контролируется диспетчером защиты, эти правила можно обойти. Чтобы сделать это, вызовите метод `setAccessible()` для объектов `Field`, `Method` или `Constructor`, например:

```
f.setAccessible(true); // Теперь можно вызывать f.get(harry);
```

Метод `setAccessible()` относится к классу `AccessibleObject`, общему суперклассу классов `Field`, `Method` и `Constructor`. Он нужен для обеспечения работы отладчиков, поддержки постоянных хранилищ и выполнения других подобных действий. Позднее в этой главе мы применим его для создания обобщенного метода `toString()`.

С методом `get()` связана еще одна проблема. Поле `name` имеет тип `String`, поэтому можно вернуть его значение как объект `Object`. Предположим, однако, что нам нужно определить значение поля `salary`. Оно имеет тип `double`, а в языке Java числовые типы не являются объектами. Чтобы решить эту проблему, можно использовать либо метод `getDouble()` класса `Field`, либо метод `get()`, который с помощью механизма рефлексии поместит поле в соответствующий интерфейсный класс, в данном случае — `Double`.

Разумеется, значение поля можно не только определять, но и задавать. Вызов `f.set(obj, value)` задает новое значение поля, представленного переменной `f` в объекте `obj`.

В листинге 5.7 показано, как создать обобщенный метод `toString()`, работающий с *любым* классом. Сначала он использует метод `getDeclaredFields()` для получения всех полей. Затем применяется метод `setAccessible()`, делающий все эти поля доступными. Далее определяется имя и значение каждого поля. В листинге 5.7 показан пример преобразования значений в строки путем рекурсивного вызова метода `toString()`.

```
class ObjectAnalyzer
{
  public String toString(Object obj)
  {
     Class cl = obj.getClass();
     . . .
     String r = cl.getName();
     // Проверка полей класса и всех его суперклассов.
     do
     {
        r += "[";
        Field[] fields = cl.getDeclaredFields();
        AccessibleObject.setAccessible(fields, true);
        // Определение имен и значений всех полей.
        for (Field f : fields)
        {
           if (!Modifier.isStatic(f.getModifiers()))
           {
              if (!r.endsWith("[")) r += ","
              r += f.getName() + "=";
              try
              {
                 Object val = f.get(obj);
                 r += toString(val);
              }
              catch (Exception e) { e.printStackTrace(); }
           }
        }
        r += "]";
        cl = cl.getSuperclass();
     }
     while (cl != null);
     return r;
  }
  . . .
}
```

Код, представленный в листинге 5.7, достаточно сложен для понимания. Циклические ссылки могут привести к бесконечной рекурсии, следовательно, объект `ObjectAnalyzer` должен отслеживать объекты, которые он уже просмотрел. Кроме того, для просмотра массивов необходим другой подход. Детали этого подхода мы рассмотрим в следующем разделе.

Для просмотра содержимого любого объекта можно воспользоваться методом `toString()`. Рассмотрим следующий вызов:

```
ArrayList<Integer> squares = new ArrayList<Integer>();
for (int i = 1; i <= 5; i++) squares.add(i * i);
  System.out.println(new ObjectAnalyzer().toString(squares));
```

В результате выполнения данных строк кода будет выведена следующая информация:

```
java.util.ArrayList[elementData=class java.lang.Object[]{
java.lang.Integer[value=1][][],
java.lang.Integer[value=4][][],
java.lang.Integer[value=9][][],
java.lang.Integer[value=16][][],
java.lang.Integer[value=25][][],null,null,null,null,null},
size=5][modCount=5][][]
```

Универсальный метод `toString()` можно использовать для реализации метода `toString()` в собственных классах. Например, сделать это можно следующим образом:

```
public String toString()
{
   return new ObjectAnalyzer().toString(this);
}
```

Такой способ применения метода `toString()` полезен при отладке программ.

Листинг 5.7. Содержимое файла `ObjectAnalyzerTest.java`

```
1. import java.lang.reflect.*;
2. import java.util.*;
3. /**
4.    * Эта программа демонстрирует использование
5.    * рефлексии для слежения за объектами.
6.    * @version 1.11 2004-02-21
7.    * @author Cay Horstmann
8.    */
9. public class ObjectAnalyzerTest
10.{
11.   public static void main(String[] args)
12.   {
13.      ArrayList<Integer> squares = new ArrayList<Integer>();
14.      for (int i = 1; i <= 5; i++)
15.         squares.add(i * i);
16.      System.out.println(new ObjectAnalyzer().toString(squares));
17.   }
18.}
19.class ObjectAnalyzer
20.{
21.   /**
22.     * Преобразует объект в строку, в которой перечислены все поля.
23.     * @param obj Объект
24.     * @return Строка, в которой указаны имя объекта,
25.     * имена всех полей и их значения
26.     */
27.   public String toString(Object obj)
28.   {
29.      if (obj == null) return "null";
30.      if (visited.contains(obj)) return "...";
31.      visited.add(obj);
32.      Class cl = obj.getClass();
33.      if (cl == String.class) return (String) obj;
34.      if (cl.isArray())
35.      {
36.         String r = cl.getComponentType() + "[]{";
37.         for (int i = 0; i < Array.getLength(obj); i++)
38.         {
39.            if (i > 0) r += ",";
40.            Object val = Array.get(obj, i);
41.            if (cl.getComponentType().isPrimitive()) r += val;
42.            else r += toString(val);
43.         }
44.         return r + "}";
45.      }
46.
```

```
47.      String r = cl.getName();
48.      // Проверка полей класса и всех его суперклассов.
49.      do
50.      {
51.         r += "[";
52.         Field[] fields = cl.getDeclaredFields();
53.         AccessibleObject.setAccessible(fields, true);
54.         // Определение имен и значений всех полей.
55.         for (Field f : fields)
56.         {
57.            if (!Modifier.isStatic(f.getModifiers()))
58.            {
59.               if (!r.endsWith("[")) r += ",";
60.               r += f.getName() + "=";
61.               try
62.               {
63.                  Class t = f.getType();
64.                  Object val = f.get(obj);
65.                  if (t.isPrimitive()) r += val;
66.                  else r += toString(val);
67.               }
68.               catch (Exception e)
69.               {
70.                  e.printStackTrace();
71.               }
72.            }
73.         }
74.         r += "]";
75.         cl = cl.getSuperclass();
76.      }
77.      while (cl != null);
78.
79.      return r;
80.   }
81.
82.   private ArrayList<Object> visited = new ArrayList<Object>();
83. }
```

API java.lang.reflect.AccessibleObject 1.2

- `void setAccessible(boolean flag)`

  Устанавливает признак доступности заданного объекта рефлексии. Значение `true` свидетельствует, что проверка доступа к полю отменена и приватные поля теперь доступны.

- `boolean isAccessible()`

  Получает значение признака доступности заданного объекта рефлексии.

- `static void setAccessible(AccessibleObject[] array, boolean flag)`

  Удобный метод для установки признака доступности массива объектов.

API java.lang.Class 1.1

- `Field getField(String name)`
- `Field[] getFields()`

  Возвращает общедоступное поле с указанным именем или массив, содержащий все поля.

- `Field getDeclaredField(String name)`
- `Field[] getDeclaredFields()`

  Возвращает поле с указанным именем, объявленное в классе, или массив, содержащий все поля.

API java.lang.reflect.Field 1.1

- `Object get(Object obj)`

  Возвращает значение поля объекта `obj`, описываемого данным объектом `Field`.

- `void set(Object obj, Object newValue)`

  Устанавливает новое значение поля объекта `obj`, описываемого данным объектом `Field`.

Использование рефлексии для написания кода обобщенного массива

Класс `Array` из пакета `java.lang.reflect` позволяет создавать массивы динамически. Например, используя метод `arrayCopy()`, о котором упоминалось в главе 3, можно увеличить массив, полностью сохранив его текущее содержимое.

Проблема, которую мы собираемся решить, весьма типична. Допустим, что у нас есть заполненный массив заданного типа и нам требуется увеличить его. Кроме того, предположим, что нам надоело каждый раз вручную создавать необходимый для этого код. Мы хотим написать обобщенный метод, позволяющий увеличивать размер массива, сохраняя его предыдущее содержание.

```
Employee[] a = new Employee[100];
...
// Массив полон.
a = (Employee[])arrayGrow(a);
```

Как написать такой метод? Нам поможет тот факт, что массив `Employee[]` можно преобразовать в массив `Object[]`. Звучит многообещающе. Итак, предпримем первую попытку создать обобщенный метод. Мы просто увеличим массив на 10%+10 элементов (поскольку одного увеличения на 10% для небольших массивов недостаточно).

```
static Object[] badArrayGrow(Object[] a) // Бесполезно.
{
   int newLength = a.length * 11 / 10 + 10;
   Object[] newArray = new Object[newLength];
   System.arraycopy(a, 0, newArray, 0 , a.length);
   return newArray;
}
```

Однако существует проблема, связанная с фактическим *использованием* полученного в результате массива. Этот массив имеет тип `Object[]`, поскольку он создан с помощью выражения

```
new Object[newLength]
```

Массив класса `Object[]` не может быть преобразован в массив класса `Employee[]`. При попытке сделать это возникнет исключение `ClassCastException`. Проблема в том, что, как мы уже указывали, в ходе выполнения программы система запоминает первоначальный тип элементов массива, т.е. тип, указанный в операции `new`. Можно временно преобразовать массив типа `Employee[]` в массив `Object[]` и обратно, но массив, изначально созданный как массив `Object[]`, преобразовать в массив `Employee[]` невозможно. Чтобы создать подходящий обобщенный метод, нужно научиться создавать новый массив, тип которого совпадал бы с типом исходного массива. Для этого нам нужны методы класса `Array` из пакета `java.lang.reflect`, особенно метод `newInstance()`, создающий новый массив. Тип элементов массива и требуемая длина должны передаваться в качестве параметров методу:

```
Object newArray = Array.newInstance(componentType, newLength);
```

Чтобы сделать это, нам нужно уметь определять длину и тип элементов нового массива.

Длину можно получить с помощью метода `Array.getLength()`. Статический метод `getLength()` класса `Array` возвращает длину любого массива. Чтобы определить тип элементов нового массива, нужно выполнить следующие действия.

1. Определить, какому классу принадлежит объект `a`.
2. Убедиться, что он действительно является массивом.
3. Воспользоваться методом `getComponentType()` класса `Class` (определен лишь для объектов, представляющих собой массивы) и получить требуемый тип массива.

Почему метод `getLength()` принадлежит классу `Array`, а `getComponentType()` — классу `Class`? Вряд ли это известно кому-либо, кроме разработчиков этих классов. Приходится принимать существующее распределение методов по классам.

В итоге мы получим следующий код:

```
static Object goodArrayGrow(Object a) // Правильный код.
{
  Class cl = a.getClass();
  if (!cl.isArray()) return null;
  Class componentType = cl.getComponentType();
  int length = Array.getLength(a);
  int newLength = length * 11/10 +10;
  Object newArray = Array.newInstance(componentType, newLength);
  System.arraycopy(a, 0, newArray, 0 , length);
  return newArray;
}
```

Отметим, что метод `arrayGrow()` можно применять для увеличения массива любого типа, а не только массива объектов:

```
int[] a = { 1, 2, 3, 4 };
a = (int[]) goodArrayGrow(a);
```

Для этого параметр метода `arrayGrow()` объявляется как `Object`, а не как *массив объектов* `Object` (т.е. `Object[]`). Массив типа `int[]` можно преобразовать в объект типа `Object`, но не в массив, состоящий из экземпляров `Object`!

В листинге 5.8 показано, как следует увеличивать размеры массивов разных типов. Отметим, что приведение типа значения, возвращаемого методом `badArrayGrow()`, вызовет исключение.



На заметку! Мы приводим эту программы, чтобы проиллюстрировать работу с массивами посредством рефлексии. Если вы просто хотите увеличить массив, используйте метод `copyOf` класса `Arrays`:

```
Employee[] a = new Employee[100];
. . .
// массив полон
a = Arrays.copyOf(a, a.length * 11 / 10 + 10);
```

Листинг 5.8. Содержимое файла `ArrayGrowTest.java`

```
1. import java.lang.reflect.*;
2.
3. /**
4.    * Эта программа демонстрирует использование
5.    * рефлексии для манипулирования массивами.
6.    * @version 1.01 2004-02-21
7.    * @author Cay Horstmann
8.    */
9. public class ArrayGrowTest
10. {
11.    public static void main(String[] args)
12.    {
13.       int[] a = { 1, 2, 3 };
14.       a = (int[]) goodArrayGrow(a);
15.       arrayPrint(a);
16.
17.       String[] b = { "Tom", "Dick", "Harry" };
18.       b = (String[]) goodArrayGrow(b);
19.       arrayPrint(b);
20.
21.       System.out.println(
22.          "Следующий вызов сгенерирует исключение.");
23.       b = (String[]) badArrayGrow(b);
24.    }
25.
26.    /**
27.       * Этот метод пытается увеличить размер массива, создавая
28.       * новый массив и копируя туда все элементы старого.
29.       * @param a Массив, подлежащий увеличению
30.       * @return Увеличенный массив, содержащий все элементы a.
31.       * Возвращаемый массив имеет тип Object[],
32.       * который не совпадает с типом массива a.
33.       */
34.    static Object[] badArrayGrow(Object[] a)
35.    {
36.       int newLength = a.length * 11 / 10 + 10;
37.       Object[] newArray = new Object[newLength];
38.       System.arraycopy(a, 0, newArray, 0, a.length);
39.       return newArray;
40.    }
41.
```

```
42.   /**
43.     * Этот метод увеличивает размер массива, создавая новый
44.     * массив того же типа и копируя туда все имеющиеся элементы.
45.     * @param a Массив, подлежащий увеличению. Может состоять из
46.     *   объектов или значений простых типов.
47.     * @return Увеличенный массив, содержащий все элементы массива a.
48.     */
49.   static Object goodArrayGrow(Object a)
50.   {
51.      Class cl = a.getClass();
52.      if (!cl.isArray()) return null;
53.      Class componentType = cl.getComponentType();
54.      int length = Array.getLength(a);
55.      int newLength = length * 11 / 10 + 10;
56.
57.      Object newArray =
58.         Array.newInstance(componentType, newLength);
59.      System.arraycopy(a, 0, newArray, 0, length);
60.      return newArray;
61.   }
62.
63.   /**
64.     * Удобный метод вывода всех элементов массива.
65.     * @param a Массив для вывода. Может состоять из
66.     *  объектов или значений простых типов.
67.     */
68.   static void arrayPrint(Object a)
69.   {
70.      Class cl = a.getClass();
71.      if (!cl.isArray()) return;
72.      Class componentType = cl.getComponentType();
73.      int length = Array.getLength(a);
74.      System.out.print(componentType.getName()
75.         + "[" + length + "] = { ");
76.      for (int i = 0; i < Array.getLength(a); i++)
77.         System.out.print(Array.get(a, i) + " ");
78.      System.out.println("}");
79.   }
80. }
```

API java.lang.reflect.Array 1.1

- `static Object get(Object array, int index)`
- `static` *xxx* `get`*Xxx*`(Object array, int index)`

  (Символы *xxx* обозначают простые типы `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`.) Эти методы возвращают значение элемента указанного массива, имеющего заданный индекс.

- `static void set(Object array, int index, Object newValue)`
- `static set`*Xxx*`(Object array, int index,` *xxx* `newValue)`

  (Символы *xxx* обозначают простые типы `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`.) Эти методы присваивают новое значение элементу указанного массива, имеющему заданный индекс.

- `static int getLength(Object array)`

  Возвращает длину указанного массива.

- `static Object newInstance(Class componentType, int length)`
- `static Object newInstance(Class componentType, int[] lengths)`

  Возвращает новый массив, состоящий из заданных компонентов и имеющий указанную размерность.

Указатели на методы!

На первый взгляд кажется, что в языке Java не предусмотрены указатели на методы, т.е. нет способа передать одному методу адрес другого, чтобы затем метод обратился по заданному адресу. Разработчики языка Java заявили, что указатели на методы небезопасны и часто порождают ошибки, а действия, для которых часто применяются такие указатели, удобно выполнять с помощью *интерфейсов* (они будут обсуждаться в следующей главе). Однако на самом деле, работая с Java 1.1 и более поздними версиями, можно создавать указатели на методы. Такая возможность является побочным (возможно, случайным) продуктом механизма рефлексии.



На заметку! Среди нестандартных расширений языка Java, которые компания Microsoft внесла в свой язык J++ (и в появившийся затем C#), есть тип указателей на методы, отличающийся от класса `Method`, обсуждаемого в этом разделе. Они называются *делегатами*. Однако внутренние классы, которые рассматриваются в следующей главе — более полезный и универсальный механизм, чем делегаты.

Чтобы понять, как работают указатели на методы, вспомните, что поле объекта можно проверить с помощью метода `get()` класса `Field`. Аналогично класс `Method` содержит метод `invoke()`, позволяющий вызывать метод, содержащийся в текущем объекте этого класса:

```
Object invoke(Object obj, Object... args)
```

Первый параметр является неявным, а остальные объекты представляют собой явные параметры. (До появления Java SE 5.0 приходилось передавать методу массив объектов или, если явные параметры отсутствовали, задавать значение `null`.)

Если метод статический, первый параметр игнорируется; вместо него можно указать `null`. Например, если `m1` представляет метод `getName()` класса `Employee`, можно использовать следующий оператор:

```
String n = (String) m1.invoke(harry);
```

Как это уже было с методами `get()` и `set()` класса `Field`, если параметр или возвращаемое значение не являются объектами какого-либо класса, а представляет собой простой тип, возникает проблема. При работе с версиями, предшествующими Java SE 5.0, нужно было сначала преобразовать все базовые типы в соответствующие типы интерфейсных классов. С появлением Java SE 5.0 можно полагаться на автоматическое преобразование простых типов в классы.

Если возвращаемое значение относится к простому типу, метод `invoke()` возвращает объект интерфейсного класса. Предположим, что `m2` представляет метод `raiseSalary()` класса `Employee`. В результате выполнения метода будет возвращен объект `Double`, после чего вы сможете выполнить приведение типов. В Java SE 5.0 преобразование произойдет автоматически.

```
double s = (Double) m2.invoke(harry);
```

Как получить объект Method? Можно, конечно, вызвать метод getDeclaredMethods() и выполнить поиск среди возвращаемого массива объектов типа Method. Кроме того, можно вызвать метод getMethod() класса Class. Его действие можно сравнить с методом getField(), получающим строку с указанием имени поля и возвращающим объект Field. Однако методов с одним и тем же именем может быть несколько, и следует тщательно выбирать требуемый. По этой причине нужно предусмотреть массив, содержащий правильные типы параметров. Сигнатура метода getMethod() выглядит следующим образом:

```
Method getMethod(String name, Class... parameterTypes)
```

Рассмотрим, например, как можно получить указатели на методы getName() и raiseSalary() класса Employee:

```
Method m1 = Employee.class.getMethod("getName");
Method m2 = Employee.class.getMethod("raiseSalary", double.class);
```

До появления Java SE 5.0 приходилось размещать объекты Class в массиве либо, при отсутствии параметров, указывать значение null.

Теперь, выяснив правила использования объектов Method, рассмотрим, как они работают. В листинге 5.9 приведена программа, которая выводит таблицу значений математической функции вроде Math.sqrt или Math.sin. Результат работы программы выглядят следующим образом:

```
public static native double java.lang.Math.sqrt(double)
     1.0000 |     1.0000
     2.0000 |     1.4142
     3.0000 |     1.7321
     4.0000 |     2.0000
     5.0000 |     2.2361
     6.0000 |     2.4495
     7.0000 |     2.6458
     8.0000 |     2.8284
     9.0000 |     3.0000
    10.0000 |     3.1623
```

Разумеется, код, осуществляющий вывод таблицы на экран, не зависит от конкретной функции:

```
double dx = (to - from) / (n - 1);
for (double x = from; x <= to; x += dx)
{
   double y = (Double) f.invoke(null, x);
   System.out.printf("%10.4f | %10.4f%n" + y, x, y);
}
```

Здесь f — это объект Method. Поскольку мы вызываем статический метод, то в качестве первого параметра invoke передается null.

Для того чтобы вывести таблицу со значениями функции Math.sqrt, мы используем следующую строку кода:

```
Math.class.getMethod("sqrt", double.class)
```

При вызове getMethod() передается имя метода sqrt класса Math и параметр типа double.

В листинге 5.9 приведен полный код универсальной программы вывода значений функции.

Листинг 5.9. Содержимое файла `MethodPointerTest.java`

```
1. import java.lang.reflect.*;
2.
3. /**
4.    * Эта программа демонстрирует использование
5.    * рефлексии для вызова методов.
6.    * @version 1.1 2004-02-21
7.    * @author Cay Horstmann
8.    */
9. public class MethodPointerTest
10.{
11.   public static void main(String[] args) throws Exception
12.   {
13.      // Получение указателей на методы square и sqrt.
14.      Method square = MethodPointerTest.class.getMethod("square",
15.         double.class);
16.      Method sqrt = Math.class.getMethod("sqrt", double.class);
17.
18.      // Вывод значений x и y в виде таблицы.
19.
20.      printTable(1, 10, 10, square);
21.      printTable(1, 10, 10, sqrt);
22.   }
23.
24.   /**
25.     * Возвращает квадрат числа.
26.     * @param x Число
27.     * @return Квадрат числа
28.     */
29.   public static double square(double x)
30.   {
31.      return x * x;
32.   }
33.
34.   /**
35.     * Выводит таблицу значений x и y метода.
36.     * @param from Нижняя граница значений x
37.     * @param to Верхняя граница значений x
38.     * @param n Количество строк в таблице
39.     * @param f Метод, получающий и возвращающий значения типа double
40.     */
41.   public static void printTable(double from, double to, int n, Method f)
42.   {
43.      // Вывод названия метода в качестве заголовка таблицы.
44.      System.out.println(f);
45.
46.      double dx = (to - from) / (n - 1);
47.
48.      for (double x = from; x <= to; x += dx)
49.      {
50.         try
51.         {
52.            double y = (Double) f.invoke(null, x);
53.            System.out.printf("%10.4f | %10.4f%n", x, y);
54.         }
```

```
55.         catch (Exception e)
56.         {
57.            e.printStackTrace();
58.         }
59.      }
60.   }
61. }
```

Таким образом, с помощью объектов `Method` можно делать то же, что и посредством указателей на функции в языке C (или делегирования в C#). Как и в C, такой стиль программирования обычно неудобен и часто приводит к ошибкам. Что случится, если мы вызовем метод с неправильными параметрами? Метод `invoke()` сгенерирует исключение.

Кроме того, и параметры, и возвращаемое значение метода `invoke()` обязательно должны иметь тип `Object`. Это значит, что нам придется осуществлять приведение типов, в результате компилятор не станет проверять программу. Следовательно, ошибки проявятся только при ее тестировании, когда исправить их будет значительно сложнее. Более того, программа, использующая механизм рефлексии для получения указателей на методы, работает гораздо медленнее, чем программа, непосредственно вызывающая эти методы.

По этой причине мы рекомендуем использовать объекты `Method` только в случае крайней необходимости. Намного лучше применять интерфейсы и вложенные классы (тема следующей главы). В частности, вслед за разработчиками языка Java мы повторим, что использовать объекты `Method` для функций обратного вызова не следует. Для этой цели хорошо подходят интерфейсы (см. следующую главу), позволяющие создавать программы, которые работают намного быстрее и надежнее.

API `java.lang.reflect.Method 1.1`

- `public Object invoke(Object implicitParameter, Object[] explicit(parameters)`

  Вызывает метод, описанный в объекте, передавая ему заданные параметры и возвращая значение, вычисленное этим методом. Для статических методов в качестве неявного параметра передается `null`. Простые типы следует передавать только в виде интерфейсных классов. Значения простых типов должны извлекаться из интерфейсных классов.

Рекомендации по применению наследования

В завершение данной главы мы предлагаем несколько советов по правильному применению очень полезного механизма наследования.

1. *Размещайте общие операции и поля в суперклассе.*

   По этой причине мы поместили поле `name` в класс `Person` и не стали повторять его в классах `Employee` и `Student`.

2. *Старайтесь не использовать защищенные поля.*

   Некоторые разработчики полагают, что следует “на всякий случай” объявлять большинство полей защищенными, чтобы по мере необходимости подклассы могли к ним обратиться. Однако есть две причины, по которым этот механизм

не гарантирует достаточной защиты. Во-первых, множество подклассов неограниченно — любой программист может создать подкласс вашего класса и написать программу, получающую непосредственный доступ к защищенным полям экземпляра, нарушая инкапсуляцию. Во-вторых, в языке Java к защищенным полям имеют доступ все классы, размещенные в том же пакете, независимо от того, являются они подклассами данного класса или нет.

Однако специализированные методы, которые переопределяются в подклассах, защищенными объявлять полезно. Ярким примером служит метод `clone()`.

3. *Используйте наследование для моделирования отношения "является".*

   Наследование позволяет экономить время и усилия при разработке программ, поэтому некоторые программисты им злоупотребляют. Предположим, нам нужен класс `Contractor`. Сотрудник, нанимаемый по контракту, имеет имя и дату заключения договора, однако у него нет оклада. У него почасовая оплата, причем он работает не так давно, чтобы возникла необходимость повышать оплату его труда. Ниже показано, как можно сделать класс `Contractor` подклассом класса `Employee`, добавив поле `hourlyWage`.

   ```
   class Contractor extends Employee
   {
      ...
      private double hourlyWage;
   }
   ```

   Эта идея *не* слишком удачна, поскольку при этом получается, что каждый сотрудник, нанятый по контракту, имеет и оклад, и почасовую оплату. Если вы попробуете реализовать методы для распечатки чеков или ведомостей, сразу проявится недостаток такого подхода. Программа, которую вам придется написать, будет гораздо длиннее той, которую вы могли бы создать, не прибегая к неоправданному наследованию.

   Отношение "сотрудник с почасовой оплатой/постоянный сотрудник" не удовлетворяет отношению "является". Сотрудники, *нанятые по контракту*, не являются разновидностью *постоянных* сотрудников.

4. *Не используйте наследование, если не все наследуемые методы имеют смысл.*

   Допустим, что нам нужно написать класс `Holiday`. Разумеется, праздники — это разновидность календарных дней, а дни можно представить в виде объектов `GregorianCalendar`, поэтому можно применять наследование:

   ```
   class Holiday extends GregorianCalendar(...)
   ```

   К сожалению, множество праздников при наследовании не закрыто. Среди общедоступных методов класса `GregorianCalendar` есть метод `add()`, который может превратить праздники в будни.

   ```
   Holiday christmas;
   cristmas.add(GregorianCalendar.DAY, 12);
   ```

   Итак, наследование в этом случае не подходит.

5. *Переопределяя метод, не изменяйте ожидаемое поведение класса.*

   При переопределении методов необходимо не только соблюдать синтаксис, но, что гораздо важнее, учитывать поведение класса. При переопределении метода вы не должны без причины изменять его поведение. В этом компилятор не по-

может вам; он не в состоянии проверить, оправдано ли переопределение метода. Предположим, что вы захотели устранить проблему с методом `add()` в классе `Holiday`, переопределив этот метод так, чтобы он, например, не выполнял никаких действий либо возвращал следующий праздничный день. Однако такое переопределение нарушает принцип подстановки. При выполнении приведенной ниже последовательности команд пользователь вправе ожидать определенного результата, независимо от того, является ли `x` экземпляром `GregorianCalendar` или `Holiday`.

```
int d1 = x.get(Calendar.DAY_OF_MONTH);
x.add(Calendar.DAY_OF_MONTH, 1);
int d2 = x.get(Calendar.DAY_OF_MONTH);
System.out.println(d2 - d1);
```

Очевидно, что приведенный пример несколько надуман. Разные пользователи посчитают естественным различное поведение программы. Так, например, можно услышать мнение, что принцип подстановки требует, чтобы в методе `Manager.equals()` не учитывалась премия, поскольку метод `Employee.equals()` игнорирует ее. Подобные споры могут длиться бесконечно и не дать никакого результата. Принимая конкретное решение, следует руководствоваться целями, для которых создается программа.

6. *Используйте полиморфизм, а не информацию о типе.*

   Вспомните о полиморфизме, как только увидите код, имеющий следующий вид:

   ```
   if (x имеет_тип 1)
      действие_1(x);
   else if (x имеет_тип 2)
      действие_2(x);
   ```

   Носят ли *действие_1* и *действие_2* общий характер? Если да, поместите соответствующие методы в общий суперкласс или интерфейс обоих типов. Тогда можно просто вызвать приведенный ниже метод и с помощью механизма динамического связывания, присущего полиморфизму, выполнить правильное действие.

   ```
   x.действие();
   ```

   Код, использующий полиморфизм или реализацию интерфейса, намного легче поддерживать и расширять, чем код, использующий многочисленные проверки типов.

7. *Не злоупотребляйте механизмом рефлексии.*

   Механизм рефлексии позволяет создавать программы с высоким уровнем абстракции, которые определяют поля и методы во время выполнения. Эта возможность чрезвычайно полезна для системного программирования, но для прикладного — практически не нужна. Рефлексия — очень хрупкий механизм, поскольку компилятор не может помочь программисту обнаружить ошибки. Все ошибки проявляются во время выполнения программы и приводят к возникновению исключений.

В этой главе было показано, как в Java поддерживаются основы объектно-ориентированного программирования: классы, наследование и полиморфизм. В следующей главе мы затронем две сложные темы, которые очень важны для эффективного использования Java: интерфейсы и вложенные классы.

Глава 6

ИНТЕРФЕЙСЫ И ВЛОЖЕННЫЕ КЛАССЫ

В ЭТОЙ ГЛАВЕ...

- Интерфейсы
- Клонирование объектов
- Интерфейсы и обратный вызов
- Вложенные классы
- Прокси-классы

Теперь вы знаете все основные инструменты, предназначенные для объектно-ориентированного программирования на языке Java. В этой главе мы ознакомим вас еще с несколькими усовершенствованными способами программирования. Несмотря на то что эти способы далеко не очевидны, владеть ими должен каждый профессиональный программист.

Первый из них — *интерфейсы* — это способ указывать, *что* должны делать классы, не описывая, *как* они это должны делать. Классы могут *реализовывать* один или несколько интерфейсов. Если возникает необходимость использовать интерфейс, применяются объекты этих классов. Изучив интерфейсы, мы рассмотрим клонирование объектов (которое иногда называется "*глубоким" копированием*). Клоном объекта называется новый объект, имеющий точно такое же состояние. В частности, вы можете модифицировать клон, никак не затрагивая оригинал.

Далее мы перейдем к механизму, основанному на применении *вложенных классов* (inner class). С технической точки зрения вложенные классы довольно сложны — они определяются внутри других классов, а их методы имеют доступ к полям окружающего класса. Вложенные классы полезны при проектировании совокупности взаимосвязанных классов. В частности, они позволяют создавать согласованный, высокопрофессиональный код для реализации событийно-ориентированного графического пользовательского интерфейса.

Глава завершается обсуждением прокси-объектов, реализующих произвольные интерфейсы. Прокси-объекты представляют собой весьма специализированные конструкции, полезные для создания инструментов системного программирования. Начинающие программисты могут спокойно пропустить этот раздел.

Интерфейсы

Интерфейс в языке Java не является классом. Он представляет собой множество *требований*, предъявляемых к классу, который должен соответствовать интерфейсу.

Обычно разработчик, который собирается использовать работу другого программиста для решения какой-либо задачи, заявляет: "Если ваш класс будет соответствовать определенному интерфейсу, я смогу выполнить свою работу". Рассмотрим конкретный пример. Метод `sort` класса `Array` позволяет упорядочить массив объектов при одном условии: объекты должны принадлежать классам, реализующим интерфейс `Comparable`.

Вот как выглядит этот интерфейс:

```
public interface Comparable
{
   int compareTo(Object other);
}
```

Это значит, что любой класс, реализующий интерфейс `Comparable`, должен иметь метод `compareTo()`, получающий параметр `Object` и возвращающий целое число.



На заметку! В Java SE 5.0 интерфейс `Comparable` стал обобщенным типом.

```
public interface Comparable<T>
{
    int compareTo(T other);    // Параметр относится к типу T
}
```

Например, если класс реализует интерфейс `Comparable<Employee>`, он должен содержать следующий метод:

```
int compareTo(Employee other);
```

При желании вы по-прежнему можете использовать "низкоуровневый" тип `Comparable`, но при этом вам придется явным образом приводить параметр метода `compareTo()` к требуемому типу.

Все методы интерфейса автоматически считаются общедоступными, поэтому, объявляя метод в интерфейсе, указывать модификатор `public` не обязательно.

Разумеется, существует и неявное требование: метод `compareTo()` должен на самом деле уметь сравнивать два объекта и возвращать признак того, что один из них больше другого. Предполагается, что метод возвращает отрицательное значение, если объект `x` меньше объекта `y`, нуль — если они равны, и положительное число — в противном случае.

Этот конкретный интерфейс имеет единственный метод. Некоторые интерфейсы могут иметь более одного метода. Как вы узнаете далее, с помощью интерфейсов можно также объявлять константы. Однако еще важнее тот факт, что интерфейсы *не могут* реализовываться в виде объектов. В интерфейсах нет ни полей, ни тела методов; они содержатся в классах, реализующих соответствующие интерфейсы. Таким образом, интерфейс можно воспринимать как абстрактный класс, лишенный каких-либо полей экземпляра. Однако между двумя этими понятиями есть существенное различие, и позднее мы подробно проанализируем его.

Теперь предположим, что мы собираемся использовать метод `sort()` класса `Array` для сортировки объектов класса `Employee`. В этом случае класс `Employee` должен *реализовать* интерфейс `Comparable`.

Для того чтобы класс реализовал интерфейс, нужно выполнить два действия.

1. Объявить, что класс реализует интерфейс.
2. Определить в классе все методы, указанные в интерфейсе.

Чтобы объявить, что класс реализует интерфейс, используется ключевое слово `implements`:

```
class Employee implements Comparable
```

Разумеется, теперь нужно реализовать метод `compareTo()`. Например, сравним зарплату сотрудников. Метод `compareTo()` будет возвращать `-1`, если зарплата первого сотрудника меньше зарплаты второго, `0`, если они равны, и `1` в противном случае.

```
public int compareTo(Object otherObject)
{
   Employee other = (Employee) otherObject;
   if (salary < other.salary) return -1;
   if (salary > other.salary) return 1;
   return 0;
}
```



Внимание! В составе интерфейса для метода `compareTo()` не указывается модификатор `public`, поскольку все методы *интерфейса* автоматически являются общедоступными. Однако при реализации интерфейса этот модификатор должен быть указан. В противном случае компилятор предположит, что область видимости этого метода ограничивается пакетом, и выдаст предупреждение, что вы пытаетесь изменить уровень доступа.

Начиная с Java SE 5.0, можно использовать более изящное решение, реализовав интерфейс Comparable<Employee>.

```
class Employee implements Comparable<Employee>
{
   public int compareTo(Employee other)
   {
      if (salary < other.salary) return -1;
      if (salary > other.salary) return 1;
      return 0;
   }
    . . .
}
```

Как видите, теперь не приходится приводить Object к требуемому типу.



Совет. Метод compareTo() интерфейса Comparable возвращает целое число. Если объекты не эквивалентны, возвращается положительное или отрицательное значение. Возможность использования в качестве признака неравенства любое число, отличное от нуля, может оказаться полезной при сравнении целочисленных полей. Предположим, что каждый сотрудник имеет уникальный номер id. В этом случае можно вернуть число id - other.id. Это значение будет отрицательным, если номер первого сотрудника меньше номера второго, 0, если они равны, и положительное в противном случае. Учтите только, что диапазон изменения целых чисел должен быть достаточно малым, чтобы вычитание не привело к переполнению. Если известно, что идентификационный номер является неотрицательным или его абсолютное значение не превышает (Integer.Max_Value - 1) / 2, то вы смело можете применять описанный выше способ.

Разумеется, этот прием с вычитанием не работает для чисел с плавающей точкой. Разность salary - other.salary может округляться до нуля, если зарплаты очень близки, но не идентичны.

Теперь ясно, что нужно сделать, чтобы класс смог сортировать объекты, — реализовать в нем метод compareTo(). Это вполне резонно. Должен существовать какой-то способ использовать метод sort() для сравнения объектов. Однако почему нельзя в классе Employee просто предусмотреть метод compareTo(), не реализуя интерфейс Comparable?

Причина заключается в том, что язык Java является *строго типизированным*. Вызывая некий метод, компилятор должен убедиться, что этот метод действительно существует. В теле метода sort() могут находиться операторы наподобие следующих:

```
if (a[i].compareTo(a[j]) > 0)
{
   // Поменять местами объекты a[i] и a[j].
   ...
}
```

Компилятор должен знать, что объект a[i] действительно содержит метод compareTo(). Если переменная a представляет собой массив объектов, реализующих интерфейс Comparable, то существование такого метода гарантируется, поскольку каждый класс, реализующий указанный интерфейс, по определению должен содержать этот метод.



На заметку! Может показаться, что метод sort() класса Array работает только с массивами типа Comparable[] и что компилятор выдаст предупреждение, как только обнаружит вызов метода sort() для массива, элементы которого не реализуют указанный интерфейс. Увы, это не так. Вместо этого метод sort() получает на вход массивы типа Object[] и выполняет приведение типов.

```
// Фрагмент кода из стандартной библиотеки
if (((Comparable) a[i]).compareTo(a[i] > 0)
{
   // Поменять местами объекты a[i] и a[j].
   ...
}
```

Если объект `a[i]` не принадлежит классу, реализующему интерфейс `Comparable`, виртуальная машина генерирует исключение.

В листинге 6.1 приведен код программы сортировки массива, состоящего из объектов класса `Employee`.

Листинг 6.1. Содержимое файла `EmployeeSortTest.java`

```
1. import java.util.*;
2.
3. /**
4.    * Эта программа демонстрирует использование интерфейса Comparable.
5.    * @version 1.30 2004-02-27
6.    * @author Cay Horstmann
7.    */
8. public class EmployeeSortTest
9. {
10.    public static void main(String[] args)
11.    {
12.       Employee[] staff = new Employee[3];
13.
14.       staff[0] = new Employee("Harry Hacker", 35000);
15.       staff[1] = new Employee("Carl Cracker", 75000);
16.       staff[2] = new Employee("Tony Tester", 38000);
17.
18.       Arrays.sort(staff);
19.
20.       // Вывод информации о всех объектах Employee.
21.       for (Employee e : staff)
22.          System.out.println("name=" + e.getName() + ",salary=" + e.getSalary());
23.    }
24. }
25.
26. class Employee implements Comparable<Employee>
27. {
28.    public Employee(String n, double s)
29.    {
30.       name = n;
31.       salary = s;
32.    }
33.
34.    public String getName()
35.    {
36.       return name;
37.    }
38.
39.    public double getSalary()
40.    {
41.       return salary;
42.    }
```

```
43.
44.   public void raiseSalary(double byPercent)
45.   {
46.      double raise = salary * byPercent / 100;
47.      salary += raise;
48.   }
49.
50.   /**
51.     * Сравнивает зарплату сотрудников.
52.     * @param otherObject Другой объект класса Employee
53.     * @return Отрицательное число, если зарплата данного
54.     *    сотрудника меньше, чем зарплата сотрудника,
55.     *    представленного объектом otherObject; 0,
56.     *    если их зарплаты равны, и положительное число
57.     *    в противном случае.
58.     */
59.   public int compareTo(Employee other)
60.   {
61.      if (salary < other.salary) return -1;
62.      if (salary > other.salary) return 1;
63.      return 0;
64.   }
65.
66.   private String name;
67.   private double salary;
68. }
```

API java.lang.Comparable<T> 5.0

- `int compareTo(T other)`

  Сравнивает текущий объект с объектом `other` и возвращает отрицательное целое число, если текущий объект меньше, чем объект `other`, нуль, если они равны между собой, и положительное число в противном случае.

API java.util.Arrays 1.2

- `static void sort(Object[] a)`

  Сортирует элементы массива `a` методом слияния. Все элементы массива должны соответствовать классам, реализующим интерфейс `Comparable`, и быть совместимыми друг с другом.



На заметку! В языке Java есть стандартное требование: "Автор реализации метода должен гарантировать, что для всех объектов `x` и `y` выполняется условие `sgn(x.compareTo(y)) = -sgn(y.compareTo(x))`. (Следовательно, если вызов `y.compareTo(x)` генерирует исключение, значит, то же самое должен делать `x.compareTo(y)`.)" Здесь `sgn` означает *знак* числа: `sgn(n)` равна `-1`, если число `n` отрицательно, `0`, если число `n` равно нулю, и `1`, если число `n` положительное. Другими словами, если поменять явный и неявный параметры метода `compareTo()` местами, знак возвращаемого числа (но не обязательно его фактическое значение) также должен измениться на противоположный.

Что касается метода `equals()`, при наследовании может возникнуть проблема.

Поскольку класс `Manager` расширяет класс `Employee`, он реализует интерфейс `Comparable<Employee>`, а не `Comparable<Manager>`.

Если разработчик класса `Manager` примет решение переопределить метод `compareTo()`, он должен предусмотреть сравнение менеджеров с обычными сотрудниками. Можно привести тип `Employee` к типу `Manager`.

```
class Manager extends Employee
{
    public int compareTo(Employee other)
    {
        Manager otherManager = (Manager) other;     // Так нельзя
        . . .
    }
    . . .
}
```

Однако такой подход нарушает правило "антисимметрии". Если объект `x` относится к классу `Employee`, а объект `y` — к классу `Manager`, то вызов `x.compareTo(y)` не приведет к возникновению исключения — он просто сравнит `x` и `y` как объекты класса `Employee`. Однако обратный вызов `y.compareTo(x)` сгенерирует исключение `ClassCastException`.

Аналогичная ситуация возникает при программировании метода `equals()`, которая обсуждалась в главе 5.

Если подклассы имеют различные критерии сравнения, то считаем, что сравнение недопустимо. Таким образом, каждый метод `compareTo()` должен начинаться со следующей проверки:

```
if (getClass() != other.getClass()) throw new ClassCastException();
```

Если существует общий алгоритм сравнения объектов подклассов, надо реализовать в суперклассе единый метод `compareTo()` и объявить его как `final`.

Предположим, например, что на предприятии, независимо от заработной платы, менеджеры считаются выше рядовых сотрудников. Как быть с другими подклассами, например `Executive` и `Secretary`? Если вы хотите учредить нечто вроде табеля о рангах, введите в состав класса `Employee` метод `rank()`. Тогда в каждом подклассе метод `rank()` должен переопределяться и результаты его работы учитываться при выполнении метода `compareTo()`.

Свойства интерфейсов

Интерфейсы — это не классы. В частности, с помощью операции `new` невозможно создать экземпляр интерфейса:

```
x = new Comparable(...); // Неправильно.
```

Но несмотря на это, можно объявлять переменные, ссылающиеся на интерфейсы:

```
Comparable x;            // Правильно.
```

Реально такая переменная ссылается на объект класса, реализующего данный интерфейс:

```
x = new Employee(...);   // Правильно, если класс Employee
                         // реализует интерфейс Comparable.
```

Как известно, операция `instanceof` проверяет, принадлежит ли объект заданному классу. Однако с помощью этой операции можно также проверять, реализует ли объект заданный интерфейс.

```
if (anObject instanceof Comparable) { ... }
```

Аналогично классам, интерфейсы также могут образовывать иерархию наследования. Это позволяет создавать цепочки интерфейсов в направлении от более абстрактных к более специализированным. Допустим, что у нас есть интерфейс `Moveable` следующего вида:

```
public interface Moveable
{
   void move(double x, double y);
}
```

Мы можем создать интерфейс `Powered`, расширяющий интерфейс `Moveable`:

```
public interface Powered extends Moveable
{
   double milesPerGallon();
}
```

Хотя в интерфейсе может не быть ни полей, ни статических методов, в нем можно объявлять константы:

```
public interface Powered extends Moveable
{
   double milesPerGallon();

   // Общедоступная статическая константа.
   double SPEED_LIMIT = 95;
}
```

Напомним, что все методы в интерфейсе автоматически считаются общедоступными. Аналогично, при объявлении константы всегда предполагаются модификаторы `public static final`.



На заметку! Если при объявлении метода в составе интерфейса вы укажете ключевое слово `public`, а поле пометите как `public static final`, это не будет ошибкой. Некоторые программисты поступают так для того, чтобы коды их программ были более читабельными. Однако в спецификации языка Java рекомендуется не использовать лишние ключевые слова. (В данной книге мы следовали этим рекомендациям.)

В некоторых интерфейсах объявляются только константы и ни одного метода. Например, в стандартной библиотеке содержится интерфейс `SwingConstants`, определяющий константы `NORTH`, `SOUTH`, `HORIZONTAL` и т.д. Любой класс, реализующий интерфейс `SwingContants`, автоматически наследует эти константы. Его методы могут непосредственно ссылаться, например, на константу `NORTH`, не используя громоздкое обозначение `SwingConstants.NORTH`. Однако многие специалисты считают, что такое использование интерфейсов со временем изживет себя. Мы также не рекомендуем применять их с этой целью.

В языке Java любой класс может иметь только один суперкласс, но при этом любой класс может реализовывать *несколько* интерфейсов. Это позволяет максимально гибко определять поведение класса. Например, в языке Java есть очень важный интерфейс `Cloneable`. (Этот интерфейс детально рассматривается в следующем разделе.) Если некий класс реализует интерфейс `Cloneable`, то для создания точных копий его объектов можно применять метод `clone()` класса `Object`. Далее, предположим, что мы хотим не только создавать клоны объектов нашего класса, но и сравнивать их между собой. Тогда нужно просто реализовать в классе оба интерфейса, `Clonable` и `Comparable`:

```
class Employee emplements Cloneable, Comparable
```

Для разделения имен интерфейсов, задающих свойства классов, используется запятая.

Интерфейсы и абстрактные классы

Если вы помните содержание раздела главы 5, посвященного абстрактным классам, у вас, возможно, возник вопрос, зачем разработчики языка Java создали механизм интерфейсов. Почему бы не сделать интерфейс `Comparable` абстрактным классом?

```
abstract class Comparable // Почему бы и нет?
{
  public abstract int compareTo(Object other);
}
```

В этом случае класс `Employee` мог бы просто расширять абстрактный класс и реализовывать метод `compareTo()`:

```
class Employee extends Comparable // Почему бы и нет?
{
  public int compareTo(Object other) { ... }
}
```

К сожалению, это породило бы массу проблем, связанных с использованием абстрактного базового класса для выражения обобщенного свойства. У класса может быть только один родитель. Предположим, что класс `Employee` уже является подклассом какого-нибудь другого класса, скажем, `Person`. Значит, еще одного родительского класса у него быть не может:

```
class Employee extends Person, Comparable // Ошибка!.
```

В то же время каждый класс может реализовывать сколько угодно интерфейсов:

```
class Employee extends Person implements Conparable // Правильно.
```

В других языках программирования, в частности в языке C++, классы могут иметь несколько суперклассов. Это свойство называется *множественным наследованием*. Разработчики языка Java решили не поддерживать множественное наследование, поскольку оно делает язык либо слишком сложным (как C++), либо менее эффективным (как Eiffel).

В то же время интерфейсы предоставляют большинство возможностей множественного наследования, не усложняя язык и не снижая эффективность.



На заметку! В языке C++ есть множественное наследование. Это вызывает много сложностей, связанных с виртуальными базовыми классами, правилами доминирования и приведением типов указателей. Множественное наследование используют лишь немногие программисты, работающие на C++. Некоторые никогда его не применяют, остальные рекомендуют использовать множественное наследование только в "смешанном" виде. Это значит, что первичный базовый класс описывает родительский объект, а дополнительные базовые классы (так называемые *смеси* (mix-ins)) описывают вспомогательные свойства. Этот стиль напоминает классы из языка Java, имеющие только один базовый класс и дополнительные интерфейсы. Однако в C++ смеси могут добавлять некоторые свойства по умолчанию, в то время как интерфейсы в языке Java этого делать не могут.

Клонирование объектов

После создания копии переменной и оригинал, и копия представляют собой ссылки на один и тот же объект (рис. 6.1). Это значит, что изменение одной переменной повлечет изменение другой.

```
Employee original = new Employee("John Public", 50000);
Employee copy = original;
copy.raiseSalary(10);  // Оригинал тоже изменился!
```

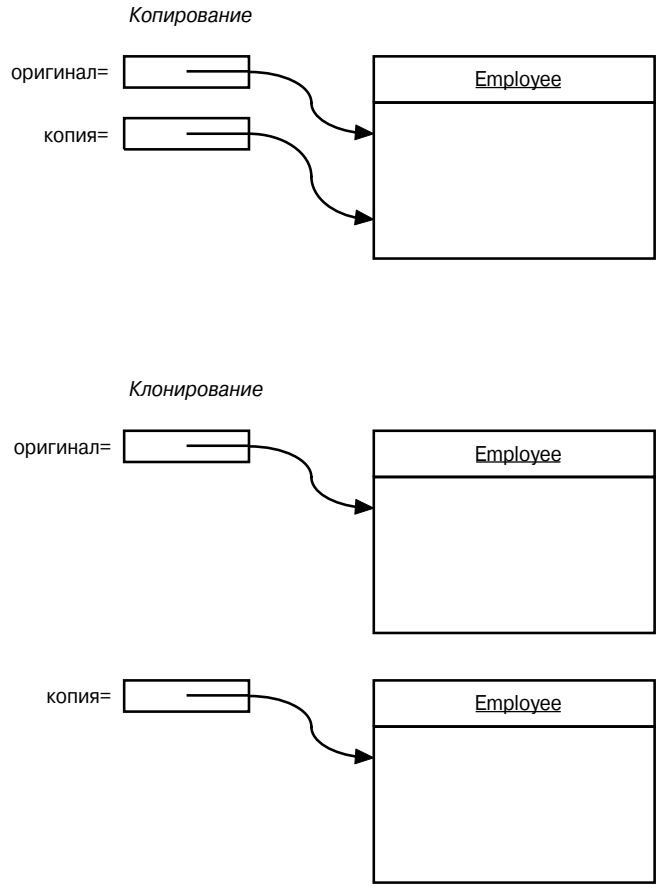


Рис. 6.1. Копирование и клонирование

Если нужно, чтобы переменная `copy` представляла собой новый объект, в первый момент своего существования идентичный объекту `original`, но совершенно независимый от него, используйте метод `clone()`.

```
Employee copy = original.clone();
copy.raiseSalary(10); // OK! Оригинал не изменился!
```

Однако не все так просто. Метод `clone()` является защищенным (`protected`) методом класса `Object`, т.е. его нельзя вызвать непосредственно. Только класс `Employee`

может клонировать объекты своего класса. С этим ограничением связана существенная причина. Подумайте о способе, которым класс `Object` может реализовать метод `clone()`. Ему вообще ничего не известно об объекте, поэтому он может копировать лишь поля. Если все поля класса являются числовыми или имеют другой основной тип, их копирование выполняется прекрасно. Однако если объект содержит ссылку на подобъект, то и оригинал, и клонированные объекты будут совместно использовать одну и ту же информацию.

Чтобы проиллюстрировать это явление, рассмотрим класс `Employee`, который, начиная с главы 4, мы используем для демонстрации различных аспектов работы с объектами. На рис. 6.2 показано, что происходит, когда метод `clone()` класса `Object` применяется к объекту класса `Employee`. Как видим, операция клонирования по умолчанию является "поверхностной" — она не клонирует объекты, на которые есть ссылки внутри объектов.

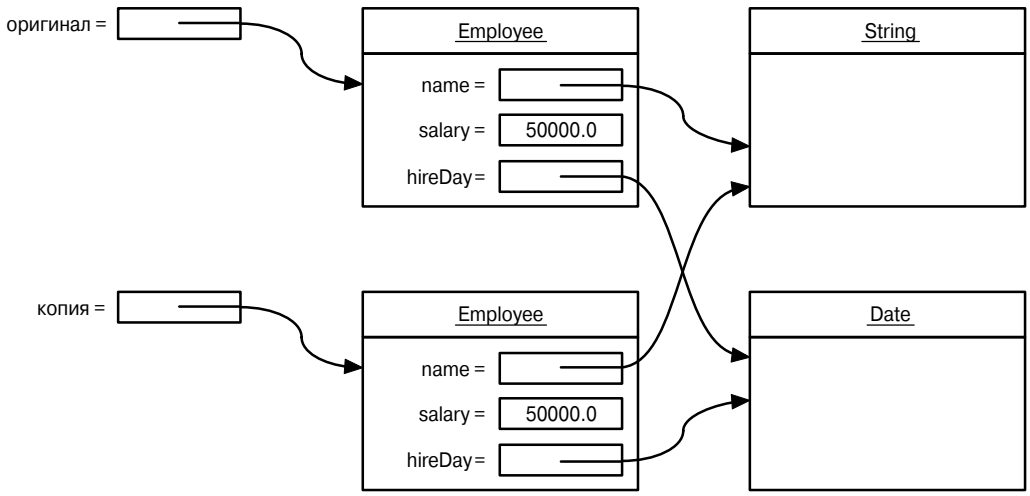


Рис. 6.2. "Поверхностная" копия

Хорошо ли это? Все зависит от конкретной ситуации. Если подобъект, используемый совместно как оригиналом, так и "поверхностным" клоном, является *неизменяемым* (immutable), это вполне безопасно. Такое происходит, если подобъект является экземпляром неизменяемого класса, например `String`. Подобъект также может оставаться постоянным на протяжении всей жизни содержащего его объекта, не подвергаясь воздействию модифицирующих методов либо методов, вычисляющих ссылку на него.

Достаточно часто, однако, подобъекты подвергаются изменениям, поэтому приходится переопределять метод `clone()`, чтобы выполнить "*глубокое" копирование*, которое позволяет клонировать подобъекты наряду с содержащими их объектами. В нашем примере поле `hireDay` ссылается на экземпляр изменяемого класса `Date`.

Для каждого класса нужно принять следующие решения.

1. Достаточно ли метода `clone()` по умолчанию?
2. Можно ли дополнить метод `clone()`, используемый по умолчанию, применяя его к изменяемым объектам?
3. Следует ли применять метод `clone()`?

Считаем, что на третий вопрос получен положительный ответ. Для того чтобы программист мог использовать метод clone(), предлагаемый по умолчанию, либо модифицированный метод clone(), класс должен удовлетворять перечисленным ниже требованиям.

1. Реализовать интерфейс Cloneable.
2. В нем должен быть переопределен метод clone() с модификатором доступа public.



На заметку! Метод clone() в классе Object объявлен как protected, поэтому его нельзя просто вызвать с помощью выражения anObject.clone(). Однако разве недоступны защищенные методы для любого подкласса, и не является ли каждый класс подклассом класса Object? К счастью, правила защищенного доступа более тонкие (см. главу 5). Подкласс может вызвать защищенный метод clone() только для клонирования *своих собственных* объектов. Чтобы клонировать другие объекты, метод clone() следует переопределить и разрешить клонирование объектов любым методом.

В данном случае интерфейс Cloneable используется не совсем привычным для вас образом. В частности, в нем не объявляется метод clone() — этот метод наследуется от класса Object. Интерфейс служит меткой, указывающей на то, что в данном случае разработчик класса понимает, как выполняется процесс клонирования. Объекты настолько настороженно относятся к клонированию, что если объект требует выполнения данной операции, но не реализует интерфейс Cloneable, генерируется исключение.



На заметку! Интерфейс Cloneable — один из немногих *интерфейсов-маркеров* (tagged interfaces) в языке Java. (Некоторые программисты называют их *маркерными интерфейсами*.) Напомним, что такие интерфейсы, как Comparable, должны гарантировать, что некий класс реализует конкретный метод или набор методов. Интерфейсы-маркеры не имеют методов; их единственное предназначение — разрешить использовать операцию instanceof для проверки типа.

```
if (obj instanceof Cloneable) ...
```

Мы не рекомендуем применять данную технологию.

Даже если применение метода clone() по умолчанию (поверхностное копирование) вполне допустимо, в любом случае нужно реализовать интерфейс Cloneable, переопределить метод clone(), сделав его общедоступным и вызвать метод super.clone(). Рассмотрим пример.

```
class Employee implements Cloneable
{
   // Область видимости расширяется до public,
   // и изменяется тип возвращаемого значения
   public Employee clone() throws CloneNotSupportedException
   {
      return super.clone();
   }
   . . .
}
```



На заметку! До появления Java SE 5.0 метод clone() всегда возвращал объект Object. Ковариантные возвращаемые типы, поддерживаемые Java SE 5.0, позволяют указывать в методе clone() корректный тип возвращаемого значения.

Рассмотренный выше метод clone() не добавляет никаких новых функциональных возможностей к методу Object.clone(), реализующему "поверхностное" копирование. Чтобы реализовать "глубокое" копирование, надо приложить дополнительные усилия и организовать клонирование изменяемых полей экземпляра.

Ниже приведен пример метода clone(), выполняющего "глубокое" копирование.

```
class Employee implements Cloneable
{
   . . .
   public Object clone() throws CloneNotSupportedException
   {
      // Вызов метода Object.clone().
      Employee cloned = (Employee) super.clone();

      // Клонирование изменяемых полей.
      cloned.hireDay = (Date) hireDay.clone()

      return cloned;
   }
}
```

Метод clone() класса Object может генерировать исключение CloneNotSupported-Exception. Это происходит в том случае, если clone() вызывается для объекта, не реализующего интерфейс Cloneable. Поскольку и класс Employee, и класс Date реализуют этот интерфейс, исключение не генерируется. Однако компилятор не знает этого, поэтому нам надо объявить о возможном исключении следующим образом:

```
public Employee clone() throws CloneNotSupportedException
```

Не лучше ли было бы вместо этого предусмотреть обработку исключения?

```
public Employee clone()
{
   try
   {
      return super.clone();
   }
   catch (CloneNotSupportedException e) { return null; }
   // Это не поизойдет, так как классы реализуют
   // интерфейс Cloneable
}
```

Такое решение вполне подходит для классов final. В других случаях уместнее прибегнуть к помощи ключевого слова throws. При этом у подкласса останется возможность сгенерировать исключение CloneNotSupportedException, в случае если он не сможет выполнить клонирование.

Клонируя подклассы, надо соблюдать осторожность. Например, если вы определите метод clone() в классе Employee, другие программисты смогут использовать его для клонирования объектов Manager. Сможет ли метод clone() из класса Employee справиться с данной задачей? Это зависит от набора полей, объявленных в классе Manager. В нашем случае проблемы не возникнут, поскольку поле bonus относится к простому типу. Но если в класс Manager будет включено поле, требующее "глубокого" копирования, клонирование перестанет выполняться. Нет никакой гарантии, что в подклассе реализован метод clone(), корректно решающий поставленную задачу. По этой причине в классе Object метод clone() объявлен как protected. Однако если

вы хотите, чтобы пользователи ваших классов могли вызывать `clone()`, подобная "роскошь" остается недоступной.

Следует ли реализовывать метод `clone()` в своих классах? Если пользователям требуется "глубокое" копирование, то ответ, конечно же, должен быть положительным. Некоторые специалисты считают, что от метода `clone()` надо вовсе отказаться и реализовать вместо него другой метод, выполняющий аналогичную задачу. Мы согласны, что метод `clone()` — не совсем удачное решение, но если вы передадите его функции другому методу, вы столкнетесь с теми же проблемами. Как бы то ни было, клонирование поддерживается не очень часто. Достаточно сказать, что метод `clone()` реализован менее чем в 5% классов из стандартной библиотеки.

Программа, код которой приведен в листинге 6.2, клонирует объекты класса `Employee`, затем вызывает два модифицирующих метода. Метод `raiseSalary()` изменяет значение поля `salary`, а метод `setHireDay()` — состояние поля `hireDay`. Ни один из модифицирующих методов не действует на исходный объект, поскольку метод `clone()` переопределен и осуществляет "глубокое" копирование.



На заметку! Все типы массивов имеют общедоступный, а не защищенный метод `clone()`. Вы можете использовать его для создания нового массива, содержащего копии всех элементов. Например:

```
int[] luckyNumbers = { 2, 3, 5, 7, 11, 13 };
int[] cloned = (int[]) luckyNumbers.clone();
cloned[5] = 12; // Не изменяет luckyNumbers[5]
```



На заметку! Во втором томе представлен альтернативный механизм клонирования объектов — с использованием средства сериализации объектов Java. Этот механизм прост в реализации и безопасен, но не слишком эффективен.

Листинг 6.2. Содержимое файла `CloneTest.java`

```
1. import java.util.*;
2.
3. /**
4.    * Эта программа демонстрирует клонирование.
5.    * @version 1.10 2002-07-01
6.    * @author Cay Horstmann
7.    */
8. public class CloneTest
9. {
10.   public static void main(String[] args)
11.   {
12.      try
13.      {
14.         Employee original = new Employee("John Q. Public", 50000);
15.         original.setHireDay(2000, 1, 1);
16.         Employee copy = original.clone();
17.         copy.raiseSalary(10);
18.         copy.setHireDay(2002, 12, 31);
19.         System.out.println("original=" + original);
20.         System.out.println("copy=" + copy);
21.      }
```

```
22.      catch (CloneNotSupportedException e)
23.      {
24.         e.printStackTrace();
25.      }
26.   }
27. }
28.
29. class Employee implements Cloneable
30. {
31.   public Employee(String n, double s)
32.   {
33.      name = n;
34.      salary = s;
35.      hireDay = new Date();
36.   }
37.
38.   public Employee clone() throws CloneNotSupportedException
39.   {
40.      // Вызов Object.clone().
41.      Employee cloned = (Employee)super.clone();
42.
43.      // Клонирование изменяемых полей.
44.      cloned.hireDay = (Date)hireDay.clone();
45.
46.      return cloned;
47.   }
48.
49.   /**
50.     * Установка даты найма на работу.
51.     * @param year Год найма
52.     * @param month Месяц найма
53.     * @param day День найма
54.     */
55.   public void setHireDay(int year, int month, int day)
56.   {
57.      // Пример изменения поля экземпляра.
58.      hireDay = new GregorianCalendar(year, month - 1, day).getTime();
59.   }
60.
61.   public void raiseSalary(double byPercent)
62.   {
63.      double raise = salary * byPercent / 100;
64.      salary += raise;
65.   }
66.
67.   public String toString()
68.   {
69.      return "Employee[name=" + name
70.         + ",salary=" + salary
71.         + ",hireDay=" + hireDay + "]";
72.   }
73.
74.   private String name;
75.   private double salary;
76.   private Date hireDay;
77. }
```

Интерфейсы и обратный вызов

Механизм *обратного вызова* (callback) широко распространен в программировании. При обратном вызове программист задает действия, которые должны выполняться всякий раз, когда происходит некоторое событие. Например, можно задать действие, которое должно быть выполнено после щелчка на некоторой кнопке или при выборе определенного пункта меню. Однако, поскольку вы еще не умеете создавать пользовательский интерфейс, мы рассмотрим похожую, но более простую ситуацию.

Пакет `javax.swing` содержит класс `Timer`, который можно использовать для отсчета интервалов времени. Например, если в программе предусмотрены часы, то с помощью класса `Timer` можно отсчитывать каждую секунду и обновлять циферблат часов.

Устанавливая таймер, мы задаем интервал времени и указываем, что должно произойти по его истечении.

Как указать таймеру, что он должен делать? Во многих языках программирования задается имя функции, которую таймер должен периодически вызывать. Однако классы из стандартной библиотеки языка Java используют объектно-ориентированный подход. Программист должен передать таймеру объект некоторого класса. После этого таймер вызывает один из методов данного объекта. Передача объекта — более гибкий механизм, чем вызов функции, поскольку объект может нести дополнительную информацию.

Разумеется, таймер должен знать, какой метод он должен вызвать. Для этого таймеру нужно указать объект класса, реализующего интерфейс `ActionListener` из пакета `java.awt.event`. Этот интерфейс выглядит следующим образом:

```
public interface ActionListener
{
   void actionPerformed(ActionEvent event);
}
```

По истечении заданного интервала времени таймер вызывает метод `actionPerformed()`.



На заметку! Как было сказано в главе 5, в языке Java есть аналоги указателей на функции — объекты класса `Method`. Однако их трудно использовать, и они медленнее работают. Кроме того, на этапе компиляции не проверяются типы. Каждый раз, когда в C++ используется указатель на функцию, в Java применяется интерфейс.

Предположим, нам нужно каждые десять секунд выводить на экран сообщение о текущем времени, сопровождаемое звуковым сигналом. Для этого необходимо определить класс, реализующий интерфейс `ActionListener`, а затем поместить операторы, которые нужно выполнить, в тело метода `actionPerformed()`.

```
class TimePrinter implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      Date now = new Date();
      System.out.println("Звонок, время: " + now);
      Toolkit.getDefaultToolkit().beep();
   }
}
```

Обратите внимание на параметр `ActionEvent` метода `actionPerformed()`. Он содержит информацию о событии, например, об объекте, его породившем (более подробно этот вопрос рассматривается в главе 8). В данной программе детальная информация о событии не важна, поэтому можно просто проигнорировать этот параметр.

Затем следует создать объект данного класса и передать его конструктору класса `Timer`:

```
ActionListener listener = new TimePrinter();
Timer t = new Timer(10000, listener);
```

Первый параметр конструктора `Timer` представляет собой интервал времени между точками отсчета, измеренный в миллисекундах. Сообщение должно выдаваться на экран каждые десять секунд. Второй параметр является объектом класса `ActionListener`.

Итак, запустим таймер:

```
t.start();
```

Каждые десять секунд на экране будет появляться следующее сообщение, сопровождаемое звуковым сигналом:

```
Звонок, время: Fri Jan 18 13:29:08 PDT 2008
```

В листинге 6.3 приведен код программы, реализующей описанный алгоритм. После запуска таймера программа выводит на экран окно с сообщением и ждет, пока пользователь не щелкнет на кнопке OK, чтобы завершить работу; при этом на экран каждые десять секунд выводится текущее время.

Будьте терпеливы. После того как диалоговое окно с вопросом Завершить программу? отобразится на экране, первое сообщение таймера появится только через десять секунд.

Отметим, что в дополнение к пакетам `javax.swing.*` и `java.util.*` программа импортирует класс `javax.swing.Timer`. Как видите, при импортировании указано имя класса. Это позволяет избежать неоднозначности. Дело в том, что существует класс `java.util.Timer`, который к нашей задаче отношения не имеет.

Листинг 6.3. Содержимое файла `TimerTest.java`

```
1. /**
2.    @version 1.00 2000-04-13
3.    @author Cay Horstmann
4. */
5. import java.awt.*;
6. import java.awt.event.*;
7. import java.util.*;
8. import javax.swing.*;
9. import javax.swing.Timer;
10. // Таким образом мы разрешаем конфликт с классом java.util.Timer
11.
12. public class TimerTest
13. {
14.    public static void main(String[] args)
15.    {
16.       ActionListener listener = new TimePrinter();
17.
18.       // Создание таймера, вызывающего обработчик каждые 10 секунд.
19.       Timer t = new Timer(10000, listener);
20.       t.start();
```

```
21.
22.      JOptionPane.showMessageDialog(null, "Завершить программу?");
23.      System.exit(0);
24.   }
25. }
26.
27. class TimePrinter implements ActionListener
28. {
29.   public void actionPerformed(ActionEvent event)
30.   {
31.      Date now = new Date();
32.      System.out.println("Звонок, время: " + now);
33.      Toolkit.getDefaultToolkit().beep();
34.   }
35. }
```

API javax.swing.JOptionPane 1.2

- `static void showMessageDialog(Component parent, Object message)`

  Отображает на экране диалоговое окно со строкой сообщения и кнопкой OK. Диалоговое окно выравнивается по центру компонента `parent`. Если `parent` является `null`, окно выравнивается по центру экрана.

API javax.swing.Timer 1.2

- `Timer(int interval, ActionListener listener)`

  Создает таймер, передающий сообщение объекту `listener`. Сообщение передается по истечении интервала времени, продолжительность которого задается параметром `interval`.

- `void start()`

  Запускает таймер, который затем вызывает метод `actionPerformed()` его слушателей.

- `void stop()`

  Останавливает таймер. После остановки таймера метод `actionPerformed()` слушателей больше не вызывается.

API java.awt.Toolkit 1.0

- `static Toolkit getDefaultToolkit()`

  Возвращает инструментальный набор, используемый по умолчанию. Этот набор содержит информацию о среде пользовательского графического интерфейса.

- `void beep()`

  Воспроизводит звуковой сигнал.

Вложенные классы

Вложенным (inner) называется класс, определенный внутри другого класса. Зачем он нужен? Назовем три причины.

- Объект внутреннего класса имеет доступ к данным объекта, в котором он определен, включая приватные данные.
- Вложенный класс можно скрыть от других классов того же пакета.
- *Анонимный* вложенный класс удобен, если нужно определить обратный вызов в процессе выполнения программы; при этом не приходится создавать большой объем дополнительного кода.

В данном разделе мы рассмотрим вложенные классы в описанном ниже порядке.

- Вначале обсудим простой вложенный класс, способный обращаться к полям экземпляра внешнего класса.
- Далее рассмотрим специальные синтаксические правила, используемые при объявлении вложенных классов.
- Потом речь пойдет о преобразовании вложенных классов в обычные. Начинающие программисты могут пропустить этот материал.
- После этого рассматриваются локальные вложенные классы, способные обращаться к локальным переменным в области видимости включающего класса.
- Затем мы введем понятие *анонимного внутреннего класса* и покажем преимущества использования таких классов при организации обратных вызовов.
- Наконец, вы увидите, как можно использовать *статические вложенные классы* для формирования вложенных вспомогательных классов.



На заметку! В языке C++ есть *вложенные классы* (nested classes). Вложенный класс находится в области видимости включающего его класса. Вот типичный пример: внутри класса для связного списка определен класс, содержащий связи, и класс, в котором определяется позиция итератора.

```
class LinkedList
{
public:
    class Iterator // Вложенный класс.
    {
    public:
         void insert(int x);
         inr erase();
         ...
    };
    ...
private:
    class Link // Вложенный класс.
    {
    public:
        Link* next;
        int data;
    };
    ...
}
```

Вложение представляет собой отношение между *классами*, а не между *объектами*. Объект класса `LinkedList` не содержит подобъектов типа `Iterator` или `Link`.

У вложения классов есть два преимущества: возможность *управления именами* и *управления доступом*. Поскольку имя `Iterator` вложено внутрь класса `LinkedList`, оно известно

исключительно как `LinkedList::Iterator` и не может конфликтовать с другими классами, имеющими имя `Iterator`. В Java это преимущество не играет такой роли, поскольку в нем подобное управление именами осуществляют *пакеты*. Отметим, что класс `Link` находится в *приватном* разделе класса `LinkedList`. Он полностью скрыт от остальной части программы. По этой причине можно объявлять его поля общедоступными. Тогда методы класса `LinkedList` получат к ним доступ (который им совершенно необходим), причем для остальных методов эти поля останутся невидимыми. В языке Java такой вид управления доступом невозможен, если не применять вложенные классы.

Однако вложенные классы Java имеют еще одно преимущество, которое делает их более полезными, чем вложенные классы C++. Объект внутреннего класса содержит неявную ссылку на объект внешнего класса, который создал его. С помощью этой ссылки объект внутреннего класса получает доступ ко всему состоянию внешнего объекта. Этот механизм более подробно рассматривается далее в настоящей главе.

В Java дополнительную ссылку не имеют лишь статические вложенные классы. Именно они представляют собой полный аналог вложенных классов C++.

Использование вложенных классов для доступа к состоянию объекта

Синтаксис, используемый для работы с вложенными классами, довольно сложен. Поэтому, чтобы продемонстрировать применение вложенных классов, мы рассмотрим простой, но несколько надуманный пример. Мы преобразуем пример `TimerTest` и сформируем класс `TalkingClock`. Для организации работы "говорящих часов" применяются два параметра: интервал между последовательными сообщениями и флаг, позволяющий включать или отключать звуковой сигнал.

```
class TalkingClock
{
   public TalkingClock(int interval, boolean beep) { . . . }
   public void start() { . . . }

   private int interval;
   private boolean beep;

   private class TimePrinter implements ActionListener
      // Вложенный класс
   {
      . . .
   }
}
```

Обратите внимание, что класс `TimePrinter` теперь расположен в классе `TalkingClock`. Это не значит, что каждый экземпляр `TalkingClock` содержит поле типа `TimePrinter`. Как вы увидите, объекты `TimePrinter` создаются с методами класса `TalkingClock`.

Рассмотрим более подробно класс `TimePrinter`. Обратите внимание, что метод `actionPerformed()`, перед тем как воспроизвести звуковой сигнал, проверяет флаг `beep`.

```
private class TimePrinter implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      Date now = new Date();
```

```
        System.out.println("Звонок, время: " + now);
        if (beep) Toolkit.getDefaultToolkit().beep();
    }
}
```

Теперь начинается самое интересное. Нетрудно заметить, что в составе класса `TimePrinter` нет поля `beep`. Вместо этого метод `actionPerformed()` обращается к соответствующему полю объекта `TalkingClock`. С подобным явлением мы еще не встречались. Обычно метод обращается к полям объекта. Однако вложенный класс имеет доступ не только к своим полям, но и к полям создавшего его объекта, т.е. экземпляра внешнего класса.

Для того чтобы это стало возможным, вложенный класс должен содержать ссылку на объект внешнего класса. Такая ссылка условно показана на рис. 6.3.

В определении внутреннего класса эта ссылка явным образом не присутствует. Для того чтобы продемонстрировать, как она работает, включим в код ссылку `outer`. Тогда метод `actionPerformed()` будет выглядеть следующим образом:

```
public void actionPerformed(ActionEvent event)
{
   Date now = new Date();
   System.out.println("Звонок, время: " + now);
   if (outer.beep) Toolkit.getDefaultToolkit().beep();
}
```

Ссылка на объект внешнего класса задается в конструкторе. Компилятор модифицирует все конструкторы внутреннего класса, добавляя параметр-ссылку на внешний класс. Поскольку в `TalkingClock` конструкторы не определены, компилятор формирует конструктор, генерируя код, подобный следующему:

```
// Автоматически генерируемый код
public TimePrinter(TalkingClock clock)
{
  outer = clock;
}
```

Обратите внимание на то, что слово `outer` не является ключевым словом Java. Оно используется для иллюстрации механизма, задействованного при работе с вложенными классами.

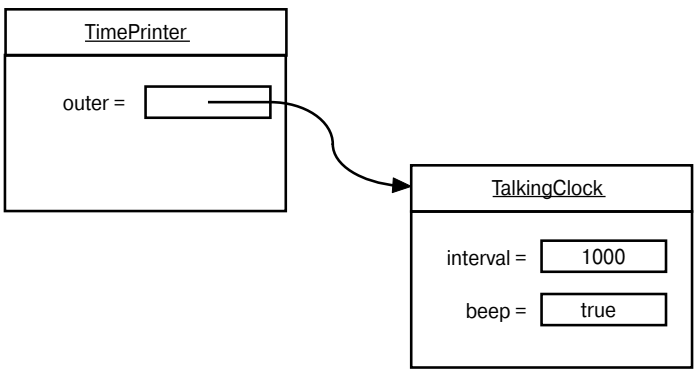


Рис. 6.3. Объект внутреннего класса содержит ссылку на объект внешнего класса

После того как метод `start()` создаст объект класса `TimePrinter`, компилятор передаст конструктору текущего объекта ссылку `this` на объект `TalkingClock`.

```
ActionListener listener = new TimePrinter(this); // Параметр добавляется
                                                 // автоматически
```

В листинге 6.4 показана законченная программа, проверяющая работу внутреннего класса. Если бы `TimePrinter` был обычным классом, он должен был бы получить доступ к флагу `beep` посредством общедоступного метода. Наличие внутреннего класса упрощает задачу. Теперь создавать специальный метод, возвращающий значение `beep`, не обязательно, так как он нужен только одному другому классу.



На заметку! Мы могли бы объявить класс `TimePrinter` как `private`. Тогда только методы `TalkingClock` смогли бы конструировать объекты `TimePrinter`. Только вложенные классы могут быть приватными. Обычные классы всегда имеют полную видимость или видимость в пределах пакета.

Листинг 6.4. Содержимое файла `InnerClassTest.java`

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.util.*;
4. import javax.swing.*;
5. import javax.swing.Timer;
6.
7. /**
8.    * Эта программа демонстрирует использование вложенных классов.
9.    * @version 1.10 2004-02-27
10.   * @author Cay Horstmann
11.   */
12.public class InnerClassTest
13.{
14.   public static void main(String[] args)
15.   {
16.      TalkingClock clock = new TalkingClock(1000, true);
17.      clock.start();
18.
19.      // Программа выполняется до тех пор, пока
20.      // пользователь не щелкнет на кнопке OK.
21.      JOptionPane.showMessageDialog(null, "Завершить программу?");
22.      System.exit(0);
23.   }
24.}
25.
26./**
27. * Часы, выводящие информацию о времени через
28. * заданный интервал.
29. */
30.class TalkingClock
31.{
32.   /**
33.    * Конструирует "говорящие часы"
34.    * @param interval Интервал в миллисекундах между сообщениями
35.    * @param beep Значение true включает звуковой сигнал
36.    */
```

```
37.   public TalkingClock(int interval, boolean beep)
38.   {
39.      this.interval = interval;
40.      this.beep = beep;
41.   }
42.
43.   /**
44.    * Запускает часы.
45.    */
46.   public void start()
47.   {
48.      ActionListener listener = new TimePrinter();
49.      Timer t = new Timer(interval, listener);
50.      t.start();
51.   }
52.
53.   private int interval;
54.   private boolean beep;
55.
56.   private class TimePrinter implements ActionListener
57.   {
58.      public void actionPerformed(ActionEvent event)
59.      {
60.         Date now = new Date();
61.         System.out.println("Звонок, время: " + now);
62.         if (beep) Toolkit.getDefaultToolkit().beep();
63.      }
64.   }
65. }
```

Специальные синтаксические правила для работы с вложенными классами

В предыдущем разделе мы называли ссылку на внешний класс именем `outer`, тем самым поясняя, что это ссылка вложенного класса на внешний класс. На самом деле синтаксис, связанный с использованием внешней ссылки, несколько сложнее. Выражение

Внешний_класс`.this`

означает ссылку на внешний класс. Например, во внутреннем классе `TimePrinter` можно создать метод `actionPerformed()`:

```
public void actionPerformed(ActionEvent event)
{
   . . .
   if (TalkingClock.this.beep) Toolkit.getDefaultToolkit().beep();
}
```

И, наоборот, можно более явно записать конструктор внутреннего класса, используя следующий синтаксис:

объект_Внешнего_Класса`.new` *Вложенный_Класс*`(`*параметры*`)`

Например:

```
ActionListener listener = this.new TimePrinter();
```

Теперь ссылке на внешний класс, содержащейся во вновь созданном объекте класса `TimePrinter`, присвоена ссылка `this` метода, создавшего объект внутреннего класса. Этот случай является наиболее распространенным. Как всегда, ключевое слово `this` излишне. Однако ссылке на объект внешнего класса можно присвоить другой объект, явно указав его. Например, если класс `TimePrinter` является общедоступным вложенным классом, можно создать его объекты для любых "говорящих часов".

```
TalkingClock jabberer = new TalkingClock(1000, true);
TalkingClock.TimePrinter listener = jabberer.new TimePrinter();
```

Отметим, что если приведенная ниже ссылка находится в области видимости внешнего класса, то она относится к внутреннему классу:

```
Внешний_Класс.Вложенный_Класс
```

Например, если класс `TimePrinter` является общедоступным, на него можно ссылаться в любом месте программы с помощью выражения `TalkingClock.TimePrinter`.

Чем полезны вложенные классы, нужны ли они и насколько они безопасны

Вложенные классы были впервые реализованы в Java 1.1. Многие программисты встретили появление вложенных классов настороженно, считая их отступлением от принципов, заложенных в основу языка Java. По их мнению, преимуществом языка Java перед C++ является его простота. Вложенные классы действительно сложны. (Такой вывод вы наверняка сделаете, ознакомившись с анонимными вложенными классами, которые будут рассматриваться далее в этой главе.) Взаимодействие вложенных классов с другими средствами языка не совсем очевидно, в особенности это касается управления доступом.

Зачем же разработчики языка Java пожертвовали преимуществами, которыми он выгодно отличался от других языков, в пользу элегантного и интересного механизма, выгода от которого настолько неочевидна?

У нас нет окончательного ответа на этот вопрос. Стоит уточнить, что обработку вложенных классов осуществляет *компилятор*, а не виртуальная машина. Для их обозначения используется символ `$`, разделяющий имена внешних и вложенных классов; таким образом, для виртуальной машины вложенные классы неотличимы от внешних.

Например, класс `TimePrinter` в составе `TalkingClock` преобразуется в класс `TalkingClock$TimePrinter`. Чтобы увидеть, как работает этот механизм, попробуйте провести следующий эксперимент: запустите программу `ReflectionTest` (см. главу 5) и выполните рефлексию класса `TalkingClock$TimePrinter`. В качестве альтернативы просто воспользуйтесь утилитой `javap`:

```
javap -private ИмяКласса
```



На заметку! Если вы используете UNIX, не забудьте защитить символ `$`, если указываете имя класса в командной строке. То есть запускайте `ReflectionTest` или `javap`, как показано ниже:

```
java ReflectionTest TalkingClock\$TimePrinter
```

или

```
javap -private TalkingClock\$TimePrinter
```

На экране появятся следующие строки:

```
public class TalkingClock$TimePrinter
{
  public TalkingClock$TimePrinter(TalkingClock);
  TalkingClock$TimePrinter(TalkingClock, TalkingClock$1);

  public void actionPerformed(java.awt.event.ActionEvent);

  final TalkingClock this$0;
}
```

Нетрудно заметить, что компилятор генерирует дополнительное поле `this$0` для ссылки на внешний класс. (Имя `this$0` синтезируется компилятором, поэтому сослаться на него невозможно.) Также вы можете видеть параметр `TalkingClock` у конструктора.

Если компилятор автоматически выполняет преобразования, нельзя ли реализовать подобный механизм вручную? Попробуем сделать это. Превратим `TimePrinter` в обычный класс, определяемый за пределами `TalkingClock`, а затем передадим ему ссылку `this` на создавший его объект.

```
class TalkingClock
{
  . . .
  public void start()
  {
    ActionListener listener = new TimePrinter(this);
    Timer t = new Timer(interval, listener);
    t.start();
  }
}

class TimePrinter implements ActionListener
{
  public TimePrinter(TalkingClock clock)
  {
    outer = clock;
  }
  . . .
  private TalkingClock outer;
 }
```

Рассмотрим теперь метод `actionPerformed()`. Ему нужен доступ к полю `outer.beep`.

```
if (outer.beep) . . . // Ошибка!
```

Здесь возникает проблема. Вложенный класс имеет доступ к приватным данным лишь внешнего класса, который его содержит, но теперь класс `TimePrinter` не является вложенным и потому доступа не имеет.

Следовательно, вложенные классы намного мощнее, чем обычные, поскольку обладают более высокими привилегиями доступа.

Может возникнуть вопрос, каким образом вложенные классы получают дополнительные привилегии доступа, ведь они преобразуются в обычные, а виртуальной машине вообще о них ничего не известно? Чтобы раскрыть эту тайну, снова воспользуемся программой `ReflectionTest`, отслеживающей поведение класса `TalkingClock`.

```
class TalkingClock
{
   public TalkingClock(int, boolean);

   static boolean access$100(TalkingClock);
   public void start();

   private int interval;
   private boolean beep;
}
```

Обратите внимание на статический метод `access$100`, добавленный компилятором во внешний класс. Он возвращает значение поля `beep`, содержащегося в объекте, который передан методу в качестве параметра.

Вложенный класс вызывает этот метод. В методе `actionPerformed()` класса `TimePrinter` есть следующий оператор:

```
if (beep)
```

Оно преобразуется в приведенный ниже вызов:

```
if (access$100(outer));
```

Не опасно ли это? В принципе, да! Посторонний может легко вызвать метод `access$100()` и прочесть данные приватного поля `beep`. Конечно, `access$100` не является допустимым именем для Java-метода. Однако хакеры, знакомые со структурой файлов, содержащих классы, легко могут создать свой аналогичный файл и вызвать данный метод с помощью соответствующих команд виртуальной машины. Разумеется, такой файл должен формироваться вручную (например, с помощью шестнадцатеричного редактора). Но поскольку область видимости секретных методов доступа ограничена пакетом, атакующий код должен размещаться в том же пакете, что и атакуемый класс.

Итак, если вложенный класс обладает доступом к приватным полям, можно создать другой класс, добавить его в тот же пакет и получить доступ к приватным данным. Правда, для этого нужен опыт и решительность. Программист не может получить такой доступ случайно, не создавая для этих целей специальный файл, содержащий модифицированные классы.



На заметку! Синтезированные методы и конструкторы могут быть довольно запутанными (пропустите эту врезку, если вы новичок). Предположим, что мы превратили `TimePrinter` в приватный вложенный класс. Но для виртуальной машины не существует вложенных классов, так что компилятор делает то, что может, а именно: порождает класс с видимостью уровня пакета, с приватным конструктором:

```
private TalkingClock$TimePrinter(TalkingClock);
```

Конечно, никто не сможет вызвать этот конструктор, так что нужен еще один конструктор с видимостью уровня пакета, который вызовет первый:

```
TalkingClock$TimePrinter(TalkingClock, TalkingClock$1);
```

Компилятор транслирует вызов конструктора в методе `start` класса `TaklingClock` следующим образом:

```
new TalkingClock$TimePrinter(this, null)
```

Локальные вложенные классы

Если внимательно изучить код класса TalkingClock, обнаружится, что имя класса TimePrinter используется лишь однажды — при создании объекта этого типа в методе start().

В таких ситуациях класс можно определить *локально в отдельном методе.*

```
public void start()
{
   class TimePrinter implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         Date now = new Date();
         System.out.println("Звонок, время: " + now);
         if (beep) Toolkit.getDefaultToolkit().beep();
      }
   }

   ActionListener listener = new TimePrinter();
   Timer t = new Timer(1000, listener);
   t.start();
}
```

Локальные классы никогда не объявляются с помощью модификаторов доступа (т.е. public и protected). Их область видимости всегда ограничивается блоком, в котором они объявлены.

Локальные классы имеют огромное преимущество — они полностью скрыты от внешнего мира и даже от остальной части класса TalkingClock. Ни один метод, за исключением start(), ничего не знает о классе TimePrinter.

Доступ к финальным переменным из внешних методов

Кроме того, локальные классы выгодно отличаются от обычных вложенных тем, что они имеют доступ не только к полям своего внешнего класса, но и к локальным переменным! Однако такие локальные переменные должны быть объявлены как final. Рассмотрим типичный пример. Переместим параметры interval и beep из конструктора TalkingClock в метод start().

```
public void start(int interval, final boolean beep)
{
   class TimePrinter implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         Date now = new Date();
         System.out.println("Звонок, время: " + now);
         if (beep) Toolkit.getDefaultToolkit().beep();
      }
   }

   ActionListener listener = new TimePrinter();
   Timer t = new Timer(1000, listener);
   t.start();
}
```

Обратите внимание, что классу TimePrinter больше не нужно хранить переменную экземпляра beep. Он просто ссылается на параметр метода, содержащего определение данного класса.

Возможно, это не так уж и неожиданно. В конце концов, приведенная ниже строка находится внутри метода start(), так почему бы ей не иметь доступ к переменной beep?

```
if (beep) . . .
```

Локальные вложенные классы обладают рядом особенностей. Чтобы понять их, рассмотрим поток управления.

1. Вызывается метод start().
2. С помощью вызова конструктора внутреннего класса TimePrinter инициализируется объектная переменная listener.
3. Ссылка listener передается конструктору класса Timer, таймер запускается, и метод start() прекращает свою работу. В этот момент параметр beep метода start() больше не существует.
4. Впоследствии метод actionPerformed() выполняет оператор if (beep) . . .

Чтобы метод actionPerformed() выполнялся успешно, класс TimePrinter должен создать копию поля beep до того, как оно перестанет существовать в качестве локальной переменной метода start(). Именно это и происходит.

В нашем примере компилятор синтезирует для локального внутреннего класса имя TalkingClock$1TimePrinter. Применив снова программу ReflectionTest для изучения класса TalkingClock$1TimePrinter, получим следующий результат:

```
class TalkingClock$1TimePrinter
{
  TalkingClock$1TimePrinter(TalkingClock, boolean);

  public void actionPerformed(java.awt.event.ActionEvent);

  final boolean val$beep;
  final TalkingClock this$0;
}
```

Обратите внимание на параметр конструктора, имеющий тип boolean, и переменную экземпляра val$beep. При создании объекта переменная beep передается конструктору и размещается в поле val$beep. Для того чтобы это стало возможным, разработчикам компилятора пришлось достаточно потрудиться. Компилятор должен обнаруживать доступ к локальным переменным, создавать для каждой из них соответствующие поля, а затем копировать локальные переменные в конструкторе так, чтобы поля данных инициализировались копиями локальных переменных.

С точки зрения разработчика прикладных программ, доступ к локальным переменным выглядит привлекательно. Благодаря такой возможности вложенные классы становятся проще, и уменьшается количество полей, которые должны программироваться явным образом.

Как уже упоминалось, методы локального класса могут ссылаться только на локальные переменные, объявленные с помощью ключевого слова final. По этой причине параметр beep в нашем примере был объявлен финальным (final). Финальная локальная переменная не может быть модифицирована. Этим гарантируется, что локальная переменная и ее копия, созданная внутри локального класса, всегда имеют одно и то же значение.



На заметку! Мы уже видели, что терминальные переменные можно использовать в качестве констант:

```
public static final double SPEED_LIMIT = 55;
```

Ключевое слово `final` можно применять и к локальным, статическим переменным, а также к переменным экземпляра. Во всех случаях это означает одно и то же: данной переменной значение присваивается только один раз, сразу после ее создания. Впоследствии это значение изменить будет невозможно.

Однако инициализировать терминальную переменную одновременно с ее определением не обязательно. Например, параметр `beep` инициализируется после создания при вызове метода `start`. (Если этот метод вызывается неоднократно, при каждом вызове будет создаваться новый параметр `beep`.) Значение переменной экземпляра `val$beep` в классе `TalkingClock$1TimePrinter` устанавливается лишь один раз, в конструкторе внутреннего класса. Финальную переменную, которая не была инициализирована при определении, часто называют *пустой финальной переменной*.

Ограничение `final` несколько противоречиво. Предположим, например, что нам нужно обновить счетчик во включающем контексте. Здесь мы хотим подсчитать, насколько часто во время сортировки вызывается метод `compareTo()`.

```
int counter = 0;
Date[] dates = new Date[100];
for (int i = 0; i < dates.length; i++)
       dates[i] = new Date()
    {
           public int compareTo(Date other)
           {
                  counter++; // Ошибка
                  return super.compareTo(other);
           }
       };
Arrays.sort(dates);
System.out.println(counter + " сравнений.");
```

Вы не можете объявить `counter`, как `final`, поскольку ясно, что ее нужно обновлять. Нельзя ее заменить на `Integer`, поскольку объекты `Integer` не изменяемы. Выход состоит в использовании массива длиной в 1 элемент.

```
final int[] counter = new int[1];
for (int i = 0; i < dates.length; i++)
    dates[i] = new Date()
    {
           public int compareTo(Date other)
           {
                  counter[0]++;
                  return super.compareTo(other);
           }
    };
```

(Переменная массива также объявлена как `final`, но это означает просто, что вы не можете заставить ее ссылаться на другой массив. Элементы массива можно изменять свободно.)

Впервые встречая вложенный класс, компилятор автоматически выполняет эту трансформацию для всех локальных переменных, модифицируемых во внутреннем классе. Однако некоторых программистов пугает способность компилятора за их спиной

создавать объекты в куче, и вместо этого принимается ограничение final. Возможно, в будущих версиях Java это решение будет пересмотрено.

Анонимные вложенные классы

Работая с локальными вложенными классами, можно использовать еще одну интересную возможность. Если нужно создать один-единственный объект данного класса, этому классу можно даже не присваивать имени. Такой класс называется *анонимным вложенным классом*.

```
public void start(int interval, final boolean beep)
{
   ActionListener listener = new ActionListener()
      {
         public void actionPerformed(ActionEvent event)
         {
            Date now = new Date();
            System.out.println("Звонок, время: " + now);
            if (beep) Toolkit.getDefaultToolkit().beep();
         }
      };
   Timer t = new Timer(1000, listener);
   t.start();
}
```

Надо признать, что синтаксис достаточно сложен. На самом деле приведенный выше фрагмент означает следующее: создается новый объект класса, реализующего интерфейс ActionListener, в котором в фигурных скобках {} определен требуемый метод actionPerformed().

Любые параметры, необходимые для создания объекта, задаются в круглых скобках, следующих за именем супертипа. В общем случае синтаксис выглядит следующим образом:

```
new СуперТип(параметры конструирования объекта)
{
  методы внутреннего класса и данные
}
```

В данном случае *СуперТип* — это интерфейс ActionListener; вложенный класс *реализует* данный интерфейс. *СуперТип* может быть и классом; тогда вложенный класс *расширяет* данный суперкласс.

Анонимный вложенный класс не может иметь конструкторов, поскольку имя конструктора должно совпадать с именем класса, а в данном случае класс не имеет имени. Вместо этого параметры, необходимые для создания объекта, передаются конструктору *суперкласса*. Казалось бы, если вложенный класс реализует какой-либо интерфейс, параметры не задаются, однако существует способ указать их.

```
new ТипИнтерфейса()
{
  методы и данные
}
```

Следует тщательно изучить различие между созданием нового объекта некоторого класса и созданием объекта анонимного внутреннего класса, расширяющего данный класс. Если вслед за скобками, содержащими список параметров, необходимых для соз-

дания объекта, указана открытая фигурная скобка, определяется анонимный вложенный класс.

```
Person queen = new Person("Mary");
   // Объект класса Person.
Person count = new Person("Dracula") {...};
   // Объект внутреннего класса, расширяющего класс Person.
```

Был ли ввод в язык анонимных вложенных классов удачной идеей, или это только великолепное средство создавать никому не понятные программы? Скорее всего, и то и другое одновременно. Если вложенный класс невелик и сводится всего к нескольким строчкам простого кода, можно сберечь время на набор текста программы, но сэкономленное время обернется трудностями при чтении исходного текста программы.

В листинге 6.5 приведена законченная программа, реализующая "говорящие часы", в которой используется анонимный вложенный класс. Сравнив эту программу с кодом, содержащимся в листинге 6.4, вы можете убедиться, что применение анонимного внутреннего класса сделало программу несколько короче, но не проще для понимания.

Листинг 6.5. Содержимое файла `AnonymousInnerClassTest.java`

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.util.*;
4. import javax.swing.*;
5. import javax.swing.Timer;
6.
7. /**
8.  * Эта программа демонстрирует использование вложенных классов.
9.  * @version 1.10 2004-02-27
10. * @author Cay Horstmann
11. */
12.public class AnonymousInnerClassTest
13.{
14.   public static void main(String[] args)
15.   {
16.      TalkingClock clock = new TalkingClock();
17.      clock.start(1000, true);
18.
19.      // Программа выполняется до тех пор, пока
20.      // пользователь не щелкнет на кнопке OK.
21.      JOptionPane.showMessageDialog(null, "Завершить программу?");
22.      System.exit(0);
23.   }
24.}
25.
26./**
27. * Часы, выводящие показания времени
28. * через определенный интервал.
29. */
30.class TalkingClock
31.{
32.  /**
33.   * Запуск часов.
34.   * @param interval Интервал времени в миллисекундах между сообщениями
35.   * @param beep Значение true включает звуковой сигнал
36.   */
```

```
37.   public void start(int interval, final boolean beep)
38.   {
39.      ActionListener listener = new
40.         ActionListener()
41.         {
42.            public void actionPerformed(ActionEvent event)
43.            {
44.               Date now = new Date();
45.               System.out.println("Звонок, время: " + now);
46.               if (beep) Toolkit.getDefaultToolkit().beep();
47.            }
48.         };
49.      Timer t = new Timer(interval, listener);
50.      t.start();
51.   }
52. }
```

Статические вложенные классы

Может случиться так, что вам нужно будет использовать вложенный класс просто для того, чтобы скрыть его внутри другого класса, а ссылка на объект внешнего класса окажется ненужной. Подавить генерацию этой ссылки можно, объявив вложенный класс статическим (`static`).

Предположим, нам нужно найти в массиве максимальное и минимальное число. Разумеется, для этого мы напишем два метода: один — для нахождения максимального, а другой — для минимального числа. При вызове обеих функций массив просматривается дважды. Было бы намного эффективнее просматривать массив только один раз, одновременно определяя и максимальное, и минимальное число.

```
double min = Double.MAX_VALUE;
double max = Double.MIN_VALUE;
for (double v : values)
{
   if (min > v) min = v;
   if (max < v) max = v;
}
```

Однако в таком случае функция должна возвращать два значения. Сделать это можно, определив класс `Pair`, содержащий два числа:

```
class Pair
{
   public Pair(double f, double s)
   {
      first = f;
      second = s;
   }
   public double getFirst() { return first; }
   public double getSecond() { return second; }

   private double first;
   private double second;
}
```

Тогда метод `minmax` сможет вернуть объект класса `Pair`:

```
class ArrayAlg
{
   public static Pair minmax(double[] values)
   {
      . . .
      return new Pair(min, max);
   }
}
```

Вызывающий метод использует для получения минимального и максимального значения методы `getFirst()` и `getSecond()`:

```
Pair p = ArrayAlg.minmax(d);
System.out.println("min = " + p.getFirst());
System.out.println("min = " + p.getSecond());
```

Разумеется, имя `Pair` слишком широко распространено, и при выполнении большого проекта у другого программиста может возникнуть такая же блестящая идея, вот только класс `Pair` у него будет содержать не числа, а строки. Эту потенциальную проблему можно решить, сделав класс `Pair` вложенным классом, определенным внутри `ArrayAlg`. Тогда настоящим именем класса будет не `Pair`, а `ArrayAlg.Pair`:

```
ArrayAlg.Pair p = ArrayAlg.minmax(d);
```

Однако в отличие от вложенных классов, использованных в предыдущих примерах, ссылка на другой объект внутри класса `Pair` не нужна. Ее можно подавить, объявив вложенный класс статическим:

```
class ArrayAlg
{
   public static class Pair
   {
      ...
   }
      ...
}
```

Разумеется, только вложенные классы можно объявлять статическими. Статический вложенный класс ничем не отличается от любого другого внутреннего класса, за исключением того, что его объект не содержит ссылку на создавший его объект внешнего класса. В нашем примере следует применять статический вложенный класс, поскольку объект внутреннего класса создается внутри статического метода.

```
public static Pair minmax(double[] d)
{
   ...
   return new Pair(min, max);
}
```

Если бы класс `Pair` не был объявлен статическим, компилятор сообщил бы о том, что при инициализации объекта внутреннего класса объект типа `ArrayAlg` не доступен.



На заметку! Статический вложенный класс используется тогда, когда доступ к объекту внешнего класса не нужен. Некоторые программисты для обозначения статических вложенных классов используют термин *вложенные классы* (nested classes).



На заметку! Вложенные классы, определенные в интерфейсах автоматически, считаются статическими и общедоступными.

В листинге 6.6 приведен полный код класса `ArrayAlg` и вложенного класса `Pair`.

Листинг 6.6. Содержимое файла `StaticInnerClassTest.java`

```
1. /**
2.    * Эта программа демонстрирует использование
3.    * статических вложенных классов.
4.    * @version 1.01 2004-02-27
5.    * @author Cay Horstmann
6.    */
7. public class StaticInnerClassTest
8. {
9.    public static void main(String[] args)
10.   {
11.      double[] d = new double[20];
12.      for (int i = 0; i < d.length; i++)
13.         d[i] = 100 * Math.random();
14.      ArrayAlg.Pair p = ArrayAlg.minmax(d);
15.      System.out.println("min = " + p.getFirst());
16.      System.out.println("max = " + p.getSecond());
17.   }
18.}
19.
20.class ArrayAlg
21.{
22.   /**
23.    * Два числа с плавающей точкой
24.    */
25.   public static class Pair
26.   {
27.      /**
28.       * Объединение двух чисел с плавающей точкой.
29.       * @param f Первое число
30.       * @param s Второе число
31.       */
32.      public Pair(double f, double s)
33.      {
34.         first = f;
35.         second = s;
36.      }
37.
38.      /**
39.       * Возвращает первое число из пары.
40.       * @return Первое число
41.       */
42.      public double getFirst()
43.      {
44.         return first;
45.      }
46.
```

```
47.      /**
48.       * Возвращает второе число из пары.
49.       * @return Второе число
50.       */
51.      public double getSecond()
52.      {
53.         return second;
54.      }
55.
56.      private double first;
57.      private double second;
58.   }
59.
60.   /**
61.    * Определяет минимальное и максимальное число в массиве.
62.    * @param d Массив чисел с плавающей точкой
63.    * @return Пара, в которой первый элемент
64.    *   является минимальным числом массива,
65.    *   а второй элемент — максимальным числом.
66.    */
67.   public static Pair minmax(double[] values)
68.   {
69.      double min = Double.MAX_VALUE;
70.      double max = Double.MIN_VALUE;
71.      for (double v : values)
72.      {
73.         if (min > v) min = v;
74.         if (max < v) max = v;
75.      }
76.      return new Pair(min, max);
77.   }
78. }
```

Прокси-классы

В последнем разделе этой главы мы обсудим понятие прокси-классов, которые также часто называют классами-посредниками. Этот новый механизм стал доступным в Java SE 1.3. Он используется для того, чтобы во время выполнения программы создавать новые классы, реализующие заданные интерфейсы. Прокси-классы необходимы, если на этапе компиляции программист еще не знает, какие интерфейсы ему следует реализовать. В прикладном программировании такая ситуация возникает крайне редко. Однако в некоторых приложениях системного программирования гибкость, обеспечиваемая прокси-классами, может оказаться очень важным обстоятельством.

Предположим, что необходимо сконструировать объект класса, реализующего один или более интерфейсов, точная природа которых во время компиляции не известна. Теперь представим, что нужно создать объект класса, реализующего эти интерфейсы. Это трудная проблема. Если объект класса `Class` представляет настоящий класс, мы можем применить метод `newInstance()` или задействовать механизм рефлексии, чтобы найти конструктор этого класса. Однако создать объект интерфейса невозможно. Следовательно, определить новый класс во время выполнения мы не можем.

Чтобы разрешить эту проблему, некоторые программы генерируют код, помещают его в файл, вызывают компилятор, а затем полученный файл класса. Естественно, это

очень медленный процесс, кроме того, при выполнении такой программы постоянно нужен компилятор. Механизм прокси-классов предлагает более удачное решение. Прокси-класс может создавать во время выполнения совершенно новые классы и реализует интерфейсы, которые указывает программист. В частности, в прокси-классе содержатся следующие методы:

- все методы, которые требуют указанные интерфейсы;
- все методы, определенные в классе `Object` (`toString()`, `equals()` и т.д.).

Однако определить новый код для этих методов в ходе выполнения программы нельзя. Вместо этого программист должен предоставить *обработчик вызовов*, представляющий собой объект, реализующий интерфейс `InvocationHandler`. В этом интерфейсе объявлен единственный метод:

```
Object invoke(Object proxy, Method method, Object[] args)
```

При вызове какого-либо метода из прокси-объекта автоматически вызывается метод `invoke()` обработчика вызовов, получающий на вход объект класса `Method` и параметры исходного вызова. Обработчик вызовов должен уметь обработать вызов.

Чтобы создать прокси-объект, используется метод `newProxyInstance()` класса `Proxy`. Этот метод получает три параметра.

- *Загрузчик класса.* Модель безопасности языка Java позволяет использовать разные загрузчики классов: для системных классов, для классов, полученных из Internet, и т.д. Загрузчики классов обсуждаются во втором томе. Пока мы будем указывать ссылку `null`, чтобы использовать загрузчик классов, предусмотренный по умолчанию.
- Массив объектов класса `Class` — по одному для каждого интерфейса, подлежащего реализации.
- Обработчик вызовов.

Остаются два вопроса. Как определить обработчик? Что можно сделать с полученным в результате прокси-объектом? Разумеется, ответы на эти вопросы зависят от задачи, которую программист хочет решить с помощью механизма прокси-объектов. Их можно использовать для достижения описанных ниже целей.

- Переадресовывать вызовы методов на удаленный сервер.
- Устанавливать связь между событиями пользовательского интерфейса и определенными действиями, выполняемыми при работе программы.
- Отслеживать вызовы методов при отладке.

В примере, который будет рассмотрен ниже, мы применим прокси-объекты и обработчики вызовов для отслеживания обращений к методам. Мы определим класс оболочки `TraceHandler`, в котором будет храниться некоторый объект-оболочка. Его метод `invoke()` лишь выводит на экран имя и параметры метода, к которому произошло обращение, а затем вызывает сам метод, задавая в качестве неявного параметра объект, хранящийся в классе `TraceHandler`.

```
class TraceHandler impements InvocationHandler
{
  public TraceHandler(Object t)
  {
    target = t;
  }
```

```
   public Object invoke(Object proxy, Method m, Object[] args) throws Throwable
   {
      // Вывод имени метода и его параметров.
      ...
      // Вызов метода.
      return m.invoke(target, args);
   }
   private Object target;
}
```

Ниже показано, как создать прокси-объект, позволяющий трассировать вызов одного из его методов.

```
Object value = ...;
// Создание объекта-оболочки.
InvocationHandler handler = new TraceHandler(value);
// Создание прокси-объекта для всех интерфейсов.
Class[] interfaces = new Class[] { Comparable.class };
Object proxy = Proxy.newProxyInstance(null, interfaces, handler);
```

Теперь каждый раз, когда метод вызывается из объекта `proxy`, его имя и параметры выводятся, а затем происходит обращение к соответствующему методу объекта `value`.

В программе, приведенной в листинге 6.7, мы применили прокси-объекты для трассировки бинарного поиска. Сначала заполняется массив, состоящий из прокси-объектов для целых чисел от 1 до 1000. Затем для поиска случайного целого числа в массиве вызывается метод `binarySearch` класса `Arrays`. В заключение на экран выводится соответствующий элемент.

```
Object[] elements = new Object[1000];
// Массив заполняется прокси-объектами
// для целых чисел от 1 до 1000.
for (int i = 0; i < elements.length; i++)
{
   Integer value = i + 1;
   elements[i] = Proxy.newInstance(. . .); // прокси-объект для значения
}

// Генерация случайного числа.
Integer key = new Random().nextInt(elements.length) + 1;

// Поиск key.
int result = Arrays.binarySearch(elements, key);

// Вывод на экран найденного числа
if (result >= 0) System.out.println(elements[result]);
```

Класс `Integer` реализует интерфейс `Comparable`. Прокси-объекты принадлежат классу, определяемому во время выполнения. (Его имя выглядит как `$Proxy0`.) Этот класс также реализует интерфейс `Comparable`. Однако его метод `compareTo()` вызывает метод `invoke` из обработчика вызовов в прокси-объекте.



На заметку! Как было сказано в начале данной главы, начиная с Java SE 5.0, в классе `Integer` был фактически реализован интерфейс `Comparable<Integer>`. Однако во время выполнения информация о типах удаляется и при создании прокси-объекта используется обычный объект `Comparable`.

Метод binarySearch() осуществляет вызов наподобие следующего:

```
if (elements[i].compareTo(key) < 0) ...
```

Поскольку массив заполнен прокси-объектами, метод compareTo() вызывает метод invoke() класса TraceHandler. Этот метод выводит имя вызванного метода и его параметры, а затем вызывает метод compareTo() объекта Integer.

В конце программы выводятся результаты ее работы:

```
System.out.println(elements[result]);
```

Метод println() вызывает метод toString() прокси-объекта, а затем это обращение также перенаправляется обработчику вызовов.

Ниже показаны результаты полной трассировки программы.

```
500.compareTo(288)
250.compareTo(288)
375.compareTo(288)
312.compareTo(288)
281.compareTo(288)
296.compareTo(288)
288.compareTo(288)
288.toString()
```

Как видите, при бинарном поиске на каждом шаге интервал уменьшается вдвое. Обратите внимание, что метод toString() также имеет прокси, даже несмотря на то, что он не относится к интерфейсу Comparable; как вы увидите в следующем разделе, определенные методы Object всегда имеют прокси.

Листинг 6.7. Содержимое файла ProxyTest.java

```
1. import java.lang.reflect.*;
2. import java.util.*;
3.
4.
5. /**
6.  * Эта программа демонстрирует использование прокси-объектов.
7.  * @version 1.00 2000-04-13
8.  * @author Cay Horstmann
9.  */
10. public class ProxyTest
11. {
12.    public static void main(String[] args)
13.    {
14.       Object[] elements = new Object[1000];
15.
16.       // заполнение прокси-объектами для целых чисел от 1 до 1000.
17.       for (int i = 0; i < elements.length; i++)
18.       {
19.          Integer value = i + 1;
20.          InvocationHandler handler = new TraceHandler(value);
21.          Object proxy = Proxy.newProxyInstance(null, new Class[]
22.                { Comparable.class } , handler);
23.          elements[i] = proxy;
24.       }
25.
26.       // Генерация случайного числа.
27.       Integer key = new Random().nextInt(elements.length) + 1;
28.
```

```
29.      // Поиск key.
30.      int result = Arrays.binarySearch(elements, key);
31.
32.      // Вывод на экран найденного числа.
33.      if (result >= 0) System.out.println(elements[result]);
34.   }
35. }
36.
37. /**
38.  * Обработчик вызова выводит имя метода и параметры,
39.  * а затем вызывает требуемый метод.
40.  */
41. class TraceHandler implements InvocationHandler
42. {
43.   /**
44.    * Конструирование TraceHandler
45.    * @param t Неявный параметр вызова метода
46.    */
47.   public TraceHandler(Object t)
48.   {
49.      target = t;
50.   }
51.
52.   public Object invoke(Object proxy, Method m, Object[] args) throws Throwable
53.   {
54.      // Вывод неявного параметра.
55.      System.out.print(target);
56.      // Вывод имени метода.
57.      System.out.print("." + m.getName() + "(");
58.      // Вывод явных параметров.
59.      if (args != null)
60.      {
61.         for (int i = 0; i < args.length; i++)
62.         {
63.            System.out.print(args[i]);
64.            if (i < args.length - 1) System.out.print(", ");
65.         }
66.      }
67.      System.out.println(")");
68.
69.      // Вызов фактического метода.
70.      return m.invoke(target, args);
71.   }
72.
73.   private Object target;
74. }
```

Свойства прокси-классов

Теперь, увидев прокси-классы в действии, вернемся к анализу некоторых их свойств. Напомним, что прокси-классы создаются во время выполнения. Однако затем они становятся обычными классами, как и все остальные классы, обрабатываемые виртуальной машиной.

Все прокси-классы расширяют класс `Proxy`. Такой класс содержит только одну переменную экземпляра, которая ссылается на обработчик вызовов, определенный в

суперклассе Proxy. Любые дополнительные данные, необходимые для выполнения задач, поставленных перед прокси-объектами, должны храниться в обработчике вызовов. Например, в листинге 6.7 при создании прокси-объектов, реализующих интерфейс Comparable, обработчик TraceHandler является классом оболочки для фактических объектов.

Во всех прокси-классах переопределяются методы toString(), equals() и hashCode() класса Object. Эти методы лишь вызывают метод invoke(), принадлежащий обработчику событий. Другие методы класса Object (например, clone() и getClass()) не переопределяются.

Имена прокси-классов не определены. В виртуальной машине, разработанной Sun, генерируются имена классов, начинающиеся строкой $Proxy.

Для конкретного загрузчика классов и заданного набора интерфейсов может существовать только один прокси-класс. Это значит, что, вызывая метод newProxyInstance() дважды для одного и того же загрузчика классов и массива интерфейсов, программист получает два объекта одного и того же класса. Имя этого класса можно определить с помощью метода getProxyClass():

```
Class proxyClass = Proxy.getProxyClass(null, interfaces);
```

Прокси-класс всегда является общедоступным и финальным. Если все интерфейсы, которые реализуются прокси-классом, объявлены как public, этот класс не принадлежит ни одному конкретному пакету. В противном случае все интерфейсы, в объявлении которых не указан модификатор public, а, следовательно, и сам прокси-класс, должны принадлежать одному пакету. Вызвав метод isProxyClass() класса Proxy, можно проверить, представляет ли объект Class определенный прокси-класс.

API java.lang.reflect.InvocationHandler 1.3

- `Object invoke(Object proxy, Method method, Object[] args)`

  Этот метод определяется для того, чтобы задать действие, которое должно быть выполнено при вызове какого-либо метода прокси-объекта.

API java.lang.reflect.Proxy 1.3

- `static Class getProxy(ClassLoader loader, Class[] interfaces)`

  Возвращает прокси-класс, реализующий заданные интерфейсы.

- `static Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler handler)`

  Создает новый экземпляр прокси-класса, реализующего заданные интерфейсы. Все методы вызывают метод invoke() обработчика вызовов.

- `static boolean isProxyClass(Class c)`

  Возвращает значение true, если c является прокси-классом.

Эта глава завершает описание основ языка Java. Понятия интерфейсов и вложенных классов — это концепции, с которыми вам придется часто встречаться. В отличие от них, прокси-классы в основном представляют интерес для разработчиков инструментальных средств, а не прикладных программистов. Теперь можно переходить к изучению графических средств и пользовательских интерфейсов, описание которых начинается в главе 7.

Глава 7

ПРОГРАММИРОВАНИЕ ГРАФИКИ

В ЭТОЙ ГЛАВЕ...

- Общие сведения о пакете Swing
- Создание фрейма
- Позиционирование фрейма
- Отображение информации в компоненте
- Двумерные фигуры
- Работа с цветом
- Использование шрифтов при выводе текста
- Вывод графических изображений

До сих пор мы учились писать программы, входные данные для которых вводились с клавиатуры. Затем программа обрабатывала их и выводила результаты на экран. Сейчас такие программы уже не соответствуют запросам большинства пользователей. Современные программы не работают подобным образом, тем более — Web-страницы. В этой главе рассматриваются вопросы создания графического пользовательского интерфейса (Graphics User Interface — GUI). В частности, вы узнаете, каким образом можно изменить размер и положение окон на экране, отобразить в окне текст, набранный разными шрифтами, вывести на экран рисунок и т.д. Эти навыки окажутся весьма полезными при создании интересных программ, описанных в следующих главах.

Следующие две главы будут посвящены обработке событий, таких как нажатие клавиши и щелчок кнопкой мыши, а также средствам создания элементов пользовательского интерфейса, например, меню и кнопок. Прочитав эту главу и две следующих, вы сможете создавать графические приложения. Более сложные вопросы, связанные с программированием графики, рассматриваются во втором томе.

Общие сведения о пакете Swing

Первая версия языка Java содержала библиотеку классов Abstract Window Toolkit (AWT), предоставляющую основные средства программирования графического пользовательского интерфейса. Создание элементов пользовательского интерфейса на конкретной платформе (Windows, Solaris, Macintosh и т.д.) библиотека AWT поручала встроенным инструментальным средствам. Например, если с помощью библиотеки AWT на экран нужно было вывести окно с текстом, то фактически оно отображалось с помощью платформенно-ориентированных средств. Теоретически созданные таким образом программы должны работать на любых платформах, при этом имея внешний вид, характерный для целевой платформы — в соответствии с рыночным слоганом Sun: "Написано однажды, работает везде".

Подход, основанный на использовании платформенно-ориентированных компонентов, очень хорош для простых приложений. Однако вскоре стало ясно, что с его помощью крайне трудно создавать высококачественные переносимые графические библиотеки, зависящие от "родных" интерфейсных элементов платформы. Элементы пользовательского интерфейса, например, меню, панели прокрутки и текстовые поля, на разных платформах могут вести себя по-разному. Следовательно, на основе этого подхода трудно создавать согласованные программы с предсказуемым поведением. Кроме того, некоторые графические среды программирования (например, X11/Motif) не имеют такого богатого набора компонентов пользовательского интерфейса, как операционные системы Windows или Macintosh. Это, в свою очередь, делало библиотеки еще более зависимыми. В результате графические приложения, созданные с помощью библиотеки AWT, выглядели по сравнению с программами для Windows или Macintosh не так привлекательно и не имели таких функциональных возможностей. Что еще хуже, в библиотеке AWT на разных платформах обнаруживались *разные* ошибки. Программистам приходилось тестировать каждое приложение на каждой платформе, что на практике означало: "Написано однажды, отлаживается везде".

В 1996 г. компания Netscape создала библиотеку программ для создания графического пользовательского интерфейса и назвала ее IFC (Internet Foundation Classes). Эта библиотека была основана на совершенно других принципах. Элементы пользовательского интерфейса, такие как меню, кнопки и т.п., *рисовались* в пустом окне. Единственная функциональность, которая требовалась от оконной системы платфор-

мы, заключалась в отображении окна и рисовании в нем. Таким образом, элементы графического пользовательского интерфейса, созданные с помощью библиотеки IFC, выглядели и вели одинаково, но не зависели от платформы, на которой запускалась программа. Компании Sun и Netscape объединили свои усилия и усовершенствовали этот подход, создав библиотеку под кодовым названием Swing. С тех пор слово Swing стало официальным названием набора инструментальных средств для создания машинонезависимого пользовательского интерфейса. Средства Swing доступны как расширение Java 1.1 и являются частью Java SE 1.2.

С тех пор, как Дюк Эллингтон сказал: "It Don't Mean a Thing If It Ain't Got That Swing"[1], Swing стало официальным именем для неоднорангового набора инструментов графического пользовательского интерфейса. Swing — часть Java Foundation Classes (JFC). Полный комплект JFC огромен и содержит много больше, чем набор инструментов GUI Swing. В средства JFC не только включены в компоненты Swing, но также API доступности (accessibility), API 2D и API перетаскивания.



На заметку! Библиотека Swing — не полная замена AWT. Она построена на основе архитектуры AWT. Swing просто дает больше возможностей по созданию пользовательского интерфейса. Основная архитектура библиотеки, в частности обработка событий, остается той же, используете ли вы AWT или Swing. В дальнейшем под словом Swing мы имеем в виду машинонезависимые классы для создания "рисованного" пользовательского интерфейса, а под аббревиатурой AWT — платформенно-ориентированные средства для работы с окнами, такие как обработка событий.

Разумеется, элементы пользовательского интерфейса библиотеки Swing появляются на экране немного медленнее, чем компоненты, предусмотренные в библиотеке AWT. Однако, по нашему опыту, на современных машинах это практически незаметно. С другой стороны, у библиотеки Swing есть несколько весьма существенных преимуществ.

- Она содержит более богатый и удобный набор элементов пользовательского интерфейса.
- Эта библиотека намного меньше зависит от платформы, на которой будет выполняться программа. Следовательно, она содержит меньше ошибок, характерных для конкретной платформы.
- Библиотека Swing обеспечивает пользователям однотипные средства для работы на разных платформах.

Третье достоинство библиотеки Swing может обернуться недостатком: если элементы пользовательского интерфейса на разных платформах выглядят одинаково, то (по крайней мере, на некоторых платформах) они обязательно будут *отличаться* от компонентов, реализуемых в данной системе, т.е. пользователи будут в них хуже разбираться.

В библиотеке Swing эта проблема решена очень элегантно. Программисты, использующие Swing, могут придать своей программе внешний вид в заданном стиле. Например, на рис. 7.1 и 7.2 показано, как выглядит одна и та же программа в средах Windows и GTK.

---

[1] "Ах, истина проста: без свинга жизнь пуста" — джазовый стандарт корифея черного "биг-бэнда" Эдварда Кеннеди (Дюка) Эллингтона (1899–1974). Свинг как джазовое направление основан на неизменном, хотя и обильно синкопируемом ритме. В первом значении "swing" означает "ритм, раскачка", а производное от него "swinger" переводится как "жизнелюб" — *Примеч. ред.*

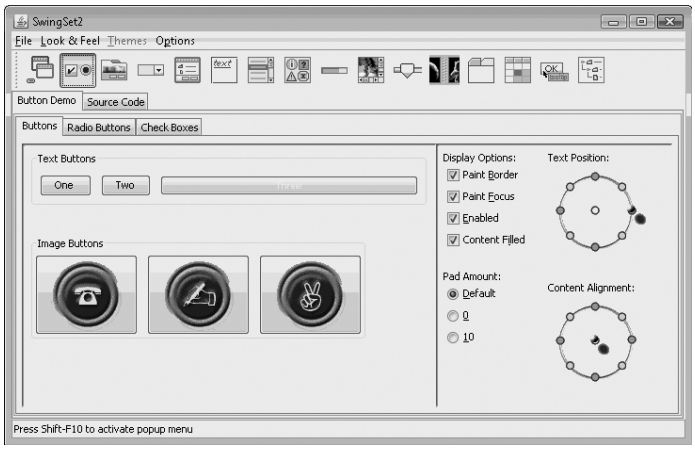


Рис. 7.1. Внешний вид окна программы в среде Windows

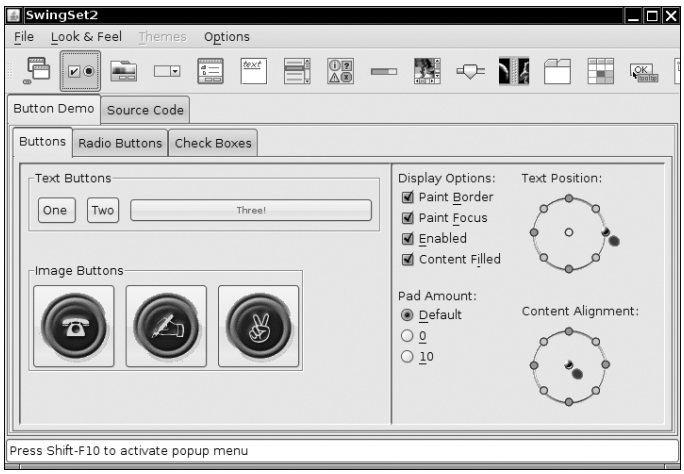


Рис. 7.2. Внешний вид окна программы в стиле GTK

Более того, Sun разработала независимый от платформы стиль, названный Metal, который сообщество Java прозвало “стилем Java”. Однако большинство программистов продолжают использовать термин Metal. Так же поступим и мы в настоящей книге.

Некоторые специалисты считали стиль Metal слишком “тяжелым”. Возможно, поэтому в JDK 5.0 он был несколько изменен (рис. 7.3). В настоящее время поддерживается несколько разновидностей Metal, слегка различающиеся цветами и шрифтами. По умолчанию используется вариант под названием Ocean. В данной книге, создавая компоненты Swing, мы будем использовать вариант Ocean стиля Metal.

В Java SE 6 Sun усовершенствовала “родной” стиль Windows и GTK. Теперь приложения Swing допускают настройку цветовой схемы и честно отображает пульсирующие кнопки и линейки прокрутки, которые стали выглядеть более современно.

Некоторые пользователи предпочитают, чтобы их приложения Java выглядели и вели себя так, как принято на данной платформе, другие предпочитают Metal или стили от независимых разработчиков. Как вы убедитесь в главе 8, очень легко обеспечить пользователя возможностью выбора предпочтительного внешнего стиля приложений.

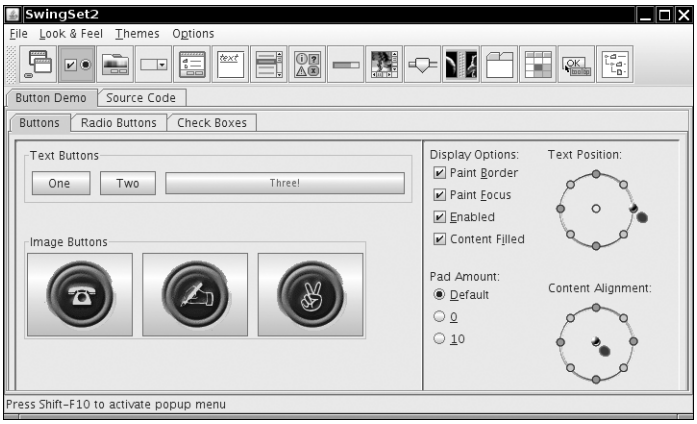


Рис. 7.3. Внешний вид программы в варианте Ocean стиля Metal



На заметку! Объем данной книги не позволяет подробно описать, как расширять средства отображения пользовательских интерфейсов. Отметим лишь, что программисты сами могут изменять стили интерфейса своих приложений и даже разрабатывать свои, совершенно новые. Это длительный процесс, при котором нужно указать, как каждый компонент библиотеки Swing должен быть отображен на экране. Некоторые разработчики уже сделали это при переносе языка Java на такие нетрадиционные платформы, как портативные терминалы или наладонные (handheld) устройства. Набор интересных реализаций средств отображения Swing можно найти по адресу `http://www.javootoo.com`.

В Java SE 5.0 были реализованы новый стиль отображения, который получил название Synth, который упрощает настройку компонентов для различных платформ. Работая с Synth, вы можете определить новый стиль, задавая лишь файлы изображений и XML-описания. Программировать при этом не нужно.



На заметку! В настоящее время большинство Java-приложений снабжаются интерфейсами, созданными на базе Swing. Исключением являются лишь программы, при разработке которых применялась интегрированная среда разработки Eclipse. В ней вместо Swing используется набор графических компонентов SWT, в котором, подобно AWT, компоненты отображаются платформено-ориентированными средствами. Информацию о SWT можно найти по адресу `http://www.eclipse.org/articles/`.

Если вы уже создавали графические пользовательские интерфейсы для Microsoft Windows, используя для этого Visual Basic или C#, то знаете, насколько легко применять готовые компоненты, предлагаемые ими. Эти средства позволяют визуально программировать интерфейс, и сами генерируют большую часть (а то и весь) кода. Хотя некоторые интегрированные среды для создания интерфейса на языке Java уже существуют, они еще не так развиты, как соответствующие средства визуального программирования в среде Windows. В любом случае, чтобы полностью понимать, как среда визуального программирования генерирует код для графического пользовательского интерфейса, и эффективно использовать нужные для этого средства, необходимо знать, как такой интерфейс создается вручную. Остальная часть этой главы посвящена основам отображения окон и рисования их компонентов.

Создание фрейма

Окно верхнего уровня (т.е. окно, не содержащееся внутри другого окна) в языке Java называется *фреймом* (frame). В библиотеке AWT для такого окна предусмотрен класс `Frame`. В библиотеке `Swing` аналогом этого класса является `JFrame`. Класс `JFrame` расширяет класс `Frame` и представляет собой один из немногих компонентов в библиотеке `Swing`, которые не отображаются на холсте (canvas). Кнопки, строка заголовков, пиктограммы и другие компоненты реализуются с помощью пользовательской оконной системы, а не библиотеки Swing.



Внимание! Большинство имен компонентов библиотеки Swing начинаются с буквы `J`. В качестве примеров можно привести классы `JButton` и `JFrame`. Они являются аналогами соответствующих компонентов AWT (например, `Button` и `Frame`). Если вы пропустите букву `J`, программа скомпилируется и будет работать, однако сочетание компонентов Swing и AWT в одном окне приведет к несогласованности внешнего вида и поведения интерфейсных элементов.

В данном разделе мы рассмотрим самые распространенные способы работы с классом `JFrame`. В листинге 7.1 приведена простая программа, отображающая на экране пустой фрейм, показанный на рис. 7.4.



Рис. 7.4. Простейший фрейм

Листинг 7.1. Содержимое файла `SimpleFrameTest.java`

```
1. import javax.swing.*;
2.
3. /**
4.    * @version 1.32 2007-06-12
5.    * @author Cay Horstmann
6.    */
7. public class SimpleFrameTest
8. {
9.    public static void main(String[] args)
10.   {
11.      SimpleFrame frame = new SimpleFrame();
12.      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13.      frame.setVisible(true);
14.   }
15.}
16.
17.class SimpleFrame extends JFrame
18.{
19.   public SimpleFrame()
20.   {
```

```
21.      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
22.   }
23.
24.   public static final int DEFAULT_WIDTH = 300;
25.   public static final int DEFAULT_HEIGHT = 200;
26. }
```

Проанализируем эту программу построчно.

Классы библиотеки Swing находятся в пакете `javax.swing`. Имя пакета `javax` означает, что данный пакет является расширением языка Java и не входит в число основных пакетов. По историческим причинам классы Swing относятся к расширению Java 1.1. Однако они присутствуют в каждой реализации Java SE, начиная с версии 1.2.

По умолчанию фрейм имеет совершенно бесполезный размер — 0×0 пикселей. В данном примере мы определяем подкласс `SimpleFrame`, конструктор которого устанавливает размер 300×200 пикселей. Это единственное отличие `SimpleFrame` от `JFrame`.

В методе `main()` класса `SimpleFrameTest` мы создаем объект `SimpleFrame` и делаем его видимым.

Есть две технические сложности, с которыми нужно справляться каждой программе Swing.

Во-первых, компоненты Swing должны быть конфигурированы в потоке-диспетчере событий — том потоке, который передает компонентам пользовательского интерфейса такие события, как щелчки кнопками мыши и нажатия клавиш. Следующий фрагмент кода используется для выполнения операторов в потоке диспетчера событий:

```
EventQueue.invokeLater(new Runnable()
    {
           public void run()
           {
                 операторы
           }
    });
```

Подробности мы обсудим в главе 14. А пока вы должны принять это просто как "волшебное заклинание", используемой для запуска программы Swing.



На заметку! Вы встретите немало программ Swing, которые не инициализируют пользовательский интерфейс в потоке диспетчера событий. Вполне допустимо инициализировать выполнять инициализацию в главном потоке. К сожалению, в связи с усложнением компонентов Swing программисты Sun не могут более гарантировать безопасность такого подхода. Вероятность ошибки чрезвычайно низка, но вы вряд ли захотите стать одним из нескольких "счастливчиков", которые столкнутся с проблемой. Лучше сделать все правильно, даже если код покажется несколько загадочным.

Далее мы определяем, что должно произойти, если пользователь закроет фрейм. В данном случае программа должна завершить свою работу. Для этого используется следующая строка кода:

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Если бы в программе использовалось несколько фреймов, завершать работу только потому, что пользователь закрыл один из них, было бы совершенно не обязательно. По умолчанию закрытый фрейм исчезает с экрана, а программа продолжает работу.

Простое создание фрейма не приводит к его автоматическому появлению на экране. В начале своего существования все фреймы невидимы. Это дает программисту возможность добавлять к фрейму компоненты еще до того, как он впервые появится на экране. Чтобы показать фрейм на экране, метод `main()` вызывает метод `setVisible()`.



На заметку! До появления Java SE 5.0 для отображения фрейма на экране можно было использовать метод `show()`. Дело в том, что суперклассом `JFrame` является класс `Window`. У класса `Window` есть суперкласс `Component`, который также содержит метод `show()`. Начиная с Java SE 1.2 метод `Component.show()` не рекомендован к использованию (deprecated), поэтому для того, чтобы сделать компонент видимым, мы рекомендуем вызывать метод `setVisible(true)`. Обратите внимание, что в Java SE 1.4 и более ранних версиях ограничения на применение метода `Window.show()` не были установлены. Метод `show()` удобен тем, что он одновременно делает окно видимым и перемещает его на передний план. Увы, данное преимущество стало достоянием прошлого. В Java SE 5.0 метод `show()` также переведен в разряд не рекомендованных к применению.

На этом работа метода `main()` завершается. Обратите внимание на то, что это не приводит к прекращению работы программы. Завершается лишь ее основной поток. Поток пользовательского интерфейса, обеспечивающий работу программы, активизируется выводом фрейма на экран.

Окно выполняющейся программы показано на рис. 7.4. Как видите, такие элементы, как строка заголовка и пиктограммы для изменения размеров окна, отображаются операционной системой, а не компонентами библиотеки Swing. При запуске этой программы в средах Windows, GTK или Mac окно будет выглядеть совершенно по-другому. Все, что находится внутри фрейма, изображается с помощью библиотеки Swing. В данной программе фрейм просто заполняется фоном, цвет которого задается по умолчанию.



На заметку! Работая с Java SE 1.4, вы можете отключить все строки заголовка и другие элементы фреймов, вызвав метод `frame.setUndecorated(true)`.

Позиционирование фрейма

В классе `JFrame` есть лишь несколько методов, позволяющих изменить внешний вид фрейма. Разумеется, благодаря наследованию можно использовать методы суперклассов класса `JFrame`, задающие размер и местоположение. К наиболее важным из них следует отнести следующие.

- Методы `setLocation()` и `setBounds()`, устанавливающие положение фрейма.
- Метод `dispose()`, закрывающий окно и освобождающий все системные ресурсы, использованные при его создании.
- Метод `setIconImage()`, сообщающий оконной системе о том, какая пиктограмма должна отображаться в строке заголовка, окне переключателя задач и т.п.
- Метод `setTitle()`, позволяющий изменить текст в строке заголовка.
- Метод `setResizable()`, получающий в качестве параметра логическое значение и определяющий, имеет ли пользователь право изменять размеры фрейма.

Иерархия наследования для класса `JFrame` показана на рис. 7.5.

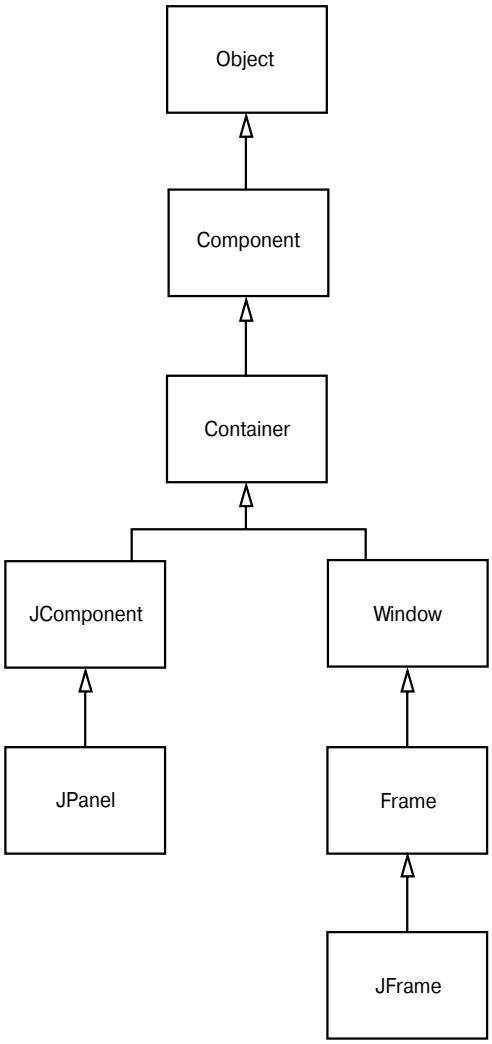


Рис. 7.5. Иерархия наследования для фрейма и классы компонентов AWT и Swing



Совет. В конце данного раздела описаны наиболее важные методы, позволяющие изменять внешний вид фреймов. Некоторые из них определены в классе `JFrame`, другие — в различных суперклассах `JFrame`. В процессе работы часто приходится выбирать наиболее подходящий метод для решения той или иной задачи. К сожалению, найти нужную информацию в документации по JDK довольно сложно. Если речь идет о подклассах, в документацию помещается только информация о переопределяемых методах. Например, метод `toFront()` можно применять к объектам `JFrame`, но, поскольку он наследуется от класса `Window`, в документации по классу `JFrame` о нем ничего не сказано. Если вы считаете, что некий метод должен существовать, а в документации по классу, с которым вы работаете, его нет, попробуйте просмотреть описание методов *суперклассов* этого класса. В начале каждой страницы документации по API есть гипертекстовые ссылки на суперклассы. Кроме того, помимо описания новых и переопределенных методов приводится список всех наследуемых методов.

Как указано в документации на API, методы для изменения размеров и формы фреймов следует искать в классе `Component` (который является предком всех объектов графического пользовательского интерфейса) и в классе `Window` (суперклассе класса `Frame`). Например, метод `show()`, который используется для отображения фрейма на экране, находится в классе `Window`, а в классе `Component` есть метод `setLocation()`, позволяющий изменить местоположение компонента. В приведенном ниже методе левый верхний угол фрейма размещается в точке, находящейся на расстоянии `x` пикселей вправо и на расстоянии `y` пикселей вниз от точки (0,0) — левого верхнего угла экрана.

```
setLocation(x,y)
```

Аналогично, метод `setBounds()` класса `Component` позволяет одновременно изменить и размер, и местоположение компонента (в частности, объекта класса `JFrame`) с помощью следующего вызова:

```
setBounds(x, y, width, height)
```

В качестве альтернативы вы можете предоставить оконной системе контроль за размещением окон. Если вызвать перед отображением окна

```
setLoationByPlatform(true);
```

то оконная система сама укажет местоположение (но не размер), обычно с небольшим смещением от предыдущего окна.



На заметку! Координаты фрейма, задаваемые методами `setLocation()` и `setBounds()`, вычисляются относительно экрана. Как будет показано в главе 9, координаты других компонентов внутри контейнера определяются относительно самого контейнера.

Определение подходящего размера фрейма

Запомните, если размер не задан явно, все фреймы по умолчанию имеют размер 0×0 пикселей. Чтобы не усложнять наш пример, мы задали размеры фреймов так, чтобы окно нормально отображалось на большинстве мониторов. Однако в профессиональных приложениях сначала нужно проверить разрешающую способность экрана и написать код, изменяющий размер фрейма в соответствии с полученной величиной: окно, которое прекрасно смотрится на экране портативного компьютера, на экране дисплея с большой разрешающей способностью будет выглядеть как почтовая марка.

Для того чтобы определить размер экрана, надо выполнить следующие действия. Сначала следует вызвать статический метод `getDefaultToolkit()` класса `Toolkit`, который вернет объект `Toolkit`. (Класс `Toolkit` содержит много методов, предназначенных для взаимодействия с оконной системой конкретной платформы.) Затем надо вызвать метод `getScreenSize()`, который вернет размер экрана в виде объекта `Dimension`. Объект `Dimension` содержит ширину и высоту в общедоступных (!) переменных `width` и `height`. Ниже представлен фрагмент кода, с помощью которого можно определить размер экрана.

```
Toolkit kit = Toolkit.getDefaultToolkit();
Dimension screenSize = kit.getScreenSize();
int screenWidth = screenSize.width;
int screenHeight = screenSize.height;
```

Мы также предоставим пиктограмму для фрейма. Поскольку процесс изображения рисунков на экране также зависит от операционной системы, для загрузки рисунка

нам снова нужен объект Toolkit. Затем этот рисунок устанавливается в качестве пиктограммы.

```
Image img = kit.getImage("icon.gif");
setIcon(img);
```

В зависимости от операционной системы эта пиктограмма выводится по-разному. Например, в системе Windows пиктограмма отображается в левом верхнем углу окна, и ее можно увидеть в списке активных задач, если нажать клавиши <Alt+Tab>.

Ниже дается еще несколько дополнительных советов по обращению с фреймами.

- Если ваш фрейм содержит только стандартные компоненты, такие как кнопки и текстовые поля, вы можете просто вызвать метод pack(), чтобы установить размер фрейма. Фрейм будет установлен в минимальный размер, позволяющий уместить все его компоненты. Довольно часто приходится устанавливать главный фрейм программы в максимальный размер. Начиная с Java SE 1.4, вы можете просто максимизировать фрейм, вызывая

  ```
  frame.setExtendedState(Frame.MAXIMIZED_BOTH);
  ```

- Также хорошей идеей будет запомнить, как пользователь позиционировал фрейм, и какой размер задал ему, чтобы восстановить эти значения при очередном запуске приложения. В главе 10 вы увидите, как использовать для этой цели Preferences API.
- Если вы пишете приложение, которое использует преимущества многоэкранного дисплея, применяйте классы GraphicsEnvironment и GraphicsDevice для нахождения размеров экранов.
- Класс GraphicsDevice также позволяет выполнять приложения в полноэкранном режиме

Листинг 7.2. Содержимое файла SizedFrameTest.java

```
1. import java.awt.*;
2.
3. import javax.swing.*;
4.
5.
6. /**
7.  * @version 1.32 2007-04-14
8.  * @author Cay Horstmann
9.  */
10. public class SizedFrameTest
11. {
12.    public static void main(String[] args)
13.    {
14. EventQueue.invokeLater(new Runnable()
15. {
16. public void run()
17. {
18. SizedFrame frame = new SizedFrame();
19. frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20. frame.setVisible(true);
21. }
22. });
23.   }
24. }
25.
```

```
26. class SizedFrame extends JFrame
27. {
28.    public SizedFrame()
29.    {
30.       // Определение размеров экрана.
31.
32.       Toolkit kit = Toolkit.getDefaultToolkit();
33.       Dimension screenSize = kit.getScreenSize();
34.       int screenHeight = screenSize.height;
35.       int screenWidth = screenSize.width;
36.
37.       // Установка ширины и высоты фрейма
38.       // и позиционирование с помощью платформы.
39.
40.       setSize(screenWidth / 2, screenHeight / 2);
41.       setLocationByPlatform(true);
42.
43.       // Установка пиктограммы и заголовка окна.
44.
45.       Image img = kit.getImage("icon.gif");
46.       setIconImage(img);
47.       setTitle("SizedFrame");
48.    }
49. }
```

API java.awt.Component 1.0

- `boolean isVisible()`
- `void setVisible(boolean b)`

  Получает или устанавливает свойство видимости. Компоненты являются видимыми изначально, за исключением таких элементов верхнего уровня, как `JFrame`.

- `get setSize(int width, int height)` **1.1**

  Устанавливает текущую ширину и длину компонента.

- `void setLocation(int x, int y)` **1.1**

  Перемещает компонент в новую точку. Если компонент не принадлежит верхнему уровню, координаты *x* и *y* (или `p.x` и `p.y`) отсчитываются относительно контейнера, в противном случае (например, для объектов класса `JFrame`) используется экранная система координат.

- `void setBounds(int x, int y, int width, int height)` **1.1**

  Перемещает компонент и изменяет его размеры. Расположение левого верхнего угла задают параметры `x` и `y`, а новый размер — параметры `width` и `height`.

- `Dimension getSize(int width, int height)` **1.1**

  Возвращает текущий размер компонента.

API java.awt.Window 1.0

- `void toFront()`

  Выводит окно на экран поверх других окон.

- `void toBack()`
  Помещает данное окно позади всех окон, выведенных на экран, и соответственно переупорядочивает остальные окна.
- `boolean isLocationByPlatform()` **5.0**
- `void setLocationByPlatform(boolean b)` **5.0**
  Получает или устанавливает свойство `locationByPlatform`. Если это свойство установлено до того, как данное окно отображено, то платформа выбирает подходящее местоположение.

API java.awt.Frame 1.0

- `boolean isResizable()`
- `void setResizable(boolean b)`
  Получает или устанавливает свойство `resizable`. Если это свойство установлено, то пользователь может изменять размеры фрейма.
- `String getTitle()`
- `void setTitle(String s)`
  Получает или устанавливает свойство `title`, определяющее текст в строке заголовка фрейма.
- `Image getIconImage()`
- `void setIconImage(Image image)`
  Получает или устанавливает свойство `iconImage`, определяющее пиктограмму фрейма. Оконная система может отображать пиктограмму как часть оформления фрейма, либо где-нибудь в другом месте.
- `boolean isUndecorated()` **1.4**
- `void setUndecorated(boolean b)` **1.4**
  Получает или устанавливает свойство `undecorated`. Когда свойство установлено, то фрейм отображается без таких деталей оформления, как линейка заголовка или кнопка закрытия. Этот метод должен быть вызван до отображения фрейма.
- `int getExtendedState()` **1.4**
- `void setExtendedState(int state)` **1.4**
  Определяет или устанавливает расширенное состояние окна. Переменная `state` может принимать следующие значения:

```
Frame.NORMAL
Frame.ICONIFIED
Frame.MAXIMIZED_HORIZ
Frame.MAXIMIZED_VERT
Frame.MAXIMIZED_BOTH
```

API java.awt.Toolkit 1.0

- `static Toolkit getDefaultToolkit()`
  Возвращает объект класса `Toolkit`, предусмотренный по умолчанию.
- `Dimension getScreenSize()`
  Определяет размеры экрана.
- `Image getImage(string filename)`
  Загружает изображение из файла `filename`.

Отображение информации в компоненте

В этом разделе мы покажем, как выводить информацию во фрейме. Например, в главе 3 мы выводили строку на консоль в текстовом режиме. Теперь мы выведем сообщение во фрейме, как показано на рис. 7.6.



Рис. 7.6. Фрейм, отображающий информацию

Строку сообщения можно вывести непосредственно во фрейм, однако никто из профессиональных программистов так не делает. В языке Java фреймы предназначены именно для того, чтобы быть контейнерами для *компонентов* (например, меню или других элементов пользовательского интерфейса). Обычно рисунки выводятся в другом компоненте, называемом *панелью* (panel), которая добавляется к фрейму.

Оказывается, структура класса `JFrame` достаточно сложна. Его внутреннее устройство показано на рис. 7.7. Как видно на рисунке, класс `JFrame` состоит из четырех слоев, каждый из которых представляет собой область (pane). Корневая область (root pane), область слоя (layered pane) и прозрачная область (glass pane) не представляют для нас интереса. Они нужны лишь для создания меню и области содержимого (content pane), выдержанных в определенном стиле. Наиболее интересной с точки зрения использования библиотеки Swing является область содержимого. При разработке фрейма компоненты добавляются в область содержимого с помощью следующего кода:

```
Container contentPane = frame.getContentPane();
Component c = . . .;
contentPane.add(c);
```

В Java SE 1.4 и более старых версиях при попытке вызвать метод `add()` класса `Jframe` генерировалось исключение с сообщением "Do not use `JFrame.add()`. Use `JFrame.getContentPane().add()` instead" ("Не используйте `JFrame.add()`. Взамен применяйте `JFrame.getContentPane().add()`"). В версии Java SE 5.0 был реализован метод `JFrame.add()`, который лишь переадресует вызов методу `add()` области содержимого. Таким образом, в Java SE 5.0 вы можете использовать следующий вызов:

```
frame.add(c);
```

В данном случае мы хотим добавить во фрейм единственный компонент, в котором будем отображать сообщение. Чтобы отобразить компонент, определим класс, расширяющий `JComponent`, и переопределим метод `paintComponent()` этого класса. Метод `paintComponent()` получает один параметр типа `Graphics`, который представляет собой набор установок для изображения рисунков и текста, например, в нем задается шрифт и цвет текста. Все рисование в языке Java осуществляется с помощью объектов класса `Graphics`. В этом классе предусмотрены методы для рисования узоров, изображений и текста.

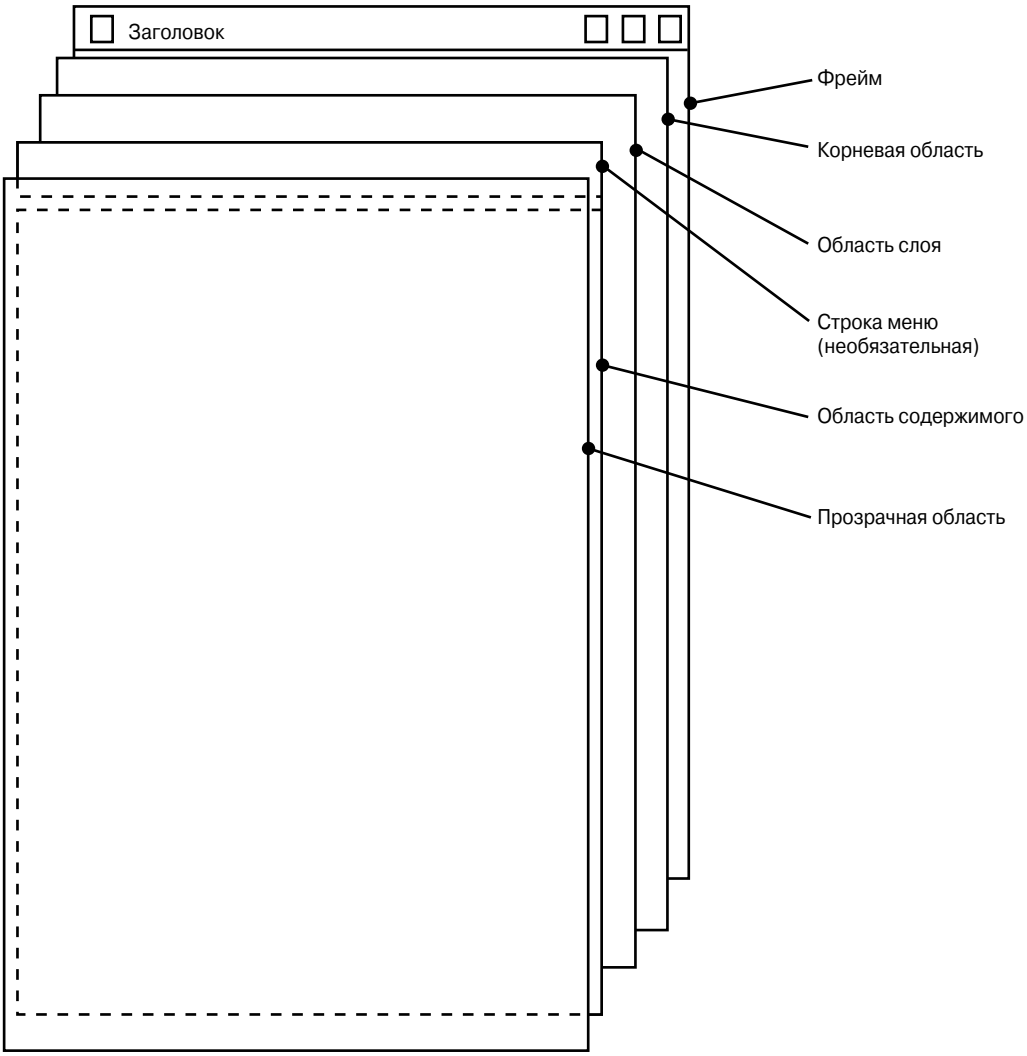


Рис. 7.7. Внутренняя структура `JFrame`



На заметку! Параметр типа `Graphics` напоминает контекст устройства в среде Windows или графический контекст в среде X11.

Ниже демонстрируется процесс создания компонента, в котором можно рисовать.

```
class MyComponent extends JComponent
{
   public void paintComponent(Graphics g)
   {
      код, выполняющий рисование
   }
}
```

Каждый раз, когда возникает необходимость перерисовать окно, неважно по какой причине, обработчик событий уведомляет об этом компонент. В результате выполняется метод `paintComponent()` всех компонентов.

Метод `paintComponent()` никогда не следует вызывать самостоятельно. Когда нужно перерисовать окно приложения, он вызывается автоматически и вмешиваться в этот процесс не следует.

В каких случаях необходима автоматическая перерисовка? Например, при увеличении или уменьшении размеров окна. Если пользователь открыл новое окно, которое перекрыло существующее, окно приложения оказывается поврежденным и также должно быть перерисовано. (Графическая система не сохраняет в памяти пиксели нижележащего окна.) Наконец, когда окно отображается на экране впервые, нужно выполнить код, указывающий, как и где должны отображаться его элементы.



Совет. Если необходимо принудительно перерисовать экран, вызывайте вместо метода `paintComponent()` метод `repaint()`. Метод `repaint()` обратится к методу `paintComponent` каждого компонента и передаст ему сконфигурированный требуемым образом объект `Graphics`.

Как видно из приведенного выше фрагмента программы, метод `paintComponent()` имеет один параметр типа `Graphics`. Размеры объекта типа `Graphics` при отображении на экран задаются в пикселях. Координата (0,0) соответствует левому верхнему углу компонента, на поверхности которого выполняется рисование.

Отображение текста на экране считается особой разновидностью рисования. Класс `Graphics` содержит метод `drawString()`, который вызывается следующим образом:

```
g.drawString(text, x, y)
```

В нашем случае строка текста `"Not a Hello, World Program"` выводится на экран в исходном окне, занимающем приблизительно четверть экрана по ширине и половину — по высоте. Несмотря на то что нам пока неизвестно, как задается размер строки, установим ее начало в точке с координатами (75, 100). Это означает, что вывод первого символа строки начинается с точки, отстоящей на 75 пикселей от правого края и на 100 пикселей — от верхнего. (На самом деле 100 пикселей отсчитывается от верхнего края до базовой линии строки.) Метод `paintComponent()` выглядит приблизительно так:

```
class NotHelloWorldPanel extends JComponent
{
  public void paintComponent(Graphics g)
  {
    ... // См. ниже.
    g.drawString("Not a Hello, World program",
                 MESSAGE_X, MESSAGE_Y);
  }

  public static final int MESSAGE_X = 75;
  public static final int MESSAGE_Y = 100;
}
```

Полностью код программы представлен в листинге 7.3.



На заметку! Вместо расширения `JComponent` некоторые программисты предпочитают расширить класс `JPanel`. `JPanel` предназначен служить контейнером, который может содержать в себе другие компоненты, но можно также рисовать непосредственно в нем. Есть только одно отличие. Панель непрозрачна, а это означает, что она отвечает за рисование всех пикселей в ее пределах. Простейший способ достичь того же — залить панель цветом фона, вызвав `super.paintComponent()` в методе `paintComponent` каждого подкласса панели:

```
class NotHelloWorldPanel extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      ... // Код для рисования.
   }
}
```

Листинг 7.3. Содержимое файла `NotHelloWorld.java`

```
1. import javax.swing.*;
2. import java.awt.*;
3.
4. /**
5.  * @version 1.32 2007-06-12
6.  * @author Cay Horstmann
7.  */
8. public class NotHelloWorld
9. {
10.   public static void main(String[] args)
11.   {
12.      EventQueue.invokeLater(new Runnable()
13.         {
14.            public void run()
15.            {
16.               NotHelloWorldFrame frame = new NotHelloWorldFrame();
17.               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18.               frame.setVisible(true);
19.            }
20.         });
21.   }
22. }
23.
24. /**
25.  * Фрейм, содержащий панель с сообщением.
26.  */
27. class NotHelloWorldFrame extends JFrame
28. {
29.   public NotHelloWorldFrame()
30.   {
31.      setTitle("NotHelloWorld");
32.      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
33.
34.      // Добавление панели к фрейму.
35.
36.      NotHelloWorldPanel panel = new NotHelloWorldPanel();
37.      add(panel);
38.   }
```

```
39.
40.   public static final int DEFAULT_WIDTH = 300;
41.   public static final int DEFAULT_HEIGHT = 200;
42. }
43.
44. /**
45.  * Панель, отображающая сообщение.
46.  */
47. class NotHelloWorldPanel extends JPanel
48. {
49.   public void paintComponent(Graphics g)
50.   {
51.      g.drawString("Not a Hello, World program", MESSAGE_X, MESSAGE_Y);
52.   }
53.
54.   public static final int MESSAGE_X = 75;
55.   public static final int MESSAGE_Y = 100;
56. }
```

API javax.swing.JFrame 1.2

- `Container getContentPane()`

  Возвращает область содержимого для объекта класса `JFrame`.

- `void add(Component c)`

  Добавляет указанный компонент в область содержимого фрейма. (До появления Java SE 5.0 вызов этого метода приводил к генерации исключения.)

API java.awt.Component 1.0

- `void repaint()`

  Вызывает перерисовку компонента. Перерисовка выполняется сразу же после того, как возникнут условия, позволяющие сделать это.

- `public void repaint(int x, int y, int width, int height)`

  Вызывает перерисовку части компонента. Перерисовка выполняется сразу же после того, как возникнут условия, позволяющие сделать это.

API javax.swing.JComponent 1.2

- `void paintComponent(Graphics g)`

  В подклассах этот метод переопределяется. Содержимое метода определяет действия по выводу данных.

Двумерные фигуры

Начиная с версии 1.0, в классе `Graphics` имеются методы для рисования линий, прямоугольников, эллипсов и т.д. Однако эти операции предоставляют слишком ограниченный набор возможностей. Например, в них нельзя изменить толщину линии и повернуть фигуру на произвольный угол.

В Java SE 1.2 появилась библиотека *Java 2D*, реализующая более мощный набор графических операций. В этой главе мы рассмотрим лишь ее основные свойства — более подробная информация представлена во втором томе в главе, посвященной расширенным средствам AWT.

Для того чтобы нарисовать фигуру с помощью библиотеки Java 2D, нужно создать объект класса `Graphics2D`. Этот класс является подклассом класса `Graphics`. Начиная с Java SE 2, такие методы, как `paintComponent()`, автоматически получают объекты класса `Graphics2d`. Нужно лишь применить приведение типов, как показано ниже.

```
public void paintComponent(Graphics g)
{
   Graphics g2 = (Graphics2D)g;
   ...
}
```

При создании геометрических фигур в библиотеке Java 2D применяется объектно-ориентированный подход. В частности, перечисленные ниже классы, предназначенные для рисования линий, прямоугольников и эллипсов, реализуют интерфейс `Shape`.

```
Line2D
Rectangle2D
Ellipse2D
```



На заметку! Библиотека Java 2D поддерживает рисование и более сложных фигур, в частности, дуг, квадратичных и кубических кривых, а также произвольных линий. Более подробная информация об этом содержится во втором томе.

Для того чтобы нарисовать фигуру, сначала нужно создать объект некоторого класса, реализующего интерфейс `Shape`, а затем вызвать метод `draw()` класса `Graphics2D`, например:

```
Rectangle2D rect = ...;
g2.draw(rect);
```



На заметку! До появления библиотеки Java 2D программисты использовали для рисования фигур методы класса `Graphics`, например, метод `drawRectangle()`. На первый взгляд старый способ вызова методов выглядит немного проще. Однако, используя библиотеку Java 2D, программист получает большую свободу действий — впоследствии он может уточнить свой рисунок с помощью многочисленных средств, предусмотренных в этой библиотеке.

Использование классов библиотеки Java 2D сопряжено с некоторыми трудностями. В отличие от методов рисования, предусмотренных в версии 1.0, в которых применялись целочисленные координаты, выраженные в пикселях, в библиотеке Java 2D используются координаты, представленные числами с плавающей точкой. Во многих случаях это очень удобно, поскольку часто программист формирует фигуру, координаты которой выражаются естественными единицами измерения (такими как дюймы или миллиметры), а уж затем переводит их в пиксели. В библиотеке Java 2D в ходе внутренних вычислений применяются числа с плавающей точкой с одинарной точностью. Этого вполне достаточно, ведь главной целью геометрических вычислений является вывод пикселей на экран или на принтер. Поскольку все округления в ходе вычислений не выходят за пределы одного пикселя, это совершенно не влияет на внешний вид

фигуры. Более того, вычисления над числами с плавающей точкой с одинарной точностью на некоторых платформах выполняются быстрее, к тому же числа типа `float` занимают вдвое меньше памяти, чем числа типа `double`.

И все же манипулировать числами типа `float` иногда бывает неудобно, поскольку язык Java в вопросах приведения типов непреклонен и требует преобразовывать числа `double` в числа `float` явно. Рассмотрим, к примеру, следующее выражение:

```
float f = 1.2;                    // Ошибка.
```

Это предложение не будет скомпилировано, поскольку константа `1.2` имеет тип `double`, и компилятор зафиксирует потерю точности. Придется, формируя константу с плавающей точкой, указать суффикс `F`:

```
float f = 1.2F;                   // Правильно.
```

Теперь рассмотрим следующий пример:

```
rectangle2D r = ...
float f = r.getWidth();           // Ошибка.
```

Этот фрагмент не скомпилируется по тем же причинам. Метод `getWidth()` возвращает число, имеющее тип `double`. На этот раз нужно выполнить приведение типов:

```
float f = (float)r.getWidth();    // Правильно.
```

Поскольку указание суффиксов и приведение типов доставляют довольно много хлопот, разработчики библиотеки Java 2D решили предусмотреть *две версии* каждого класса для рисования фигур: в первой версии все координаты выражаются числами, имеющими тип `float` (для дисциплинированных программистов), а во второй — числами типа `double` (для ленивых). (В рамках этой книги мы относим себя ко второй группе и везде используем тип `double`.)

Разработчики библиотеки выбрали необычный и на первый взгляд запутанный способ создания пакетов, соответствующих этим двум версиям. Рассмотрим класс `Rectangle2D`. Это абстрактный класс, имеющий два конкретных подкласса, каждый из которых является внутренним и статическим.

```
Rectangle2D.Float
Rectangle2D.Double
```

На рис. 7.8 показана диаграмма наследования этих классов.

Лучше всего совсем проигнорировать тот факт, что эти два конкретных класса являются внутренними и статическими, — это лишь уловка, чтобы избежать применения таких имен, как `FloatRectangle2D` и `DoubleRectangle2D`. (Статические внутренние классы детально рассмотрены в главе 6.)

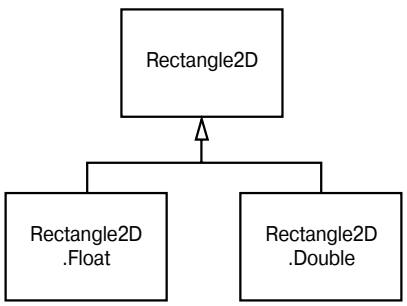


Рис. 7.8. Классы для рисования прямоугольника

При создании объекта класса Rectangle2d.Float используются координаты, имеющие тип float, а в объектах класса Rectangle2d.Double они выражаются числами типа double.

```
Rectangle2D.Float floatRect = new Rectangle2d.Float(10.0F,
   25.0F, 22.5F, 20.0F);
Rectangle2D.Double doubleRect = new Rectangle2d.Double(10.0,
   25.0, 22.5, 20.0);
```

Поскольку и класс Rectangle2d.Float, и класс Rectangle2D.Double расширяют один и тот же класс Rectangle2D, а методы в этих подклассах переопределяют методы суперкласса, нет нужды помнить точный тип координат, используемых при создании фигуры. Достаточно применять в качестве ссылок на прямоугольники переменные типа Rectangle2D:

```
Rectangle2D floatRect = new Rectangle2d.Float (10.0F,
   25.0F, 22.5F, 20.0F);
Rectangle2D doubleRect = new Rectangle2d.Double(10.0,
   25.0, 22.5, 20.0);
```

Таким образом, программисту приходится работать с внутренними классами лишь при создании объектов, описывающих фигуры.

Параметры конструктора задают координаты верхнего левого угла, ширину и высоту прямоугольника.



На заметку! В действительности класс Rectangle2D.Float имеет один дополнительный метод, который не наследуется от класса Rectangle2d, — setRect(float x, float y, float h, float w). Если вместо ссылок типа Rectangle2D.Float использовать ссылки типа Rectangle2D, этот метод станет недоступным. Однако это не большая потеря — класс Rectangle2D имеет свой метод setRect() с параметрами типа double.

И параметры, и значения, возвращаемые методами класса Rectangle2D, имеют тип double. Например, метод getWidth() возвращает значение типа double, даже если ширина хранится в переменной, имеющей тип float, в объекте класса Rectangle2D.Float.



Совет. Чтобы избежать значений типа float, используйте классы, в которых координаты выражаются числами типа double. Однако если вам нужно создать тысячи объектов, следует подумать об экономии памяти. В этом вам помогут классы, в которых координаты задаются числами типа float.

Все, что было сказано нами о классах Rectangle2D, относится и к любым другим классам, предназначенным для рисования плоских фигур. Кроме того, существует еще и класс Point2D с подклассами Point2D.Float и Point2D.Double. Ниже приведен пример создания соответствующего объекта.

```
Point2D p = new Point2D.Double(10, 20);
```



Совет. Класс point2D очень полезен. Работа с объектами класса Point2D больше соответствует объектно-ориентированной парадигме, чем работа с отдельными координатами *x* и *y*. Многие конструкторы и методы получают на входе параметры, имеющие тип Point2D. Мы полагаем, что при малейшей возможности программист должен использовать именно эти объекты, так как они делают геометрические вычисления более понятными.

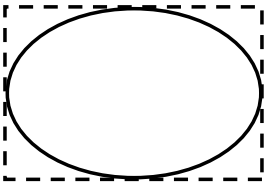


Рис. 7.9. Прямоугольник, ограничивающий эллипс

И класс `Rectangle2D`, и класс `Ellipse2D` являются производными классами одного и того же суперкласса `RectangularShape`. Всем известно, что эллипс не является прямоугольной фигурой, однако он ограничен прямоугольником (рис. 7.9).

В классе `RectangularShape` определено более 20 методов, общих для всех фигур, среди которых находятся и такие полезные методы, как `getWidth()`, `getHeight()`, `getCenterX()` и `getCenterY()` (однако, к сожалению, на время написания этой книги в нем не было метода `getCenter()`, возвращающего центр фигуры в виде объекта класса `Point2D`).

Отметим, что в новую библиотеку вошли и старые классы из библиотеки Java 1.0, заняв свое место в иерархии наследования. Так, например, классы `Rectangle` и `Point`, которые хранят координаты прямоугольника и его центра в виде целых чисел, расширяют классы `Rectangle2D` и `Point2D`.

На рис. 7.10 показаны отношения между классами, представляющими фигуры. Однако подклассы `Float` и `Doubl`, на рисунке не приведены. Классы, унаследованные от предыдущей версии, представлены закрашенными прямоугольниками.

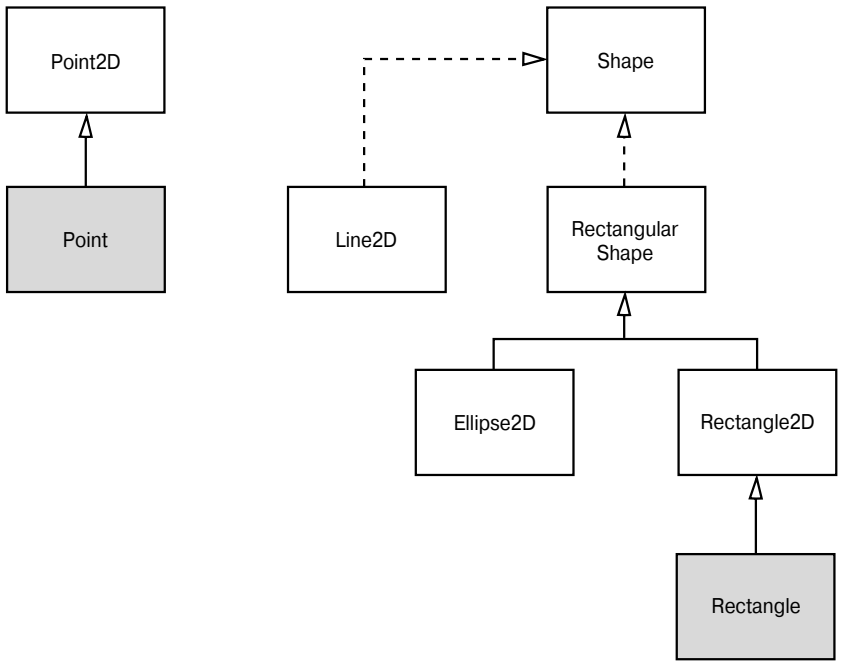


Рис. 7.10. Отношения между классами, представляющими геометрические фигуры

Объекты классов `Rectangle2D` и `Ellipse2D` создаются довольно легко. Достаточно указать следующие данные:

- координаты x и y левого верхнего угла;
- ширина и высота фигуры.

Для эллипсов эти параметры относятся к ограничивающим их прямоугольникам.

Например, приведенная ниже строка кода создает эллипс, ограниченный прямоугольником, левый верхний угол которого находится в точке (150, 200), а сам он имеет ширину и высоту, равную соответственно 100 и 50 пикселям.

```
Ellipse2D e = new Ellipse2D.Double(150, 200, 100, 50);
```

Иногда бывает нелегко определить, где должен находиться левый верхний угол фигуры. Довольно часто прямоугольник задается противоположными вершинами, но не обязательно они соответствуют левому верхнему и нижнему правому углу. В этом случае невозможно построить прямоугольник с помощью следующего оператора:

```
Rectangle2D rect = new Rectangle2D.Double(px, py, qx - px, qy - py);
   // Ошибка.
```

Если параметр `p` не соответствует координатам верхнего левого угла, одна или обе разности между координатами будут отрицательными и прямоугольник окажется пустым. В этом случае сначала нужно создать пустой прямоугольник, а затем использовать метод `setFrameFromDiagonal()`.

```
Rectangle2D rect = new Rectangle2D.Double();
rect.setFrameFromDiagonal(px, py, qx, qy);
```

Еще лучше, если противоположные вершины известны и представлены в виде объектов `p` и `q` класса `Point2D`. В этом случае можно воспользоваться оператором

```
rect.setFrameFromDiagonal(p, q);
```

При создании эллипса обычно известны центр, ширина и высота ограничивающего прямоугольника, но не координаты его левого верхнего угла. Существует метод `setFrameFromCenter()`, использующий центр, но ему нужно задать одну из четырех угловых точек. Следовательно, эллипс лучше создать следующим образом:

```
Ellipse2D ellipse = new Ellipse2D.Double(centerX - width / 2,
   centerY - height / 2, width, height);
```

Для того чтобы нарисовать линию, нужно задать начальную и конечную точки, заданные либо как объекты класса `Point2D`, либо в виде пары чисел:

```
Line2D line = new Line2D.Double(start, end);
```

или

```
Line2D line = new Line2D.Double(startX,
startY, endX, endY);
```

Программа, приведенная в листинге 7.4, рисует прямоугольник, эллипс, вписанный в прямоугольник, диагональ прямоугольника, а также окружность, центр которой совпадает с центром прямоугольника.

Результат работы программы показан на рис. 7.11.

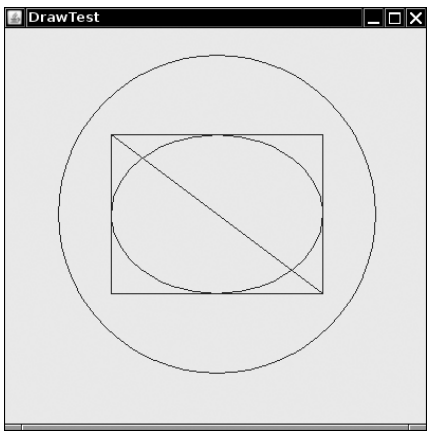


Рис. 7.11. Рисование геометрических фигур

Листинг 7.4. Содержимое файла `DrawTest.java`

```
1. import java.awt.*;
2. import java.awt.geom.*;
3. import javax.swing.*;
4.
```

```
5. /**
6.  * @version 1.32 2007-04-14
7.  * @author Cay Horstmann
8.  */
9. public class DrawTest
10.{
11.   public static void main(String[] args)
12.   {
13.      EventQueue.invokeLater(new Runnable()
14.         {
15.            public void run()
16.            {
17.               DrawFrame frame = new DrawFrame();
18.               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.               frame.setVisible(true);
20.            }
21.         });
22.   }
23.}
24.
25./**
26. * Фрейм, содержащий панель для рисования.
27. */
28.class DrawFrame extends JFrame
29.{
30.   public DrawFrame()
31.   {
32.      setTitle("DrawTest");
33.      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34.
35.      // Добавление панели к фрейму.
36.
37.      DrawComponent component = new DrawComponent();
38.      add(component);
39.   }
40.
41.   public static final int DEFAULT_WIDTH = 400;
42.   public static final int DEFAULT_HEIGHT = 400;
43.}
44.
45./**
46. * Компонент, отображающий прямоугольники и эллипсы.
47. */
48.class DrawComponent extends JComponent
49.{
50.   public void paintComponent(Graphics g)
51.   {
52.      Graphics2D g2 = (Graphics2D) g;
53.
54.      // Рисование прямоугольника.
55.
56.      double leftX = 100;
57.      double topY = 100;
58.      double width = 200;
59.      double height = 150;
60.
```

```
61.      Rectangle2D rect = new Rectangle2D.Double(leftX, topY, width, height);
62.      g2.draw(rect);
63.
64.      // Рисование эллипса, вписанного в прямоугольник.
65.
66.      Ellipse2D ellipse = new Ellipse2D.Double();
67.      ellipse.setFrame(rect);
68.      g2.draw(ellipse);
69.
70.      // Рисование диагонали.
71.
72.      g2.draw(new Line2D.Double(leftX, topY, leftX + width, topY + height));
73.
74.      // Рисование окружности, центр которой совпадает
75.      // с центром прямоугольника.
76.
77.      double centerX = rect.getCenterX();
78.      double centerY = rect.getCenterY();
79.      double radius = 150;
80.
81.      Ellipse2D circle = new Ellipse2D.Double();
82.      circle.setFrameFromCenter(centerX, centerY, centerX + radius, centerY + radius);
83.      g2.draw(circle);
84.   }
85. }
```

API java.awt.geom.RectangularShape 1.2

- `double getCenterX()`
- `double getCenterY()`
- `double getMinX()`
- `double getMinY()`
- `double getMaxX()`
- `double getMaxY()`

  Возвращает координаты центра, наименьшую и наибольшую координату *x* и *y* описанного прямоугольника.

- `double getWidth()`
- `double getHeight()`

  Возвращает ширину и высоту описанного прямоугольника.

- `double getX()`
- `double getY()`

  Возвращает координаты *x* и *y* левого верхнего угла описанного прямоугольника.

API java.awt.geom.Rectangle2D.Double 1.2

- `Rectangle2D.Double(double x, double y, double w, double h)`

  Создает прямоугольник, заданный посредством координат верхнего левого угла, ширины и высоты.

API java.awt.geom.Rectangle2D.Float 1.2

- `Rectangle2D.Float(float x, float y, float w, float h)`

  Создает прямоугольник, заданный посредством координат верхнего левого угла, ширины и высоты.

API java.awt.geom.Ellipse2D.Double 1.2

- `Ellipse2D.Double(double x, double y, double w, double h)`

  Создает эллипс, включающий прямоугольник которого имеет заданные координаты верхнего левого угла, ширину и высоту.

API java.awt.geom.Point2D.Double 1.2

- `Point2D.Double(double x, double y)`

  Рисует точку с заданными координатами.

API java.awt.geom.Line2D.Double 1.2

- `Line2D.Double(Point2D start, Point2D end)`
- `Line2D.Double(Point2D startX, Point2D startY, Point2D endX, Point2D startX)`

  Рисует линию с заданными координатами начала и конца.

Работа с цветом

Метод `setPaint()` класса `Graphics2D` позволяет выбрать цвет, который будет применяться при всех дальнейших операциях, связанных с рисованием в графическом контексте. Например:

```
g2.setPaint(Color.RED)
g2.drawString("Warning!", 100, 100);
```

Вы можете закрашивать цветом интерьеры или замкнутые фигуры (такие как прямоугольники или эллипсы). Для этого вместо `draw` просто вызывайте `fill`:

```
Rectangle2D rect = . . .;
g2.setPaint(Color.RED);
g2.fill(rect); // закрашивает прямоугольник красным цветом
```

Для того чтобы иметь возможность создавать многоцветные рисунки, следует выбрать некоторый цвет, нарисовать фигуру, затем выбрать новый цвет, нарисовать другую фигуру и т.д.

Цвет определяется с помощью класса `Color`. Класс `java.awt.Color` содержит константы, которые соответствуют 13 стандартным цветам: `BLACK`, `BLUE`, `CYAN`, `DARK_GRAY`, `GRAY`, `GREEN`, `LIGHT_GRAY`, `MAGENTA`, `ORANGE`, `PINK`, `RED`, `WHITE`, `YELLOW`.



На заметку! До появления Java SE 1.4 имена констант, определяющих цвет, задавались строчными буквами, например `Color.red`. Это довольно странно, поскольку для констант принято формировать имена, состоящие из прописных букв. Начиная с Java SE 1.4, стандартные названия цветов указываются как прописными, так и строчными буквами (для сохранения обратной совместимости).

Вы можете указать свой произвольный цвет, создав объект класса Color из красной, зеленой и синей составляющих. В конструкторе класса Color компоненты цветов задаются целыми числами в диапазоне от 0 до 255.

```
Color(int redness, int greenness, int blueness)
```

Ниже приведен пример указания собственного цвета.

```
g2.setPaint(new Color(0, 128, 128); // Тусклый сине-зеленый цвет.
g2.drawString("Welcome!", 75, 125);
```



На заметку! Кроме сплошного цвета, можно выбирать более сложные конструкции с изменением оттенков. Подробно этот вопрос будет обсуждаться во втором томе в главе, посвященной расширенным средствам AWT. Если вместо Graphics2D используется объект класса Graphics, то для установки цвета нужно применять метод setColor().

Для того чтобы установить цвет фона, следует применять метод setBackground() класса Component, являющегося предком класса JPanel:

```
MyPanel p = new MyPanel();
p.setBackground(Color.PINK);
```

Существует также метод setForeground(). Он задает цвет, который используется при рисовании компонента.



Совет. Методы brighter() и darker() класса Color создают (в полном соответствии со своими именами) более светлые или темные варианты текущего цвета. Используя метод brighter(), можно легко выделить заголовок или пункт меню. В действительности одного вызова метода brighter() для увеличения яркости недостаточно. Для того чтобы достичь заметного эффекта, желательно применить его трижды: c.brighter().brighter().brighter().

Для большинства известных цветов в классе SystemColor предусмотрены символьные имена. Константы в этом классе определяют цвета, которые используются для разнообразных элементов пользовательского интерфейса. Например, приведенный ниже вызов выбирает в качестве фона фрейма цвет, заданный по умолчанию для всех окон пользовательского интерфейса.

```
p.setBackground(SyetemColor.window)
```

Фон заполняет окно при каждой перерисовке. Использовать цвета, заданные в классе SystemColor, особенно удобно, если вы хотите, чтобы пользовательский интерфейс был выдержан в той же цветовой гамме, что и рабочий стол. В табл. 7.1 приведены названия системных цветов и их описание.

Таблица 7.1. Системные цвета

Название	Описание
desktop	Цвет фона для рабочего стола
activeCaption	Цвет фона для заголовков
activeCaptionText	Цвет текста для заголовков
activeCaptionBorder	Цвет рамки для неактивных заголовков
inactiveCaption	Цвет фона для неактивных заголовков

Окончание табл. 7.1

Название	Описание
inactiveCaptionText	Цвет текста для неактивных заголовков
inactiveCaptionBorder	Цвет рамки для неактивных заголовков
window	Цвет фона для окна
windowBorder	Цвет рамки для окна
windowText	Цвет текста в окне
menu	Фон для меню
menuText	Цвет текста в меню
text	Цвет фона для текста
textText	Цвет текста
textInactiveText	Цвет текста для неактивных управляющих элементов
textHighlight	Цвет фона для выделенного текста
textHighlightText	Цвет выделенного текста
control	Цвет фона для управляющих элементов
controltext	Цвет текста для элементов управления
controlLtHighlight	Цвет для слабого выделения управляющих элементов
controlHighlight	Цвет для выделения управляющих элементов
controlShadow	Цвет тени для управляющих элементов
controlDkShadow	Темный цвет тени для управляющих элементов
scrollbar	Цвет фона для полосы прокрутки
info	Цвет фона для информационных сообщений
infoText	Цвет текста для информационных сообщений

API java.awt.Color 1.0

- `Color(int r, int g, int b)`

  Создает объект класса `Color`.

  Параметры `r` Интенсивность красного цвета (0–255),
  `g` Интенсивность зеленого цвета (0–255).
  `b` Интенсивность синего цвета (0–255).

API java.awt.Graphics 1.0

- `Color getColor()`
- `void setColor(Color c)`

  Получает или устанавливает текущий цвет. При выполнении последующих графических операций будет использоваться новый цвет.

  Параметр `c` Новый цвет.

API java.awt.Graphics2D 1.0

- `Paint getPaint()`
- `void setPaint(Paint p)`

  Получает или устанавливает атрибуты рисования для данного графического контекста. Класс `Color` реализует интерфейс `Paint`. Этот метод можно использовать для задания сплошного цвета при рисовании.

- `void fill(Shape s)`

  Заполняет текущую фигуру.

API java.awt.Component 1.0

- `Color getBackground()`
- `void setBackground(Color c)`

  Получает или устанавливает цвет фона.

  Параметр `c` Новый цвет фона.

- `Color getForeground()`
- `void setForeground(Color c)`

  Получает или устанавливает цвет переднего плана.

  Параметр `c` Новый цвет переднего плана.

Использование шрифтов при выводе текста

Программа, приведенная в начале этой главы, выводила на экран строку текста шрифтом по умолчанию. Часто требуется, чтобы текст отображался разными шрифтами. Шрифты определяются *начертанием* символов. Имя начертания состоит из названия семейства шрифтов (например, Helvetica) и необязательного суффикса (например, Bold). Так, например, названия Helvetica и Helvetica Bold считаются частью одного и того же семейства Helvetica.

Для того чтобы выяснить, какие шрифты доступны на конкретном компьютере, вызовите метод `getAvailableFamilyNames()` класса `GraphicsEnvironment`. Этот метод возвращает массив строк, состоящих из названий всех доступных шрифтов. Чтобы создать экземпляр класса `GraphicsEnvironment`, описывающего графическую среду, используется статический метод `getLocalGraphicsEnvironment()`. Таким образом, приведенная ниже программа выводит названия всех шрифтов, доступных в вашей системе.

```
import java.awt.*

public class ListFonts
{
   public static void main(String[] args)
   {
      String[] fontNames = GraphicsEnvironment
         .getLocalGraphicsEnvironment()
         .getAvailableFontFamilyNames();
      for (int i = 0; i  fontNames.length; i++)
         System.out.println(fontNames[i]);
   }
}
```

В одной из систем список начинается такими строками:

```
Abadi MT Condenced light
Arial
Arial Black
Arial Narrow
Arioso
Baskerville
Binner Gothic
...
```

И так далее для следующих семидесяти шрифтов.

Имена шрифтов представляют собой торговые марки, а сами шрифты могут быть защищены авторскими правами. Распространение шрифтов часто сопряжено с оплатой. Разумеется, аналогично дешевым подделкам дорогих духов, существуют имитации фирменных шрифтов. Например, имитация шрифта Helvetica распространяется с операционной системой Windows под именем Arial.

Для того чтобы поддерживать определенную систему, в AWT принято пять *логических* названий шрифтов:

```
SansSerif
Serif
Monospaced
Dialog
DialogInput
```

Эти имена всегда соответствуют шрифтам, фактически существующим в данном клиентском компьютере. Например, в системе Windows шрифт SanSerif отображается на шрифт Arial.

В дополнение Sun JDK всегда включает три семейства шрифтов под названием “Lucida Sans”, “Lucida Bright” и “Lucida Sans Typewriter”.

Для того чтобы изобразить букву с помощью заданного шрифта, сначала необходимо создать объект класса `Font`, а затем указать название шрифта, его стиль и размер символов. Ниже показано, как можно создать объект класса `Font`.

```
Font helvb14 = new Font("Helvetica", Font.BOLD, 14);
```

Третий параметр задает размер шрифта. Для указания размеров используется единица измерений, которая называется пунктом (point). В одном дюйме содержится 72 пункта.

В конструкторе класса `Font` вместо фактического названия начертания можно использовать логическое имя шрифта. Затем нужно указать стиль, задав второй аргумент конструктора равным одному из следующих значений:

```
Font.PLAIN
Font.BOLD
Font.ITALIC
Font.BOLD + Font.ITALIC
```



На заметку! Отображение логических шрифтов на имена физических шрифтов определено в файле `fontconfig.properties` в подкаталоге `jre/lib` инсталляции Java. Информация об этом файле доступна по адресу `http://java.sun.com/javase/6/docs/technotes/guides/intl/fontconfig.html`.

Начиная с версии JDK 1.3, можно использовать шрифты TrueType. Для этого нужно создать отдельный поток ввода для конкретного шрифта — обычно с файла на диске или с узла сети (потоки будут подробно рассматриваться во втором томе). Затем необходимо вызвать статический метод `FontCreateFont()`.

```
URL url = new URL("http://www.fonts.com/Wingbats.ttf");
InputStream in = url.openStream();
Font f = Font.createFont(Font.TRUETYPE_FONT, in);
```

Этот шрифт не является ни курсивным, ни полужирным, а его размер равен одному пункту. Чтобы получить шрифт заданного размера, используйте метод `deriveFont()`:

```
Font df = f.deriveFont(14.0F);
```



Внимание! Существуют две перегруженные версии метода `deriveFont()`. Одна из них (с параметром типа `float`) задает размер шрифта, другая (с параметром типа `int`) — стиль шрифта. Таким образом, вызов `f.deriveFont(14)` задает стиль, а не размер! (В результате будет установлен курсив, поскольку двоичное представление числа 14 содержит единицу в разряде, соответствующем константе `ITALIC`, но не `BOLD`.)

В языке Java шрифты состоят из букв, цифр, знаков пунктуации и некоторых других символов. Например, если вывести шрифтом Dialog символ `'\u2297'`, на экране появится знак ⊗. Доступными являются только символы, указанные в множестве Unicode.

Ниже приводится код, который выводит на экран строку “Hello, World” с помощью полужирного шрифта SanSerif размером 14 пунктов.

```
Font sansbold14 = new Font("SansSerif", Font.BOLD, 14);
g2.setFont(sansbold14);
String message = "Hello, World!";
g2.drawString(message, 75, 100);
```

Теперь *выровняем строку по центру* компонента. Для этого нужно знать ширину и высоту строки в пикселях. Процесс выравнивания зависит от трех факторов.

- Используемый шрифт (в данном случае — sans serif, полужирный, 14 пунктов).
- Символы строки (в данном случае — “Hello, World!”).
- Устройство, на котором будет отображаться строка (в данном случае — экран).

Чтобы получить объект, представляющий характеристики экрана, нужно вызвать метод `getFontRenderContext()` класса `Graphics2D`. Он возвращает объект класса `FontRenderContext`. Этот объект необходимо передать методу `getStringBounds()` класса `Font`.

```
FontRenderContext context = g2.getFontRenderContext();
Rectangle2D bounds = f.getStringBounds(message, context);
```

Метод `getStringBounds()` возвращает прямоугольник, ограничивающий строку.

Чтобы выяснить, от чего зависят размеры этого прямоугольника, надо рассмотреть некоторые основные термины, применяемые при наборе текстов (рис. 7.12). *Базовая линия* (baseline) — это воображаемая линия, которая касается снизу таких символов, как, например, “e”. *Максимальная высота* (ascent) — максимальное расстояние от базовой линии до вершин *выступающих над базовой линией* символов (asceder), например, верхнего края буквы “b” или “k”. *Максимальная глубина посадки* символов (descent) — это расстояние от базовой линии до *нижнего края* (descender), например, нижнего края буквы “p” или “g”.

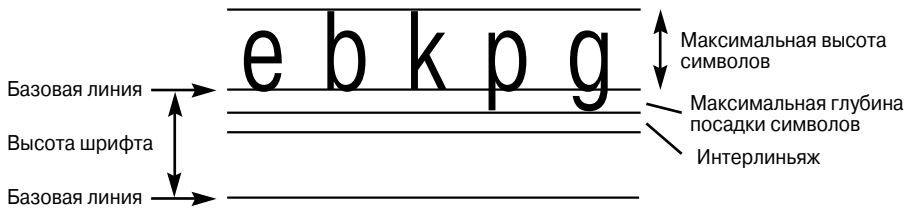


Рис. 7.12. Основные термины, применяемые при формировании строки текста

Интерлиньяж (leading) – это разность между максимальной глубиной посадки символов предыдущей строки и максимальной высотой символов следующей строки. *Высота* (height) шрифта – это расстояние между соседними базовыми линиями, т.е. максимальная высота и максимальная глубина посадки символов плюс интерлиньяж.

Ширина прямоугольника, возвращаемого методом `getStringBounds()`, задает горизонтальный размер строки. Высота прямоугольника равна сумме максимальной высоты и максимальной глубины посадки символов плюс интерлиньяж. Координата *y* отсчитывается от базовой линии. Для верхней части прямоугольника она является отрицательной. Таким образом, ширину, высоту и максимальную высоту символов можно вычислить следующим образом:

```
double stringWidth = bounds.getWidth();
double stringHeight = bounds.getHeight();
double ascent = -bounds.getY();
```

Если нужно узнать, чему равен интерлиньяж или максимальная глубина посадки символов, следует вызвать метод `getLineMetrics()` класса `Font`. Этот метод возвращает объект класса `LineMetrics()`, в котором есть методы для определения указанных выше характеристик.

```
LineMetrics metrics = f.getLineMetrics(message, context);
float descent = metrics.getDescent();
float leading = metrics.getLeading();
```

В приведенном ниже фрагменте эта информация используется для выравнивания строки по центру содержащего ее компонента.

```
FontRenderContext context = g2.getFontRenderContext();
Rectangle2D bounds = f.getStringBounds(message, context);

// (x,y) — координаты верхнего левого угла строки текста
double x = (getWidth() - bounds.getWidth()) / 2;
double y = (getHeight() - bounds.getHeight()) / 2;

// Для определения базовой линии максимальная высота
// складывается с y
double ascent = -bounds.getY();
double baseY = y + ascent;
g2.drawString(message, (int) x, (int) baseY);
```

Для того чтобы понять, как происходит выравнивание текста по центру, отметим, что метод `getWidth()` возвращает ширину компонента. Часть этой ширины занимает строка сообщения. Ее горизонтальный размер возвращает метод `bounds.getWidth()`. Оставшееся пространство нужно равномерно распределить по обеим сторонам строки. Следовательно, размер незаполненного пространства с каждой стороны должен рав-

няться половине разности между шириной компонента и длиной строки. Те же рассуждения справедливы и для выравнивания строки по высоте.



На заметку! Когда необходимо вычислить размерности за пределами метода `paintComponent`, вы не можете получить контекст визуализации шрифта из объекта `Graphics2D`. В этом случае вызывайте метод `getFontMetrics` класса `JComponent`, а потом `getFontRenderContext`:

```
FontRenderContext context = getFontMetrics(f).getFontRenderContext();
```

Итак, полученная программа рисует базовую линию и ограничивающий прямоугольник. На рис. 7.13 показано, как это выглядит на экране, а сама программа приведена в листинге 7.5.



Рис. 7.13. При выполнении программы отображаются ограничивающий прямоугольник и базовая линия строки

Листинг 7.5. Содержимое файла `FontTest.java`

```
1. import java.awt.*;
2. import java.awt.font.*;
3. import java.awt.geom.*;
4. import javax.swing.*;
5.
6. /**
7.  * @version 1.33 2007-04-14
8.  * @author Cay Horstmann
9.  */
10. public class FontTest
11. {
12.    public static void main(String[] args)
13.    {
14.       EventQueue.invokeLater(new Runnable()
15.          {
16.             public void run()
17.             {
18.                FontFrame frame = new FontFrame();
19.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20.                frame.setVisible(true);
21.             }
22.          });
23.    }
24. }
25.
26. /**
27.  * Фрейм с компонентом для вывода текста.
28.  */
```

```
29. class FontFrame extends JFrame
30. {
31.    public FontFrame()
32.    {
33.       setTitle("FontTest");
34.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
35.
36.       // Добавление компонента к фрейму.
37.
38.       FontComponent component = new FontComponent();
39.       add(component);
40.    }
41.
42.    public static final int DEFAULT_WIDTH = 300;
43.    public static final int DEFAULT_HEIGHT = 200;
44. }
45.
46. /**
47.  * Компонент, в центре которого выводится сообщение в рамке.
48.  */
49. class FontComponent extends JComponent
50. {
51.    public void paintComponent(Graphics g)
52.    {
53.       Graphics2D g2 = (Graphics2D) g;
54.
55.       String message = "Hello, World!";
56.
57.       Font f = new Font("Serif", Font.BOLD, 36);
58.       g2.setFont(f);
59.
60.       // Определение размеров сообщения.
61.
62.       FontRenderContext context = g2.getFontRenderContext();
63.       Rectangle2D bounds = f.getStringBounds(message, context);
64.
65.       // Определение координат левого верхнего угла
66.       // строки текста.
67.
68.       double x = (getWidth() - bounds.getWidth()) / 2;
69.       double y = (getHeight() - bounds.getHeight()) / 2;
70.
71.       // Для определения базовой линии используется
72.       // максимальная высота.
73.
74.       double ascent = -bounds.getY();
75.       double baseY = y + ascent;
76.
77.       // Вывод сообщения.
78.
79.       g2.drawString(message, (int) x, (int) baseY);
80.
81.       g2.setPaint(Color.LIGHT_GRAY);
82.
```

```
83.      // Вывод базовой линии.
84.
85.      g2.draw(new Line2D.Double(x, baseY, x + bounds.getWidth(), baseY));
86.
87.      // Вывод ограничивающего прямоугольника.
88.
89.      Rectangle2D rect =
90.         new Rectangle2D.Double(x, y, bounds.getWidth(), bounds.getHeight());
91.      g2.draw(rect);
92.   }
93. }
```

API java.awt.Font 1.0

- `Font(string name, int style, int size)`

  Создает новый объект класса `Font`.

  Параметры `name` Имя шрифта, представляющее собой либо имя начертания (например, Helvetica Bold), либо логическое название шрифта (например, Serif или SansSerif).

  `style` Стиль (`Font.PLAIN`, `Font.BOLD`, `Font.ITALIC` или `Font.BOLD+Font.ITALIC`).

  `size` Размер в пунктах (например, `12`).

- `String getFontName()`

  Возвращает имя начертания (например, Helvetica Bold).

- `String getFamily()`

  Возвращает имя семейства шрифтов (например, Helvetica).

- `String getName()`

  Возвращает логическое имя (например, SansSerif), если оно было присвоено шрифту при его создании; в противном случае возвращает имя начертания.

- `Rectangle2D getStringBounds(String s, FontRenderContext)` **1.2**

  Возвращает прямоугольник, ограничивающий данную строку. Координата y прямоугольника отсчитывается от базовой линии. Координата *y* верхней части прямоугольника равна максимальной высоте со знаком "минус". Высота прямоугольника равна сумме максимальной высоты и максимальной глубины посадки. Ширина прямоугольника равна ширине строки.

- `LineMetrics getLineMetrics(String s, FontRenderContext context)` **1.2**

  Возвращает объект класса `LineMetrics`, позволяющий определить характеристики строки.

- `Font deriveFont(int style)` **1.2**
- `Font deriveFont(float size)` **1.2**
- `Font deriveFont(int style, float size)` **1.2**

  Возвращает новый объект класса `Font`, совпадающий с текущим, за исключением размера и стиля, заданных в качестве параметров.

API java.awt.font.LineMetrics 1.2

- `float getAscent()`

  Определяет максимальную высоту — расстояние от базовой линии до верхней части прописных букв.

- `float getDescent()`

  Определяет максимальную глубину посадки — расстояние от базовой линии до нижней части букв.

- `float getLeading()`

  Определяет интерлиньяж — расстояние между нижней частью предыдущей строки и верхом следующей.

- `float getHeight()`

  Определяет общую высоту шрифта — расстояние между двумя базовыми линиями текста (максимальная высота и интерлиньяж плюс максимальная глубина посадки).

API java.awt.Graphics 1.0

- `Font getFont()`
- `void setFont(Font font)`

  Извлекает или устанавливает шрифт в графическом контексте. Этот шрифт будет использоваться в дальнейших операциях с графикой.

- `void drawString(String str, int x, int y)`

  Выводит на экран строку, шрифт и цвет которой определяются текущими установками графического контекста.

Параметры	`str`	Строка, подлежащая выводу
	`x`	Координата *x* начала строки
	`y`	Координата *y* начала строки

API java.awt.Graphics2D 1.2

- `FontRenderContext getFontRenderContext()`

  Определяет контекст, который задает характеристики шрифта для выполнения операции с графикой.

- `void drawString(String str, float x, float y)`

  Выводит строку, шрифт и цвет которой определяются текущими установками.

Параметры	`str`	Строка, подлежащая выводу.
	`x`	Координата *x* начала строки.
	`y`	Координата *y* начала строки.

API java.awt.JComponent 1.2

- `FontMetrics getFontMetrics()` **5.0**

  Извлекает метрики заданного шрифта. Класс `FontMetrics` — это предшественник класса `LineMetrics`.

API **java.awt.FontMetrics 1.0**

- `FontRenderContext getFontRenderContext()` **1.2**

  Получает контекст отображения для шрифта.

Вывод графических изображений

Вы уже видели, как путем вывода линий и фигур создаются простые изображения. Сложные изображения, такие как фотографии, обычно создаются отдельно, например, в результате сканирования или применения графических редакторов. (Прочитав второй том, вы узнаете, что можно создавать изображения и помещать набор пикселей в массив. Эта процедура широко применяется при генерировании фрактальных изображений.)

Если изображения хранятся в файлах на диске компьютера или в глобальной сети, их можно прочитать, а затем отобразить на экране с помощью объектов класса `Graphics`. Начиная с Java SE 1.4, организовать чтение изображений очень легко. Если рисунок хранится в локальном файле, достаточно использовать следующий фрагмент кода:

```
String filename = "...";
Image image = ImageIO.read(new File(filename));
```

При вызове метода можно также указать URL:

```
String filename = "...";
Image image = ImageIO.read(new URL(urlname));
```

Если рисунок не доступен, метод `read()` генерирует исключение `IOException`. Обработка исключений описана в главе 11. Пока будем считать, что наша простая программа перехватывает исключение и выводит на экран содержимое стека.

Теперь переменная `image` содержит ссылку на объект, инкапсулирующий графическое изображение. Этот объект можно отобразить на экране с помощью метода `drawImage()` класса `Graphics`.

```
public void paintComponent(Graphics g)
{
   ...
   g.drawImage(image, x, y, null);
}
```

Программа, приведенная в листинге 7.6, делает немного больше, многократно отображая указанное изображение. Результат работы программы показан на рис. 7.14. Вывод осуществляется с помощью метода `paintComponent()`. Сначала одна копия рисунка помещается в левый верхний угол окна, а затем вызывается метод `copyArea()`, который копирует его.

```
for (int i = 0; i * imageWidth , = getWidth(); i++)
   for (int j = 0; j * imageHeight <= getHeight(); j++)
     if ( i + j > 0)
        g.copyArea(0, 0, imageWidth, imageHeight,
           i * imageWidth, j * imageHeight);
```

В листинге 7.6 приведен полный исходный текст программы для работы с изображениями.

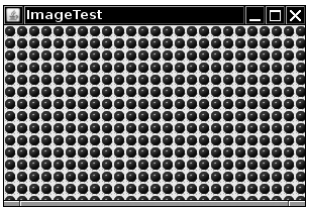


Рис. 7.14. Окно, содержащее копии одного и того же изображения

Листинг 7.6. Содержимое файла `ImageTest.java`

```
1. import java.awt.*;
2. import java.io.*;
3. import javax.imageio.*;
4. import javax.swing.*;
5.
6. /**
7.  * @version 1.33 2007-04-14
8.  * @author Cay Horstmann
9.  */
10. public class ImageTest
11. {
12.    public static void main(String[] args)
13.    {
14.       EventQueue.invokeLater(new Runnable()
15.          {
16.             public void run()
17.             {
18.                ImageFrame frame = new ImageFrame();
19.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20.                frame.setVisible(true);
21.             }
22.          });
23.    }
24. }
25.
26. /**
27.  * Фрейм с компонентом, на котором выводится изображение.
28.  */
29. class ImageFrame extends JFrame
30. {
31.    public ImageFrame()
32.    {
33.       setTitle("ImageTest");
34.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
35.
36.       // Добавление компонента к фрейму.
37.
38.       ImageComponent component = new ImageComponent();
39.       add(component);
40.    }
41.
42.    public static final int DEFAULT_WIDTH = 300;
43.    public static final int DEFAULT_HEIGHT = 200;
44. }
```

```
45.
46. /**
47.  * Компонент, на котором многократно выводится изображение.
48.  */
49. class ImageComponent extends JComponent
50. {
51.    public ImageComponent()
52.    {
53.       // Получение изображения.
54.       try
55.       {
56.          image = ImageIO.read(new File("blue-ball.gif"));
57.       }
58.       catch (IOException e)
59.       {
60.          e.printStackTrace();
61.       }
62.    }
63.
64.    public void paintComponent(Graphics g)
65.    {
66.    if (image == null) return;
67.
68.    int imageWidth = image.getWidth(this);
69.    int imageHeight = image.getHeight(this);
70.
71.    // Отображение рисунка в левом верхнем углу.
72.
73.    g.drawImage(image, 0, 0, null);
74.
75.    // Многократный вывод изображения в панели.
76.
77.    for (int i = 0; i * imageWidth <= getWidth(); i++)
78.       for (int j = 0; j * imageHeight <= getHeight(); j++)
79.          if (i + j > 0) g.copyArea(0, 0, imageWidth, imageHeight,
80.                                    i * imageWidth, j * imageHeight);
81.    }
82.
83.    private Image image;
84. }
```

API javax.swing.ImageIO 1.4

- `static BufferedImage read(File f)`
- `static BufferedImage read(URL u)`

  Чтение изображения из указанного файла или копирование его с узла сети.

API java.awt.Graphics 1.0

- `boolean drawImage(Image img, int x, int y, ImageObserver observer)`

  Выводит изображение. (Этот метод может вернуть управление до того, как изображение будет полностью выведено.)

Параметры `img` Изображение, предназначенное для вывода.
`x` Координата *x* левого верхнего угла.
`y` Координата *y* левого верхнего угла.
`observer` Объект, обеспечивающий уведомление о ходе процесса (может быть нулевой ссылкой `null`).

- `boolean drawImage(Image img, int x, int y, int width, int height, ImageObserver observer)`

  Выводит масштабированное изображение. Система масштабирует изображение для того, чтобы оно умещалось в области с заданной шириной и высотой. (Этот метод может вернуть управление до того, как изображение будет полностью выведено.)

  Параметры `img` Изображение, предназначенное для вывода.
  `x` Координата *x* левого верхнего угла.
  `y` Координата *y* левого верхнего угла.
  `width` Требуемая ширина изображения
  `height` Требуемая высота изображения.
  `observer` Объект, обеспечивающий уведомление о ходе процесса (может быть нулевой ссылкой `null`).

- `boolean copyArea(int x, int y, int width, int height, int dx, int dy)`

  Копирует область экрана.

  Параметры `x` Координата *x* левого верхнего угла копируемой области.
  `y` Координата *y* левого верхнего угла копируемой области.
  `width` Ширина исходной области.
  `height` Высота исходной области.
  `dx` Расстояние по горизонтали от копируемой области до области, в которую выполняется копирование.
  `dy` Расстояние по вертикали от копируемой области до области, в которую выполняется копирование.

На этом введение в программирование графики на Java завершено. Более сложные приемы программирования двумерной графики и манипуляции изображениями будут изложены во втором томе. В следующей главе вы узнаете, как заставить вашу программу реагировать на пользовательский ввод.

Глава 8

ОБРАБОТКА СОБЫТИЙ

В ЭТОЙ ГЛАВЕ...

- Общие сведения об обработке событий
- Действия
- События мыши
- Иерархия событий AWT

Обработка событий имеет огромное значение для создания программ с графическим пользовательским интерфейсом. Чтобы профессионально реализовать пользовательский интерфейс, необходимо очень хорошо разбираться в этом механизме. В этой главе описано, как работает модель событий Java AWT. Вы узнаете, как перехватывать события от компонентов пользовательского интерфейса и устройств ввода. Также мы покажем, как работать с *действиями*, предлагающими более структурированный подход к обработке событий.

Общие сведения об обработке событий

Каждая операционная среда, поддерживающая графический пользовательский интерфейс, непрерывно отслеживает такие события, как нажатие клавиш или щелчок мыши, а затем сообщают о них выполняемой программе. Программа решает, как реагировать на эти события. В языках, подобных Visual Basic, соответствие между событиями и кодом очевидно: некто пишет код, реагирующий на каждое интересующее событие, и помещает его в так называемую *процедуру обработки события* (event procedure). Например, кнопка в Visual Basic по имени `HelpButton` будет вызвана процедура `HelpButton_Click`. Каждому компоненту пользовательского интерфейса в языке Visual Basic соответствует фиксированное множество событий, состав которого изменить невозможно.

С другой стороны, если для создания программы, реагирующей на события, используется язык, подобный C, программист должен создать код, непрерывно проверяющий очередь событий, которая передается ему операционной системой. (Обычно для этого код обработки событий помещается в гигантский цикл с большим количеством операторов `switch`.) Очевидно, что такая технология довольно неуклюжа, да и создавать такие программы намного сложнее. Однако подобный способ имеет преимущество, заключающееся в том, что множество событий ничем не ограничено, в отличие от языка Visual Basic, в котором очередь событий скрыта от программиста.

Подход, принятый в языке Java, представляет собой нечто среднее между подходами, применяемыми в языках Visual Basic и C. Он довольно сложен. При обработке событий, предусмотренных в библиотеке AWT, программист полностью контролирует передачу событий от *источников* (например, кнопок или полос прокруток) к *слушателю событий* (event listener). *Любой* объект можно назначить слушателем события — на практике вы указываете объект, который может надлежащим образом отреагировать на событие. Эта *модель делегирования событий* позволяет достичь большей гибкости по сравнению с языком Visual Basic, в котором обработчики предопределены.

Источники событий содержат методы, позволяющие связывать их со слушателями. Когда происходит соответствующее событие, источник извещает всех зарегистрированных слушателей.

Как можно ожидать от такого объектно-ориентированного языка, как Java, информация о событии инкапсулируется в *объекте* события. В языке Java все события описываются подклассами `java.util.EventObject`. Конечно, существуют подклассы для каждого типа событий — вроде `ActionEvent` и `WindowEvent`.

Различные источники могут порождать разные виды событий. Например, кнопка может создавать объекты класса `ActionEvent`, а окно — объекты класса `WindowEvent`.

Кратко механизм обработки событий AWT можно описать следующим образом.

- Объект-слушатель представляет собой экземпляр класса, реализующего специальный интерфейс, называемый (естественно) интерфейсом слушателя.

- Источник события — это объект, который может регистрировать слушателей и посылать им объекты событий.
- При наступлении события источник посылает объекты события всем зарегистрированным слушателям.
- Слушатели используют информацию, инкапсулированную в объекте события, для того, чтобы решить, как реагировать на это событие.

На рис. 8.1 показаны отношения между классами-обработчиками событий и интерфейсами.

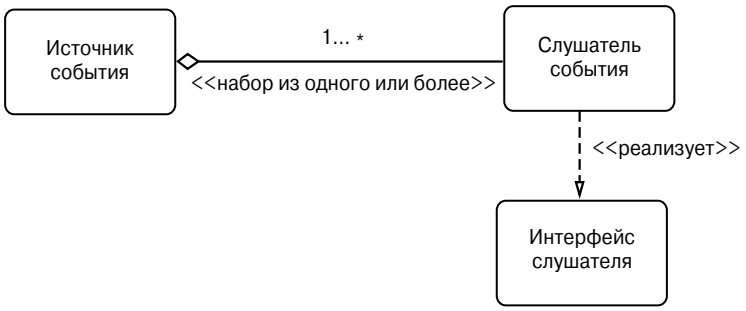


Рис. 8.1. Отношения между классами-обработчиками событий и интерфейсами

Приведем пример спецификации слушателя:

```
ActionListener listener = ...;
Jbutton button = new JButton("OK");
button.addActionListener(listener);
```

Теперь слушатель оповещается о возникновении "события действия", произошедшем в кнопке. Для кнопки, как и следовало ожидать, таким событием действия является щелчок кнопкой мыши.

Для того чтобы реализовать интерфейс `ActionListener`, класс слушателя должен иметь метод с именем `actionPerformed()`, получающий в качестве параметра объект класса `ActionEvent`.

```
class MyListener implements ActionListener
{
   ...
   public void actionPerformed(ActionEvent event)
   {
      // Реакция на щелчок на кнопке.
      ...
   }
}
```

Когда пользователь щелкает на кнопке, объект класса `JButton` создает объект класса `ActionEvent` и вызывает метод `listener.actionPerformed(event)`, передавая ему объект события. Обработчиками события (например, щелчка мышью на кнопке) можно назначать и несколько объектов одновременно. В этом случае после щелчка на кнопке объект класса `JButton` вызовет методы `actionPerformed()` всех зарегистрированных слушателей.

На рис. 8.2 демонстрирует взаимодействие между источником события, слушателем события и объектом события.

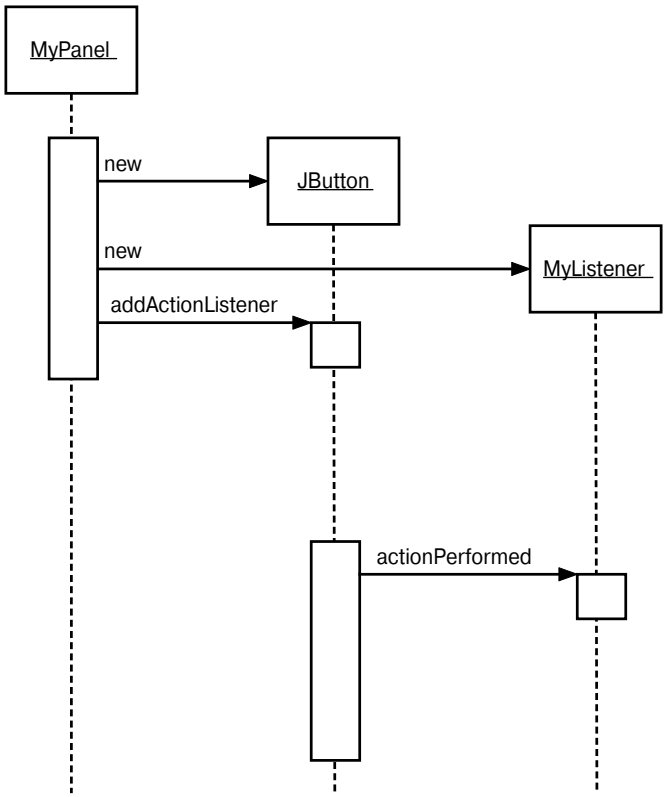


Рис. 8.2. Уведомление о событии

Пример: обработка щелчка на кнопке

Для того чтобы лучше освоить модель делегирования событий, рассмотрим подробно, как обрабатывается щелчок на кнопке. Для этого нам понадобятся три кнопки, расположенные в панели, и три объекта-слушателя, обеспечивающие реакцию на события, связанные с кнопками.

В рамках этого сценария каждый раз, когда пользователь щелкает мышью на какой-нибудь кнопке, расположенной в панели, соответствующий слушатель получает объект класса `ActionEvent`, указывающий на факт щелчка. В нашей программе в ответ на щелчок будет изменяться цвет фона панели.

Прежде чем перейти к рассмотрению программы, реагирующей на щелчок кнопки, нужно объяснить, как создаются кнопки и как их размещают в панели. (Более подробно элементы графического пользовательского интерфейса рассматриваются в главе 9.)

Для того чтобы создать кнопку, в ее конструкторе нужно задать метки, пиктограмму или оба параметра одновременно. Рассмотрим два примера:

```
JButton yellowButton = new JButton("Yellow");
JButton blueButton = new JButton(new ImageIcon("blue-ball.gif"));
```

Вызовем метод add, чтобы добавить кнопки в панель:

```
JButton yellowButton = new JButton("Yellow");
JButton blueButton = new JButton("Blue");
JButton redButton = new JButton("Red");
buttonPanel.add(yellowButton);
buttonPanel.add(blueButton);
buttonPanel.add(redButton);
```

Результат работы этого фрагмента кода показан на рис. 8.3.

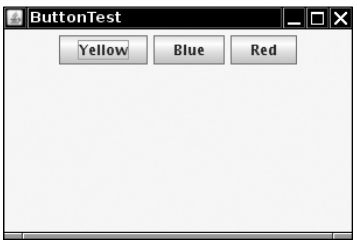


Рис. 8.3. Панель с кнопками

Теперь надо добавить код, позволяющий "прослушивать" эти кнопки. Для этого нужен класс, реализующий интерфейс ActionListener, в котором, как мы уже говорили, объявлен один метод — actionPerformed(). Сигнатура этого метода выглядит следующим образом:

```
public void actionListener(ActionEvent event)
```



На заметку! Интерфейс ActionListener, использованный в нашем примере, не ограничивается отслеживанием щелчков на кнопках. Он применяется во многих ситуациях.

- После двойного щелчка на пункте списка.
- После выбора пункта меню.
- После нажатия клавиши <Enter>, когда курсор находится в текстовом поле.
- По истечении заданного периода времени, отслеживаемого компонентом Timer.

Дополнительные детали вы еще увидите в этой и следующей главах.

Во всех случаях интерфейс ActionListener используется совершенно одинаково: метод actionPerformed() — единственный метод в интерфейсе ActionListener — получает в качестве параметра объект типа ActionEvent. Этот объект несет в себе информацию о событии.

Предположим, что мы хотим после щелчка на кнопке изменить цвет фона панели. Новый цвет нужно указать в классе обработчика.

```
class ColorAction implementsListener
{
   public ColorAction(Color c)
   {
      backgroundColor = c;
   }
   public void actionPerformed(ActionEvent event)
   {
      // Устанавливаем цвет панели.
      ...
   }
   private Color backgroundColor;
}
```

Затем мы создаем по одному объекту для каждого цвета и задаем их в качестве слушателей событий, связанных с кнопкой:

```
ColorAction yellowAction = new ColorACtion(Color.YELLOW);
ColorAction blueAction = new ColorACtion(Color.BLUE);
ColorAction redAction = new ColorAction(Color.RED);

yellowButton.addActionListener(yellowAction);
blueButton.addActionListener(blueAction);
redButton.addActionListener(redAction);
```

Например, если пользователь щелкает на кнопке с надписью Yellow, вызывается метод `actionPerformed()` объекта `yellowAction`. Поле экземпляра `backgroundColor` хранит значение `Color.YELLOW`, поэтому выполняющийся метод может установить требуемый цвет панели.

Остался один небольшой вопрос. Объект `ColorAction` не имеет доступа к переменной `panel`. Эту проблему можно решить разными способами. Переменную `panel` можно указать в конструкторе класса `ColorAction`. Но удобнее сделать класс `ColorAction` вложенным по отношению к классу `ButtonPanel`. В этом случае его методы получат доступ к внешним переменным автоматически. (Более подробно о вложенных классах рассказывается в главе 6.)

Мы будем применять второй способ. Ниже показан пример включения класса `ColorAction` в класс `ButtonPanel`.

```
class ButtonFrame extends JFrame
{
   ...
   private class ColorAction implements ActionListener
   {
      ...
      public void actionPerformed(ActionEvent event)
      {
         buttonPanel.setBackground(backgroundColor);
      }
      private Color backgroundColor;
   }

   private JPanel buttonPanel;
}
```

Рассмотрим метод `actionPerformed()` более подробно. Класс `ColorAction` не имеет поля `buttonPanel`. Но внешний класс `ButtonFrame` — имеет.

Такая ситуация встречается довольно часто. Объекты-слушатели событий должны выполнять какие-то действия, влияющие на другие объекты. Часто бывает очень удобным заранее поместить класс обработчика в состав другого класса, который данный обработчик должен модифицировать.

Листинг 8.1 содержит полный код программы, в которой реализуется обработка событий. Как только пользователь щелкает на какой-нибудь кнопке, соответствующий слушатель изменяет цвет панели.

Листинг 8.1. Содержимое файла `ButtonTest.java`

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
```

```
5. /**
6.  * @version 1.33 2007-06-12
7.  * @author Cay Horstmann
8.  */
9. public class ButtonTest
10.{
11.   public static void main(String[] args)
12.   {
13.      EventQueue.invokeLater(new Runnable()
14.         {
15.            public void run()
16.            {
17.               ButtonFrame frame = new ButtonFrame();
18.               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.               frame.setVisible(true);
20.            }
21.         });
22.   }
23.}
24.
25./**
26. * Фрейм, содержащий панель с кнопками.
27. */
28.class ButtonFrame extends JFrame
29.{
30.   public ButtonFrame()
31.   {
32.      setTitle("ButtonTest");
33.      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34.
35.      // Создание кнопок.
36.      JButton yellowButton = new JButton("Yellow");
37.      JButton blueButton = new JButton("Blue");
38.      JButton redButton = new JButton("Red");
39.
40.      buttonPanel = new JPanel();
41.
42.      // Добавление кнопок к панели.
43.      buttonPanel.add(yellowButton);
44.      buttonPanel.add(blueButton);
45.      buttonPanel.add(redButton);
46.
47.      // Добавление панели к фрейму.
48.      add(buttonPanel);
49.
50.      // Создание действий кнопок.
51.      ColorAction yellowAction = new ColorAction(Color.YELLOW);
52.      ColorAction blueAction = new ColorAction(Color.BLUE);
53.      ColorAction redAction = new ColorAction(Color.RED);
54.
55.      // Связывание действий с кнопками.
56.      yellowButton.addActionListener(yellowAction);
57.      blueButton.addActionListener(blueAction);
58.      redButton.addActionListener(redAction);
59.   }
60.
```

```
61.   /**
62.    * Слушатель действия, изменяющий цвет панели.
63.    */
64.   private class ColorAction implements ActionListener
65.   {
66.      public ColorAction(Color c)
67.      {
68.         backgroundColor = c;
69.      }
70.
71.      public void actionPerformed(ActionEvent event)
72.      {
73.         buttonPanel.setBackground(backgroundColor);
74.      }
75.
76.      private Color backgroundColor;
77.   }
78.
79.   private JPanel buttonPanel;
80.
81.   public static final int DEFAULT_WIDTH = 300;
82.   public static final int DEFAULT_HEIGHT = 200;
83. }
```

API **javax.swing.JButton 1.2**

- `JButton(String label)`
- `JButton(Icon icon)`
- `JButton(String label, Icon icon)`

  Создает кнопку. Строка, передаваемая в качестве параметра, может представлять собой текст или, начиная с версии Java SE 1.3, HTML-выражение, например `<HTML><B>OK</B></HTML>`

API **java.awt.Container 1.0**

- `Component add(Component c)`

  Добавляет компонент `c` в контейнер.

API **javax.swing.ImageIcon 1.2**

- `ImageIcon(String filename)`

  Создает пиктограмму, изображение которой хранится в файле.

Удобство вложенных классов

Некоторым не нравятся вложенные классы, потому что им кажется, что вложение классов и объектов замедляет выполнение программы. Попробуем разобраться в этом. Для каждого компонента пользовательского интерфейса совсем не нужен новый класс. В нашем примере все три кнопки делят между собой один класс-слушатель. Разумеется, каждой из них соответствует отдельный объект этого класса. Однако эти объекты невелики. Такой объект содержит обозначение цвета и ссылку на панель. К тому же традицион-

ное решение с операторами `if...else` также ссылается на те же объекты цветов, которые хранят и слушатели событий, как на локальные переменные, а не поля экземпляра.

Приведем хороший пример, показывающий, как анонимные вложенные классы позволяют упростить исходный код программы. Вернемся к листингу 8.1. Обратите внимание на то, что обработка событий для всех кнопок совершенно одинакова и состоит из следующих операций.

1. Создание кнопки с соответствующей меткой.
2. Размещение кнопки в панели.
3. Создание слушателя, изменяющего цвет панели.
4. Добавление слушателя.

Чтобы упростить эту задачу, создадим вспомогательный метод:

```
void makeButton(String name, Color backgroundColor)
{
   JButton button = new JButton(name);
   add(button);
   ColorAction action = new ColorAction(backgroundColor);
   button.addActionListener(action);
}
```

Тогда мы просто вызываем его:

```
   makeButton("yellow", Color.YELLOW);
   makeButton("blue", Color.BLUE);
   makeButton("red", Color.RED);
```

Однако возможности упрощения программы этим не исчерпываются. Отметим, что класс `ColorAction` нужен лишь один раз: в методе `makeButton()`. Следовательно, его можно преобразовать в анонимный класс.

```
void makeButton(String name, final Color backgroundColor)
{
   JButton button = new JButton(name);
   buttonPanel.add(button);
   button.addActionListener(new
       ActionListener()
      {
         public void actionPerformed(ActionEvent event)
         {
            buttonPanel.setBackgroiund(backgroundColor);
         }
      });
}
```

Код слушателя стал еще немного проще. Метод `actionPerformed()` теперь просто ссылается на параметр `backgroundColor`. (Как и все локальные переменные, доступные во вложенном классе, этот параметр должен быть финальным.)

Явный конструктор теперь не нужен. Как мы уже видели в главе 6, механизм вложенных классов автоматически генерирует конструктор, размещающий в памяти все локальные финальные переменные, используемые в одном из методов этого класса.



Совет. На первый взгляд может показаться, что вложенные классы существенно усложняют код программы. Однако если вы научитесь выделять главное, абстрагируясь на время от остального кода, то легко освоите синтаксис вложенных классов. Рассмотрим пример.

```
button.addActionListener(new ActionListener()
    {
      public void actionPerformed(ActionEvent event)
      {
        buttonPanel.etBackground(backgroundColor);
      }
    });
```

Таким образом, действие кнопки заключается в установке цвета фона. Пока обработчик события состоит всего лишь из нескольких операторов, мы полагаем, что в этом коде вполне можно разобраться, особенно, если программист не боится применять механизм вложенных классов.



На заметку! Вы совершенно свободны в назначении *любого* объекта класса, реализующего интерфейс `ActionListener`, слушателем кнопки. Мы предпочитаем использовать объекты нового класса, которые специально созданы для выполнения нужных действий кнопки. Однако некоторые программисты не чувствуют себя комфортно в применении вложенных классов и выбирают другую стратегию. Они создают контейнер источников событий, реализующий интерфейс `ActionListener`. Затем контейнер устанавливает сам себя в качестве слушателя — примерно так:

```
yellowButton.addActionListener(this);
blueButton.addActionListener(this);
redButton.addActionListener(this);
```

Теперь три кнопки не имеют индивидуальных слушателей. Они разделяют единственный объект слушателя — фрейм кнопок. Таким образом, его метод `actionPerformed` должен определять, какая именно кнопка нажата.

```
class ButtonFrame extends JFrame implements ActionListener
{
 . . .
 public void actionPerformed(ActionEvent event)
 {
    Object source = event.getSource();
    if (source == yellowButton) . . .
    else if (source == blueButton) . . .
    else if (source == redButton ) . . .
    else . . .
 }
}
```

Как видите, все становится несколько запутанным, и потому мы не рекомендуем такой подход.



java.util.EventObject 1.1

- `Object getSource()`

  Возвращает ссылку на объект, являющийся источником события.

API java.awt.event.ActionEvent 1.1

- `String getActionCommand()`

  Возвращает командную строку, связанную с событием. Если событие произошло с кнопкой, командной строкой является метка строки (обратите внимание, что она может быть изменена с помощью метода `setActionCommand()`).

API java.beans.EventHandler 1.4

- `static Object create(Class listenerInterface, Object target, String action)`
- `static Object create(Class listenerInterface, Object target, String action, String eventProperty)`
- `static Object create(Class listenerInterface, Object target, String action, String eventProperty, String listenerMethod)`

  Эти методы создают прокси-объект, реализующий заданный интерфейс. В результате либо указанный метод, либо все методы интерфейса выполняют заданное действие с целевым объектом.

  Параметр `action` может быть именем метода или свойством объекта. Если это свойство — выполняется метод `set...()`. Например, если параметр `action` имеет значение `text`, оно превращается в вызов метода `setText()`.

  Свойство события состоит из одного или нескольких имен свойств, разделенных точками. Первое свойство задает считывание параметра метода из обработчика, второе свойство — считывание результирующего объекта и т.д. Например, свойство `source.text` превращается в вызовы методов `getSource()` и `getText()`.

Создание слушателей, содержащих единственный вызов метода

В Java SE 1.4 появился механизм, позволяющий специфицировать простые слушатели событий без программирования вложенных классов. Например, предположим, что имеется кнопка с меткой "Load" (Загрузить), чей обработчик событий содержит единственный вызов метода:

```
frame.loadData();
```

Конечно, вы можете использовать анонимный вложенный класс:

```
loadButton.addActionListener(new ActionListener()
{
  public void actionPerformed(ActionEvent event)
  {
    frame.loadData();
  }
});
```

Но класс `EventHandler` может создать такой слушатель автоматически, следующим вызовом:

```
EventHandler.create(ActionListener.class, frame, "loadData")
```

Конечно, вы все равно должны установить обработчик:

```
loadButton.addActionListener(
  EventHandler.create(ActionListener.class, frame, "loadData"));
```

Если слушатель вызывает метод с одним параметром, который может быть получен из параметра события, вы можете использовать другую форму создания метода. Например, следующий вызов:

```
EventHandler.create(ActionListener.class, frame, "loadData", "source.text")
```

эквивалентен такому:

```
new ActionListener()
{
   public void actionPerformed(ActionEvent event)
   {
         frame.loadData(((JTextField) event.getSource()).getText());
   }
}
```

Имена свойств `source` и `text` превращаются в вызовы методов `getSource` и `getText`.

Пример: изменение визуального стиля

По умолчанию программы Swing используют визуальный стиль Metal. Изменить стиль можно двумя способами. В подкаталоге JDK `jre/lib` можно предусмотреть файл `swing.properties` и задать в нем с помощью свойства `swing.defaultlaf` имя класса, определяющего нужный стиль, например:

```
swing.defaultlaf=com.sun.java.swing.plaf.motif.MotifLookAndFeel
```

Отметим, что стиль Metal находится в пакете `javax.swing`. Другие стили расположены в пакете `com.sun.java`. Они не являются обязательным атрибутом каждой реализации языка Java. В данный момент по соображениям, связанным с авторским правом, пакеты стилей для операционных систем Windows и Mac поставляются только с исполняющими средами, ориентированными на эти системы.



Совет. Поскольку строки, начинающиеся символом `#`, в файлах, описывающих свойства стилей, игнорируются, в файле `swing.properties` можно предусмотреть несколько стилей, "закомментировав" ненужные.

```
#swing.defaulflaf=javax.swing.plaf.metal.MetalLookAndFeel
swing.defaultlaf=com.sun.java.swing.plaf.motif.MotifLookAndFeel
#swing.defaultlaf=com.sun.java.swing.plaf.windows.WindowsLookAndFeel
```

Чтобы изменить стиль интерфейса программы, ее нужно запустить заново. Программа прочитывает файл `swing.properties` лишь один раз, при запуске.

Второй способ изменяет стиль внешнего вида динамически. Для того чтобы сделать это, вызовите метод `UIManager.setLookAndFeel()` и задайте ему имя класса требуемого стиля.

Затем вызовите статический метод `SwingUtilities.updateComponentTreeUI()` и обновите с его помощью весь набор компонентов. Этому методу достаточно передать один компонент, остальные он найдет автоматически. Если нужный стиль не будет обнаружен или произойдет ошибка при его загрузке, метод `UIManager.setLookAndFeel` может сгенерировать различные исключения. Как обычно, вы можете не уделять внимания обработке исключений, поскольку эта тема будет полностью освещена в главе 11.

Рассмотрим пример, в котором задается стиль Motif графического интерфейса программы.

```
String plaf = "com.sun.java.swing.plaf.motif.MotifLookAndFeel";
try
{
  UIManager.setLookAndFeel(plaf);
  SwingUtilities.updateComponentTreeUI(panel);
}
catch(Exception e); { e.printStackTrace(); }
```

Для того чтобы пронумеровать инсталлированные реализации стилей, надо использовать следующий оператор:

```
UIManager.LookAndFeelInfo[] infos =
  UIManager.getInstalledLookAndFeels();
```

После этого вы можете с помощью приведенного ниже кода получить имя каждого стиля и реализующего его класса.

```
String name = infos[i].getName();
String className = infos[i].getClassName();
```

В листинге 8.2 приведена законченная программа, демонстрирующая переключение стилей (рис. 8.4). Эта программа очень похожа на ту, которая представлена в листинге 8.1. Следуя советам, изложенным выше, для того, чтобы задать действия с кнопками, т.е. переключение стилей, мы применили вспомогательный метод `makeButton()` и анонимный вложенный класс.

У этой программы есть одно преимущество. Метод `actionPerformed()` во вложенном классе-обработчике должен передать методу `updateComponentTreeUI()` ссылку `this` на объект внешнего класса `PlafPanel`. Напомним, что ссылка `this` на объект внешнего класса должна сопровождаться префиксом — именем этого класса.

```
SwingUtilities.updateComponentTreeUI(PlafPanel.this)
```

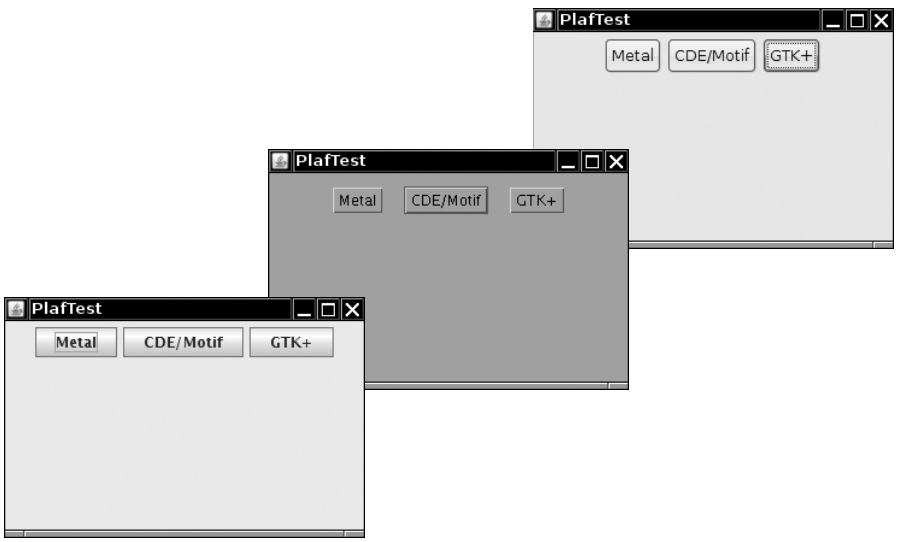


Рис. 8.4. Переключение стилей

Листинг 8.2. Содержимое файла `PlafTest.java`

```
1. import java.awt.EventQueue;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * @version 1.32 2007-06-12
7.  * @author Cay Horstmann
8.  */
9. public class PlafTest
10.{
11.   public static void main(String[] args)
12.   {
13.      EventQueue.invokeLater(new Runnable()
14.         {
15.            public void run()
16.            {
17.               PlafFrame frame = new PlafFrame();
18.               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.               frame.setVisible(true);
20.            }
21.         });
22.   }
23.}
24.
25./**
26. * Фрейм с панелью кнопок для изменения визуального стиля.
27. */
28.class PlafFrame extends JFrame
29.{
30.   public PlafFrame()
31.   {
32.      setTitle("PlafTest");
33.      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34.
35.      buttonPanel = new JPanel();
36.
37.      UIManager.LookAndFeelInfo[] infos = UIManager.getInstalledLookAndFeels();
38.      for (UIManager.LookAndFeelInfo info : infos)
39.      makeButton(info.getName(), info.getClassName());
40.
41.      add(buttonPanel);
42.   }
43.
44./**
45. * Создает кнопку, изменяющую визуальный стиль.
46. * @param name Имя кнопки
47. * @param plafName Имя класса, описывающего стиль
48. */
49.void makeButton(String name, final String plafName)
50.{
51.   // Добавление кнопки к панели.
52.
53.   JButton button = new JButton(name);
54.   buttonPanel.add(button);
55.
```

```
56.   // Связывание действия с кнопкой.
57.
58.   button.addActionListener(new ActionListener()
59.      {
60.         public void actionPerformed(ActionEvent event)
61.         {
62.            // Действие кнопки: переключение на другой стиль.
63.            try
64.            {
65.               UIManager.setLookAndFeel(plafName);
66.               SwingUtilities.updateComponentTreeUI(PlafFrame.this);
67.            }
68.            catch (Exception e)
69.            {
70.               e.printStackTrace();
71.            }
72.         }
73.      });
74.   }
75.
76.   private JPanel buttonPanel;
77.
78.   public static final int DEFAULT_WIDTH = 300;
79.   public static final int DEFAULT_HEIGHT = 200;
80. }
```

API **javax.swing.UIManager 1.2**

- `static UIManager.LookAndFeelInfo[] getInstalledLookAndFeels()`

  Получение массива объектов, описывающих реализации инсталлированных стилей.

- `static setLookAndFeel(String className)`

  Установка текущего стиля, используя заданное имя класса (вроде "`javax.swing.plaf.metal.MetalLookAndFeel`").

API **javax.swing.UIManager.LookAndFeelInfo 1.2**

- `String getName()`

  Возвращает имя стиля.

- `String getClassName()`

  Возвращает имя класса, реализующего стиль.

Классы-адаптеры

Не все события обрабатываются так же просто, как события, связанные с кнопками. Если программа представляет собой профессиональное приложение, она обязательно должна завершаться корректно, и пользователь должен иметь гарантию, что он не потеряет результаты выполненной работы. Например, если пользователь закрыл фрейм, можно вывести на экран диалоговое окно и предупредить его о необходимости сохранить результаты работы и только после его согласия завершить выполнение приложения.

Если программа пытается закрыть фрейм, объект JFrame становится источником события WindowEvent. Для того чтобы перехватить это событие, нужен соответствующий объект-слушатель, который следует добавить в список слушателей окна.

```
WindowListener listener = ...;
frame.addWindowListener(listener);
```

В данном случае слушатель должен быть экземпляром класса, реализующего интерфейс WindowListener. В интерфейсе WindowListener имеется семь методов. Фрейм вызывает их в ответ на семь разных событий, которые могут произойти с окном. Их имена отражают их назначение. Исключением, может быть, является слово "iconified", что в системе Windows означает "свернутое" окно. Вот как выглядит полный интерфейс WindowsListener.

```
public interface WindowListener
{
   void windowOpened(WindowEvent e);
   void windowClosing(WindowEvent e);
   void windowClosed(WindowEvent e);
   void windowIconified(WindowEvent e);
   void windowDeiconified(WindowEvent e);
   void windowActivated(WindowEvent e);
   void windowDeactivated(WindowEvent e);
}
```



На заметку! Если нужно проверить, было ли окно увеличено до максимальных размеров, следует инсталлировать WindowStateListener. Подробную информацию о нем можно найти в документации на API.

Как обычно, любой класс, реализующий какой-либо интерфейс, должен реализовывать все его методы. В данном случае это означает создание тела *семи* методов. Напомним, что нас интересует только один из них — метод windowClosing().

Разумеется, можно определить класс, реализующий интерфейс WindowListener, добавить в метод windowClosing() вызов System.exit(0) и написать пустые функции для остальных шести методов.

```
class Terminator implements WindowListener
{
   public void windowClosing(WindowEvent e)
   {
      System.exit(0);
   }

   public void windowOpened(WindowEvent e) {}
   public void windowClosing(WindowEvent e) {}
   public void windowClosed(WindowEvent e) {}
   public void windowIconified(WindowEvent e) {}
   public void windowDeiconified(WindowEvent e) {}
   public void windowActivated(WindowEvent e) {}
   public void windowDeactivated(WindowEvent e) {}
}
```

Создавать код для шести методов, которые ничего не делают, — неблагодарное занятие. Чтобы упростить нашу задачу, для каждого из интерфейсов слушателей AWT, в котором содержится несколько методов, создается *класс-адаптер*, реализующий все эти методы, причем тело их пусто. Например, класс WindowAdapter содержит семь мето-

дов, не выполняющих никаких операций. Следовательно, класс-адаптер автоматически удовлетворит требованиям, предъявляемым к реализации соответствующего интерфейса. Программист может расширить класс-адаптер и уточнить нужные реакции на некоторые и даже на все типы событий, происходящих в интерфейсе. (Обратите внимание, что интерфейс `ActionListener` содержит только один метод, поэтому для него класс-адаптер не нужен.)

Рассмотрим применение адаптера для окна. Расширим класс `WindowAdapter`, унаследовав шесть пустых методов и переопределив метод `windowClosing()`:

```
class Terminator extends WindowAdapter
{
  public void windowClosing(WindowEvent e)
  {
    if (пользователь согласен)
       System.exit(0);
  }
}
```

Теперь объект класса `Terminator` можно зарегистрировать как слушатель событий:

```
WindowListener listener = new Terminator();
frame.addWindowListener(listener);
```

С этого момента, как только фрейм сгенерирует событие, он передаст его слушателю, вызвав один из его семи методов (рис. 8.5). Шесть из этих методов ничего не делают, а метод `windowClosing()` выполняет вызов `System.exit(0)`, завершая работу приложения.

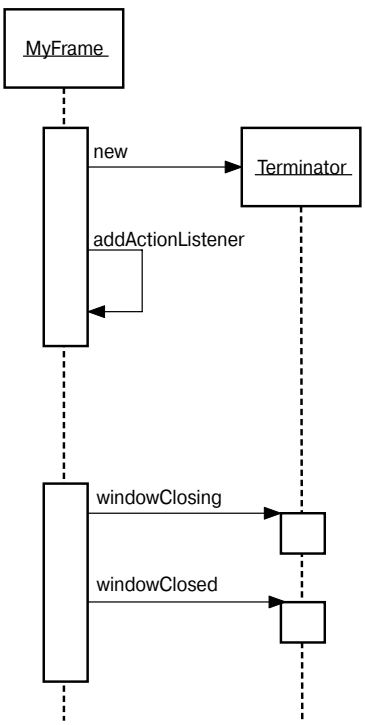


Рис. 8.5. Обработка событий, связанных с окном



Внимание! Если, расширяя класс адаптера, вы неправильно укажете имя метода, компилятор не сообщит об ошибке. Например, если в подклассе `WindowAdapter` вы создадите метод `windowIsClosing()`, то получите новый класс, содержащий восемь методов, причем метод `windowClosing()` не будет выполнять никаких действий.

Создание класса слушателя, расширяющего класс `WindowAdapter`, — несомненный шаг вперед, но можно достичь еще большего. Объекту слушателя не обязательно давать имя, достаточно написать следующий оператор:

```
frame.addWindowListener(new Terminator());
```

Но и это еще не все! Можно использовать в качестве обработчика анонимный вложенный класс фрейма:

```
frame.addWindowListener(new
   WindowAdapter()
   {
      public void windowClosing(WindowEvent e)
      {
        if(пользователь согласен)
             System.exit(0);
      }
   } );
```

В этом фрагменте кода выполняются описанные ниже действия.

- Определяется анонимный класс, расширяющий класс `WindowAdapter`.
- В этом анонимном классе переопределяется метод `windowClosing()`. Как и прежде, данный метод отвечает за выход из программы.
- Новый класс наследует от класса `WindowAdapter` шесть методов, не выполняющих никаких действий.
- Создается объект класса. Этот объект также не имеет имени.
- Анонимный объект передается методу `addWindowListener()`.

Еще раз подчеркнем — к использованию анонимного вложенного класса нужно привыкнуть. Наградой будет предельная краткость полученного кода.

API java.awt.event.WindowListener 1.1

- `void windowOpened(WindowEvent e)`
  Этот метод вызывается после открытия окна.
- `void windowClosing(WindowEvent e)`
  Этот метод вызывается, когда пользователь отдает оконному диспетчеру команду закрыть окно. Обратите внимание на то, что окно закрывается, только если вызван метод `hide()` или `dispose()`.
- `void windowClosed(WindowEvent e)`
  Этот метод вызывается после закрытия окна.
- `void windowIconified(WindowEvent e)`
  Этот метод вызывается после свертывания окна.
- `void windowDeiconified(WindowEvent e)`
  Этот метод вызывается после развертывания окна.

- `void windowActivated(WindowEvent e)`

  Этот метод вызывается после активизации окна. Активным может быть только фрейм или диалоговое окно. Обычно оконный диспетчер специально отмечает активное окно, например, подсвечивает его заголовок.

- `void windowDeactivated(WindowEvent e)`

  Этот метод вызывается после того, как окно стало неактивным.

API java.awt.event.WindowStateListener 1.4

- `void windowStateChanged(WindowEvent event)` **1.4**

  Этот метод вызывается после того, как окно было увеличено до максимальных размеров, свернуто или если были восстановлены его нормальные размеры.

API java.awt.event.WindowEvent 1.1

- `int getNewState()` **1.4**
- `int getOldState()` **1.4**

  Эти методы возвращают новое или старое состояние окна при изменении его состояния. Возвращаемое целое число может принимать одно из следующих значений:

```
Frame.NORMAL
Frame.ICONIFIED
Frame.MAXIMIZED_HORIZ
Frame.MAXIMIZED_VERT
Frame.MAXIMIZED_BOTH
```

Действия

Одну и ту же команду можно выполнить разными способами. Пользователь может выбрать пункт меню, нажать соответствующую клавишу или щелкнуть на кнопке панели инструментов. В этом случае очень удобна рассматриваемая нами модель событий AWT: достаточно связать все эти события с одним и тем же слушателем. Предположим, что `blueAction` — это слушатель действия, чей метод `actionPerformed` изменяет цвет фона на синий. Можно связать один и тот же объект с разными источниками событий, перечисленными ниже.

- Кнопка Blue (Синий) в панели инструментов.
- Пункт меню Blue.
- Нажатие клавиш <Ctrl+B>.

Команда, изменяющая цвет, выполняется одинаково, независимо от того, что привело к ее выполнению — щелчок на кнопке, выбор пункта меню или нажатие комбинации клавиш.

В пакете Swing предусмотрен очень полезный механизм, позволяющий инкапсулировать команды и связывать их с несколькими источниками событий. Этот механизм представляет собой интерфейс `Action`. *Действие* (action) — это объект, инкапсулирующий:

- описание команды (в виде текстовой строки или пиктограммы);
- параметры, необходимые для выполнения программы (например, нужный цвет).

Интерфейс Action содержит следующие методы:

```
Void actionPerformed(ActionEvent event)
void setEnabled(boolean b)
boolean isEnabled()
void pitValue(String key, Object value)
Object getValue(String key)
void addPropertyChangeListener(PropertyChangeListener listener)
void removePropertyChangeListener(PropertyChangeListener listener)
```

Первый метод похож на метод интерфейса ActionListener: действительно, интерфейс Action расширяет интерфейс ActionListener. Следовательно, вместо объекта класса, реализующего интерфейс ActionListener, можно использовать объект класса, реализующего интерфейс Action.

Следующие два метода позволяют блокировать и разблокировать действие, а также проверить, является ли указанное действие разблокированным (т.е. доступным) в данный момент. Если пункт меню или кнопка панели инструментов связаны с заблокированным действием, они выделяются светло-серым цветом.

Методы putValue() и getValue() позволяют записывать и извлекать из памяти произвольные пары "имя–значение" объектов класса, реализующего интерфейс Action. Пара предопределенных строк, таких как Action.NAME и Action.SMALL_ICON, содержат имена и пиктограммы объектов действий:

```
Action.putValue(Action.NAME, "Blue");
action.putValue(Action.SMALL_ICON,
   new ImageIcon("blue-ball.gif"));
```

В табл. 8.1 приведены предопределенные имена действий.

Таблица 8.1. Предопределенные имена действий

Имя	Значение
NAME	Имя действия. Отображается на кнопке и в названии пункта меню.
SMALL_ICON	Место для хранения пиктограммы, которая изображается на кнопке, в пункте меню или в панели инструментов.
SHORT_DESCRIPTION	Краткое описание пиктограммы, отображается в строке подсказки.
LONG_DESCRIPTION	Подробное описание пиктограммы. Может использоваться для подсказки. Не используется ни одним компонентом из библиотеки Swing.
MNEMONIC_KEY	Мнемоническое сокращение. Отображается в пункте меню (см. главу 9).
ACCELERATOR_KEY	Место для хранения комбинации клавиш. Не используется ни одним компонентом из библиотеки Swing.
ACTION_COMMAND_KEY	Ранее использовалось в теперь уже устаревшем методе registeredKeyBoardAction().
DEFAULT	Может быть полезным для хранения разнообразных объектов. Не используется ни одним компонентом из библиотеки Swing.

Если в меню или к панели инструментов добавляется какое-то действие, его имя и пиктограмма автоматически извлекаются из памяти и отображаются в меню и в панели. Значение SHORT_DESCRIPTION выводится в строке подсказки.

Последние два метода интерфейса Action позволяют уведомлять другие объекты, в частности, меню и панели инструментов, об изменении свойств действий.

Например, если меню создано как обработчик, предназначенный для отслеживания изменения свойств действия, и это действие впоследствии было заблокировано, то при отображении меню на экране соответствующее имя действия может быть подсвечено серым. Обработчики изменения свойств представляют собой часть модели JavaBeans. Подробно компоненты JavaBeans и их свойства будут рассматриваться во втором томе.

Обратите внимание на то, что `Action` является *интерфейсом*, а не классом. Любой класс, реализующий этот интерфейс, должен реализовать семь методов, которые мы только что рассмотрели. К счастью, это сделать достаточно легко, поскольку все они, кроме первого метода, содержатся в классе `AbstractAction`. Класс `AbstractAction` предназначен для хранения пар, состоящих из имен и значений, а также для управления обработчиками изменения свойств. Все, что нужно для этого сделать для практического использования, — это создать подкласс и добавить метод `actionPerformed()`.

Попробуем создать действие, изменяющее цвет фона. Поместим в память имя команды, соответствующую пиктограмму и желаемый цвет. Код цвета запишем в таблицу, состоящую из пар "имя–значение", предусмотренных классом `AbstractAction`. Ниже приводится класс `ColorAction`. Конструктор задает пары, состоящие из имен и значений, а метод `ActionPerformed()` изменяет цвет фона.

```
public class ColorAction extends AbstractAction
{
   public ColorAction(String name, Icon icon, Color c)
   {
      putValue(Action.NAME, name);
      putValue(Action.SMALL_ICON, icon);
      putValue("color", c);
      putValue(Action.SHORT_DESCRIPTION,
         "Set panel color to " + name.toLowerCase());
   }

   public void actionPerformed(ActionEvent event)
   {
      Color c = (Color) getValue("color");
      setBackground(c);
   }
}
```

В программе создаются три объекта этого класса:

```
Action blueAction = new ColorAction("Blue",
   new ImageIcon("blue-ball.gif"), Color.BLUE);
```

Теперь свяжем это действие с кнопкой. Сделать это можно с помощью конструктора `JButton`, получающего объект `Action` в качестве параметра:

```
JButton blueButton = new JButton(blueAction);
```

Этот конструктор считывает имя и пиктограмму действия, помещает короткое описание во всплывающую подсказку и регистрирует объект `Action` в качестве обработчика. Пиктограмма и строка подсказки показаны на рис. 8.6. Как мы увидим в следующей главе, точно так же легко можно добавлять действия в меню.

И, наконец, нужно связать объекты действия клавишами, чтобы они выполнялись, когда пользователь нажимает соответствующие клавиши. Для того чтобы ассоциировать действия с нажатием клавиш, сначала нужно создать объект класса `KeyStroke`. Это удобный класс, инкапсулирующий описание клавиш.

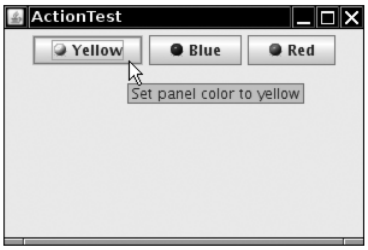


Рис. 8.6. Кнопки с пиктограммами из объектов действий

Чтобы создать объект этого класса, не нужно вызывать конструктор. Вместо него следует использовать статический метод `getKeyStroke()` класса `KeyStroke`.

```
KeyStroke ctrlBKey = KeyStroke.getKeyStroke("ctrl B");
```

Чтобы понять следующий шаг, нужно понимать концепцию *фокуса клавиатуры*. Пользовательский интерфейс может иметь много кнопок, меню, линеек прокрутки и других компонентов. Нажатия кнопок передаются компонентам, обладающим фокусом ввода. Этот компонент обычно (но не всегда) выделен визуально. Например, в визуальном стиле Java Look and Feel кнопка с фокусом имеет тонкую прямоугольную рамку вокруг текста. Вы можете использовать клавишу <Tab> для перемещения фокуса между компонентами. Когда вы нажимаете клавишу пробела, происходит щелчок на кнопке, обладающей фокусом в данный момент. Другие клавиши вызывают другие действия; например, клавиши со стрелками могут управлять линейками прокрутки.

Однако в нашем случае нам не нужно посылать нажатия клавиш компоненту, владеющему фокусом. Вместо этого каждая из наших кнопок должна обрабатывать события, связанные с нажатием клавиш, и реагировать на сочетания клавиш <Ctrl+Y>, <Ctrl+B> и <Ctrl+R>.

Это часто встречающаяся проблема, и проектировщики Swing предложили для нее удобное решение. Каждый объект класса `JComponent` содержит три *отображения ввода* (input map), связывающих объекты класса `KeyStroke` с действиями. Эти отображения соответствуют различным условиям (табл. 8.2).

При нажатии клавиши отображения ввода обрабатываются в следующем порядке.

1. Проверяется отображение `WHEN_FOCUSED` компонента, владеющего фокусом ввода. Если предусмотрена реакция на нажатие клавиши, выполняется соответствующее действие, если действие не заблокировано, процесс прекращается.
2. Начиная с компонента, содержащего фокус ввода, выполняется проверка условия `WHEN_ANCESTOR_OF_FOCUSED_COMPONENT` его родительского компонента. Как только условие будет выполнено, производится соответствующее действие. Если действие не заблокировано, процесс прекращается.
3. Проверяются все видимые и незаблокированные компоненты в окне, обладающем фокусом ввода. Компоненты должны быть зарегистрированы в условии `WHEN_IN_FOCUSED_WINDOW`. Каждый из этих компонентов (в порядке регистрации) получает возможность выполнить соответствующее действие. Когда будет выполнено первое незаблокированное действие, процесс прекратится. Если клавиша присутствует в нескольких отображениях `WHEN_IN_FOCUSED_WINDOW`, то результаты, полученные на данном этапе, могут быть неоднозначными.

Таблица 8.2. Условия, соответствующие трем отображениям ввода

Флаг	Условие
WHEN_FOCUSED	Когда данный компонент находится в фокусе клавиатуры.
WHEN_ANCESTOR_OF_FOCUSED_COMPONENT	Когда данный компонент содержит другой компонент, находящийся в фокусе клавиатуры.
WHEN_IN_FOCUSED_WINDOW	Когда данный компонент содержится в том же окне, что и компонент, находящийся в фокусе клавиатуры.

Отображение ввода извлекается из компонента с помощью метода getInputMap():

```
InputMap imap = panel.getInputMap(JComponent.WHEN_FOCUSED);
```

Условие WHEN_FOCUSED означает, что данное отображение проверяется, если компонент обладает фокусом ввода. В нашем случае это условие не проверяется, поскольку фокусом владеет одна кнопка, а не панель в целом. Каждое из оставшихся двух отображений также позволяет очень легко изменить цвет фона в ответ на нажатие клавиш. В программе мы будем использовать WHEN_ANCESTOR_OF_FOCUSED_COMPONENT.

Класс InputMap не связывает напрямую объекты класса KeyStroke с объектами класса ColorAction, реализующего интерфейс Action. Вместо этого объектам класса KeyStroke он ставит в соответствие произвольные объекты, а с помощью второго отображения, реализованного классом ActionMap, связывает их с действиями. Это облегчает связывание одного и того же действия с разными клавишами, зарегистрированными в разных отображениях ввода.

Итак, каждый компонент имеет три отображения ввода и одно отображение, связывающее нажатие клавиш с действиями. Чтобы объединить их, нужно присвоить каждому действию имя. Вот как это можно сделать:

```
imap.put(KeyStroke.getKeyStroke("ctrl Y"), "panel.yellow");
ActionMap amap = panel.getActionMap();
amap.put("panel.yellow", yellowAction);
```

Если нужно задать "пустое" действие, можно использовать строку "none". Это позволяет легко деактивизировать клавишу:

```
imap.put(KeyStroke.getKeyStroke("ctrl C", "none");
```



Внимание! В документации JDK предполагается, что название клавиш и соответствующего действия совпадают. Такое решение вряд ли можно назвать оптимальным. Имя действия отображается на кнопке и в пункте меню. Оно может изменяться в процессе разработки, в частности, это неизбежно при адаптации пользовательского интерфейса для разных языков. Мы рекомендуем делать имена действий независимыми от имен, отображаемых на экране.

Итак, чтобы одна и та же операция выполнялась в ответ на щелчок на кнопке, выбор пункта меню или нажатие клавиши, потребуется выполнить описанные ниже шаги.

1. Реализовать класс, расширяющий класс AbstractAction. Один класс можно будет использовать для программирования разных взаимосвязанных действий.
2. Создать объект класса действия.
3. Создать кнопку или пункт меню на основе объекта действия. Конструктор прочтет текст метки и иконку из объекта действия.

4. Для действий, которые выполняются в ответ на нажатие клавиш, нужно сделать один дополнительный шаг. Сначала следует идентифицировать компонент окна верхнего уровня, например, панель, содержащую все остальные компоненты.
5. Проверить отображение `WHEN_ANCESTOR_OF_FOCUSED_COMPONENT` компонента верхнего уровня. Создать объект класса `KeyStroke` для нужного нажатия клавиш. Создать объект, соответствующий нажатию клавиш, например, строку, описывающую нужное действие. Добавить пару (нажатие клавиши, действие) к отображению ввода.
6. Наконец, получить информацию о соответствии действий и клавиш для компонента верхнего уровня. Добавить пару (действие, объект действия) в отображение.

В листинге 8.3 приведен полный текст программы, задающей соответствие между кнопками, нажатием клавиш и действиями. Действия выполняются в ответ на активизацию кнопок или нажатие клавиш <Ctrl+Y>, <Ctrl+B> или <Ctrl+R>.

Листинг 8.3. Содержимое файла `ActionTest.java`

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * @version 1.33 2007-06-12
7.  * @author Cay Horstmann
8.  */
9. public class ActionTest
10. {
11.    public static void main(String[] args)
12.    {
13.       EventQueue.invokeLater(new Runnable()
14.          {
15.             public void run()
16.             {
17.                ActionFrame frame = new ActionFrame();
18.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.                frame.setVisible(true);
20.             }
21.          });
22.    }
23. }
24.
25. /**
26.  * Фрейм с панелью, демонстрирующей действия изменения цвета.
27.  */
28. class ActionFrame extends JFrame
29. {
30.    public ActionFrame()
31.    {
32.       setTitle("ActionTest");
33.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34.
35.       buttonPanel = new JPanel();
36.
37.       // Определение действий.
38.       Action yellowAction =
39.  new ColorAction("Yellow", new ImageIcon("yellow-ball.gif"),Color.YELLOW);
```

```
40.     Action blueAction =
41.  new ColorAction("Blue", new ImageIcon("blue-ball.gif"), Color.BLUE);
42.     Action redAction =
43.  new ColorAction("Red", new ImageIcon("red-ball.gif"), Color.RED);
44.
45.     // Добавление кнопок для действий.
46.     buttonPanel.add(new JButton(yellowAction));
47.     buttonPanel.add(new JButton(blueAction));
48.     buttonPanel.add(new JButton(redAction));
49.
50.     // Добавление панели к фрейму.
51.     add(buttonPanel);
52.
53.     // Связывание клавиш Y, B и R с именами.
54.     InputMap imap =
55.  buttonPanel.getInputMap(JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT);
56.     imap.put(KeyStroke.getKeyStroke("ctrl Y"), "panel.yellow");
57.     imap.put(KeyStroke.getKeyStroke("ctrl B"), "panel.blue");
58.     imap.put(KeyStroke.getKeyStroke("ctrl R"), "panel.red");
59.
60.     // Связывание имен с действиями.
61.     ActionMap amap = buttonPanel.getActionMap();
62.     amap.put("panel.yellow", yellowAction);
63.     amap.put("panel.blue", blueAction);
64.     amap.put("panel.red", redAction);
65.   }
66.
67.   public class ColorAction extends AbstractAction
68.   {
69.     /**
70.      * Создание действия для изменения цвета.
71.      * @param name Надпись на кнопке
72.      * @param icon Пиктограмма на кнопке
73.      * @param c Цвет фона
74.      */
75.     public ColorAction(String name, Icon icon, Color c)
76.     {
77.       putValue(Action.NAME, name);
78.       putValue(Action.SMALL_ICON, icon);
79.       putValue(Action.SHORT_DESCRIPTION, "Set panel color to "
80.                + name.toLowerCase());
81.       putValue("color", c);
82.     }
83.
84.     public void actionPerformed(ActionEvent event)
85.     {
86.       Color c = (Color) getValue("color");
87.       buttonPanel.setBackground(c);
88.     }
89.   }
90.
91.   private JPanel buttonPanel;
92.
93.   public static final int DEFAULT_WIDTH = 300;
94.   public static final int DEFAULT_HEIGHT = 200;
95. }
```

API javax.swing.Action 1.2

- `boolean isEnabled()`
- `void setEnabled(boolean b)`

  Получает или устанавливает свойство `enabled` действия.

- `void putValue(String key, Object value)`

  Помещает пару (имя, значение) в объект действия.

  Параметры `key` Имя, которое должно сохраняться в объекте действия. Оно может быть строкой, но несколько имен имеют специальное значение.

  `value` Объект, связанный с именем.

- `Object getValue(String key)`

  Возвращает значение из сохраненной пары (имя, значение).

API javax.swing.KeyStroke 1.2

- `static KeyStroke getKeyStroke(String description)`

  Создает объект `KeyStroke` из читабельного для человека описания (последовательности строк с пробельными разделителями). Описание начинается с нуля или более модификаторов `shift control ctrl meta alt altGraf` и заканчивается либо строкой `typed`, за которой следует односимвольная строка (например, `"typed a"`), либо необязательный спецификатор события (по умолчанию — `pressed` или `released`), за которым идет код клавиши. Код клавиши, снабженный префиксом `VK_`, должен соответствовать константе `KeyEvent`; например, `"INSERT"` соответствует `KeyEvent.VK_INSERT`.

API javax.swing.JComponent 1.2

- `ActionMap getActionMap()` **1.3**

  Возвращает отображения клавиш в действия.

- `InputMap getInputMap(int flag)` **1.3**

  Получает отображение ввода, которое переводит нажатия клавиш в клавиши действий.

  Параметр `flag` Одно из значений, представленных в табл. 8.4.

События мыши

Если пользователь должен только щелкать на кнопках или выбирать пункты меню, обрабатывать явным образом события, связанные с мышью, не нужно. Эти операции автоматически поддерживаются компонентами пользовательского интерфейса, а затем преобразовываются в соответствующие семантические события. Однако если пользователь должен иметь возможность рисовать с помощью мыши, необходимо отслеживать ее перемещения, щелчки и события, связанные с перетаскиванием объектов.

В данном разделе мы рассмотрим простой графический редактор, позволяющий создавать, перемещать и стирать квадраты на холсте (рис. 8.7).

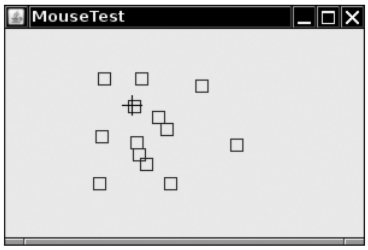


Рис. 8.7. Программа, демонстрирующая обработку событий, связанных с мышью

Когда пользователь щелкает мышью, вызываются три метода объекта-обработчика: `mousePressed()`, если кнопка мыши была нажата, `mouseReleased()`, если кнопка была отпущена, и `mouseClicked`. Если надо отследить только сам щелчок, первые два метода использовать не обязательно. Пользуясь методами `getX()` и `getY()` объекта `MouseEvent`, передаваемого в качестве параметра, можно определить координаты курсора мыши в момент щелчка. Если нужно различать обычный, двойной и тройной (!) щелчки, используется метод `getClickCount()`.

Некоторые разработчики так реализуют свое программное обеспечение, что для управления интерфейсными элементами пользователю приходится щелкать кнопкой мыши при нажатых модифицирующих клавишах, например <Control+Shift+щелчок>. Мы находим такую практику нецелесообразной, но если вы не согласны, то обнаружите, что проверка кнопок мыши с клавиатурными модификаторами довольно запутана. Для проверки установленных модификаторов используются битовые маски. В исходной библиотеке API маски, соответствующие двум кнопкам мыши, совпадали с масками клавиш-модификаторов:

```
BUTTON2_MASK == ALT_MASK
BUTTON3_MASK == META_MASK
```

Это было сделано для того, чтобы пользователи однокнопочных мышей могли имитировать нажатие других кнопок, удерживая модифицирующие клавиши на клавиатуре. Однако в Java SE 1.4 использован другой подход. Теперь в нем предусмотрены новые маски:

```
BUTTON1_DOWN_MASK
BUTTON2_DOWN_MASK
BUTTON3_DOWN_MASK
SHIFT_DOWN_MASK
CTRL_DOWN_MASK
ALT_DOWN_MASK
ALT_GRAPH_DOWN_MASK
META_DOWN_MASK
```

Метод `getModifiersEx()` позволяет распознавать события, связанные с нажатием кнопок мыши при нажатых клавишах.

Обратите внимание, что маска `BUTTON3_DOWN_MASK` позволяет проверять нажатие правой кнопки мыши в среде Windows. Например, для определения, нажата ли правая кнопка, можно воспользоваться следующим кодом:

```
if ((event.getModifiersEx() & InputEvent.BUTTON3_DOWN_MASK) != 0)
  // Код, обрабатывающий нажатие правой кнопки мыши
```

В программе, которая будет рассмотрена ниже, используются методы `mousePressed()` и `mouseClicked()`. Если щелкнуть кнопкой мыши на пикселе, не принадлежащем ни одному из нарисованных квадратов, на экране появится новый квадрат. Эта процедура реализована в методе `mousePressed()`, поэтому компьютер отреагирует на щелчок мыши немедленно, не дожидаясь освобождения кнопки мыши. Двойной щелчок внутри какого-нибудь квадрата приведет к тому, что он будет стерт. Эта процедура реализована в методе `mouseClicked()`, поскольку щелчки мыши нужно подсчитывать.

```
public void mousePressed(MouseEvent event)
{
   current = find(event.getPoint());
   if (current == null) // Вне квадрата.
      add(event.getPoint());
}

public void mouseClicked(MouseEvent event)
{
   current = find(event.getPoint());
   if (current != null && event.getClickCount() >= 2)
      remove(current);
}
```

При перемещении мыши по окну последнее получает постоянный поток событий, связанных с движением мыши. В большинстве приложений эти события игнорируются. Однако в нашем примере программа отслеживает события, чтобы изменить вид курсора (на крестообразный), как только он выйдет за пределы квадрата. Эта процедура реализована в методе `getPredefinedCursor()` в классе `Cursor`. В табл. 8.3 приведены константы, используемые в этом методе, и показаны виды курсоров, как они выглядят в среде Windows.

Таблица 8.3. Виды курсоров

Пиктограмма	Константа	Пиктограмма	Константа
	DEFAULT_CURSOR		NE_RESIZE_CURSOR
	CROSSHAIR_CURSOR		E_RESIZE_CURSOR
	HAND_CURSOR		SE_RESIZE_CURSOR
	MOVE_CURSOR		S_RESIZE_CURSOR
	TEXT_CURSOR		SW_RESIZE_CURSOR
	WAIT_CURSOR		W_RESIZE_CURSOR
	N_RESIZE_CURSOR		NW_RESIZE_CURSOR

Рассмотрим метод `mouseMoved()`, объявленный в интерфейсе `MouseMotionListener` и реализованный в нашей программе.

```
public void mouseMoved(MouseEvent event)
{
   if (find(event.getPoint()) == null)
      setCursor(Cursor.getDefaultCursor());
```

```
   else
      setCursor(Cursor.getPredefinedCursor
        (Cursor.CROSSHAIR_CURSOR));
}
```



На заметку! Используя метод `createCustomCursor()` из класса `Toolkit`, можно определить новую форму курсора.

```
Toolkit tk = Toolkit.getDefaultToolkit();
Image img = tk.getImage("dynamite.gif");
Cursor dynamiteCursor = tk.createCustomCursor(img,
   new Point(10, 10), "dynamite stick");
```

Первый параметр метода `createCustomCursor()` задает изображение курсора. Второй параметр определяет координаты точки, используемой в качестве указателя. Третий параметр — строка, описывающая курсор. Эту строку можно применять для вспомогательных целей, сообщить голосом вид курсора. Такая возможность важна при адаптации программ для пользователей с нарушениями зрения.

Если перемещение мыши осуществляется при нажатой кнопке, вместо `mouseClicked()` вызывается метод `mouseDragged()`. В нашем примере квадраты можно перетаскивать по экрану. При этом квадрат перемещается так, чтобы его центр располагался в той точке, на которую указывает мышь. Содержимое холста перерисовывается.

```
public void mouseDragged(MouseEvent event)
{
   if (cursor >= null)
   {
      int x = event.getX();
      int y = event.getY();

      current.setFrame(
        x - SIDELENGTH / 2,
        y - SIDELENGTH / 2,
        SIDELENGTH,
        SIDELENGTH);
      repaint();
   }
}
```



На заметку! Метод `mouseMoved()` вызывается, только если указатель мыши находится внутри компонента. Однако метод `mouseDragged()` вызывается даже тогда, когда указатель мыши пребывает вне компонента.

Есть еще два метода, обрабатывающих события, связанные с мышью: `mouseEntered()` и `mouseExited()`. Эти методы вызываются, когда указатель мыши входит в пределы компонента и выходит из него.

Наконец, мы объясним, как прослушивать события мыши. В ответ на щелчок мышью вызывается метод `mouseClicked()`, являющийся частью интерфейса `MouseListener`. Поскольку во многих приложениях отслеживаются только щелчки мышью, а также потому, что перемещения мыши происходят слишком часто, события, связанные с перемещением мыши и перетаскиванием, определяются в отдельном интерфейсе `MouseMotionListener`.

В нашей программе нас интересуют оба типа событий, связанных с мышью. В ней определены два вложенных класса: `MouseHandler` и `MouseMotionHandler`. Класс `MouseHandler` расширяет класс `MouseAdapter`, поскольку он определяет только два из пяти методов класса `MouseListener`. Класс `MouseMotionHandler` реализует интерфейс `MouseMotionListener` и определяет оба его метода. Текст программы приведен в листинге 8.4.

Листинг 8.4. Содержимое файла `MouseTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.util.*;
4.  import java.awt.geom.*;
5.  import javax.swing.*;
6.
7.  /**
8.   * @version 1.32 2007-06-12
9.   * @author Cay Horstmann
10.  */
11. public class MouseTest
12. {
13.    public static void main(String[] args)
14.    {
15.       EventQueue.invokeLater(new Runnable()
16.          {
17.             public void run()
18.             {
19.                MouseFrame frame = new MouseFrame();
20.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21.                frame.setVisible(true);
22.             }
23.          });
24.    }
25. }
26.
27. /**
28.  * Фрейм, содержащий панель, предназначенную для проверки операций с мышью.
29.  */
30. class MouseFrame extends JFrame
31. {
32.    public MouseFrame()
33.    {
34.       setTitle("MouseTest");
35.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
36.
37.       // Добавление компонентов к фрейму.
38.
39.       MouseComponent component = new MouseComponent();
40.       add(component);
41.    }
42.
43.    public static final int DEFAULT_WIDTH = 300;
44.    public static final int DEFAULT_HEIGHT = 200;
45. }
46.
47. /**
48.  * Компонент, поддерживающий добавление и удаление квадратов с помощью мыши.
49.  */
```

```
50. class MouseComponent extends JComponent
51. {
52.    public MouseComponent()
53.    {
54.       squares = new ArrayList<Rectangle2D>();
55.       current = null;
56.
57.       addMouseListener(new MouseHandler());
58.       addMouseMotionListener(new MouseMotionHandler());
59.    }
60.
61.    public void paintComponent(Graphics g)
62.    {
63.       Graphics2D g2 = (Graphics2D) g;
64.
65.       // Рисование всех квадратов.
66.       for (Rectangle2D r : squares)
67.       g2.draw(r);
68.    }
69.
70. /**
71.  * Нахождение первого квадрата, который содержит заданную точку.
72.  * @param p Точка
73.  * @return Первый квадрат, содержащий точку p
74.  */
75. public Rectangle2D find(Point2D p)
76. {
77.    for (Rectangle2D r : squares)
78.    {
79.       if (r.contains(p)) return r;
80.    }
81.    return null;
82. }
83.
84. /**
85.  * Добавление квадрата к набору.
86.  * @param p Центр квадрата
87.  */
88. public void add(Point2D p)
89. {
90.    double x = p.getX();
91.    double y = p.getY();
92.
93.    current = new Rectangle2D.Double(x - SIDELENGTH / 2,
94.          y - SIDELENGTH / 2, SIDELENGTH, SIDELENGTH);
95.    squares.add(current);
96.    repaint();
97. }
98.
99. /**
100.  * Удаление квадрата из набора.
101.  * @param s Удаляемый квадрат
102.  */
103. public void remove(Rectangle2D s)
104. {
105.    if (s == null) return;
```

```
106.    if (s == current) current = null;
107.    squares.remove(s);
108.    repaint();
109. }
110.
111. private static final int SIDELENGTH = 10;
112. private ArrayList<Rectangle2D> squares;
113. private Rectangle2D current;
114.
115. // Квадрат, в пределах которого находится курсор.
116.
117. private class MouseHandler extends MouseAdapter
118. {
119.    public void mousePressed(MouseEvent event)
120.    {
121.       // Добавить новый квадрат, если курсор находится
122.       // за пределами всех квадратов
123.       current = find(event.getPoint());
124.       if (current == null) add(event.getPoint());
125.    }
126.
127.    public void mouseClicked(MouseEvent event)
128.    {
129.       // После двойного щелчка текущий квадрат удаляется.
130.       current = find(event.getPoint());
131.       if (current != null && event.getClickCount() >= 2) remove(current);
132.    }
133. }
134.
135. private class MouseMotionHandler implements MouseMotionListener
136. {
137.    public void mouseMoved(MouseEvent event)
138.    {
139.       // Если курсор находится в пределах квадрата,
140.       // он принимает крестообразный вид.
141.
142.       if (find(event.getPoint()) == null) setCursor(Cursor.getDefaultCursor());
143.       else setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
144. }
145.
146.       public void mouseDragged(MouseEvent event)
147.       {
148.          if (current != null)
149.          {
150.             int x = event.getX();
151.             int y = event.getY();
152.
153.             // Перетаскивание текущего квадрата в точку (x, y)
154.             current.setFrame(x - SIDELENGTH / 2, y - SIDELENGTH / 2,
155.                              SIDELENGTH, SIDELENGTH);
156.             repaint();
157.          }
158.       }
159.    }
160. }
```

API java.awt.event.MouseEvent 1.1

- `int getX()`
- `int getY()`
- `Point getPoint()`

  Возвращает горизонтальную (*x*) и вертикальную (*y*) координаты или точку, соответствующую событию. Отсчет производится от левого верхнего угла компонента, являющегося источником события.

- `int getClickCount()`

  Возвращает количество последовательных щелчков мыши, связанных с данным событием. (Интервал времени, в пределах которого подсчитываются щелчки, зависит от операционной системы.)

API java.awt.event.InputEvent 1.1

- `int getModifiersEx()` **1.4**

  Возвращает модификаторы для события. При проверке возвращаемого значения используются перечисленные ниже маски.

  ```
  BUTTON1_DOWN_MASK
  BUTTON2_DOWN_MASK
  BUTTON3_DOWN_MASK
  SHIFT_DOWN_MASK
  CTRL_DOWN_MASK
  ALT_DOWN_MASK
  ALT_GRAPH_DOWN_MASK
  META_DOWN_MASK
  ```

- `static int getModifiersExText(int modifiers)` **1.4**

Возвращает строку, описывающую модификаторы, например `Shift+Button1`.

API java.awt.Toolkit 1.0

- `public Cursor createCustomCursor(Image image, Point hotSpot, String name)` **1.2**

  Создает новый объект курсора.

  Параметры `image` Изображение активного курсора.
  `hotSpot` Точка, считающаяся указателем.
  `name` Описание курсора.

API java.awt.Component 1.0

- `public void setCursor(Cursor cursor)` **1.1**

  Изменяет внешний вид курсора.

Иерархия событий AWT

После обсуждения механизма обработки событий завершим эту главу общим обзором архитектуры обработки событий AWT.

Как мы уже вкратце упоминали ранее, в языке Java при обработке событий используется объектно-ориентированный подход. Все события являются потомками класса `EventObject` из пакета `java.util`. (Общий суперкласс не назвали именем `Event`, так как это имя носил класс событий в старой модели. Хотя старая модель в настоящее время не рекомендована к использованию, ее классы все еще являются частью библиотеки Java.)

Класс `EventObject` имеет подкласс `AWTEvent`, являющийся родительским по отношению ко всем классам событий из пакета AWT. На рис. 8.8 показана диаграмма наследования событий из пакета AWT.

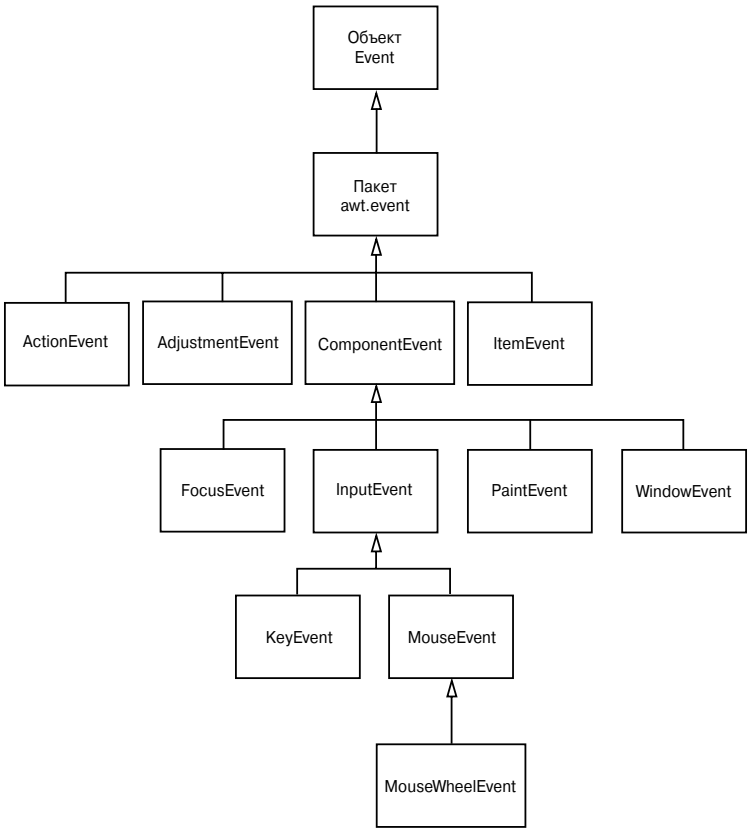


Рис. 8.8. Диаграмма наследования классов событий AWT

Некоторые компоненты пакета Swing генерируют другие объекты событий и непосредственно расширяют класс `EventObject`.

Объекты событий инкапсулируют информацию, которую источник события передает слушателям. По мере необходимости объекты событий можно проанализировать с помощью методов `getSource()` и `getActionSource()`.

Некоторые из классов событий AWT не имеют практической ценности для программиста на Java. Например, пакет AWT помещает объекты `PaintEvent` в очередь событий,

однако эти объекты не доставляются слушателю. Программисты должны сами переопределять метод `paintComponent()`, чтобы управлять перерисовкой. AWT также генерирует ряд событий, интересующих лишь системных программистов; они могут использоваться для поддержки иероглифических языков, автоматизации тестирования роботов и выполнения других самых разнообразных задач. В этой книге мы не будем обсуждать эти специализированные типы событий.

Семантические и низкоуровневые события в библиотеке AWT

В AWT события разделены на *низкоуровневые* (low-level) и *семантические* (semantic). Семантические события описывают действия пользователя, например, щелчок на кнопке; следовательно, событие `ActionEvent` является семантическим. Низкоуровневые события обеспечивают возможность таких действий. Если пользователь щелкнул на кнопке, значит, он нажал кнопку мыши, возможно, переместил курсор по экрану и отпустил кнопку мыши (причем курсор мыши должен находиться в пределах кнопки). Семантические события могут быть также сгенерированы путем нажатия клавиш, например для перемещения по кнопкам с помощью клавиши <Tab>. Аналогично, действие с полосой прокрутки относится к семантическим событиям, в то время как перетаскивание объекта с помощью мыши относится к низкоуровневым.

Из классов пакета `java.awt.event`, описывающих семантические события, наиболее часто используются следующие.

- `ActionEvent` (щелчок на кнопке, выбор пункта меню, выбор пункта в списке, нажатие клавиши <Enter> при работе с полм редактирования).
- `AdjustmentEvent` (перемещение ползунка на полосе прокрутки).
- `ItemEvent` (выбор положения переключателя опций или пункта в списке).

Из низкоуровневых событий наиболее часто встречаются перечисленные ниже.

- `KeyEvent` (нажатие или отпускание клавиши).
- `MouseEvent` (нажатие и отпускание кнопки мыши, перемещение курсора мыши, перетаскивание курсора, т.е. его перемещение при нажатой кнопке).
- `MouseWheelEvent` (вращение колесика мыши).
- `FocusEvent` (получение или потеря фокуса ввода.).
- `WindowEvent` (изменение состояния окна).

Прослушиваются эти события с помощью следующих интерфейсов:

```
ActionListener          MouseMotionListener
AdjustmentListener      MouseWheelListener
FocusListener           WindowListener
ItemListener            WindowFocusListener
KeyListener             WindowStateListener
MouseListener
```

Некоторые из интерфейсов слушателей AWT, а именно те, у которых более одного метода, сопровождаются классом адаптера, реализующего все его методы как пустые и ничего не делающие. (Другие интерфейсы имеют лишь по одному методу, так что никакой выгоды от класса адаптера для этих интерфейсов нет.) Ниже перечислены наиболее часто используемые классы адаптеров.

```
FocusAdapter        MouseMotionAdapter
KeyAdapter          WindowAdapter
MouseAdapter
```

В табл. 8.4 приведены наиболее важные интерфейсы слушателей событий AWT, события и источники событий. Пакет `java.swing.event` содержит дополнительные события, специфичные для компонентов Swing. Некоторые из них мы рассмотрим в следующей главе.

Таблица 8.4. Объекты и интерфейсы, участвующие в обработке событий

Интерфейс	Методы	Параметр/ методы доступа	Источник события
ActionListener	actionPerformed	ActionEvent getActionCommand getModifiers	AbstractButton JComboBox JTextField Timer
AdjustmentListener	adjustmentValueChanged	AdjustmentEvent getAdjustable getAdjustmentType getValue	JScrollbar
ItemListener	itemStateChanged	ItemEvent getItem getItemSelectable getStateChange	AbstractButton JComboBox
FocusListener	focusGained focusLost	FocusEvent isTemporary	Component
KeyListener	keyPressed keyReleased keyTyped	KeyEvent getKeyChar getKeyCode getKeyModifiersText getKeyText isActionKey	Component
MouseListener	mousePressed mouseReleased mouseEntered mouseExited mouseClicked	MouseEvent getClickCount getX getY getPoint translatePoint	Component
MouseMotionListener	mouseDragged mouseMoved	MouseEvent	Component
MouseWheelListener	mouseWheelMoved	MouseWheelEvent getWheelRotation getScrollAmount	Component
WindowListener	windowClosing windowOpened windowIconified windowDeiconified windowClosed windowActivated windowDeactivated	WindowEvent getWindow	Window
WindowFocusListener	windowGainedFocus windowLostFocus	WindowEvent getOppositeWindow	Window
WindowStateListener	windowStateChanged	WindowEvent getOldState getNewState	Window

На этом обсуждение обработки событий AWT завершено. В следующей главе мы покажем, как собрать вместе наиболее часто используемые компоненты, представленные в Swing, а также детально рассмотрим генерируемые ими события.

Глава 9

КОМПОНЕНТЫ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА SWING

В ЭТОЙ ГЛАВЕ...

- Swing и шаблон проектирования "Модель–представление–контроллер"
- Общие сведения о компоновке элементов
- Ввод текста
- Компоненты для выбора вариантов
- Меню
- Расширенные средства компоновки
- Диалоговые окна

Предыдущая глава была в основном посвящена рассмотрению модели обработки событий Java. Прочитав ее, вы получили сведения, без которых немыслимо создать приложение с графическим интерфейсом. В этой главе мы обсудим самые важные инструменты, позволяющие создавать полнофункциональные пользовательские интерфейсы.

Сначала мы вкратце рассмотрим архитектурные принципы, лежащие в основе пакета Swing. Если вы хотите эффективно использовать современные компоненты пользовательского интерфейса, необходимо разбираться в том, как они функционируют. Затем мы покажем вам, как применять обычные компоненты пользовательского интерфейса из пакета Swing, например, поля редактирования, переключатели и меню. Вы узнаете, как использовать возможности диспетчеров компоновки Java для того, чтобы разместить компоненты в окне, независимо от его конкретных особенностей. В заключение мы покажем, как с помощью средств Swing создаются диалоговые окна.

В этой главе описываются основные составляющие пакета Swing, такие как текстовые компоненты, кнопки и полосы прокрутки. Все это наиболее важные компоненты пользовательского интерфейса, которые вам понадобятся чаще всего. Более сложные компоненты будут описаны во втором томе.

Swing и шаблон проектирования “Модель–представление–контроллер”

Начнем эту главу с раздела, в котором опишем архитектуру компонентов пакета Swing. Сначала мы обсудим концепцию *шаблонов проектирования* (design patterns), а затем обратимся к шаблону “Модель–представление–контроллер”, который оказал огромное влияние на дизайн каркаса Swing.

Шаблоны проектирования

Решая задачу, большинство людей руководствуются своим опытом или спрашивают совета у других. Шаблоны проектирования — это способ представления опыта в структурированном виде.

В последние годы разработчики программного обеспечения стали собирать каталоги таких схем. Пионеров этой области вдохновили схемы проектирования, созданные архитектором Кристофером Александером (Christopher Alexander). В своей книге *Timeless Way of Building* (Oxford University Press, 1979) он составил каталог схем для проектирования общественных и частных жилых помещений. Вот типичный пример.

Окна

Каждый любит посидеть в удобном кресле на балконе, у эркера или большого окна с низким подоконником... В комнате, где нет такого места, редко чувствуешь себя удобно и спокойно...

В такой комнате человек хочет удовлетворить два желания:

- сидеть удобно;
- сидеть лицом к свету.

Разумеется, если удобные места, т.е. места, где вам больше всего хотелось бы сесть, находятся далеко от окна, то разрешить этот конфликт невозможно.

Следовательно, в каждой комнате, где человек проводит продолжительное время, должно быть, по крайней мере, одно окно в подходящем для этого месте.

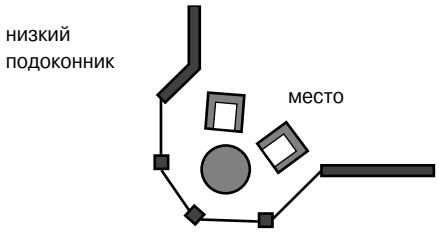


Рис. 9.1. Место у окна

Каждая схема в каталоге Александера, как и в каталоге схем программного обеспечения, следует определенному формату. Сначала описывается контекст, т.е. ситуация, которая порождает проблему. Затем ставится задача, обычно в виде конфликтующих между собой ограничений. В заключение предлагается ее решение, в котором достигается компромисс между противоречивыми требованиями.

В приведенном выше примере контекстом является комната, в которой человек проводит значительную часть дня. Конфликтующие ограничения заключаются в его желании сидеть удобно и при этом лицом к свету. Решение заключается в создании "места у окна".

В архитектуре "Модель–представление–контроллер" контекстом является система пользовательского интерфейса, которая отображает информацию и получает данные. В ней существует несколько противоречий. К ним могут относиться разные визуальные представления одних и тех же данных, которые должны одновременно обновляться. Визуальное представление может изменяться в зависимости от стиля интерфейса. Механизмы взаимодействия также могут изменяться, например, для того, чтобы поддерживать выполнение команд, которые вводятся голосом. Решение этой задачи заключается в распределении ответственности между тремя разными взаимодействующими компонентами: моделью, представлением и контроллером.

Шаблон "Модель–представление–контроллер" — не единственный шаблон проектирования, применяемый в AWT и Swing. Вот несколько дополнительных примеров.

- Контейнеры и компоненты — примеры шаблона "Композиции".
- Прокручиваемая панель — "Декоратор".
- Диспетчеры компоновки следуют шаблону "Стратегия".

Важным аспектом шаблонов проектирования является то, что они стали частью общей культуры программирования. Программисты в любой части света знают, что такое "Модель–представление–контроллер", и что собой представляет шаблон "Декоратор". Таким образом, шаблоны стали эффективным способом описания проблем проектирования программного обеспечения.

Формальное описание многочисленных полезных программных шаблонов вы найдете в академической книге Эриха Гаммы (Erich Gamma) *Design Patterns–Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995). Кроме того, мы рекомендуем блестящую книгу Франка Бушмана (Frank Buschmann) *A System of Patterns* (John Wiley & Sons, 1996), которую находим менее академичной и более доступной.

Шаблон "Модель-представление-контроллер"

Теперь вернемся назад и задумаемся о том, из чего состоят компоненты графического пользовательского интерфейса, например кнопка, флажок опции, поле редактирования или сложное окно управления каталогами.

Каждый компонент имеет следующие характеристики.

- *Содержимое*, например состояние кнопки (нажата или нет) или текст в поле редактирования.
- *Внешний вид* (цвет, размер и т.д.).
- *Поведение* (реакция на события).

Даже в таких, казалось бы, простых компонентах, как кнопки, эти характеристики тесно связаны между собой. Очевидно, внешний вид кнопки зависит от стиля интерфейса в целом. Кнопка в стиле Metal отличается от кнопки в стиле Windows или Motif. Кроме того, внешний вид зависит от состояния кнопки: нажатая кнопка должна выглядеть иначе, чем обычная. Состояние, в свою очередь, зависит от событий. Если пользователь щелкнул на кнопке, она считается нажатой.

Разумеется, когда вы используете кнопку в ваших программах, вы просто рассматриваете ее как кнопку, не вдаваясь в подробности ее внутреннего устройства и характеристик. В конце концов, технические детали — удел программиста, реализовавшего эту кнопку. Однако программисты, реализующие кнопки, должны тщательно продумывать их устройство и работу. В конце концов, им нужно реализовать кнопки и все прочие компоненты пользовательского интерфейса так, чтобы они правильно работали независимо от выбранного визуального стиля.

Для того чтобы помочь в решении подобных задач, разработчики пакета Swing обратились к хорошо известному шаблону *"Модель–представление–контроллер"* (model-view-controller — MVC). В основе этого шаблона лежат объектные принципы, описанные в главе 5, один из которых гласит: не перегружать объект слишком большим количеством задач. Класс, предусмотренный для кнопки, не обязан делать сразу все. Поэтому стиль компонента связывается с одним объектом, а его содержимое — с другим. Шаблон проектирования "Модель–представление–контроллер" позволяет достичь этой цели. Следуя ей, нужно реализовать три отдельных класса.

- Класс *модели*, в котором хранится содержимое.
- Класс *представления*, который отображает содержимое.
- Класс *контроллера*, управляющий вводом данных.

Шаблон "Модель–представление–контроллер" точно специфицирует взаимодействие между этими классами. Модель хранит содержимое и не реализует *пользовательский интерфейс*. Содержимое кнопки тривиально — это небольшой набор флагов, означающих, нажата кнопка или нет, активна или нет и т.д. Содержимое поля редактирования — это строка. Она не совпадает с *представлением* — если содержимое превышает длину текстового поля, пользователь видит лишь часть отображаемого текста (рис. 9.2).

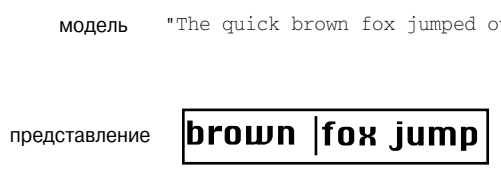


Рис. 9.2. Модель и представление текстового поля

Модель должна реализовывать методы, изменяющие содержимое и раскрывающие его смысл. Например, модель поля редактирования имеет методы для добавления

символов к строке, удаления их и возврата текста в виде строки. Учтите, что модель полностью невидима. Изображать на экране данные, хранящиеся в модели, — задача *представления*.



На заметку! Термин "модель", видимо, не совсем удачен, поскольку его часто ассоциируют со способом представления абстрактного понятия. Например, авиаконструкторы и конструкторы автомобилей строят модели для того, чтобы имитировать реальные самолеты и автомобили. Однако эта аналогия не применима к архитектуре "Модель–представление–контроллер". В этом случае модель хранит содержимое, а представление отвечает за ее полное или неполное визуальное изображение. Гораздо более точной аналогией является натурщица, позирующая художнику. Художник должен смотреть на модель и создавать ее изображение. В зависимости от стиля, в котором работает художник, представление может оказаться классическим портретом, картиной в стиле импрессионизма или набором фигур в стиле кубизма.

Одно из преимуществ шаблона MVC состоит в том, что модель может иметь несколько представлений, каждое из которых отражает отдельный аспект ее содержимого. Например, редактор HTML-кода часто предлагает одновременно два представления одних и тех же данных: WYSIWYG (What You See Is What You Get) и набор дескрипторов (рис. 9.3). Когда контроллер обновляет модель, изменяются оба представления. Получив уведомление об изменении, представление обновляется автоматически. Разумеется, для таких простых компонентов пользовательского интерфейса, как кнопки, нет никакой необходимости предусматривать несколько представлений одной и той же модели.

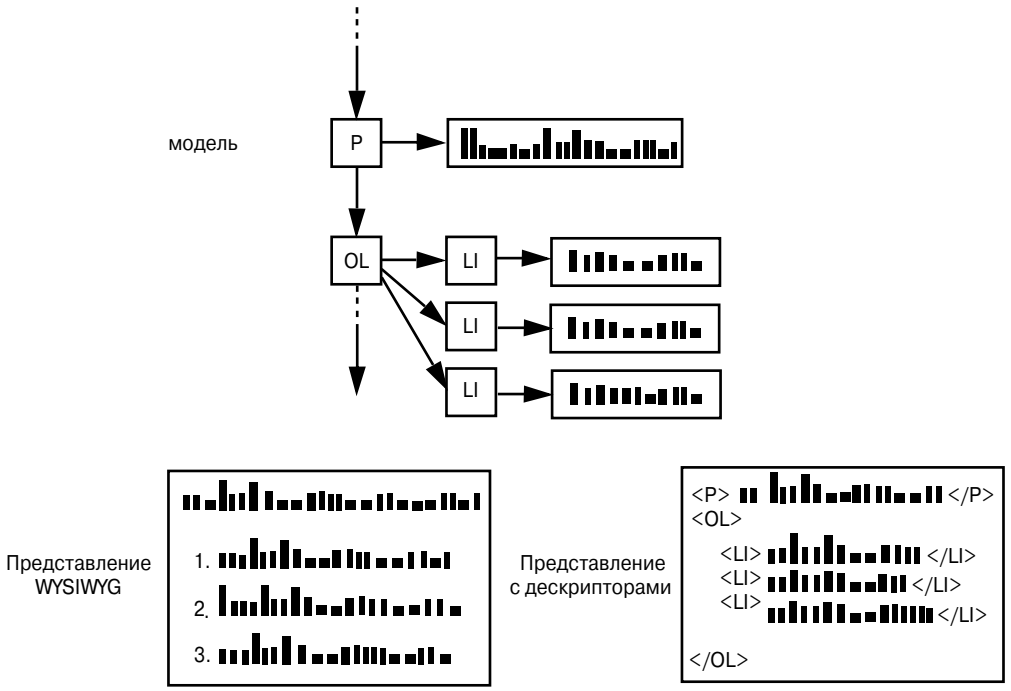


Рис. 9.3. Два разных представления одной и той же модели

Контроллер обрабатывает события, связанные с вводом информации, например, щелчки мышью и нажатия клавиш, а затем решает, преобразовывать ли эти события в изменения модели или представления. Например, если пользователь щелкнул на кнопке в текстовом поле, контроллер вызовет команду модели “вставить символ”. Затем модель заставит представление обновить изображение. Представление никогда не узнает, почему изменился текст. Однако если пользователь нажал клавишу управления курсором, контроллер может отдать представлению команду, вызывающую прокрутку. Прокрутка не изменяет текст, поэтому модель никогда не узнает об этом событии.

Взаимодействие модели, представления и контроллера схематически показано на рис. 9.4.

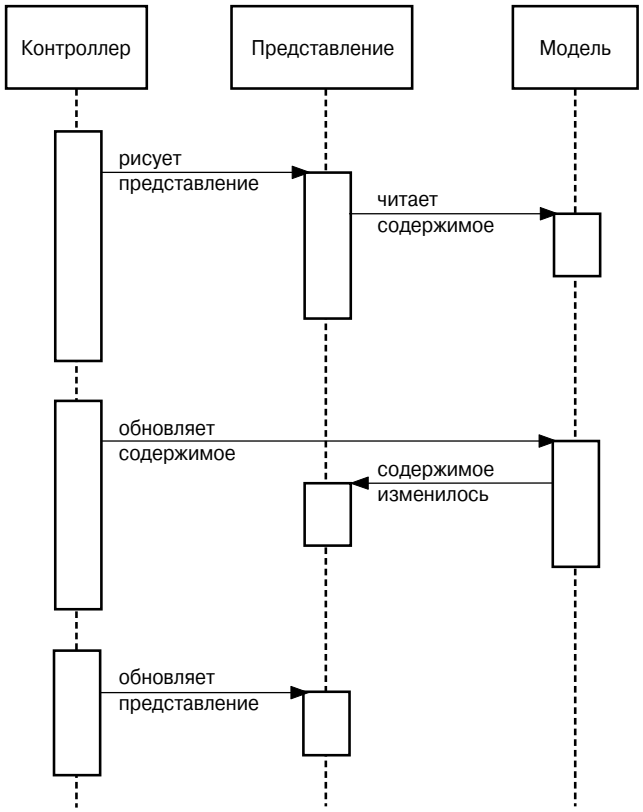


Рис. 9.4. Взаимодействие между объектами модели, представления и контроллера

Как программист, использующий компоненты Swing, вы совершенно не обязаны постоянно думать об архитектуре “Модель–представление–контроллер”. Каждый компонент пользовательского интерфейса имеет так называемый класс-оболочку (например, `JButton` или `JTextField`), в котором хранятся модель и ее представление. Если вас интересует содержимое (например, текст в соответствующем поле), класс-оболочка запросит модель и вернет ответ. Если вас интересует изменение внешнего вида (например, положение курсора в поле редактирования), класс-оболочка пошлет запрос представлению. Однако иногда класс-оболочка работает недостаточно хорошо. Тогда нужно извлечь модель и работать непосредственно с ней. (Работать непосредственно с представлением необязательно — это задача кода, задающего стиль.)

Помимо того, что шаблон "Модель–представление–контроллер" выполняет возложенные на нее задачи, она оказалась очень привлекательной для проектировщиков Swing, поскольку облегчает поддержку различных стилей пользовательского интерфейса. Модель кнопки или поля редактирования не зависит от стиля, но от него полностью зависит визуальное представление компонента. Контроллер также может изменяться. Например, в устройствах, управляемых голосом, контроллер должен обрабатывать наборы событий, резко отличающиеся от стандартных событий, связанных с клавиатурой и мышью. Отделяя модель от пользовательского интерфейса, разработчики, использующие пакет Swing, получили возможность повторно применять код модели и даже переключать стиль интерфейса в ходе выполнения программы.

Разумеется, шаблоны — это лишь руководство к действию, а не догма. Нет такого шаблона, который идеально подходил бы для любых ситуаций. Например, может оказаться сложным следовать схеме "место у окна", если вам нужно перестроить, скажем, больничную палату. Аналогично разработчики пакета Swing обнаружили, что в реальной жизни не всегда возможно точно следовать шаблону "Модель–представление–контроллер". Модели легко разделяются, и каждый компонент пользовательского интерфейса имеет свой класс модели. Однако задачи, стоящие перед представлением и контроллером, не всегда четко отделяются одна от другой, поэтому при распределении их между различными классами возникают сложности. Конечно, пользователю этих классов совершенно безразлично, как они реализованы. Как мы уже говорили, о моделях часто можно не думать вовсе, а использовать классы-оболочки.

Кнопки Swing с точки зрения шаблона "Модель–представление–контроллер"

В предыдущей главе мы показали, как работает кнопка, ничего не говоря о модели, представлении и контроллере. Поскольку кнопка — почти простейший элемент пользовательского интерфейса, ее удобно использовать в качестве наглядного пособия для изучения шаблона "Модель–представление–контроллер". При разработке более сложных компонентов Swing применяется аналогичный подход.

Для большинства компонентов классы модели реализуют интерфейсы, имена которых оканчиваются словом `Model`. В частности, для кнопок используется интерфейс `ButtonModel`. Классы, реализующие этот интерфейс, могут определять состояние кнопок разного вида. Кнопки настолько просты, что для них в библиотеке Swing предусмотрен отдельный класс `DefaultButtonModel`, реализующий этот интерфейс.

Понять, какого рода данные поддерживаются моделью кнопки, можно, просмотрев методы, объявленные в интерфейсе `ButtonModel` (табл. 9.1).

Таблица 9.1. Свойства, объявленные в интерфейсе `ButtonModel`

Имя свойства	Описание
`actionCommand`	Строка команды действия, ассоциированной с кнопкой.
`mnemonic`	Мнемоническое обозначение для данной кнопки.
`armed`	Равно `true`, если кнопка была нажата и курсор мыши все еще находится на кнопке.
`enabled`	Равно `true`, если кнопка не заблокирована.
`pressed`	Равно `true`, если кнопка была нажата и кнопка мыши все еще не отпущена.
`rollover`	Равно `true`, если курсор мыши находится на кнопке.
`selected`	Равно `true`, если кнопка включена (используется для флажков опций и переключателей).

В каждом объекте JButton хранится объект модели кнопки, который можно извлечь оттуда:

```
JButton button = new JButton("Blue");
ButtonModel model = button.getModel();
```

На практике подробности, касающиеся состояния кнопки, интересуют лишь представление, которое рисует ее на экране. Однако из класса JButton можно извлечь и другую полезную информацию — в частности, заблокирована ли кнопка. (Для этого класс JButton запрашивает ее модель.)

Обратимся еще раз к интерфейсу ButtonModel и попробуем определить, *чего в нем нет.* Оказывается, модель *не хранит* название кнопки и ее пиктограмму. По этой причине анализ модели не дает информации о внешнем виде кнопки. (При реализации групп переключателей эта особенность может стать для программиста источником осложнений.)

Стоит также отметить, что *одна и та же модель* (а именно DefaultButtonModel) используется для поддержки нажимаемых кнопок, переключателей, флажков опций и даже для выбора пункта меню. Разумеется, каждый из упомянутых типов кнопок имеет свое собственное представление и отдельный контроллер. Если реализуется интерфейс в стиле Metal, класс JButton использует для представления класс BasicButtonUI, а для контроллера — класс ButtonUIListener. В общем, каждый компонент библиотеки Swing имеет связанный с ним объект представления, название которого заканчивается на UI. При этом не все компоненты пакета Swing имеют свой собственный объект контроллера.

Итак, прочитав это краткое вступление, посвященное классу JButton, вы можете спросить: а что на самом деле представляет собой класс JButton? Это просто класс-оболочка, подкласс класса JComponent, который содержит объект DefaultButtonModel, некоторые данные, необходимые для отображения (например, метку кнопки и ее пиктограмму), а также объект класса BasicButtonUI, реализующий представление кнопки.

Общие сведения о компоновке элементов

Прежде чем перейти к обсуждению таких компонентов пакета Swing, как текстовые поля редактирования и переключатели, кратко рассмотрим, каким образом они располагаются во фрейме. Поскольку, в отличие от Visual Basic, в JDK нет визуального конструктора форм, программист должен сам позаботиться о коде, размещающем компоненты пользовательского интерфейса во фрейме.

Разумеется, если вы работаете в интегрированной среде разработки на Java, в ней вполне могут быть предусмотрены средства, которые автоматизируют некоторые из этих задач. Несмотря на это, программист обязан четко представлять себе, как осуществляется размещение компонентов, поскольку интерфейсы, полученные с помощью наилучших систем автоматической генерации, обычно нуждаются в ручной "доводке".

Рассмотрим программу из предыдущей главы, в которой кнопки использовались для изменения цвета фрейма (рис. 9.5).

Кнопки содержатся в объекте JPanel и управляются диспетчером "плавающей" компоновки (flow layout manager) — стандартным диспетчером компоновки для панели. На рис. 9.6 показано, что происходит, когда вы добавляете дополнительные кнопки к панели. Как видим, если кнопки не помещаются в текущей строке, они переносятся на новую строку.

Более того, кнопки будут центрированы в панели, даже если пользователь изменит размер фрейма (рис. 9.7).

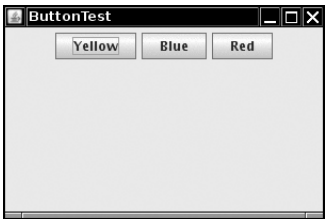


Рис. 9.5. Панель с тремя кнопками



Рис. 9.6. Панель с шестью кнопками, управляемая диспетчером "плавающей" компоновки

В общем случае все *компоненты* размещаются в *контейнерах* и управляются *диспетчером компоновки* (layout manager), который определяет позиции и размеры компонентов в контейнере.

Кнопки, текстовые поля и другие элементы пользовательского интерфейса расширяют класс `Component`. Компоненты могут помещаться внутрь контейнеров, таких как панели. Поскольку контейнеры сами могут быть помещены внутри других контейнеров, класс `Container` расширяет `Component`. На рис. 9.8 показана иерархия наследования `Component`.



Рис. 9.7. При изменении размеров панели автоматически изменяется положение кнопок



На заметку! К сожалению, иерархия наследования несколько неясна по двум причинам. Во-первых, окна верхнего уровня, такие как `JFrame`, являются подклассами `Container`, а, следовательно, и `Component`, но они не могут быть помещены внутри других контейнеров. Более того, `JComponent` — это подкласс `Container`, а не `Component`, и потому можно добавлять другие компоненты в `JButton`. (Хотя эти компоненты и не будут отображаться.)

У каждого контейнера есть диспетчер компоновки по умолчанию, но вы всегда можете установить свой собственный. Например, следующий оператор:

```
panel.setLayout(new GridLayout(4, 4));
```

использует класс `GridLayout` для размещения компонентов в панели. Вы добавляете компоненты в контейнер. Метод `add()` контейнера принимает компонент и необходимые директивы по размещению для диспетчера компоновки.

API `java.awt.Container 1.0`

- `setLayout(LayoutManager m)`

  Задает диспетчер компоновки для данного контейнера.

- `Component add(Component c)`
- `Component add(Component c, Object constraints)` **1.1**

  Добавляет компонент к данному контейнеру и возвращает ссылку на него.

  Параметры `c` Добавляемый компонент.

  `constraints` Идентификатор, понятный диспетчеру компоновки.

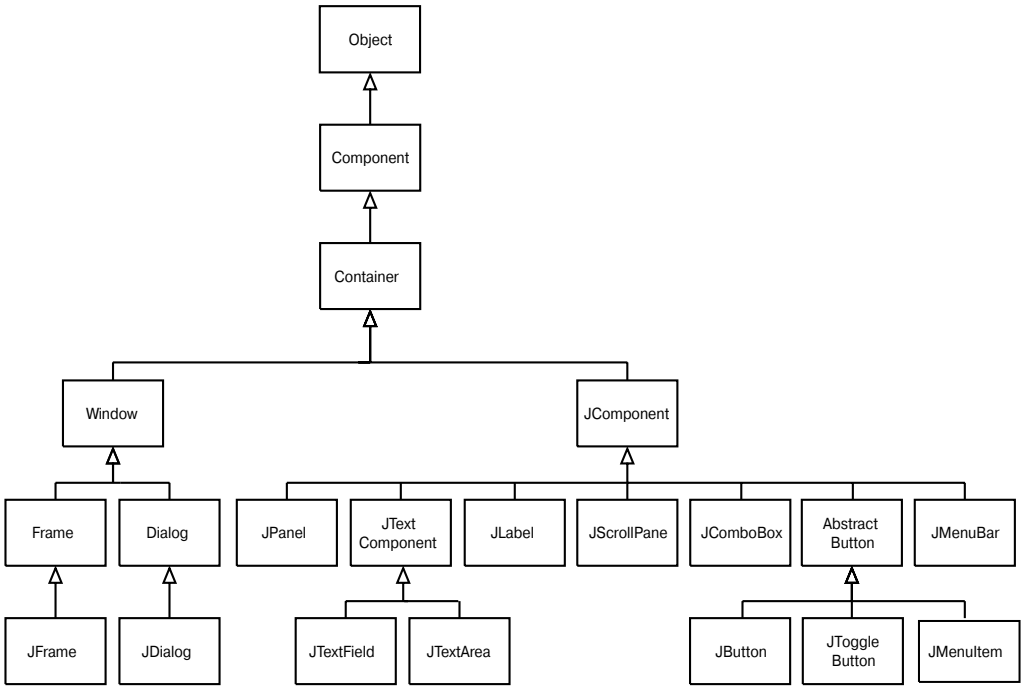


Рис. 9.8. Иерархия наследования для класса `Component`

API **`java.awt.FlowLayout 1.0`**

- `FlowLayout()`
- `FlowLayout(int align)`
- `FlowLayout(int align, ing hgap, int vgap)`

  Создает новый объект класса `FlowLayout`.

  Параметры: `align` `LEFT`, `CENTER` или `RIGHT`.

  `hgap` Расстояние между компонентами по горизонтали, заданное в пикселях (при отрицательных значениях компоненты перекрываются).

  `vgap` Расстояние между компонентами по вертикали, заданное в пикселях (при отрицательных значениях компоненты перекрываются).

Диспетчер компоновки `BorderLayout`

Диспетчер компоновки `BorderLayout` по умолчанию устанавливается для панели содержимого, присутствующей в объекте `JFrame`. В отличие от диспетчера `FlowLayout`, который полностью контролирует позицию каждого компонента, диспетчер `BorderLayout` позволяет вам выбрать место для каждого компонента. Компонент можно разместить в центре панели, в ее верхней или нижней части, а также слева или справа (рис. 9.9).

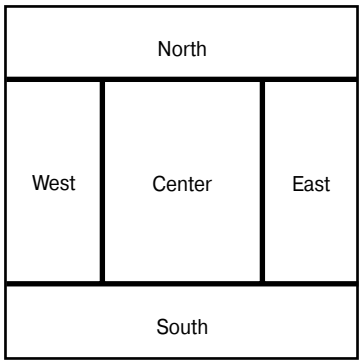


Рис. 9.9. Области, поддерживаемые диспетчером компоновки `BorderLayout`

Например:

```
frame.add(component, BorderLayout.SOUTH);
```

При размещении компонентов сначала выделяется место по краям контейнера, а оставшееся свободным пространство считается центральной областью. При изменении размера контейнера компонентов, расположенных по краям, остается постоянной; изменяется лишь размер центральной области. При добавлении компонента к панели указываются константы `CENTER` (центральная), `NORTH` (северная), `SOUTH` (южная), `EATH` (восточная) или `WEST` (западная), определенные в классе `BorderLayout`. Занимать все позиции в панели не обязательно. Если не указано никакого значения, по умолчанию принимается `CENTER`.



На заметку! Константы в классе `BorderLayout` определены как строки. Например, константа `BorderLayout.SOUTH` — это строка `"South"`. Многие программисты предпочитают использовать непосредственно строки, поскольку они короче, например `contentPane.add(component, "South")`. Однако если вы случайно сделаете в строке опечатку, компилятор не распознает этой ошибки.

В отличие от `FlowLayout`, диспетчер компоновки `BorderLayout` растягивает все компоненты, так чтобы они заполняли все свободное пространство. (Как вы помните, диспетчер `FlowLayout` оставляет размеры компонента неизменными.). Это может привести к проблеме при добавлении кнопки:

```
frame.add(yellowButton, BorderLayout.SOUTH); // не делайте так!
```

На рис. 9.10 показано, что произойдет, если мы попробуем применить фрагмент кода, приведенный выше. Размеры кнопки увеличатся, и она заполнит собой всю нижнюю часть фрейма. Если мы попытаемся вставить в нижнюю часть еще одну кнопку, она заменит собой предыдущую кнопку.

Один из способов решить эту проблему заключается в использовании дополнительных панелей. Например, обратите внимание на рис. 9.11. В панели, расположенной в нижней части панели содержимого, размещены три кнопки.

Чтобы достичь такой конфигурации, сначала создается новый объект `JPanel`, к которому затем добавляются отдельные кнопки. Как вы уже знаете, по умолчанию с обычной панелью связывается диспетчер компоновки `FlowLayout`. В данной ситуации он вполне подходит. В панель с помощью метода `add()` помещаются отдельные кнопки (эту процедуру мы уже рассматривали ранее).



Рис. 9.10. Одиночная кнопка, управляемая диспетчером компоновки `BorderLayout`



Рис. 9.11. Панель, размещенная в нижней части фрейма

Поскольку мы не переопределяли диспетчер компоновки, то положением кнопок управляет `FlowLayout`. Это значит, что кнопки будут выровнены по центру панели и масштаб их не будет увеличен для заполнения всего свободного пространства. Следующий фрагмент кода добавляет в нижнюю часть контейнера панель, содержащую три кнопки:

```
JPanel panel = new JPanel();
panel.add(yellowButton);
panel.add(blueButton);
panel.add(redButton);
frame.add(panel, BorderLayout.SOUTH);
```

`BorderLayout` расширяет размер панели с тремя кнопками, чтобы она заняла всю нижнюю область.

API `javax.awt.BorderLayout 1.0`

- `BorderLayout()`
- `BorderLayout(int hgap, int vgap)`

  Конструирует новый объект `BorderLayout`.

  Параметры `hgap` Расстояние между компонентами по горизонтали, заданное в пикселях (при отрицательных значениях компоненты перекрываются).

  `vgap` Расстояние между компонентами по вертикали, заданное в пикселях (при отрицательных значениях компоненты перекрываются).

Диспетчер компоновки `GridLayout`



Рис. 9.12. Калькулятор

Диспетчер компоновки `GridLayout` размещает компоненты по строкам и столбцам, подобно электронной таблице. Однако при этом размер всех ячеек одинаков. На рис. 9.12 показано окно, в котором для размещения кнопок калькулятора применен диспетчер `GridLayout`. При изменении размера окна кнопки автоматически увеличиваются или уменьшаются, причем размеры всех кнопок остаются одинаковыми.

Требуемое количество строк и столбцов указывается в конструкторе объекта `GridLayout`.

```
panel.setLayout(new GridLayout(5, 4));
```

Компоненты добавляются построчно, сначала — в первую ячейку первой строки, затем — во вторую ячейку первой строки и т.д.

```
panel.add(new JButton("1"));
panel.add(new JButton("2"));
```

В листинге 9.1 приведен исходный текст программы, имитирующей работу калькулятора. Это обычный калькулятор; он не использует польскую инверсную запись, получившую необъяснимую популярность в учебниках по языку Java. В этой программе после включения элементов в состав фрейма мы вызываем метод `pack()`. Он сохраняет рекомендуемый размер компонентов и на их основе вычисляет ширину и высоту фрейма.

На практике лишь немногие программы используют диспетчер компоновки `GridLayout` для размещения компонентов, подобно кнопкам калькулятора. Диспетчер, реализующий сетку из одной строки или из одного столбца, может применяться для разбиения окна на равные части. Например, если вы хотите разместить в окне ряд кнопок одинакового размера, вам надо включить их в панель, управляемую с помощью диспетчера `GridLayout`. Очевидно, что в конструкторе диспетчера надо задать одну строку, а число столбцов должно равняться количеству кнопок.

Листинг 9.1. Содержимое файла `Calculator.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import javax.swing.*;
4.
5.  /**
6.   * @version 1.33 2007-06-12
7.   * @author Cay Horstmann
8.   */
9.  public class Calculator
10. {
11.   public static void main(String[] args)
12.   {
13.      EventQueue.invokeLater(new Runnable()
14.         {
15.            public void run()
16.            {
17.               CalculatorFrame frame = new CalculatorFrame();
18.               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.               frame.setVisible(true);
20.            }
21.         });
22.   }
23. }
24.
25. /**
26.  * Фрейм с панелью калькулятора.
27.  */
28. class CalculatorFrame extends JFrame
29. {
30.   public CalculatorFrame()
31.   {
32.      setTitle("Calculator");
33.      CalculatorPanel panel = new CalculatorPanel();
```

```
34.       add(panel);
35.       pack();
36.    }
37. }
38.
39. /**
40.  * Панель с кнопками калькулятора и элементом
41.  * для отображения результатов вычислений.
42.  */
43. class CalculatorPanel extends JPanel
44. {
45.    public CalculatorPanel()
46.    {
47.       setLayout(new BorderLayout());
48.
49.       result = 0;
50.       lastCommand = "=";
51.       start = true;
52.
53.       // Добавление элемента для отображения результатов.
54.
55.       display = new JButton("0");
56.       display.setEnabled(false);
57.       add(display, BorderLayout.NORTH);
58.
59.       ActionListener insert = new InsertAction();
60.       ActionListener command = new CommandAction();
61.
62.       // Размещение кнопок в виде сетки 4x4.
63.
64.       panel = new JPanel();
65.       panel.setLayout(new GridLayout(4, 4));
66.
67.       addButton("7", insert);
68.       addButton("8", insert);
69.       addButton("9", insert);
70.       addButton("/", command);
71.
72.       addButton("4", insert);
73.       addButton("5", insert);
74.       addButton("6", insert);
75.       addButton("*", command);
76.
77.       addButton("1", insert);
78.       addButton("2", insert);
79.       addButton("3", insert);
80.       addButton("-", command);
81.
82.       addButton("0", insert);
83.       addButton(".", insert);
84.       addButton("=", command);
85.       addButton("+", command);
86.
87.       add(panel, BorderLayout.CENTER);
88.    }
89.
```

```
90.    /**
91.     * Добавление кнопки к центральной панели.
92.     * @param label Надпись на кнопке
93.     * @param listener Слушатель кнопки
94.     */
95.    private void addButton(String label, ActionListener listener)
96.    {
97.       JButton button = new JButton(label);
98.       button.addActionListener(listener);
99.       panel.add(button);
100.   }
101.
102.   /**
103.    * При обработке события строка, связанная с кнопкой,
104.    * помещается в конец отображаемого текста.
105.    */
106.   private class InsertAction implements ActionListener
107.   {
108.      public void actionPerformed(ActionEvent event)
109.      {
110.         String input = event.getActionCommand();
111.         if (start)
112.         {
113.            display.setText("");
114.            start = false;
115.         }
116.         display.setText(display.getText() + input);
117.      }
118.   }
119.
120.   /**
121.    * При обработке события выполняется команда, которая
122.    * определяется строкой, связанной с кнопкой.
123.    */
124.   private class CommandAction implements ActionListener
125.   {
126.      public void actionPerformed(ActionEvent event)
127.      {
128.         String command = event.getActionCommand();
129.
130.         if (start)
131.         {
132.            if (command.equals("-"))
133.            {
134.               display.setText(command);
135.               start = false;
136.            }
137.            else lastCommand = command;
138.         }
139.         else
140.         {
141.            calculate(Double.parseDouble(display.getText()));
142.            lastCommand = command;
143.            start = true;
144.         }
145.      }
146.   }
```

```
147.
148.   /**
149.    * Выполнение вычислений.
150.    * @param x Значение, накапливающее предыдущие результаты
151.    */
152.   public void calculate(double x)
153.   {
154.      if (lastCommand.equals("+")) result += x;
155.      else if (lastCommand.equals("-")) result -= x;
156.      else if (lastCommand.equals("*")) result *= x;
157.      else if (lastCommand.equals("/")) result /= x;
158.      else if (lastCommand.equals("=")) result = x;
159.      display.setText("" + result);
160.   }
161.
162.   private JButton display;
163.   private JPanel panel;
164.   private double result;
165.   private String lastCommand;
166.   private boolean start;
167.}
```

API java.awt.GridLayout 1.0

- `GridLayout(int rows, int columns)`
- `GridLayout(int rows, int columns, int hgap, int vgap)`

Создает новый объект класса `GridLayout` с заданным расстоянием между объектами по горизонтали и по вертикали.

Параметры

`rows`	Количество строк в сетке.
`columns`	Количество столбцов в сетке.
`hgap`	Расстояние по горизонтали, выраженное в пикселях (отрицательное значение приводит к перекрытию).
`vgap`	Расстояние по вертикали, выраженное в пикселях (отрицательное значение приводит к перекрытию).

API java.awt.Window 1.0

- `void pack()`

Определяет размеры окна на основании рекомендуемых размеров содержащихся в нем компонентов.

Ввод текста

Теперь мы готовы приступить к изучению интерфейсных элементов, содержащихся в пакете Swing. Начнем с компонентов, позволяющих пользователю вводить и редактировать текст. Для этого в языке Java предусмотрены два компонента: *поле редактирования*, `JTextField`, и *текстовая область*, `JTextArea`. В поле редактирования можно ввести только одну строку, а в текстовую область — несколько. `JPasswordField` принимает строку текста, не отображая его содержимого.

Все эти три класса расширяют класс `JTextComponent`. Создать компонент этого класса невозможно, поскольку он является абстрактным. С другой стороны, как это часто бывает при программировании на языке Java, при просмотре документации по API оказывается, что существуют методы, которые определены именно в классе `JTextComponent` и лишь наследуются подклассами. В качестве примера можно привести методы, позволяющие поместить текст в поле редактирования или текстовую область, а также извлечь текст из этих компонентов.

API javax.swing.text.JTextComponent 1.2

- `String getText()`
- `void setText(String t)`

  Получает и устанавливает текст в текстовом компоненте.

- `boolean isEditable()`
- `void setEditable(boolean b)`

  Получает и устанавливает свойство `editable`, определяющее, может ли пользователь редактировать содержимое текстового компонента.

Текстовые поля

Обычный способ добавить текстовое поле редактирования в окно — поместить его в панель или в другой контейнер (точно так же, как и кнопку):

```
JPanel panel = new JPanel();
JTextField textField = new JTextField("Default input", 20);
panel.add(textField);
```

Этот фрагмент кода добавляет текстовое поле, инициализируя его строкой `"Default input"`. Второй параметр конструктора задает длину строки. В данном случае длина строки равна 20 символам. К сожалению, символы — не очень точная единица измерения. Их ширина зависит от выбранного шрифта. Идея заключается в том, что если вы ожидаете ввода n символов или меньше, то должны указать n в качестве ширины колонки. На практике это измерение работает не очень хорошо и потому длину строки приходится завышать на один или два символа. Кроме того, следует учесть, что заданное количество символов считается пакетом AWT лишь *желательной* длиной строки. Решив уменьшить или увеличить поле редактирования, диспетчер компоновки изменит длину строки. Длина строки, задаваемая параметром конструктора `JTextField`, не является максимально допустимым количеством символов, которое может ввести пользователь в данное поле редактирования. Пользователь может набирать и более длинные строки; если текст выйдет за пределы поля редактирования, произойдет прокрутка содержимого. Часто прокрутка текста плохо воспринимается пользователями, поэтому размер поля редактирования следует задавать с запасом. Если во время выполнения возникает необходимость изменить этот размер, следует применить метод `setColumns()`.

! **Совет.** Изменив размер поля редактирования с помощью метода `setColumns()`, следует вызвать метод `revalidate()` контейнера, содержащего данный элемент.

```
textField.setColumns(10);
panel.revalidate();
```

Метод `revalidate()` вычисляет заново размер и взаимное расположение всех компонентов в контейнере. Затем диспетчер компоновки перерисовывает контейнер, изменяя размер поля редактирования.

Метод `revalidate()` принадлежит классу `JComponent`. Его выполнение не приводит к немедленному изменению размеров компонента; компонент лишь помечается специальным образом. Такой подход исключает постоянные вычисления при запросе на изменение размеров нескольких компонентов. Если же вы хотите изменить размеры компонентов в составе `JFrame`, вам следует вызвать метод `validate()`; `JFrame` не является подклассом `JComponent`.

Обычно при работе с программой пользователь должен иметь возможность вводить текст или редактировать содержимое текстового поля. Нередко в начале работы поля редактирования пусты. Для того чтобы создать пустое текстовое поле редактирования, не указывайте соответствующий параметр в конструкторе класса `JTextField`:

```
JTextField textField = new JTextField(20);
```

Содержимое поля редактирования можно изменить в любое время, используя метод `setText()` из родительского класса `TextComponent`:

```
textField.setText("Hello!");
```

Как было сказано ранее, определить, какой текст содержится в поле редактирования, можно с помощью метода `getText()`. Этот метод возвращает в точности тот текст, что набрал пользователь. Чтобы отбросить лишние пробелы до начала текста и после его окончания, надо к значению, возвращаемому методом `getText`, применить метод `trim()`:

```
String text = textField.getText().trim();
```

Для того чтобы задать шрифт для отображения текста, следует использовать метод `setFont()`.

API javax.swing.JTextField 1.2

- `JTextField(int cols)`

  Создает пустое поле `JTextField` с указанным числом колонок.

- `JTextField(String text, int cols)`

  Создает поле указанного размера, инициализированное заданной строкой и указанным числом колонок.

- `int getColumns()`
- `void setColumns(int cols)`

  Задает или получает размер поля редактирования, т.е. количество символов, которое должно умещаться в поле.

API javax.swing.JComponent 1.2

- `void revalidate()`

  Указывает на то, что размеры компонента и его позиция должны быть вычислены заново.

- `void setFont(Font f)`

  Устанавливает шрифт компонента.

API **java.awt.Component 1.0**

- `void validate()`

  Вызывает повторное вычисление позиции и размеров компонента. Если компонент является контейнером, позиции содержащихся в нем компонентов также должны быть определены заново.

- `Font getFont()`

  Получает шрифт компонента.

Метки и маркирующие компоненты

Метки — это компоненты, хранящие текст. Они не имеют обрамления и других видимых элементов (например, не имеют границ). Кроме того, они не реагируют на ввод пользователя. Метки можно использовать для идентификации компонентов. Например, в отличие от кнопок, текстовые компоненты не имеют меток, которые позволили бы их различать. Чтобы пометить компонент, не имеющий идентификатора, необходимо выполнить следующие действия.

1. Создать компонент `JLabel`, содержащий заданный текст.
2. Поместить его достаточно близко к компоненту, чтобы пользователь мог видеть, что эта метка относится именно к данному интерфейсному элементу.

Конструктор класса `JLabel` позволяет задать текст или пиктограмму и, при необходимости, выровнять содержимое компонента. Для этой цели используются константы, объявленные в интерфейсе `SwingConstants`. Интерфейс определяет несколько полезных констант, таких как `LEFT`, `RIGHT`, `CENTER`, `NORTH`, `EAST` и т.п. Класс `JLabel` является одним из нескольких классов библиотеки Swing, реализующих этот интерфейс. Таким образом, можно, например, задать метку, текст в которой будет выровнен по левому краю:

```
JLabel label = new JLabel("Minutes", SwingConstants.RIGHT);
```

или

```
JLabel label = new JLabel("Minutes", JLabel.RIGHT);
```

Методы `setText` и `setIcon` позволяют задать текст и пиктограмму метки во время выполнения.

! **Совет.** Начиная с версии Java SE 1.3 на кнопки, метки и пункты меню можно помещать как обычный текст, так и текст в формате HTML. Мы не рекомендуем помещать на кнопки текст в формате HTML — он нарушает стиль интерфейса. Однако в метках такой текст очень эффективен. Строку метки надо поместить между дескрипторами `<HTML>...</HTML>`.

```
label = new JLabel("<html><b>Необходимый</b> элемент:</html>");
```

Учтите, что первый компонент с меткой, содержащей текст в формате HTML, отображается на экране с запаздыванием, поскольку для этого нужно загрузить код, выполняющий разбор HTML-выражений.

Метки можно помещать в контейнер подобно любому другому компоненту. Это означает, что для их размещения применяются те же подходы, что вы видели ранее.

API javax.swing.JLabel 1.2

- `JLabel(String text)`
- `JLabel(Icon icon)`
- `JLabel(String text, int align)`
- `JLabel(String text, Icon icon, int align)`

Создает метку с текстом и пиктограммой. Пиктограмма располагается левее текста.

Параметры `text` Текст метки.
`icon` Пиктограмма метки.
`align` Одна из констант: `LEFT`, `CENTER` или `RIGHT`.

- `String getText()`
- `void setText(String text)`

Извлекает или устанавливает текст метки.

- `Icon getIcon()`
- `void setIcon(Icon icon)`

Извлекает или устанавливает пиктограмму метки.

Поля для ввода пароля

Поля для ввода пароля представляют собой особый вид полей редактирования. Чтобы скрыть пароль от посторонних наблюдателей, его символы не отображаются на экране. Вместо этого каждый символ в пароле заменяется *эхо-символом* (echo symbol), обычно звездочкой (`*`). В библиотеке Swing предусмотрен класс `JPasswordField`, реализующий такое поле редактирования.

Поле для ввода пароля — это еще одна демонстрация преимуществ архитектуры "Модель–представление–контроллер". Для хранения данных пароль использует ту же модель, что и обычное поле редактирования, но его представление изменено и заменяет все символы пароля эхо-символами.

API javax.swing.JPasswordField 1.2

- `JPasswordFirld(String text, int columns)`

Создает новое поле для ввода пароля.

- `void setEchoChar(char echo)`

Задает эхо-символ, который может зависеть от стиля интерфейса. Если задано значение `0`, выбирается эхо-символ по умолчанию.

- `char[] getPassword()`

Возвращает текст, содержащийся в поле для ввода пароля. Для обеспечения большей безопасности возвращаемый массив после использования следует перезаписать. Пароль возвращается как массив символов, а не как объект класса `String`. Причина такого решения заключается в том, что строка может оставаться в виртуальной машине до тех пор, пока не будет уничтожена механизмом сборки мусора.

Текстовые области

Иногда при работе программы возникает необходимость ввести несколько строк. Как указывалось ранее, для этого применяется компонент `JTextArea`. Поместив данный компонент в свою программу, разработчик предоставляет пользователю возможность вводить любое количество текста, разделяя строки нажатием клавиши <Enter>. Каждая строка заканчивается символом `'\n'`, как это предусмотрено в языке Java. Пример работы текстовой области показан на рис. 9.13.



Рис. 9.13. Текстовая область

В конструкторе компонента `JTextArea` указывается количество строк и их длина. Например:

```
textArea = new JTextArea(8, 40);
   // 8 строк по 40 символов в каждой.
```

Параметр, задающий количество символов в строке, действует так же, как и для поля редактирования; его значение рекомендуется немного завысить. Пользователь не ограничен количеством строк и их длиной. Если длина строки или число строк выйдет за пределы заданных параметров, текст будет прокручиваться в окне. Для изменения длины строк можно применять метод `setColumns()`, а для изменения их количества — метод `setRows()`. Эти параметры задают лишь рекомендуемые размеры — диспетчер компоновки может самостоятельно увеличивать или уменьшать размеры поля редактирования.

Если пользователь ввел больше текста, чем умещается в текстовой области, то произойдет отсечение. Этого можно избежать, установив автоматический перенос строки (line-wrapping).

```
textArea.setLineWrap(true); // В длинных строках выполняется перенос.
```

Автоматический перенос проявляется лишь визуально. Текст, хранящийся в документе, изменен не будет — в него не вставляются символы `'\n'`.

Панели прокрутки

В пакете Swing текстовая область не имеет полос прокрутки. Если они необходимы, текстовую область следует включить в *панель прокрутки* (scroll pane):

```
textArea = new JtextArea(8, 40);
JScrollPane scrollPane = new JScrollPane(textArea);
```

Теперь панель прокрутки управляет представлением текстовой области. Полоса прокрутки автоматически появляется, когда текст выходит за рамки отведенной для него области, и исчезает, когда "лишняя" часть текста удаляется. Прокрутка поддерживается панелью прокрутки — сама программа не обязана обрабатывать события, связанные с ней.

Это универсальный механизм, который работает с любым компонентом, а не только с текстовыми областями. Чтобы добавить линейки прокрутки к компоненту, поместите его в панель прокрутки.

Программа в листинге 9.2 демонстрируются различные текстовые компоненты. Эта программа отображает текстовое поле, поле пароля и текстовую область с линейками прокрутки. Текстовое поле и поле пароля снабжены метками. Чтобы вставить предложение в конец текста, щелкните на кнопке Insert (Вставить).

Листинг 9.2. Содержимое файла `TextComponentTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import javax.swing.*;
4.
5.  /**
6.   * @version 1.40 2007-04-27
7.   * @author Cay Horstmann
8.   */
9.  public class TextComponentTest
10. {
11.   public static void main(String[] args)
12.   {
13.     EventQueue.invokeLater(new Runnable()
14.     {
15.        public void run()
16.        {
17.           TextComponentFrame frame = new TextComponentFrame();
18.           frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.           frame.setVisible(true);
20.        }
21.     });
22.   }
23. }
24.
25. /**
26.  * Фрейм с простыми текстовыми компонентами.
27.  */
28. class TextComponentFrame extends JFrame
29. {
30.   public TextComponentFrame()
31.   {
32.     setTitle("TextComponentTest");
33.     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34.
35.     final JTextField textField = new JTextField();
36.     final JPasswordField passwordField = new JPasswordField();
37.
38.     JPanel northPanel = new JPanel();
39.     northPanel.setLayout(new GridLayout(2, 2));
40.     northPanel.add(new JLabel("User name: ", SwingConstants.RIGHT));
41.     northPanel.add(textField);
42.     northPanel.add(new JLabel("Password: ", SwingConstants.RIGHT));
43.     northPanel.add(passwordField);
44.
45.     add(northPanel, BorderLayout.NORTH);
46.
47.     final JTextArea textArea = new JTextArea(8, 40);
48.     JScrollPane scrollPane = new JScrollPane(textArea);
49.
50.     add(scrollPane, BorderLayout.CENTER);
51.
52.     // Вставка кнопки для добавления текста
53.     // к текстовой области.
54.
```

```
55.     JPanel southPanel = new JPanel();
56.
57.     JButton insertButton = new JButton("Insert");
58.     southPanel.add(insertButton);
59.     insertButton.addActionListener(new ActionListener()
60.     {
61.       public void actionPerformed(ActionEvent event)
62.       {
63.          textArea.append("User name: " + textField.getText() + " Password: "
64.             + new String(passwordField.getPassword()) + "\n");
65.       }
66.     });
67.
68.     add(southPanel, BorderLayout.SOUTH);
69.
70.     // Добавление текстовой области с линейками прокрутки.
71.
72.   }
73.
74.   public static final int DEFAULT_WIDTH = 300;
75.   public static final int DEFAULT_HEIGHT = 300;
76. }
```

API javax.swing.JTextArea 1.2

- `JTextArea()`
- `JTextArea(int rows, int cols)`
- `JTextArea(String text, int rows, int cols)`

  Создает новую текстовую область.

- `void setColumns(int cols)`

  Задает предпочтительную длину строки.

- `void setRows()`

  Задает предпочтительное количество строк.

- `void append(String newText)`

  Добавляет заданный текст в конец содержимого текстовой области.

- `void setLineWrap(boolean wrap)`

  Включает и отключает режим автоматического переноса строк.

- `void setWrapStyleForward(boolean word)`

  Если параметр `word` равен `true`, то перенос в длинных строках выполняется по границе слова, в противном случае границы слов во внимание не принимаются.

- `void setTabSize(int c)`

  Устанавливает табуляторы через каждые `c` символов. Обратите внимание на то, что пробелы не включаются; текст лишь выравнивается по следующей позиции табуляции.

API javax.swing.JScrollPane 1.2

- `JScrollPane(Component c)`

  Создает панель прокрутки, которая отображает содержание указанного компонента. Полоса прокрутки появляется, только если текст превышает размеры представления.

Компоненты для выбора вариантов

Теперь вы знаете, как принимать текстовый ввод пользователя. Однако во многих случаях предпочтительнее ограничить действия пользователя выбором среди конечного набора возможностей. Эти возможности могут быть представлены кнопками или списком пунктов. (Во многих случаях такой подход освободит вас от необходимости отслеживать ошибки.) В этом разделе описаны флажки опций, переключатели, списки и линейные регуляторы (ползунки).

Флажки опций



Рис. 9.14. Флажки опций

Если данные сводятся к ответу “да” или “нет”, используйте компонент — флажок опций (checkbox). Чтобы установить флажок, нужно щелкнуть мышью на компоненте, а для того чтобы сбросить флажок — щелкнуть повторно. Установить или сбросить флажок можно также с помощью клавиши пробела, нажав ее в момент, когда на компоненте находится фокус ввода.

На рис. 9.14 показано простое окно программы с двумя флажками опций, один из которых включает и отключает курсив, а другой — полужирный шрифт. Обратите внимание на то, что первый флажок обладает фокусом ввода. Об этом говорит прямоугольная рамка вокруг его метки. Каждый раз, когда пользователь щелкает на флажке, содержимое окна обновляется с учетом нового атрибута шрифта.

Флажки опций сопровождаются метками, указывающими их предназначение. Текст метки задается в конструкторе:

```
bold = new JCheckBox("Bold");
```

Для установки и сброса флажка используется метод `setSelected()`. Например:

```
bold.setSelected(true);
```

Метод `isSelected()` позволяет определить текущее состояние каждого флажка. Если значение равно `false`, значит, он сброшен, а если `true` — установлен.

Щелкая на флажке, пользователь инициирует события. Как всегда, вы можете связать с данным компонентом объект-слушатель. В нашей программе для двух кнопок предусмотрен один и тот же слушатель действий:

```
ActionListener listener = ...
bold.addActionListener(listener);
italic.addActionListener(listener);
```

В методе `actionPerformed()` обработчика запрашивается текущее состояние флажков `bold` и `italic`, а затем устанавливается шрифт, которым должен отображаться обычный текст: полужирный, курсив или полужирный курсив.

```
public void actionPerformed(ActionEvent event)
{
   int mode = 0;
   if (bold.isSelected()) mode += Font.BOLD;
   if (italic.isSelected()) mode += Font.ITALIC;
   label.setFont(new Font("Serif", mode, FONTSIZE));
}
```

В листинге 9.3 приведен полный текст программы, демонстрирующей работу с флажками опций.

Листинг 9.3. Содержимое файла `CheckBoxTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import javax.swing.*;
4.
5.  /**
6.   * @version 1.33 2007-06-12
7.   * @author Cay Horstmann
8.   */
9.  public class CheckBoxTest
10. {
11.   public static void main(String[] args)
12.   {
13.      EventQueue.invokeLater(new Runnable()
14.         {
15.            public void run()
16.            {
17.               CheckBoxFrame frame = new CheckBoxFrame();
18.               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.               frame.setVisible(true);
20.            }
21.         });
22.   }
23. }
24.
25. /**
26.  * Фрейм с текстовой меткой и флажками
27.  * для установки атрибутов шрифта.
28.  */
29. class CheckBoxFrame extends JFrame
30. {
31.   public CheckBoxFrame()
32.   {
33.      setTitle("CheckBoxTest");
34.      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
35.
36.      // Добавление текстовой метки.
37.
38.      label = new JLabel("The quick brown fox jumps over the lazy dog.");
39.      label.setFont(new Font("Serif", Font.PLAIN, FONTSIZE));
40.      add(label, BorderLayout.CENTER);
41.
42.      // Данный слушатель устанавливает атрибуты шрифта,
43.      // которым отображается текст метки, в соответствии
44.      // с состоянием флажков опций.
```

```
45.
46.       ActionListener listener = new ActionListener()
47.       {
48.          public void actionPerformed(ActionEvent event)
49.          {
50.             int mode = 0;
51.             if (bold.isSelected()) mode += Font.BOLD;
52.             if (italic.isSelected()) mode += Font.ITALIC;
53.             label.setFont(new Font("Serif", mode, FONTSIZE));
54.          }
55.       };
56.
57.       // Добавление флажков опций.
58.
59.       JPanel buttonPanel = new JPanel();
60.
61.       bold = new JCheckBox("Bold");
62.       bold.addActionListener(listener);
63.       buttonPanel.add(bold);
64.
65.       italic = new JCheckBox("Italic");
66.       italic.addActionListener(listener);
67.       buttonPanel.add(italic);
68.
69.       add(buttonPanel, BorderLayout.SOUTH);
70.    }
71.
72.    public static final int DEFAULT_WIDTH = 300;
73.    public static final int DEFAULT_HEIGHT = 200;
74.
75.    private JLabel label;
76.    private JCheckBox bold;
77.    private JCheckBox italic;
78.
79.    private static final int FONTSIZE = 12;
80. }
```

API javax.swing.JCheckBox 1.2

- `JCheckBox(String label)`
- `JCheckBox(String label, bodean state)`

  Создает флажок опции, снабженный заданной меткой и находящийся в указанном состоянии.

- `JCheckBox(String label, Icon icon)`

  Создает флажок с указанной меткой и пиктограммой. Первоначально флажок сброшен.

- `boolean isSelected()`
- `void setSelected(boolean state)`

  Извлекает или устанавливает новое состояние флажка опции.

Переключатели

В предыдущем примере пользователь мог установить оба флажка, один из них или ни одного. Во многих ситуациях необходимо выбрать только один из предложенных вариантов. Если пользователь установит другой флажок, то предыдущий флажок будет сброшен. Такую группу флажков часто называют *группой переключателей* (radio button). Эти кнопки напоминают переключатели диапазонов на радиоприемниках — при нажатии одной из таких кнопок ранее нажатая кнопка возвращается в исходное состояние. На рис. 9.15 показан типичный пример окна программы, содержащей группу переключателей. Пользователь может выбрать размер шрифта — Small (Малый), Medium (Средний), Large (Большой) и Extra large (Очень большой). Разумеется, выбрать можно лишь один размер.



Рис. 9.15. Группа переключателей

Пакет Swing позволяет легко реализовать группы переключателей. Для этого нужно создать для каждой группы по одному объекту `ButtonGroup`. Затем в группу переключателей необходимо добавить объекты типа `JRadioButton`. Объект класса `ButtonGroup` предназначен для того, чтобы отключать ранее выбранный переключатель, если пользователь щелкнул на новой кнопке.

```
ButtonGroup group = new ButtonGroup();

JRadioButton smallButton
   = new JRadioButton("Small", false);
group.add(smallButton);

JRadioButton smallButton
   = new JRadioButton("Medium", true);
group.add(mediumButton);
...
```

Второй параметр конструктора принимает значение `true`, если изначально переключатель должен быть включен, и `false` — если выключен. Отметим, что объект типа `ButtonGroup` управляет лишь *поведением* переключателей. Если нужно объединить несколько групп переключателей, их следует поместить в контейнер, например в объект класса `JPanel`.

Вернемся к рис. 9.14 и 9.15. Обратите внимание на то, что переключатели отличаются по внешнему виду от флажков опций. Флажки опций изображаются в виде квадратов, причем для установленных флажков внутри имеется галочка, в то время как переключатели имеют круглую форму: включенные — с точкой внутри, а выключенные — без обозначений.

Механизм извещения о наступлении событий, связанных с переключателями, точно такой же, как и для любых других кнопок. Если пользователь включил переключа-

тель, соответствующий объект генерирует событие. В рассматриваемом здесь примере установлен слушатель, задающий конкретный размер шрифта.

```
ActionListener listener = new
   ActionListener()
   {
      public void actionPerformed(ActionListener evt)
      {
         // Размер шрифта задается последним параметром
         // метода addRadioButton.
         label.setFont(new Font("Serif", Font.PLAIN, size));
      }
   };
```

Сравните этот обработчик с обработчиком для флажка опций. Каждому переключателю соответствует свой объект-слушатель. Каждый объект-слушатель точно знает, что нужно делать — установить конкретный размер шрифта. Для флажков ситуация иная. Оба флажка в нашем примере были связаны с одним и тем же слушателем. Он вызывал метод, определяющий текущее состояние обоих флажков.

Можно ли реализовать такой подход для переключателей? Зададим один слушатель, устанавливающий конкретный размер шрифта:

```
if (smallButton.isSelected()) size = 8;
else is (mediumButton.isSelected()) size = 12;
...
```

Однако предпочтительнее использовать отдельные объекты-слушатели поскольку они более явно связывают размер шрифта с конкретным переключателем.



На заметку! В группе может быть выбран только один переключатель. Хорошо бы сразу знать, какой именно, не проверяя каждый переключатель в группе. Поскольку объект класса `ButtonGroup` управляет всеми переключателями, было бы удобно, если бы он давал нам ссылку на выбранный переключатель. Действительно, в классе `ButtonGroup` есть метод `getSelection()`, но этот метод не возвращает ссылку на выбранный переключатель. Вместо этого он возвращает ссылку на объект класса `ButtonModel` — модель, связанную с этим переключателем. К сожалению, все методы класса `ButtonModel` не представляют собой ничего ценного. Интерфейс `ButtonModel` наследует метод `getSelectedObjects()` из интерфейса `ItemSelectable`, возвращающий совершенно бесполезную ссылку `null`. Метод `getActionCommand()` выглядит предпочтительнее, поскольку позволяет определить текстовую строку, связанную с переключателем. Однако команда в модели этого переключателя снова представляет собой нулевую ссылку `null`. Только если программист явно задаст команды для каждого переключателя с помощью метода `setActionCommand()`, установив значение, соответствующее каждой команде, впоследствии можно будет определить команду включенного переключателя с помощью метода `buttonGroup.getSelection().getActionCommand()`.

В листинге 9.4 представлен полный текст программы, позволяющей установить размер шрифта с помощью переключателей.

Листинг 9.4. Содержимое файла `RadioButtonTest.java`

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
```

```
5. /**
6.  * @version 1.33 2007-06-12
7.  * @author Cay Horstmann
8.  */
9. public class RadioButtonTest
10.{
11.   public static void main(String[] args)
12.   {
13.      EventQueue.invokeLater(new Runnable()
14.         {
15.            public void run()
16.            {
17.               RadioButtonFrame frame = new RadioButtonFrame();
18.               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.               frame.setVisible(true);
20.            }
21.         });
22.   }
23.}
24.
25./**
26. * Фрейм с текстовой меткой и переключателями,
27. * предназначенными для выбора размера шрифта.
28. */
29.class RadioButtonFrame extends JFrame
30.{
31.   public RadioButtonFrame()
32.   {
33.      setTitle("RadioButtonTest");
34.      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
35.
36.      // Добавление метки.
37.
38.      label = new JLabel("The quick brown fox jumps over the lazy dog.");
39.      label.setFont(new Font("Serif", Font.PLAIN, DEFAULT_SIZE));
40.      add(label, BorderLayout.CENTER);
41.
42.      // Добавление переключателей.
43.
44.      buttonPanel = new JPanel();
45.      group = new ButtonGroup();
46.
47.      addRadioButton("Small", 8);
48.      addRadioButton("Medium", 12);
49.      addRadioButton("Large", 18);
50.      addRadioButton("Extra large", 36);
51.
52.      add(buttonPanel, BorderLayout.SOUTH);
53.   }
54.
55.   /**
56.    * Добавление переключателей, устанавливающих размер шрифта
57.    * для отображения текстовой метки.
58.    * @param name Строка, которой помечается кнопка
59.    * @param size Размер шрифта для кнопки
60.    */
```

```
61.   public void addRadioButton(String name, final int size)
62.   {
63.      boolean selected = size == DEFAULT_SIZE;
64.      JRadioButton button = new JRadioButton(name, selected);
65.      group.add(button);
66.      buttonPanel.add(button);
67.
68.      // Слушатель, устанавливающий размер шрифта для метки.
69.
70.      ActionListener listener = new ActionListener()
71.      {
72.         public void actionPerformed(ActionEvent event)
73.         {
74.            // Размер соответствует последнему параметру метода addRadioButton
75.            label.setFont(new Font("Serif", Font.PLAIN, size));
76.         }
77.      };
78.
79.      button.addActionListener(listener);
80.   }
81.
82.   public static final int DEFAULT_WIDTH = 400;
83.   public static final int DEFAULT_HEIGHT = 200;
84.
85.   private JPanel buttonPanel;
86.   private ButtonGroup group;
87.   private JLabel label;
88.
89.   private static final int DEFAULT_SIZE = 12;
90. }
```

API javax.swing.JRadioButton 1.2

- `JRadioButton(String label, Icon icon)`

  Создает кнопку переключателя с заданной строкой и пиктограммой. В исходном состоянии кнопка не выбрана.

- `JRadioButton(String label, boolean state)`

  Создает кнопку переключателя, помеченную заданной строкой и находящуюся в указанном состоянии.

API javax.swing.ButtonGroup 1.2

- `void add(AbstractButton b)`

  Добавляет переключатель к группе.

- `ButtonModel getSelection()`

  Возвращает модель выбранной кнопки.

API javax.swing.ButtonModel 1.2

- `String getActionCommand(String s)`

  Возвращает команду для модели кнопки.

API `javax.swing.AbstractButton 1.2`

- `void setActionCommand(String s)`

  Задает команду для данной кнопки и ее модели.

Рамки

Если в одном окне расположено несколько групп переключателей, нужно иметь возможность различать их между собой. Для этого в библиотеке Swing предусмотрен набор *рамок* (borders). Рамку можно задать для каждого компонента, расширяющего класс `JComponent`. Обычно рамка окружает панель, заполненную ее элементами пользовательского интерфейса, например, переключателями.

Выбор рамок невелик, и все они задаются с помощью одинаковых действий.

1. Вызывается статический метод класса `BorderFactory`, создающий рамку, для которой предусмотрены следующие стили (рис. 9.16):
   - Lowered bevel (утопленная фаска)
   - Raised bevel (приподнятая фаска)
   - Etched (гравировка)
   - Line (линия)
   - Matte (матовая)
   - Empty (пустая — создается пустое пространство, окружающее компонент)
2. При желании с помощью метода `BorderFactory.createTitledBorder()` добавляется заголовок.
3. По мере необходимости несколько рамок можно объединить в одну, воспользовавшись методом `BorderFactory.createCompoundBorder()`.
4. Получившаяся рамка добавляется к компоненту с помощью метода `setBorder()` класса `JComponent`.

Ниже приведен фрагмент кода, предназначенный для создания рамки, имитирующей гравировку.

```
Border etched = BorderFactory.createEtchedBorder()
Border titled = BorderFactory.createTitledBorder(etched, "A Title");
panel.setBorder(titled);
```

Запустите программу, код которой приведен в листинге 9.5, и вы увидите, как выглядят рамки, выполненные в разных стилях.

Разные рамки имеют разные возможности для задания ширины и цвета. Детали описаны в документации по API. Истинные энтузиасты использования рамок обрадуются, узнав, что существуют классы `SoftBevelBorder` и `LineBorder`, позволяющие сглаживать углы. Такие рамки можно создать только с помощью конструкторов этих классов — для них не существует методов `BorderFactory`.

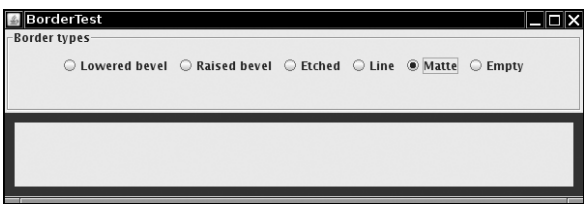


Рис. 9.16. Тестирование рамок

Листинг 9.5. Содержимое файла `BorderTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import javax.swing.*;
4.  import javax.swing.border.*;
5.
6.  /**
7.   * @version 1.33 2007-06-12
8.   * @author Cay Horstmann
9.   */
10. public class BorderTest
11. {
12.    public static void main(String[] args)
13.    {
14.       EventQueue.invokeLater(new Runnable()
15.          {
16.             public void run()
17.             {
18.                BorderFrame frame = new BorderFrame();
19.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20.                frame.setVisible(true);
21.             }
22.          });
23.    }
24. }
25.
26. /**
27.  * Фрейм с переключателями, позволяющими выбрать стиль рамки.
28.  */
29. class BorderFrame extends JFrame
30. {
31.    public BorderFrame()
32.    {
33.       setTitle("BorderTest");
34.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
35.
36.       demoPanel = new JPanel();
37.       buttonPanel = new JPanel();
38.       group = new ButtonGroup();
39.
40.       addRadioButton("Lowered bevel", BorderFactory.createLoweredBevelBorder());
41.       addRadioButton("Raised bevel", BorderFactory.createRaisedBevelBorder());
42.       addRadioButton("Etched", BorderFactory.createEtchedBorder());
43.       addRadioButton("Line", BorderFactory.createLineBorder(Color.BLUE));
44.       addRadioButton("Matte", BorderFactory.createMatteBorder(
45.             10, 10, 10, 10, Color.BLUE));
46.       addRadioButton("Empty", BorderFactory.createEmptyBorder());
47.
48.       Border etched = BorderFactory.createEtchedBorder();
49.       Border titled = BorderFactory.createTitledBorder(etched, "Border types");
50.       buttonPanel.setBorder(titled);
51.
52.       setLayout(new GridLayout(2, 1));
53.       add(buttonPanel);
54.       add(demoPanel);
55.    }
```

```
56.
57.    public void addRadioButton(String buttonName, final Border b)
58.    {
59.       JRadioButton button = new JRadioButton(buttonName);
60.       button.addActionListener(new ActionListener()
61.       {
62.          public void actionPerformed(ActionEvent event)
63.          {
64.             demoPanel.setBorder(b);
65.          }
66.       });
67.       group.add(button);
68.       buttonPanel.add(button);
69.    }
70.
71.    public static final int DEFAULT_WIDTH = 600;
72.    public static final int DEFAULT_HEIGHT = 200;
73.
74.    private JPanel demoPanel;
75.    private JPanel buttonPanel;
76.    private ButtonGroup group;
77. }
```

API javax.swing.BorderFactory 1.2

- `static Border createLineBorder(Color color)`
- `static Border createLineBorder(Color color, int thickness)`

Создает простую рамку в виде линии.

- `static MatteBorder createMatteBorder(int top, int left, int bottom, int right, Color color)`
- `static MatteBorder createMatteBorder(int top, int left, int bottom, int right, Icon tileIcon)`

Создает широкую рамку, заполненную цветом или мозаикой, состоящей из повторяющихся пиктограмм.

- `static Border createEmptyBorder()`
- `static Border createEmptyBorder(int top, int left, int bottom, int right)`

Создает пустую рамку.

- `static Border createEtchedBorder()`
- `static Border createEtchedBorder(Color highlight, Color shadow)`
- `static Border createEtchedBorder(int type)`
- `static Border createEtchedBorder(int type, Color highlight, Color shadow)`

Создает рамку, имитирующую трехмерное изображение.

Параметры	`highlight`, `shadow`	Цвета для создания эффекта трехмерного изображения.
	`type`	Одна из констант: `EtchedBorder.RAISED` или `Etched.Border.LOWERED`.

- `static Border createBevelBorder(int type)`
- `static Border createBevelBorder(int type, Color highlight, Color shadow)`
- `static Border createLoveredBevelBorder()`
- `static Border createRaisedBevelBorder()`

Создает рамку, имитирующую утопленную или приподнятую поверхность.

Параметры		
	`type`	Одна из констант: `BevelBorder.RAISED` или `BevelBorder.LOWERED`.
	`highlight`, `shadow`	Цвета для создания эффекта трехмерного изображения.

- `static TitledBorder createTitledBorder(String title)`
- `static TitledBorder createTitledBorder(Border border)`
- `static TitleBorder createTitleBorder(Border Border, String title)`
- `static TitleBorder createTitleBorder(Border Border, String title, int justification, int position)`
- `static TitleBorder createTitleBorder(Border Border, String title, int justification, int position, Font font)`
- `static TitleBorder createTitleBorder(Border border, String title, int justification, int position, Font font, Color color)`

Создает рамку с заданными свойствами, снабженную заголовком.

Параметры		
	`title`	Строка заголовка.
	`border`	Рамка, снабженная заголовком.
	`justification`	Одна из констант: `LEFT`, `CENTER`, `RIGHT`, `LEADING`, `TRAILING` или `DEFAULT_JUSTIFICATION` (аналогично `LEFT`).
	`position`	Одна из констант класса `TitleBorder`: `ABOVE_TOP`, `TOP`, `BELOW_TOP`, `ABOVE_BOTTOM`, `BOTTOM`, `BELOW_BOTTOM` или `DEFAULT_POSITION` (аналогично `TOP`).
	`font`	Шрифт заголовка.
	`color`	Цвет заголовка.

- `static CompoundBorder createCompoundBorder(Border outsideBorder, Border inside Border)`

Объединяет две рамки в одно целое.

API javax.swing.border.SoftBevelBorder 1.2

- `SoftBevelBorder(int type)`
- `SoftBevelBorder(int type, Color highlight, color shadow)`

Создает фаску со сглаженными углами.

Параметры		
	`type`	Одна из констант: `BevelBorder.RAISED` или `BevelBorder.LOWERED`.
	`highlight`, `shadow`	Цвета для имитации трехмерного изображения.

API javax.swing.border.LineBorder 1.2

- `public LineBorder(Color color, int thickness, boolean roundedCorners)`

Создает рамку в виде линии заданной ширины и цвета. Если значение параметра `roundedCorners` равно `true`, рамка имеет скругленные углы.

API **`javax.swing.JComponent 1.2`**

- `void setBorder(Border border)`

  Задает рамку для компонента.

Выпадающие списки

Если вариантов для выбора слишком много, переключатели не подойдут, поскольку для них не хватит места на экране. В этом случае следует использовать выпадающие списки (combo box). Если пользователь щелкнет на этом компоненте, откроется список из нескольких пунктов и пользователь сможет выбрать один из них (рис. 9.17).



Рис. 9.17. Выпадающий список

Если выпадающий список является *редактируемым*, то строку, соответствующую выбранному пункту, *можно редактировать* подобно содержимому обычного текстового поля. Таким образом, редактируемый выпадающий список объединяет в себе гибкость текстового поля и возможность выбора из фиксированного набора возможностей, предусмотренных заранее. Выпадающие списки реализуются с помощью класса `JComboBox`.

Для того чтобы сделать выпадающий список редактируемым, следует вызвать метод `setEditable()`. Обратите внимание на то, что изменения вносятся только в текущий пункт. Остальная часть списка остается неизменной.

Выбранную опцию или редактируемый текст можно получить с помощью метода `getSelectedItem()`.

В программе, рассматриваемой здесь в качестве примера, пользователь может выбрать стиль шрифта из заданного заранее списка (Serif, SunsSerif, Monospaced и т.д.). Кроме того, пользователь может внести в список новый шрифт.

Добавить пункт в список можно с помощью метода `addItem()`. В нашей программе метод `addItem()` вызывается только в конструкторе, но при необходимости к нему можно обратиться из любого метода:

```
faceCombo = new JComboBox();
faceCombo.setEditable(true);
faceCombo.addItem("Serif");
faceCombo.addItem("SansSerif");
...
```

Этот метод добавляет строку в конец списка. Если строку нужно вставить в другую позицию, следует вызвать метод `insertItemAdd()`:

```
faceCombo.insertItemAt("Monospaced", 0);  // Вставка в начало списка.
```

В список можно включать элементы любого типа — для отображения их будет вызван метод `toString()`.

Если во время выполнения возникает необходимость удалить пункт списка, используются методы `removeItem()` или `removeItemAt()`, в зависимости от того, хотите ли вы удалить текущий элемент или тот, который находится в указанной позиции списка:

```
faceCombo.removeItem("Monospaced");
faceCombo.removeItemAt(0); // Удаляется первый элемент.
```

Для удаления сразу всех элементов списка предусмотрен метод `removeAllItems()`.



Совет. Если вам надо включить в выпадающий список большое количество объектов, не следует использовать метод `addItem()`, поскольку такой подход существенно снизит производительность программы. Вместо этого создайте объект `DefaultComboBoxModel`, заполните его, вызывая метод `addElement()`, а затем вызовите метод `setModel()` класса `JComboBox`.

Когда пользователь выбирает пункт выпадающего списка, компонент генерирует событие. Чтобы определить, какой пункт списка был выбран, необходимо вызвать метод `getSource()`, параметром которого является данное событие. Он возвращает ссылку на список, являющийся источником события. Затем следует вызвать метод `getSelectedItem()`, возвращающий выбранный пункт списка. Значение, возвращенное этим методом, нужно привести к соответствующему типу, обычно к типу `String`.

```
public void actionPerformed(ActionEvent event);
{
   label.setFont(new Font(
      (String)faceCombo.getSelectedtem(),
      Font.Plain,
      DEFAULT_SIZE));
}
```

Полный текст программы приведен в листинге 9.6.



На заметку! Если вы хотите, чтобы вместо выпадающего отображался обычный список, т.е. чтобы пункты списка постоянно отображались на экране, используйте компонент `JList`. Он будет рассмотрен во втором томе.

Листинг 9.6. Содержимое файла `ComboBoxTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import javax.swing.*;
4.
5.  /**
6.   * @version 1.33 2007-06-12
7.   * @author Cay Horstmann
8.   */
9.  public class ComboBoxTest
10. {
```

```
11.    public static void main(String[] args)
12.    {
13.       EventQueue.invokeLater(new Runnable()
14.          {
15.             public void run()
16.             {
17.
18.                ComboBoxFrame frame = new ComboBoxFrame();
19.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20.                frame.setVisible(true);
21.             }
22.          });
23.    }
24. }
25.
26. /**
27.  * Фрейм с текстовой меткой и выпадающим списком,
28.  * предназначенным для выбора начертаний.
29.  */
30. class ComboBoxFrame extends JFrame
31. {
32.    public ComboBoxFrame()
33.    {
34.       setTitle("ComboBoxTest");
35.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
36.
37.       // Добавление текстовой метки.
38.
39.       label = new JLabel("The quick brown fox jumps over the lazy dog.");
40.       label.setFont(new Font("Serif", Font.PLAIN, DEFAULT_SIZE));
41.       add(label, BorderLayout.CENTER);
42.
43.       // Создание выпадающего списка и включение
44.       // в него пунктов, соответствующих начертаниям.
45.
46.       faceCombo = new JComboBox();
47.       faceCombo.setEditable(true);
48.       faceCombo.addItem("Serif");
49.       faceCombo.addItem("SansSerif");
50.       faceCombo.addItem("Monospaced");
51.       faceCombo.addItem("Dialog");
52.       faceCombo.addItem("DialogInput");
53.
54.       // Слушатель событий, связанных с выпадающим списком,
55.       // изменяет шрифт, которым отображается метка.
56.
57.       faceCombo.addActionListener(new ActionListener()
58.       {
59.          public void actionPerformed(ActionEvent event)
60.          {
61.             label.setFont(new Font((String) faceCombo.getSelectedItem(),
62.                            Font.PLAIN, DEFAULT_SIZE));
63.          }
64.       });
65.
```

```
66.       // Включение выпадающего списка в состав панели
67.       // (панель располагается в нижней части фрейма).
68.
69.       JPanel comboPanel = new JPanel();
70.       comboPanel.add(faceCombo);
71.       add(comboPanel, BorderLayout.SOUTH);
72.    }
73.
74.    public static final int DEFAULT_WIDTH = 300;
75.    public static final int DEFAULT_HEIGHT = 200;
76.
77.    private JComboBox faceCombo;
78.    private JLabel label;
79.    private static final int DEFAULT_SIZE = 12;
80. }
```

API **javax.swing.JComboBox 1.2**

- `boolean isEditable()`
- `void setEditable(boolean b)`

  Получает или устанавливает свойство `editable` выпадающего списка.

- `void addItem(Object item)`

  Включает в список новый элемент.

- `void insertItem(Object item, int index)`

  Добавляет элемент в заданную позицию списка.

- `void removeItem(Object item)`

  Удаляет заданный элемент из списка.

- `void removeItem(int index)`

  Удаляет из списка элемент с заданным номером.

- `void removeAllItems()`

  Удаляет все элементы из списка.

- `Object getSelectedItem()`

  Возвращает выбранный элемент списка.

Линейные регуляторы

Выпадающие списки позволяют пользователю делать выбор из дискретного набора значений. Линейные регуляторы (ползунки) дают возможность выбрать конкретное значение из диапазона возможных, например, любое число из интервала 1–100.

Наиболее часто линейные регуляторы создаются следующим образом:

```
JSlider slider = new JSlider(min, max, initialValue);
```

Если не указывать минимальное, максимальное и начальное значение, принимаются соответственно величины 0, 100 и 50.

Если линейный регулятор должен располагаться вертикально, используется приведенный ниже конструктор:

```
JSlider slider = new JSlider(SwingConstants.VERTICAL, min, max, initialValue);
```

Каждый из этих конструкторов создает простой регулятор. В качестве примера можно привести регулятор, который расположен в верхней части окна, представленного на рис. 9.18. Позже мы рассмотрим более сложные варианты линейных регуляторов.

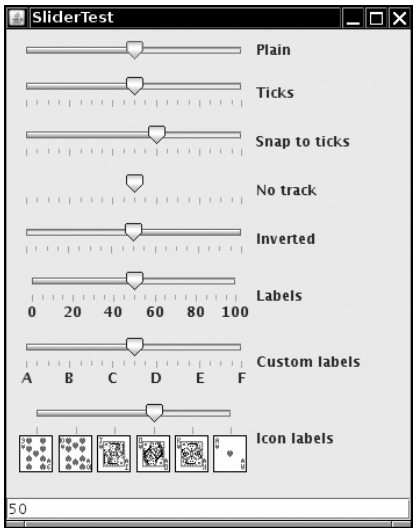


Рис. 9.18. Линейные регуляторы

Когда пользователь перемещает ползунок регулятора, *значение* компонента изменяется в пределах от минимального до максимального. Все слушатели, связанные с линейным регулятором, получают событие `ChangeEvent`. Для того чтобы получать извещения об изменении значения регулятора, нужно создать объект, реализующий интерфейс `ChangeListener`, и вызвать метод `addChangeListener()`. В этом интерфейсе объявлен лишь один метод `stateChanged()`. При его выполнении определяется значение регулятора.

```
public void stateChanged(ChangeEvent event)
{
   JSlider slider = (JSlider)event.getSource();
   int value = slider.getValue();
   ...
}
```

Линейный регулятор можно дополнить *делениями* (tic). Например, в программе, рассматриваемой в качестве примера, для второго регулятора производятся следующие установки:

```
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(5);
```

Регулятор снабжен основными делениями, следующими через каждые 20 единиц измерения, и вспомогательными, которые следуют через каждые 5 единиц. В данном случае единицы измерения связываются со значениями регулятора и не имеют никакого отношения к пикселям.

Приведенные выше выражения лишь устанавливают деления регулятора. Чтобы вывести их на экран, нужно вызвать следующий метод:

```
slider.setpaintTicks(true);
```

Основные и вспомогательные деления не зависят друг от друга. Например, можно задать расстановку основных делений через каждые 20 единиц, а вспомогательных — через 7 единиц, хотя такая шкала будет довольно неудобной.

Можно перевести регулятор в режим *привязки к делениям*. Как только пользователь отпустит ползунок регулятора, он тут же переместится на ближайшее деление. Чтобы задать такой режим работы, нужно вызвать следующий метод:

```
slider.setSnapToTicks(true);
```



Внимание! В режиме привязки к делениям компонент работает несколько неожиданно. Пока регулятор не установится точно на деление, обработчик событий изменений получает значения, не соответствующие делениям шкалы. Если же пользователь перемещает ползунок щелчком мыши, тот все равно не попадает на ближайшее деление.

С помощью приведенного ниже вызова можно задать *метки основных делений*:

```
slider.setPaintLabels(true);
```

Например, если значение регулятора изменяется в пределах от 0 до 100, а шаг основных делений равен 20, на экране появятся метки 0, 20, 40, 60, 80 и 100.

Можно задать и другие типы меток, например строки или пиктограммы (см. рис. 9.18). Эта задача не тривиальна. Сначала нужно заполнить хеш-таблицу с ключами типа `Integer` и значениями типа `Component` (автоматическое преобразование, реализованное в Java SE 5.0 и последующих версиях, упрощает этот процесс). Затем следует вызвать метод `setLabelTable()`. Компоненты размещаются под делениями. Обычно используются объекты класса `JLabel`. Ниже показано, как можно задать метки делений A, B, C, D, E и F.

```
Hashtable<Integer, Component> labelTable =
   new Hashtable<Integer, Component>();
labelTable.put(0, new JLabel("A"));
labelTable.put(20, new JLabel("B"));
. . .
labelTable.put(100, new JLabel("F"));
slider.setLabelTable(labelTable);
```

Более подробно хеш-таблицы описаны во втором томе.

В листинге 9.7 показан пример создания регулятора, у которого метками деления являются пиктограммы.



Совет. Если деления или их метки не выводятся на экран, проверьте, вызваны ли методы `setPaintTicks(true)` и `setpaintLabels(true)`.

Четвертый регулятор на рис. 9.18 не имеет “дорожки”. Подавить изображение “дорожки”, по которой двигается ползунок, можно с помощью вызова

```
slider.setPaintTrack(false);
```

Для пятого регулятора на этом рисунке направление движение изменено посредством следующего метода:

```
slider.setInverted(true);
```

Набор регуляторов, создаваемых в приведенном здесь примере, демонстрирует различные визуальные эффекты. Для каждого регулятора установлен обработчик событий, который отображает текущее значение выбранного регулятора в поле редактирования, расположенном в нижней части окна.

Листинг 9.7. Содержимое файла `SliderTest.java`

```
1.  import java.awt.*;
2.  import java.util.*;
3.  import javax.swing.*;
4.  import javax.swing.event.*;
5.
6.  /**
7.   * @version 1.13 2007-06-12
8.   * @author Cay Horstmann
9.   */
10. public class SliderTest
11. {
12.    public static void main(String[] args)
13.    {
14.       EventQueue.invokeLater(new Runnable()
15.          {
16.             public void run()
17.             {
18.                SliderTestFrame frame = new SliderTestFrame();
19.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20.                frame.setVisible(true);
21.             }
22.          });
23.    }
24. }
25.
26. /**
27.  * Фрейм с линейными регуляторами и полем редактирования,
28.  * отображающим текущее значение выбранного регулятора.
29.  */
30. class SliderTestFrame extends JFrame
31. {
32.    public SliderTestFrame()
33.    {
34.       setTitle("SliderTest");
35.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
36.
37.       sliderPanel = new JPanel();
38.       sliderPanel.setLayout(new FlowLayout(FlowLayout.LEFT));
39.
40.       // Общий слушатель для всех регуляторов.
41.       listener = new ChangeListener()
42.       {
43.          public void stateChanged(ChangeEvent event)
44.          {
45.             // Обновление поля редактирования при
46.             // изменении значения регулятора.
47.             JSlider source = (JSlider) event.getSource();
48.             textField.setText("" + source.getValue());
49.          }
50.       };
51.
52.       // Добавление простого регулятора.
53.
54.       JSlider slider = new JSlider();
55.       addSlider(slider, "Plain");
56.
```

```
57.      // Добавление регулятора с основными
58.      // и вспомогательными делениями.
59.
60.      slider = new JSlider();
61.      slider.setPaintTicks(true);
62.      slider.setMajorTickSpacing(20);
63.      slider.setMinorTickSpacing(5);
64.      addSlider(slider, "Ticks");
65.
66.      // Добавление регулятора с привязкой к делениям.
67.
68.      slider = new JSlider();
69.      slider.setPaintTicks(true);
70.      slider.setSnapToTicks(true);
71.      slider.setMajorTickSpacing(20);
72.      slider.setMinorTickSpacing(5);
73.      addSlider(slider, "Snap to ticks");
74.
75.      // Добавление регулятора без "дорожки"
76.
77.      slider = new JSlider();
78.      slider.setPaintTicks(true);
79.      slider.setMajorTickSpacing(20);
80.      slider.setMinorTickSpacing(5);
81.      slider.setPaintTrack(false);
82.      addSlider(slider, "No track");
83.
84.      // Добавление регулятора с обратным направлением движения.
85.
86.      slider = new JSlider();
87.      slider.setPaintTicks(true);
88.      slider.setMajorTickSpacing(20);
89.      slider.setMinorTickSpacing(5);
90.      slider.setInverted(true);
91.      addSlider(slider, "Inverted");
92.
93.      // Добавление регулятора с числовыми метками.
94.
95.      slider = new JSlider();
96.      slider.setPaintTicks(true);
97.      slider.setPaintLabels(true);
98.      slider.setMajorTickSpacing(20);
99.      slider.setMinorTickSpacing(5);
100.     addSlider(slider, "Labels");
101.
102.     // Добавление регулятора с буквенными метками.
103.
104.     slider = new JSlider();
105.     slider.setPaintLabels(true);
106.     slider.setPaintTicks(true);
107.     slider.setMajorTickSpacing(20);
108.     slider.setMinorTickSpacing(5);
109.
110.     Dictionary<Integer, Component> labelTable = new Hashtable<Integer, Component>();
111.     labelTable.put(0, new JLabel("A"));
112.     labelTable.put(20, new JLabel("B"));
113.     labelTable.put(40, new JLabel("C"));
114.     labelTable.put(60, new JLabel("D"));
```

```
115.     labelTable.put(80, new JLabel("E"));
116.     labelTable.put(100, new JLabel("F"));
117.
118.     slider.setLabelTable(labelTable);
119.     addSlider(slider, "Custom labels");
120.
121.     // Добавление регулятора с метками в виде пиктограмм.
122.
123.     slider = new JSlider();
124.     slider.setPaintTicks(true);
125.     slider.setPaintLabels(true);
126.     slider.setSnapToTicks(true);
127.     slider.setMajorTickSpacing(20);
128.     slider.setMinorTickSpacing(20);
129.
130.     labelTable = new Hashtable<Integer, Component>();
131.
132.     // Добавление изображений карт.
133.
134.     labelTable.put(0, new JLabel(new ImageIcon("nine.gif")));
135.     labelTable.put(20, new JLabel(new ImageIcon("ten.gif")));
136.     labelTable.put(40, new JLabel(new ImageIcon("jack.gif")));
137.     labelTable.put(60, new JLabel(new ImageIcon("queen.gif")));
138.     labelTable.put(80, new JLabel(new ImageIcon("king.gif")));
139.     labelTable.put(100, new JLabel(new ImageIcon("ace.gif")));
140.
141.     slider.setLabelTable(labelTable);
142.     addSlider(slider, "Icon labels");
143.
144.     // Добавление поля редактирования для
145.     // отображения значения регулятора.
146.
147.     textField = new JTextField();
148.     add(sliderPanel, BorderLayout.CENTER);
149.     add(textField, BorderLayout.SOUTH);
150.   }
151.
152.   /**
153.    * Добавление регулятора в панель и связывание его со слушателем.
154.    * @param s Регулятор
155.    * @param description Описание регулятора
156.    */
157.   public void addSlider(JSlider s, String description)
158.   {
159.     s.addChangeListener(listener);
160.     JPanel panel = new JPanel();
161.     panel.add(s);
162.     panel.add(new JLabel(description));
163.     sliderPanel.add(panel);
164.   }
165.
166.   public static final int DEFAULT_WIDTH = 350;
167.   public static final int DEFAULT_HEIGHT = 450;
168.
169.   private JPanel sliderPanel;
170.   private JTextField textField;
171.   private ChangeListener listener;
172.}
```

API javax.swing.JSlider 1.3

- `JSlider()`
- `JSlider(int direction)`
- `JSlider(int min, int max)`
- `JSlider(int min, int max, ini initialValue)`
- `JSlider(int direction, int min, int max, ini initialValue)`

Создает линейный регулятор с заданным направлением, минимальным и максимальным значениями.

Параметры `direction` Одна из констант: `SwingConstantc.HORIZONTAL` или `swingConstants.VERTICAL`. По умолчанию принимается значение `SwingConstantc.HORIZONTAL`.

`min, max` Минимальное и максимальное значения регулятора. По умолчанию равны 0 и 100.

`initialValue` Начальное значение регулятора. По умолчанию равно 50.

- `void setPaintTicks(boolean b)`

Если значение параметра `b` равно `true`, деления отображаются на экране.

- `void setMajorTickSpacing(int units)`
- `void setMinorTickSpacing(int units)`

Устанавливают основную и вспомогательную шкалы.

- `void setPaintLabels(boolean b)`

Если значение параметра `b` равно `true`, деления отображаются на экране.

- `slider.setlabelTable(Dictionary table)`

Устанавливает компоненты, которые используются в качестве меток делений. Каждая пара "ключ–значение" представлена в виде `new Integer(`*значение*`)` / *компонент*.

- `void setSnapToTicks(boolean b)`

Если значение параметра `b` равно `true`, ползунок регулятора всегда перемещается на деление, ближайшее к установленному значению регулятора.

- `void setPaintTrack(boolean b)`

Если значение параметра равно `true`, на экране отображается "дорожка", по которой перемещается ползунок.

Меню

В начале этой главы мы описали наиболее распространенные компоненты, которые можно разместить в окне, такие как разнообразные кнопки, поля редактирования и выпадающие списки. В пакете Swing предусмотрены также выпадающие меню, хорошо известные пользователям, которые когда-либо работали с программами с графическим интерфейсом.

Строка меню в верхней части окна содержит названия выпадающих меню. Щелкая на имени, пользователь открывает меню, состоящее из *пунктов* и *подменю*. Если пользователь щелкнет на пункте меню, все меню закроются, и в программу будет передано сообщение. На рис. 9.19 показано типичное меню, содержащее пункты и подменю.



Рис. 9.19. Меню и подменю

Построение меню

Создать меню достаточно просто. Сначала создается строка меню.

```
JMenuBar menuBar = new JMenuBar();
```

Строка меню — это обычный компонент, который можно разместить где угодно. Как правило, его помещают в верхнюю часть фрейма с помощью метода `setJMenuBar()`:

```
frame.setJMenuBar(menuBar);
```

Для каждого меню создается свой объект:

```
Jmenu editMenu = new Jmenu("Edit");
```

Меню верхнего уровня помещаются в строке меню:

```
menuBar.add(editMenu);
```

Затем в объект меню добавляются пункты, разделители и подменю:

```
JMenuItem pasteItem = new JMenuItem("Paste");
editMenu.add(pasteItem);
editMenu.addSeparator();
Jmenu optionsMenu = ...; // Подменю.
editMenu.add(optionsMenu);
```

Разделители показаны на рис. 9.19. Они отображаются под пунктами Paste (Вставка) и Read-only (Только для чтения).

Когда пользователь выбирает меню, генерируются события. Следовательно, для каждого пункта меню надо определить обработчик:

```
ActionListener listener = ...;
pasteItem.addActionListener(listener);
```

Существует удобный метод `JMenu.add(String)`, позволяющий добавлять новый пункт в конец меню, например, следующим образом:

```
editMenu.add("Paste");
```

Этот метод возвращает созданный пункт меню, для которого можно легко задать обработчик:

```
JMenuItem pasteItem = editMenu.add("Paste");
pasteItem.addActionListener(listener);
```

Довольно часто пункты меню связываются с командами, которые могут активизировать другие элементы пользовательского интерфейса, например, кнопки. В главе 8 говорилось о том, как задаются команды с помощью объектов `Action`. Следует определить класс, реализующий интерфейс `Action`. Обычно для этого класс расширяет `AbstractAction`. Затем в конструкторе `AbstractAction` указывается метка пункта

меню и переопределяется метод `actionPerformed()`, что позволяет реализовать обработку события, связанного с данным пунктом. Например:

```
Action exitAction = new
   AbstractAction("Exit") // Пункт меню "Exit".
   {
     public void actionPerformed(ActionEvent event)
     {
        // Фрагмент программы, обрабатывающий событие.
        System.exit(0);
     }
};
```

Затем объект `Action` добавляется в меню:

```
JMenuItem exitItem = fileMenu.add(exitAction);
```

В результате выполнения данного выражения в меню добавляется новый пункт. Объект `Action` становится обработчиком. Такой способ заменяет следующие строки кода:

```
JMenuItem exitItem = new JMenuItem(exitAction);
fileMenu.add(exitItem);
```



На заметку! В программах для Windows и Macintosh меню обычно определяются во внешних файлах ресурсов и связываются с приложениями с помощью идентификаторов ресурсов. Меню можно создавать непосредственно в программе, однако, как правило, так не поступают. В Java меню создаются обычным образом, поскольку возможности взаимодействия с внешними ресурсами еще более ограничены, по сравнению с теми, которые предоставляют среды Windows и Mac OS.



`javax.swing.JMenu 1.2`

- `JMenu(String label)`

  Создает меню с указанной меткой.

- `JMenuItem add(JMenuItem item)`

  Добавляет пункт (или целое меню).

- `JMenuItem add(String label)`

  Добавляет пункт в меню с указанной меткой и возвращает этот пункт меню.

- `JMenuItem add(Action a)`

  Добавляет пункт и связанное с ним действие и возвращает этот пункт.

- `void addSeparator()`

  Добавляет в меню разделитель.

- `JMenuItem insert(JMenuItem menu, int index)`

  Добавляет в указанную позицию меню новый пункт (или подменю).

- `JMenuItem insert(Action a, int index)`

  Добавляет в указанную позицию меню новый пункт и связанный с ним объект `Action`.

- `void insertSeparator(int index)`

  Добавляет в меню разделитель.

- `void remove(int index)`
- `void remove(JMenuItem item)`

  Удаляет указанный пункт меню.

API javax.swing.JMenuItem 1.2

- `JMenuItem(String label)`

  Создает пункт меню с указанной меткой.

- `JMenuItem(Action a)`

  Создает пункт меню и связывает с ним указанное действие.

API javax.swing.AbstractButton 1.2

- `void setAction(Action a)` **1.3**

  Устанавливает действие для кнопки или пункта меню.

API javax.swing.JFrame 1.2

- `void setJMenuBar(JMenuBar menubar)`

  Задает строку меню в указанном фрейме.

Пиктограммы в пунктах меню

Пункты меню очень похожи на кнопки. Фактически класс `JMenuItem` расширяет класс `AbstractButton`. Как и кнопки, меню могут иметь текстовую метку, пиктограмму или и то и другое одновременно. Пиктограмму можно указать в конструкторе `JMenuItem(String, Icon)` или `JmenuItem(Icon)`, либо задать с помощью метода `setIcon()`, унаследованного классом `JMenuItem` от класса `AbstractButton`. Вот пример:

```
JMenuItem cutItem = new JMenuItem("Cut", new ImageIcon("cut.gif"));
```

На рис. 9.19 показано меню с пиктограммами. По умолчанию названия пунктов меню располагаются справа от пиктограмм. Если вам хочется, чтобы справа находились пиктограммы, используйте метод `setHorizontalTextPosition()`, унаследованный классом `JMenuItem` от `AbstractButton`. Например, приведенный ниже оператор перемещает текст пункта меню влево от пиктограммы.

```
cutItem.setHorizontalTextPosition(SwingConstants.LEFT);
```

Пиктограмму можно связать с объектом `Action`:

```
cutAction.putValue(Action.SMALL_ICON, new ImageIcon("Cut.gif"));
```

Если пункт меню создается независимо от `Action`, то значением поля `Action.NAME` становится название пункта меню, а значением поля `Action.SMALL_ICON` — его пиктограмма. Кроме того, пиктограмму можно задать в конструкторе класса `AbstractAction`:

```
cutAction = new AbstractAction("Cut", new ImageIcon("cut.gif"))
   {
     public void actionPerformed(ActionEvent event)
     {
        // Фрагмент, реализующий обработку.
     }
   };
```

API javax.swing.JMenuItem 1.2

- `JMenuItem(String label, Icon icon)`

  Создает пункт меню с указанной меткой и пиктограммой.

API javax.swing.AbstractButton 1.2

- `void setHorizontalTextPosition(int pos)`

  Задает взаимное расположение текста и пиктограммы.

  Параметр `pos` Константа `SwingConstants.RIGHT` (текст располагается справа от пиктограммы) или `SwingConstants.LEFT`.

API javax.swing.AbstractAction 1.2

- `AbstractAction(String name, Icon smallIcon)`

  Создает объект `AbstractAction` с указанным именем и пиктограммой.

Флажки опций и переключатели в меню

Пункты меню, представляющие собой *флажки опций* или *переключатели*, показаны на рис. 9.19. Когда пользователь щелкает на пункте меню, флажки автоматически устанавливаются или сбрасываются, а состояние переключателей изменяется в соответствии с выбранным пунктом.

Если не принимать во внимание внешний вид, флажки и опции можно рассматривать как обычные пункты меню. Рассмотрим, например, как создается пункт меню с флажком опции.

```
JCheckBoxMenuItem readonlyItem = new JCheckBoxMenuItem("Read-only");
optionsMenu.add(readonlyItem);
```

Пункты меню с переключателями работают точно так же, как и обычные переключатели. В меню нужно добавить группу переключателей. Когда выбирается один из них, все остальные автоматически отключаются.

```
ButtonGroup group = new ButtonGroup();
JRadioButtonMenuItem insertItem = JRadioButtonMenuItem("Insert");
insertItem.setSelected(true);
JRadioButtonMenuItem overtypeItem = new JRadioButtonMenuItem("Overtype");
group.add(insertItem);
group.add(overtypeItem);
optionsMenu.add(insertItem);
optionsMenu.add(overtypeItem);
```

В этих пунктах меню не обязательно точно определять, когда именно пользователь сделал выбор. Вместо этого для проверки текущего состояния пункта меню достаточно применить метод `isSelected()`. (Разумеется, это означает, что в какой-то переменной экземпляра придется хранить ссылку на данный пункт меню.) Кроме того, задать состояние пункта меню можно с помощью метода `setSelected()`.

API javax.swing.JCheckBoxMenuItem 1.2

- `JCheckBoxMenuItem(String label)`

  Создает пункт меню, представляющий собой флажок опции с заданной меткой.

- `JCheckBoxMenuItem(String label, boolean state)`

  Создает пункт меню, представляющий собой флажок опции с заданными меткой и состоянием (если пункт выбран, значение переменной `state` равно `true`).

API javax.swing.JRadioButtonMenuItem 1.2

- `JRadioButtonMenuItem(String label)`

  Создает пункт меню, представляющий собой переключатель с заданной меткой.

- `JRadioButtonMenuItem(String label, boolean state)`

  Создает пункт меню, представляющий собой переключатель с заданными меткой и состоянием (если пункт выбран, значение переменной `state` равно `true`).

API javax.swing.AbstractButton 1.2

- `boolean isSelected()`

  Возвращает состояние пункта (значение `true`, если пункт выбран).

- `void setSelected(bboolean state)`

  Задает состояние пункта.

Всплывающие меню

Всплывающие, или *контекстные, меню* (pop-up menu) — это меню, не связанные со строкой, а отображающиеся в произвольно выбранной позиции на экране (рис. 9.20).

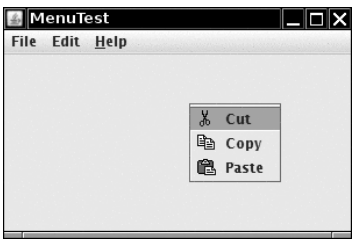


Рис. 9.20. Всплывающее меню

Всплывающее меню создается точно так же, как и обычное, за одним исключением — оно не имеет заголовка:

```
JpopupMenu popup = new JPopupMenu();
```

Пункты меню добавляются, как обычно:

```
JmenuItem item = new JmenuItem("Cut");
item.addActionListener(listener);
popup.add(item);
```

В отличие от строки меню, которая всегда находится в верхней части фрейма, всплывающее меню нужно принудительно выводить на экран с помощью метода `show()`. При этом задается родительский компонент и расположение всплывающего меню в его системе координат.

```
popup.show(panel, x, y);
```

Обычно всплывающее меню отображается на экране, когда пользователь щелкает специально предназначенной для этого кнопкой — так называемым *триггером всплывающего меню* (pop-up trigger). В системах Windows и Linux это не первая (обычно правая) кнопка мыши. Для вызова меню после щелчка кнопкой вызывается следующий метод:

```
component.setComponentPopupMenu(popup);
```

Часто приходится размещать компонент внутри другого компонента, с которым связано всплывающее меню. Для того чтобы дочерний компонент наследовал меню родительского компонента, надо вызвать метод

```
child.setInheritsPopupMenu(true);
```

Приведенные выше методы были реализованы в Java SE 5.0 для того, чтобы исключить зависимость программиста от особенностей работы с контекстным меню в конкретной системе. В предшествующих версиях Java SE приходилось устанавливать обработчик событий, связанных с мышью, и задавать представленный ниже код как в методе `mousePressed()`, так и в методе `mouseReleased()`:

```
if (popup.isPopupTrigger(event))
  popup.show(event.getComponent(), event.getX(), event.getY());
```

Причина в том, что в некоторых системах всплывающее меню отображается при нажатии кнопки мыши, а в других — при отпускании кнопки.

API javax.swing.JPopupMenu 1.2

- `void show(Component c, int x, int y)`

  Отображает всплывающее меню.

  Параметры `c` Компонент, соответствующий всплывающему меню.

  `x,y` Координаты (в системе компонента `c`) левого верхнего угла всплывающего меню.

- `boolean isPopupTrigger(MouseEvent event)` **1.3**

  Возвращает значение `true`, если событие, связанное с мышью, соответствует вызову всплывающего меню.

API java.awt.event.MouseEvent 1.1

- `boolean isPopupTrigger()`

  Возвращает значение `true`, если событие, связанное с мышью, соответствует вызову всплывающего меню.

API javax.swing.JComponent 1.2

- `JPopup getComponentPopupMenu()` **5.0**
- `void setComponentPopupMenu(JPopupMenu popup)` **5.0**

  Устанавливает или возвращает всплывающее меню, связанное с компонентом.

- `void setInheritsPopupMenu(boolean b)` **5.0**
- `boolean getInheritsPopupMenu()` **5.0**

  Устанавливает или возвращает свойство `inheritsPopupMenu`. Если данное свойство установлено, а вместо всплывающего меню компоненту соответствует значение `null`, используется меню родительского компонента.

Клавиши быстрого доступа и клавиатурные сокращения

Опытному пользователю удобно выбирать пункты меню с помощью *клавиш быстрого доступа* (keyboar mnemonics). Связать пункт меню с клавишей быстрого доступа можно, задав эту клавишу в конструкторе пункта:

```
JMenuItem cutItem = new JmenuItem("About", "A");
```

Буква в названии пункта меню, соответствующая клавише быстрого доступа выделяется подчеркиванием (рис. 9.21). Например, если клавиша быстрого доступа, задана с помощью приведенного выше выражения, то метка будет отображена как About, т.е. с подчеркнутой буквой "A". Теперь для выбора этого пункта меню пользователю достаточно нажать клавишу <A>. (Если буква, соответствующая назначенной клавише, не входит в название пункта меню, она не отображается на экране, но при ее нажатии этот пункт все равно будет выбран. Естественно, польза от таких "невидимых" клавиш сомнительна.)



Рис. 9.21. Так выглядят пункты меню, для которых назначены клавиши быстрого доступа

Задавая клавишу быстрого доступа, не всегда целесообразно выделять первое вхождение соответствующей буквы в название пункта меню. Так, например, если для пункта Save As (Сохранить) назначена клавиша <A>, то гораздо уместнее выделить подчеркиванием букву "A" в слове "As" (Save As). Начиная с Java SE 1.4, можно указывать символ, который должен быть выделен, с помощью метода `setDisplayedMnemonicIndex()`.

Имея объект класса `Action`, клавишу быстрого доступа можно назначить с помощью значения поля `Action.MNEMONIC_KEY`:

```
cutAction.putValue(Action.MNEMONIC_KEY, new Integer('A'));
```

Букву, соответствующую клавише быстрого доступа, следует задавать только в конструкторе пункта меню (но не в конструкторе меню в целом). Чтобы связать какую-либо клавишу с меню в целом, нужно вызвать метод `setMnemonic()`:

```
JMenu helpMenu = new JMenu("Help");
helpMenu.setMnemonic('H');
```

Теперь, чтобы выбрать меню в строке, нужно нажать клавишу <Alt> вместе с клавишей назначенной буквы. Например, чтобы выбрать меню Help (Справка), нужно нажать клавиши <Alt+H>.

Клавиши быстрого доступа позволяют выбрать пункт в уже открытом меню. В противоположность этому, *клавиатурные сокращения* (accelerator) позволяют выбрать пункт, не открывая меню. Например, во многих программах предусмотрены сочетания клавиш <Ctrl+O> и <Ctrl+S> для пунктов Open (Открыть) и Save (Сохранить) меню File (Файл). Для связывания сочетания клавиш с пунктом меню используется метод `setAccelerator()`. В качестве параметра он получает объект `Keystroke`. Например, приведенный ниже вызов назначает клавиатурное сокращение <Ctrl+O> для пункта меню `openItem`.

```
openItem.setAccelerator(KeyStroke.getKeyStroke("Ctrl O"));
```

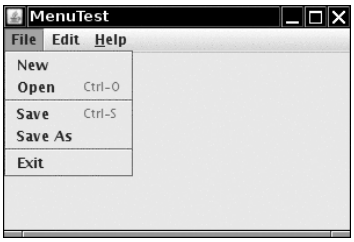


Рис. 9.22. Клавиатурные сокращения

Когда пользователь нажимает требуемое сочетание клавиш, он автоматически выбирает соответствующий пункт меню и генерирует событие, так же, как если бы он выбирал пункт меню обычным способом. Клавиатурные сокращения можно связывать только с пунктами меню, но не с меню в целом. Они не открывают меню, а лишь генерируют событие, связанное с указанным пунктом.

В принципе, клавиатурные сокращения для пунктов меню добавляются так же, как и для остальных компонентов пакета Swing. (Мы обсуждали эту технику в главе 8.) Однако если клавиатурное сокращение назначено пункту меню, соответствующая клавиатурная комбинация автоматически отображаются в меню (рис. 9.22).



На заметку! В операционной системе Windows сочетание клавиш <Alt+F4> закрывает окно. Однако сочетание не имеет отношение к языку Java. Это клавиатурное сокращение определено в операционной системе и всегда генерирует событие `WindowClosing` для активного окна, независимо от того, есть ли в меню пункт Close.



javax.swing.JMenuItem 1.2

- `JMenuItem(String label, int mnemonic)`

  Создает пункт меню с указанной меткой и клавишей быстрого доступа.

  Параметры `label` Метка пункта меню.

  `mnemonic` Мнемоническое обозначение пункта, соответствующее клавише быстрого доступа; в метке этот символ подчеркивается.

- `void setAccelerator(Keystroke k)`

  Задает клавиатурное сокращение `k` для указанного пункта меню. Соответствующее сочетание клавиш отображается в меню рядом с меткой.



javax.swing.AbstractButton 1.2

- `void setMnemonic(char mnemonic)`

  Назначает клавишу быстрого доступа для кнопки. В метке кнопки этот символ подчеркивается.

- `void setDisplayedMnemonicIndex(int index)` **1.4**

  Задает расположение символа, который должен быть выделен подчеркиванием. Данный метод вызывается в том случае, если выделять первое вхождение символа, соответствующего клавише быстрого доступа, нецелесообразно.

Блокирование и разблокирование пунктов меню

Бывают случаи, когда некоторые пункты меню должны выбираться лишь в определенных ситуациях. Например, если документ открыт лишь для чтения, то пункт меню Save не имеет смысла. Разумеется, этот пункт можно удалить с помощью метода `JMenu.remove()`, но пользователя может удивить постоянно изменяющееся меню. Лучше все же заблокировать пункты меню, временно лишив пользователя возможности выполнять ряд операций. На рис. 9.23 заблокированный пункт меню выделен светло-серым цветом.

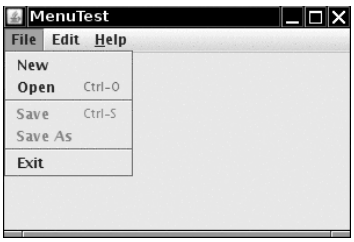


Рис. 9.23. Заблокированные пункты меню

Для того чтобы разблокировать или заблокировать пункт меню, используют метод `setEnabled()`:

```
saveItem.setEnabled(false);
```

Существуют две стратегии блокирования и разблокирования пунктов меню. При каждом изменении состояния программы можно вызывать метод `setEnabled()`, применяя его к соответствующему пункту меню. Например, открыв документ только для чтения, можно заблокировать пункты Save и Save As (Сохранить как). Однако, пользуясь этой стратегией, программист рискует "засорить" код фрагментами, предназначенными для изменения состояния пунктов меню. Гораздо лучше совсем не беспокоиться о состоянии пунктов меню в основной части программы и *задавать их параметры непосредственно перед отображением на экране*. Для этого нужно зарегистрировать обработчик событий, связанных с выбором меню. В пакете `javax.swing.event` содержится интерфейс `MenuListener`, в котором объявлены три метода:

```
void menuSelected(MenuEvent event)
void menuDeselected(MenuEvent event)
void menuCanceled(MenuEvent event)
```

Метод `menuSelected()` вызывается *перед* отображением меню на экране. Это самый подходящий момент для того, чтобы заблокировать или разблокировать пункты меню. В приведенном ниже фрагменте кода показано, как блокируются и разблокируются пункты Save и Save As, в зависимости от состояния флажка Read Only.

```
public void menuSelected(MenuEvent event)
{
   saveAction.setEnabled(!readonlyItem.isSelected());
   saveAsAction.setEnabled(!readonlyItem.isSelected());
}
```



Внимание! Блокировка пунктов меню непосредственно перед его отображением на экране — отличная идея, однако она не работает для пунктов меню, связанных с клавиатурными сокращениями. При нажатии соответствующего сочетания клавиш меню не открывается, поэтому пункт меню не будет заблокирован.

API javax.swing.JMenuItem 1.2

- `void setEnabled(boolean b)`

  Блокирует и разблокирует пункт меню.

API javax.swing.event.MenuListener 1.2

- `void menuSelected(menuEvent e)`

  Вызывается, когда меню уже выбрано, но еще не отображается на экране.

- `void menuDeselected(menuEvent e)`

  Вызывается, когда меню уже закрыто.

- `void menuCanceled(menuEvent e)`

  Вызывается, когда действия с меню отменены, например, если пользователь щелкнул мышью за пределами меню.

В листинге 9.8 приведен пример программы, генерирующей набор меню. В нем продемонстрированы все особенности меню, описанные в этом разделе: вложенные меню, заблокированные пункты, флажки опций и переключатели в пунктах меню, а также клавиши быстрого доступа и клавиатурные сокращения.

Листинг 9.8. Содержимое файла `MenuTest.java`

```
1.  import java.awt.EventQueue;
2.  import java.awt.event.*;
3.  import javax.swing.*;
4.
5.  /**
6.   * @version 1.23 2007-05-30
7.   * @author Cay Horstmann
8.   */
9.  public class MenuTest
10. {
11.    public static void main(String[] args)
12.    {
13.       EventQueue.invokeLater(new Runnable()
14.          {
15.             public void run()
16.             {
17.                MenuFrame frame = new MenuFrame();
18.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.                frame.setVisible(true);
20.             }
21.          });
22.    }
23. }
24.
25. /**
26.  * Фрейм со строкой меню.
27.  */
28. class MenuFrame extends JFrame
29. {
```

```
30.    public MenuFrame()
31.    {
32.       setTitle("MenuTest");
33.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34.
35.       JMenu fileMenu = new JMenu("File");
36.       fileMenu.add(new TestAction("New"));
37.
38.       // Клавиатурные сокращения.
39.
40.       JMenuItem openItem = fileMenu.add(new TestAction("Open"));
41.       openItem.setAccelerator(KeyStroke.getKeyStroke("ctrl O"));
42.
43.       fileMenu.addSeparator();
44.
45.       saveAction = new TestAction("Save");
46.       JMenuItem saveItem = fileMenu.add(saveAction);
47.       saveItem.setAccelerator(KeyStroke.getKeyStroke("ctrl S"));
48.
49.       saveAsAction = new TestAction("Save As");
50.       fileMenu.add(saveAsAction);
51.       fileMenu.addSeparator();
52.
53.       fileMenu.add(new AbstractAction("Exit")
54.       {
55.          public void actionPerformed(ActionEvent event)
56.          {
57.             System.exit(0);
58.          }
59.       });
60.
61.       // Флажки опций и переключатели в составе меню.
62.
63.       readonlyItem = new JCheckBoxMenuItem("Read-only");
64.       readonlyItem.addActionListener(new ActionListener()
65.       {
66.          public void actionPerformed(ActionEvent event)
67.          {
68.             boolean saveOk = !readonlyItem.isSelected();
69.             saveAction.setEnabled(saveOk);
70.             saveAsAction.setEnabled(saveOk);
71.          }
72.       });
73.
74.       ButtonGroup group = new ButtonGroup();
75.
76.       JRadioButtonMenuItem insertItem = new JRadioButtonMenuItem("Insert");
77.       insertItem.setSelected(true);
78.       JRadioButtonMenuItem overtypeItem = new JRadioButtonMenuItem("Overtype");
79.
80.       group.add(insertItem);
81.       group.add(overtypeItem);
82.
83.       // Пиктограммы в составе меню.
84.
```

```
85.       Action cutAction = new TestAction("Cut");
86.       cutAction.putValue(Action.SMALL_ICON, new ImageIcon("cut.gif"));
87.       Action copyAction = new TestAction("Copy");
88.       copyAction.putValue(Action.SMALL_ICON, new ImageIcon("copy.gif"));
89.       Action pasteAction = new TestAction("Paste");
90.       pasteAction.putValue(Action.SMALL_ICON, new ImageIcon("paste.gif"));
91.
92.       JMenu editMenu = new JMenu("Edit");
93.       editMenu.add(cutAction);
94.       editMenu.add(copyAction);
95.       editMenu.add(pasteAction);
96.
97.       // Вложенные меню.
98.
99.       JMenu optionMenu = new JMenu("Options");
100.
101.      optionMenu.add(readonlyItem);
102.      optionMenu.addSeparator();
103.      optionMenu.add(insertItem);
104.      optionMenu.add(overtypeItem);
105.
106.      editMenu.addSeparator();
107.      editMenu.add(optionMenu);
108.
109.      // Клавиши быстрого доступа.
110.
111.      JMenu helpMenu = new JMenu("Help");
112.      helpMenu.setMnemonic('H');
113.
114.      JMenuItem indexItem = new JMenuItem("Index");
115.      indexItem.setMnemonic('I');
116.      helpMenu.add(indexItem);
117.
118.      // Задать клавишу быстрого доступа можно также
119.      // с помощью объекта Action.
120.      Action aboutAction = new TestAction("About");
121.      aboutAction.putValue(Action.MNEMONIC_KEY, new Integer('A'));
122.      helpMenu.add(aboutAction);
123.
124.      // Включение меню верхнего уровня в строку меню.
125.
126.      JMenuBar menuBar = new JMenuBar();
127.      setJMenuBar(menuBar);
128.
129.      menuBar.add(fileMenu);
130.      menuBar.add(editMenu);
131.      menuBar.add(helpMenu);
132.
133.      // Всплывающее меню.
134.
135.      popup = new JPopupMenu();
136.      popup.add(cutAction);
137.      popup.add(copyAction);
138.      popup.add(pasteAction);
139.
```

```
140.      JPanel panel = new JPanel();
141.      panel.setComponentPopupMenu(popup);
142.      add(panel);
143.
144.      // Следующая строка необходима из-за ошибки 4966109
145.      panel.addMouseListener(new MouseAdapter()
146.      {
147.      });
148.   }
149.
150.   public static final int DEFAULT_WIDTH = 300;
151.   public static final int DEFAULT_HEIGHT = 200;
152.
153.   private Action saveAction;
154.   private Action saveAsAction;
155.   private JCheckBoxMenuItem readonlyItem;
156.   private JPopupMenu popup;
157.}
158.
159./**
160. * Обработчик события, который выводит
161. * в System.out имя действия.
162. */
163.class TestAction extends AbstractAction
164.{
165.   public TestAction(String name)
166.   {
167.      super(name);
168.   }
169.
170.   public void actionPerformed(ActionEvent event)
171.   {
172.      System.out.println(getValue(Action.NAME) + " selected.");
173.   }
174.}
```

Панели инструментов

Панель инструментов представляет собой набор кнопок, обеспечивающих быстрый доступ к наиболее часто используемым командам (рис. 9.24).

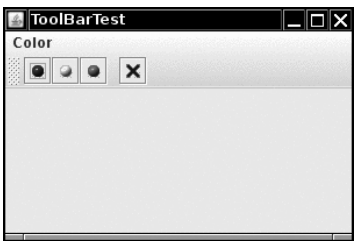


Рис. 9.24. Панель инструментов

Что отличает панель инструментов от остальных элементов пользовательского интерфейса? Ее можно перетаскивать на любую из четырех сторон фрейма (рис. 9.25). При отпускании кнопки мыши панель инструментов зафиксируется на новом месте (рис. 9.26).



На заметку! Перетаскивание панели инструментов допускается, только если она находится в контейнере, управляемом диспетчером компоновки `BorderLayout` или любым другим диспетчером, поддерживающим области `North`, `East`, `South` и `West`.

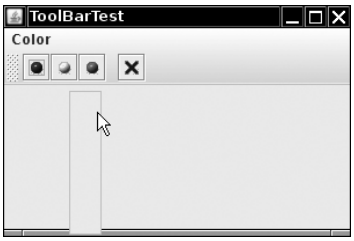


Рис. 9.25. Перетаскивание панели инструментов

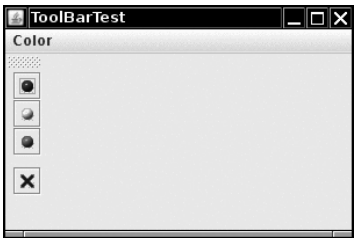


Рис. 9.26. В результате перетаскивания панель инструментов размещена вдоль левой стороны фрейма

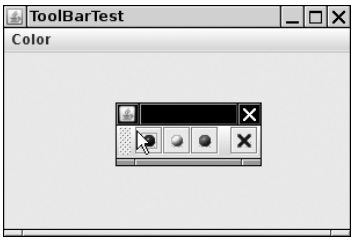


Рис. 9.27. Независимая панель инструментов

Панель инструментов может быть даже независимой от фрейма. Такая панель содержится в своем собственном фрейме (рис. 9.27). Если пользователь закрывает фрейм, содержащий независимую панель инструментов, она перемещается в исходный фрейм.

Панель инструментов легко программировать. Ниже приведен пример создания панели и добавления к ней компонента.

```
JToolBar bar = new JToolBar();
bar.add(blueButton);
```

В классе `JToolBar` также есть методы, предназначенные для добавления объекта класса `Action`. Просто наполните панель инструментов объектами действий, как показано ниже:

```
bar.add(blueAction);
```

Пиктограмма, соответствующая такому объекту, отображается в панели инструментов. Группы кнопок можно отделять друг от друга с помощью разделителей:

```
bar.addSeparator();
```

Например, d панели инструментов, показанной на рис. 9.24, присутствует разделитель, отделяющий третью кнопку от четвертой.

Панель инструментов помещается в контейнер:

```
add(bar, BorderLayout.NORTH);
```

Можно также указать заголовок панели, который появится, когда панель будет отображаться независимо от фрейма:

```
bar = new JToolBar(titleString);
```

По умолчанию панель инструментов располагается горизонтально. Чтобы сделать ее вертикальной, нужно использовать оператор

```
bar = new JToolBar(SwingCostants.VERTICAL)
```

или

```
bar = new JToolBar(titleString, SwingCostants.VERTICAL)
```

Чаще всего в панель инструментов помещают кнопки. Однако ограничений на вид компонентов, которые можно размещать в панели инструментов, не существует. Например, в нее можно поместить выпадающий список.

Всплывающие подсказки

Панели инструментов имеют существенный недостаток. По внешнему виду кнопки трудно догадаться о ее назначении. Чтобы решить эту проблему, были созданы *всплывающие подсказки* (tool tips), которые появляются на экране, когда курсор задерживается на кнопке. Текст подсказки отображается внутри закрашенного прямоугольника (рис. 9.28). Когда пользователь перемещает курсор мыши в другое место, подсказка исчезает.

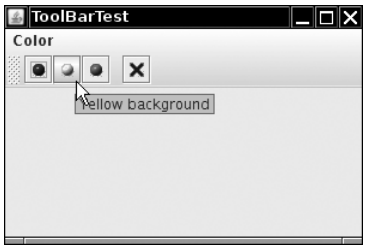


Рис. 9.28. Всплывающая подсказка

В пакете Swing подсказки можно добавлять к любому объекту класса `JComponent`, просто вызывая метод `setToolTipText()`:

```
exitButton.setToolTipText("Exit");
```

Альтернативно, если вы используете объекты класса `Action`, подсказка связывается с ключом `SHORT_DESCRIPTION`:

```
exitAction.putValue(Action.SHORT_DESCRIPTION, "Exit");
```

В листинге 9.9 приведена программа, показывающая, как одни и те же объекты класса `Action` можно добавлять в меню и в панель инструментов. Обратите внимание, что имена появляются на экране и в виде названий пунктов меню, и в виде подсказок к элементам панели инструментов.

Листинг 9.9. Содержимое файла `ToolBarTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import javax.swing.*;
4.
5.  /**
6.   * @version 1.13 2007-06-12
7.   * @author Cay Horstmann
8.   */
9.  public class ToolBarTest
10. {
11.    public static void main(String[] args)
12.    {
13.       EventQueue.invokeLater(new Runnable()
14.          {
15.             public void run()
16.             {
17.                ToolBarFrame frame = new ToolBarFrame();
18.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.                frame.setVisible(true);
20.             }
21.          });
22.    }
23. }
24.
25. /**
26.  * Фрейм с панелью инструментов и меню для изменения цвета.
27.  */
```

```
28. class ToolBarFrame extends JFrame
29. {
30.    public ToolBarFrame()
31.    {
32.       setTitle("ToolBarTest");
33.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34.
35.       // Добавление панели с изменяемым цветом.
36.
37.       panel = new JPanel();
38.       add(panel, BorderLayout.CENTER);
39.
40.       // Настройка действий.
41.
42.       Action blueAction =
43.         new ColorAction("Blue", new ImageIcon("blue-ball.gif"), Color.BLUE);
44.       Action yellowAction =
45. new ColorAction("Yellow", new ImageIcon("yellow-ball.gif"), Color.YELLOW);
46.       Action redAction =
47.         new ColorAction("Red", new ImageIcon("red-ball.gif"), Color.RED);
48.
49.       Action exitAction = new AbstractAction("Exit", new ImageIcon("exit.gif"))
50.          {
51.             public void actionPerformed(ActionEvent event)
52.             {
53.                System.exit(0);
54.             }
55.          };
56.       exitAction.putValue(Action.SHORT_DESCRIPTION, "Exit");
57.
58.       // Заполнение панели инструментов.
59.
60.       JToolBar bar = new JToolBar();
61.       bar.add(blueAction);
62.       bar.add(yellowAction);
63.       bar.add(redAction);
64.       bar.addSeparator();
65.       bar.add(exitAction);
66.       add(bar, BorderLayout.NORTH);
67.
68.       // Заполнение меню.
69.
70.       JMenu menu = new JMenu("Color");
71.       menu.add(yellowAction);
72.       menu.add(blueAction);
73.       menu.add(redAction);
74.       menu.add(exitAction);
75.       JMenuBar menuBar = new JMenuBar();
76.       menuBar.add(menu);
77.       setJMenuBar(menuBar);
78.    }
79.
80.    public static final int DEFAULT_WIDTH = 300;
81.    public static final int DEFAULT_HEIGHT = 200;
82.
```

```
83.    private JPanel panel;
84.
85.    /**
86.     * ColorAction устанавливает требуемый цвет фона.
87.     */
88.    class ColorAction extends AbstractAction
89.    {
90.       public ColorAction(String name, Icon icon, Color c)
91.       {
92.          putValue(Action.NAME, name);
93.          putValue(Action.SMALL_ICON, icon);
94.          putValue(Action.SHORT_DESCRIPTION, name + " background");
95.          putValue("Color", c);
96.       }
97.
98.       public void actionPerformed(ActionEvent event)
99.       {
100.         Color c = (Color) getValue("Color");
101.         panel.setBackground(c);
102.      }
103.   }
104.}
```

API javax.swing.JToolBar 1.2

- `JToolBar()`
- `JToolBar(String titleString)`
- `JToolBar(int orientation)`
- `JToolBar(String titleString, int orientation)`

  Создает панель инструментов, имеющую заданную строку заголовка и ориентацию. Параметр `orientation` может принимать значения `SwingConstants.HORIZONTAL` (по умолчанию) и `SwingConstants.VERTICAL`.

- `JButton add(Action a)`

  Создает новую кнопку в панели инструментов с именем, пиктограммой и обратным вызовом. Кнопка добавляется в конец панели инструментов.

- `void addSeparator()`

  Добавляет разделитель в конец панели инструментов.

API javax.swing.JComponent 1.2

- `void setToolTipText(String text)`

  Задает текст, подлежащий выводу на экран в качестве подсказки, когда курсор мыши останавливается на компоненте.

Расширенные средства компоновки

До сих пор при разработке интерфейса для наших приложений мы пользовались лишь диспетчерами компоновки `BorderLayout`, `FlowLayout` и `GridLayout`. Для решения более сложных задач этого недостаточно. В этом разделе мы детально рассмотрим расширенные средства компоновки.

У программистов, работающих на платформе Windows, может возникнуть вопрос: отчего в языке Java столько внимания уделяется диспетчерам компоновки? Ведь во многих интегрированных средах разработки достаточно перетащить компоненты в диалоговое окно и выровнять их, используя средства редактирования. При разработке большого проекта программист может вообще не думать о компоновке — квалифицированный дизайнер возьмет на себя все заботы об интерфейсе.

Такой подход имеет существенный недостаток. Если размеры элементов изменяются, приходится перекомпоновывать их вручную. Почему компоненты могут изменить свои размеры? Это происходит в двух случаях. Во-первых, пользователь может выбрать более крупный шрифт для надписей на кнопках и других элементов диалогового окна. Если вы попробуете это сделать, работая под управлением операционной системы Windows, то убедитесь сами, что многие приложения крайне неудачно справляются с этой задачей. Кнопки не увеличиваются, и более крупный шрифт "втискивается" в те же самые размеры. Такая же проблема может возникнуть при переводе интерфейса на иностранный язык. Например, слово "Cancel" на немецкий язык переводится как "Abbrechen". Если кнопка разработана так, что в ней помещается слово "Cancel", то немецкий перевод на той же кнопке будет усечен.

Почему кнопки, созданные на платформе Windows, не увеличиваются, чтобы уместить метки на своей поверхности? Это происходит потому, что разработчики интерфейса не предусмотрели никаких команд, задающих направление, в котором должны увеличиваться размеры кнопок. После того как элементы интерфейса будут перемещены в окно и выровнены, редактор диалога просто забывает координаты и размеры всех компонентов. Таким образом, информация, позволяющая судить о том, *почему* компоненты были расположены именно таким образом, теряется.

Диспетчеры компоновки в языке Java справляются с размещением компонентов намного лучше. По сути, компоновка сводится к созданию инструкций, описывающих отношения между компонентами. Это особенно важно при работе с библиотекой AWT, использующей платформенно-зависимые элементы пользовательского интерфейса. Размер кнопки или списка зависит от платформы, а разработчики приложения или аплета не могут *заранее* знать, на какой платформе будет отображаться их пользовательский интерфейс. Размер кнопки или окна списка в Motif, Windows и Macintoch могут варьироваться в широких пределах, и приложение или аплет должны знать *заранее*, на какой платформе будет отображен его интерфейс. До некоторой степени такая степень изменчивости исключается в Swing. Если ваше приложение принудительно принимает определенный стиль внешности и поведения, такой как Metal, то оно будет выглядеть идентично на всех платформах. Однако если программист хочет предоставить пользователю возможность самому выбирать стиль интерфейса, придется положиться на гибкость диспетчеров компоновки.

Со времен Java 1.0 AWT включает в себя *сеточную компоновку* (grid bag layout), которая размещает компоненты по строкам и колонкам. Размеры строки и колонки устанавливаются гибко, а компоненты могут занимать несколько строк и колонок. Такой диспетчер компоновки очень гибок, но он также и очень сложен — настолько, что само словосочетание "сеточная компоновка" может вселить страх в сердца программистов на Java.

Безуспешные попытки спроектировать диспетчер компоновки, который избавил бы программистов от тирании сеточной компоновки, привели создателей Swing к *ящичному диспетчеру компоновки* (box layout). Согласно документации JDK, класс BoxLayout "вкладывает множество панелей с разными комбинациями горизонтальных и вертикальных размеров, создавая эффект, подобный `GridLayout`, но без усложнения". Однако поскольку каждый "ящик" размещается независимо, вы не можете ис-

пользовать такой компоновщик для упорядочивания соседних компонентов одновременно по вертикали и горизонтали.

В Java SE 1.4 была предпринята еще одна попытка создания замены сеточной компоновки, а именно — *"пружинная компоновка"* (spiring layout). Вы используете воображаемые "пружины" для соединения компонентов в контейнере. При изменении размера контейнера эти пружины растягиваются и сжимаются, таким образом, регулируя размещение компонентов. Это выглядит утомительно и запутано — и так оно и есть. Поэтому пружинная компоновка быстро канула в небытие.

В 2005 г. команда разработчиков NetBeans изобрела технологию Matisse, которая комбинирует в себе инструмент компоновки и диспетчер компоновки. Проектировщики пользовательского интерфейса применяют инструмент для помещения компонентов в контейнер и для указания того, по каким компонентам они должны равняться. Инструмент транслирует замысел дизайнера в инструкции для *группового диспетчера компоновки* (group layout manager). Это намного удобнее, чем ручное написание кода управления компоновкой. Групповой диспетчер компоновки теперь стал частью Java SE 6. Даже если вы не используете NetBeans в качестве интегрированной среды разработки, мы полагаем, вам стоит рассмотреть применение его инструмента построения графических пользовательских интерфейсов. Вы можете проектировать графический интерфейс в NetBeans и вставлять полученный в результате код в интегрированную среду разработки.

В последующих разделах мы поговорим о диспетчере сеточной компоновки, потому что он достаточно широко используется и все еще представляет собой самый простой механизм генерации кода компоновки для более старых версий Java. Мы опишем стратегию, благодаря которой применение этого диспетчера компоновки может быть сравнительно безболезненным в некоторых распространенных ситуациях.

Затем мы опишем инструмент Matisse и групповой диспетчер компоновки. Вам нужно будет разобраться в том, как он работает, чтобы иметь возможность убедиться, что Matisse генерирует корректные инструкции, когда вы визуально позиционируете компоненты.

В завершение дискуссии о диспетчерах компоновки мы расскажем, как можно вообще обойтись без них и размещать компоненты вручную, а также как написать свой собственный диспетчер компоновки.

Диспетчер компоновки `GridBagLayout`

`GridBagLayout` — прародитель всех диспетчеров компоновки. Его можно рассматривать как диспетчер `GridLayout` без ограничений, т.е. его строки и столбцы могут иметь переменный размер. Для того чтобы разместить более крупный компонент, который не помещается в одну ячейку, несколько смежных ячеек можно объединить в одну. (Многие текстовые процессоры, а также HTML, предоставляют такие же возможности при редактировании таблиц: заполнение начинается с обычной сетки, а при необходимости некоторые ее ячейки объединяются в одну.) Компоненты не обязаны заполнять всю ячейку, поэтому можно задать выравнивание внутри ячеек.

Рассмотрим диалоговое окно, показанное на рис. 9.29. Оно содержит перечисленные ниже компоненты.

- Два выпадающих списка, предназначенных для выбора начертания и размера шрифта.
- Метки указанных выпадающих списков.
- Два флажка опций для выбора полужирного шрифта и курсива.
- Текстовая область для отображения строки-примера.

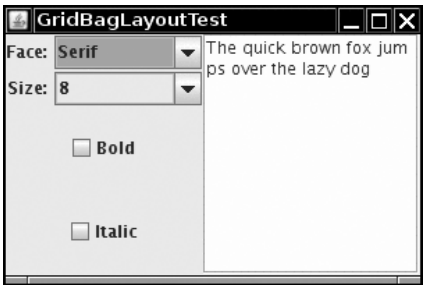


Рис. 9.29. Диалоговое окно, предназначенное для выбора шрифта



Рис. 9.30. Сетка диалогового окна, используемая при разработке

Теперь разделим диалоговое окно на ячейки так, как показано на рис. 9.30. (Строки и столбцы не обязаны иметь одинаковый размер.) Как видим, каждый флажок занимает два столбца, а текстовая область — четыре строки.

Чтобы описать эту схему в терминах диспетчера `GridBagLayout`, нужно выполнить ряд действий.

1. Создать объект `GridBagLayout`. Конструктору не следует знать, сколько строк и столбцов содержит сетка. Позднее он попытается угадать эти параметры по информации, которую он получит от вас.
2. Связать объект `GridLayout` с контейнером.
3. Для каждого компонента создать объект `GridBagConstraints`. Этот объект будет задавать расположение компонентов.
4. Добавить компонент с ограничениями с помощью следующего вызова:

   ```
   add(Component, constraints);
   ```

Ниже приведен соответствующий фрагмент кода. (Ограничения будут детально рассмотрены в дальнейшем — пока не следует беспокоиться о том, что они собой представляют.)

```
GridBagLayout layout = new GridBagLayout();
panel.setLayout(layout);
GridBagConstraints constraints = new GridBagConstraints();
constraints.weightx = 100;
constraints.weighty = 100;
constraints.gridx = 0;
constraints.gridy = 2;
constraints.gridwidth = 2;
constraints.gridheight = 1;
panel.add(style, bold);
```

Главная задача заключается в том, как задать состояние объекта класса `GridBagConstrains`. Наиболее важные сведения об этом объекте приводятся ниже.

Параметры `gridx`, `gridy`, `gridwidth` и `gridheight`

Эти параметры определяют место компонента в сетке. Значения `gridx` и `gridy` задают столбец и строку, где расположен левый верхний угол компонента. Значения `gridwidth` и `gridheight` определяют, сколько строк и столбцов занимает данный компонент.

Координаты сетки отсчитываются от нуля. В частности, выражения `gridx=0` и `gridy=0` означают левый верхний угол.

Например, текстовая область в нашем примере имеет параметры `gridx=2` и `gridy=0`, поскольку она начинается со столбца, имеющего номер 2 (т.е. третьего). Поскольку текстовая область занимает один столбец и четыре строки, `gridwidth=1`, а `gridheight=4`.

Весовые поля

Для каждой области необходимо задать *весовые поля* (weight field). В нашем примере им соответствуют поля `weightx` и `weighty`. Если вес равен нулю, то область всегда сохраняет свои первоначальные размеры. В примере, показанном на рис. 9.35, поле `weightx` для метки задано равным 0. Это позволяет сохранять постоянную ширину метки при изменении размера окна. С другой стороны, если задать вес всех областей равным 0, то контейнер разместит компоненты в центре, и пространство окна не будет заполнено.

Проблема, связанная с весовыми параметрами, заключается в том, что они являются характеристиками строк и столбцов, а не отдельных ячеек. Несмотря на это, их нужно задавать для ячеек, поскольку диспетчер `GridBagLayout` не различает отдельные строки и столбцы. В качестве веса строки или столбца принимается максимальный из весов содержащихся в них ячеек. Таким образом, если нужно, чтобы строки и столбцы сохраняли фиксированный размер, необходимо задать вес всех компонентов в этих строках и столбцах равным нулю.

Обратите внимание, что вес на самом деле не позволяет определить относительные размеры столбцов. Он лишь указывает, какая часть "свободного" пространства должна быть выделена каждой области, если контейнер превышает рекомендуемый размер. Чтобы научиться правильно подбирать вес, надо иметь определенный опыт работы с данным диспетчером компоновки. Мы рекомендуем поступать так. Установите вес равным 100, затем запустите программу на выполнение и посмотрите, как будет выглядеть интерфейс. Для того чтобы определить, как выровнены строки и столбцы, попробуйте изменить размеры диалогового окна. Если окажется, что какая-то строка или столбец не должны увеличиваться, установите вес всех компонентов, находящихся в них, равным нулю.

Параметры `fill` и `anchor`

Если вы не хотите, чтобы компонент растягивался и заполнял все пространство, нужно установить параметр `fill`. Этот параметр принимает четыре возможных значения: `GridBagConstraints.NONE`, `GridBagConstraints.HORIZONTAL`, `GridBagConstraints.VERTICAL` и `GridBagConstraints.BOTH`.

Если компонент не должен заполнять собой все пространство, можно указать, к какой позиции вы хотели бы его привязать, задав параметр `anchor`. Этот параметр может принимать следующие значения: `GridBagConstraints.CENTER` (по умолчанию), `GridBagConstraints.NORTH`, `GridBagConstraints.NORTHEAST`, `GridBagConstraints.EAST` и т.д.

Выделение пустого пространства

Задав значение поля `insets` объекта `GridBagConstraints`, компонент можно окружить дополнительным пустым пространством. Для этого необходимо задать значения полей `left`, `top`, `right` и `bottom` в объекте `Insets`. Этот процесс называется *внешним заполнением* (external padding).

Значения полей ipadx и ipady определяют *внутреннее заполнение*. Эти значения добавляются к минимальным ширине и высоте компонента, гарантируя тем самым, что компонент никогда не уменьшится до своих минимальных размеров.

Альтернативные способы указания параметров gridx, gridy, gridwidth и gridheight

В документации по библиотеке AWT рекомендуется не задавать абсолютные значения параметров gridx и gridy, а вместо этого использовать константу GridBagConstraints.RELATIVE. Затем нужно, как обычно, слева направо строка за строкой включать компоненты в контейнер, управляемый диспетчером компоновки GridBagLayout.

Количество строк и столбцов, занятых ячейкой, нужно задавать с помощью полей gridheight и gridwidht. Исключение составляет компонент, занимающий *последнюю* строку или столбец. Для него задается не числовое значение, а специальная константа GridBagConstraints.REMAINDER. Это позволяет диспетчеру компоновки распознавать последние компоненты.

Данная схема вполне работоспособна. Однако кажется странным сначала скрывать от диспетчера компоновки информацию о фактическом расположении компонентов, а затем полагаться на то, что он сам угадает правильные параметры.

У читателя может сложиться впечатление, что пользоваться диспетчером GridBagLayout очень сложно. Однако, следуя приведенным ниже рекомендациям, удается размещать элементы, не затрачивая значительных усилий.

1. Изобразите схему расположения компонентов на бумаге.
2. Определите такую сетку, чтобы каждый компонент умещался в ячейке, а большие компоненты могли занимать несколько ячеек.
3. Пометьте строки и столбцы вашей сетки числами 0, 1, 2, 3,... Теперь можно определять значения параметров gridx, gridy, gridwidth и gridheight.
4. Для каждого компонента решите, нужно ли, чтобы он заполнял ячейку в горизонтальном или вертикальном направлении? Если нет, как его выровнять внутри ячейки? Это определяют параметры fill и anchor.
5. Установите вес равным 100. Однако если вы хотите, чтобы конкретная строка или столбец всегда сохраняли свой первоначальный размер, задайте параметры weightx или weighty всех компонентов, находящихся в этой строке или столбце, равными 0.
6. Напишите программу. Тщательно проверьте установки класса GridBagConstraints. Одно неверное ограничение может разрушить всю компоновку.
7. Скомпилируйте программу и запустите ее на выполнение.

Некоторые инструменты для построения графических интерфейсов включают в себя инструменты для визуального указания ограничений. На рис. 9.31 показан диалог конфигурации из NetBeans.

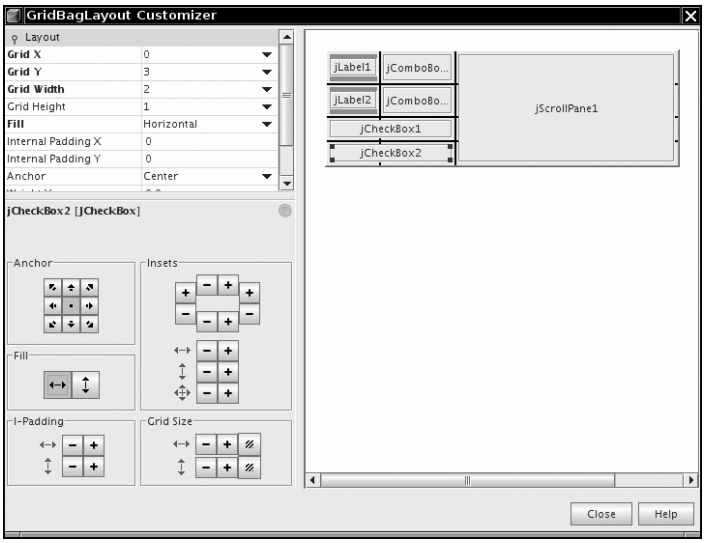


Рис. 9.31. Установка ограничений в NetBeans

Вспомогательный класс для работы с объектами `GridBagConstraints`

Наиболее трудоемкая задача при работе с диспетчером компоновки `GridBagLayout` — это написание кода, устанавливающего ограничения. Многие программисты создают для этой цели вспомогательные функции или небольшие вспомогательные классы. Код одного из таких классов приведен после примера, демонстрирующего создание диалогового окна. Характеристики вспомогательного класса приведены ниже.

- Имя класса — `GBC` (прописные буквы из имени `GridBagConstraints`).
- Данный класс является подклассом `GridBagConstraints`, поэтому константы можно указывать, используя более короткое имя, например `GBC.EAST`.
- Объект `GBC` используется при добавлении компонента, например:

```
add(component, new GBC(1, 2));
```

- Для установки наиболее часто используемых параметров, `gridx` и `gridy` или `gridx`, `gridy`, `gridwidth` и `gridheight`, предусмотрены два конструктора.

```
add(component, new GBC(1, 2, 1, 4));
```

- В данном классе предусмотрены удобные метки для установки полей, содержащих значения x и y.

```
add(component, new GBC(1, 2).setWeight(100, 100));
```

- Методы установки значений полей возвращают значение `this`, поэтому их можно объединять в цепочки.

```
add(component, new GBC(1, 2).setAnchor(GBC.EAST).setWeight(100, 100));
```

- Метод `setInsets()` создает объекты `Insets`. Так, например, чтобы создать пустое пространство размером в один пиксель, надо использовать следующее выражение:

```
add(component, new GBC(1, 2).setAnchor(GBC.EAST).setInsets(1));
```

Полный текст программы, реализующей диалоговое окно для выбора шрифта, содержится в листинге 9.10. Для добавления компонентов в контейнер, управляемый диспетчером GridBagLayout, используется приведенный ниже фрагмент кода.

```
add(faceLabel, new GBC(0, 0).setAnchor(GBC.EAST));
add(face, new GBC(1, 0).setFill(GBC.HORIZONTAL).
   setWeight(100, 0).setInsets(1));
add(sizeLabel, new GBC(0, 1).setAnchor(GBC.EAST));
add(size, new GBC(1, 1).setFill(GBC.HORIZONTAL).
   setWeight(100, 0).setInsets(1));
add(bold, new GBC(0, 2, 2, 1).setAnchor(GBC.CENTER).
   setWeight(100, 100));
add(italic, new GBC(0, 3, 2, 1).
   setAnchor(GBC.CENTER).setWeight(100, 100));
add(sample, new GBC(2, 0, 1, 4).setFill(GBC.BOTH).
   setWeight(100, 100));
```

Понимая принцип использования ограничений при работе с диспетчером, GridBagLayout, в этом коде разобраться несложно.



На заметку! В руководстве Sun, доступном по адресу http://java.sun.com/docs/books/tutorial/uiswing/layout/gridbag.html, предлагается использовать один и тот же объект GridBagConstraints для всех компонентов. На наш взгляд, получившийся в результате код труден для восприятия и при его написании легко допустить ошибку. Обратите внимание на демонстрационный код, расположенный в документе http://java.sun.com/docs/books/tutorial/uiswing/events/containerlistener.html. Действительно ли разработчик стремился растягивать кнопки по горизонтали, или он забыл отключить установку BOTH параметра fill?

Листинг 9.10. Содержимое файла GridBagLayoutTest.java

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import javax.swing.*;
4.
5.  /**
6.   * @version 1.33 2007-06-12
7.   * @author Cay Horstmann
8.   */
9.  public class GridBagLayoutTest
10. {
11.    public static void main(String[] args)
12.    {
13.       EventQueue.invokeLater(new Runnable()
14.          {
15.             public void run()
16.             {
17.                FontFrame frame = new FontFrame();
18.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.                frame.setVisible(true);
20.             }
21.          });
22.    }
23. }
24.
```

```
25. /**
26.  * Фрейм, в котором диспетчер компоновки GridBagLayout
27.  * использован  для выравнивания компонентов,
28.  * предназначенных для выбора шрифта.
29.  */
30. class FontFrame extends JFrame
31. {
32.    public FontFrame()
33.    {
34.       setTitle("GridBagLayoutTest");
35.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
36.
37.       GridBagLayout layout = new GridBagLayout();
38.       setLayout(layout);
39.
40.       ActionListener listener = new FontAction();
41.
42.       // Создание компонентов.
43.
44.       JLabel faceLabel = new JLabel("Face: ");
45.
46.       face = new JComboBox(new String[] { "Serif", "SansSerif",
47.             "Monospaced", "Dialog", "DialogInput" });
48.
49.       face.addActionListener(listener);
50.
51.       JLabel sizeLabel = new JLabel("Size: ");
52.
53.       size = new JComboBox(new String[]
54.             { "8", "10", "12", "15", "18", "24", "36", "48" });
55.
56.       size.addActionListener(listener);
57.
58.       bold = new JCheckBox("Bold");
59.       bold.addActionListener(listener);
60.
61.       italic = new JCheckBox("Italic");
62.       italic.addActionListener(listener);
63.
64.       sample = new JTextArea();
65.       sample.setText("The quick brown fox jumps over the lazy dog");
66.       sample.setEditable(false);
67.       sample.setLineWrap(true);
68.       sample.setBorder(BorderFactory.createEtchedBorder());
69.
70.       // Размещение компонентов с помощью класса GBC.
71.
72.       add(faceLabel, new GBC(0, 0).setAnchor(GBC.EAST));
73.       add(face, new GBC(1, 0).setFill(GBC.HORIZONTAL).setWeight(100, 0).
74.           setInsets(1));
75.       add(sizeLabel, new GBC(0, 1).setAnchor(GBC.EAST));
76.       add(size, new GBC(1, 1).setFill(GBC.HORIZONTAL).setWeight(100, 0).
77.           setInsets(1));
78.       add(bold, new GBC(0, 2, 2, 1).setAnchor(GBC.CENTER).setWeight(100, 100));
79.       add(italic, new GBC(0, 3, 2, 1).setAnchor(GBC.CENTER).setWeight(100, 100));
80.       add(sample, new GBC(2, 0, 1, 4).setFill(GBC.BOTH).setWeight(100, 100));
81.    }
```

```
82.
83.    public static final int DEFAULT_WIDTH = 300;
84.    public static final int DEFAULT_HEIGHT = 200;
85.
86.    private JComboBox face;
87.    private JComboBox size;
88.    private JCheckBox bold;
89.    private JCheckBox italic;
90.    private JTextArea sample;
91.
92.    /**
93.     * Слушатель, изменяющий текст строки.
94.     */
95.    private class FontAction implements ActionListener
96.    {
97.       public void actionPerformed(ActionEvent event)
98.       {
99.          String fontFace = (String) face.getSelectedItem();
100.         int fontStyle = (bold.isSelected() ? Font.BOLD : 0)
101.            + (italic.isSelected() ? Font.ITALIC : 0);
102.         int fontSize = Integer.parseInt((String) size.getSelectedItem());
103.         Font font = new Font(fontFace, fontStyle, fontSize);
104.         sample.setFont(font);
105.         sample.repaint();
106.      }
107.   }
108.}
```

В листинге 9.11 показан код вспомогательного класса GBC.

Листинг 9.11. Содержимое файла GBC.java

```
1.  import java.awt.*;
2.
3.
4.  /**
5.   * Этот класс упрощает работу с классом GridBagConstraints.
6.   * @version 1.01 2004-05-06
7.   * @author Cay Horstmann
8.   */
9.  public class GBC extends GridBagConstraints
10. {
11.    /**
12.     * Создает объект GBC, определяя gridx, gridy.
13.     * Значения остальных параметров
14.     * принимаются по умолчанию.
15.     * @param gridx Позиция gridx
16.     * @param gridy Позиция gridy
17.     * @param gridwidth Расширение ячейки в направлении x
18.     * @param gridheight Расширение ячейки в направлении y
19.     */
20.    public GBC(int gridx, int gridy)
21.    {
22.       this.gridx = gridx;
23.       this.gridy = gridy;
24.    }
25.
```

```
26.    /**
27.     * Создает объект GBC, определяя gridx, gridy,
28.     * gridwidth и gridheight. Значения остальных
29.     * параметров принимаются по умолчанию.
30.     * @param gridx Позиция gridx
31.     * @param gridy Позиция gridy
32.     * @param gridwidth Расширение ячейки в направлении x
33.     * @param gridheight Расширение ячейки в направлении y
34.     */
35.    public GBC(int gridx, int gridy, int gridwidth, int gridheight)
36.    {
37.       this.gridx = gridx;
38.       this.gridy = gridy;
39.       this.gridwidth = gridwidth;
40.       this.gridheight = gridheight;
41.    }
42.
43.    /**
44.     * Устанавливает параметр anchor.
45.     * @param anchor Значение параметра
46.     * @return Объект this, пригодный для дальнейшей модификации
47.     */
48.    public GBC setAnchor(int anchor)
49.    {
50.       this.anchor = anchor;
51.       return this;
52.    }
53.
54.    /**
55.     * Устанавливает параметр fill.
56.     * @param fill Значение параметра
57.     * @return Объект this, пригодный для дальнейшей модификации
58.     */
59.    public GBC setFill(int fill)
60.    {
61.       this.fill = fill;
62.       return this;
63.    }
64.
65.    /**
66.     * Устанавливает веса ячейки.
67.     * @param weightx Вес в направлении x
68.     * @param weighty Вес в направлении y
69.     * @return Объект this, пригодный для дальнейшей модификации
70.     */
71.    public GBC setWeight(double weightx, double weighty)
72.    {
73.       this.weightx = weightx;
74.       this.weighty = weighty;
75.       return this;
76.    }
77.
78.    /**
79.     * Устанавливает размеры свободного пространства для ячейки.
80.     * @param distance Размеры по всем направлениям
81.     * @return Объект this, пригодный для дальнейшей модификации
82.     */
```

```
83.    public GBC setInsets(int distance)
84.    {
85.       this.insets = new Insets(distance, distance, distance, distance);
86.       return this;
87.    }
88.
89.    /**
90.     * Устанавливает размеры свободного пространства для ячейки.
91.     * @param top Размер верхней части свободного пространства
92.     * @param left Размер левой части свободного пространства
93.     * @param bottom Размер нижней части свободного пространства
94.     * @param right Размер правой части свободного пространства
95.     * @return Объект this, пригодный для дальнейшей модификации
96.     */
97.    public GBC setInsets(int top, int left, int bottom, int right)
98.    {
99.       this.insets = new Insets(top, left, bottom, right);
100.      return this;
101.   }
102.
103.   /**
104.    * Устанавливает внутреннее заполнение.
105.    * @param ipadx Внутреннее заполнение в направлении x
106.    * @param ipady Внутреннее заполнение в направлении y
107.    * @return Объект this, пригодный для дальнейшей модификации
108.    */
109.   public GBC setIpad(int ipadx, int ipady)
110.   {
111.      this.ipadx = ipadx;
112.      this.ipady = ipady;
113.      return this;
114.   }
115.}
```

API java.awt.GridConstraints 1.0

- `int gridx, gridy`

  Задают столбец и строку для размещения ячейки. По умолчанию принимаются нулевые значения.

- `int gridwidth, gridheight`

  Задают количество столбцов и строк, занимаемых ячейкой. По умолчанию принимаются значения, равные 1.

- `double weightx, weighty`

  Определяют способность ячейки увеличиваться в размерах. По умолчанию принимается значение 0.

- `int anchor`

  Задает вид выравнивания компонента внутри ячейки. Вы можете указывать константы, определяющие абсолютную позицию:

  ```
  NORTHWEST  NORTH   NORTHEAST
  WEST       CENTER  EAST
  SOUTHWEST  SOUTH   SOUTHEAST
  ```

либо значения, не зависящие от ориентации:

```
FIRST_LINE_START    LINE_START   FIRST_LINE_END
PAGE_START          CENTER       PAGE_END
LAST_LINE_START     LINE_END     LAST_LINE_END
```

Вторая группа констант удобна в том случае, если приложение адаптируется для языков, в которых символы следуют справа налево или сверху вниз.

- `int fill`

  Задает способ заполнения компонентом ячейки. Допустимы значения `NONE`, `BOTH`, `HORIZONTAL` и `VERRTICAL`. По умолчанию принимается `NONE`.

- `int ipadx, ipady`

  Задает внутреннее заполнение вокруг компонента. По умолчанию принимается значение 0.

- `Insets insets`

  Задает внешнее заполнение вокруг границ ячейки. По умолчанию заполнение отсутствует.

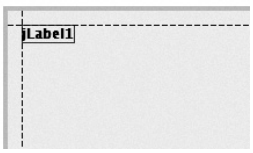
- `GridBagConstraints(int gridx, int gridy, int grigwidth, int gridheight, double weightx, double weighty, int anchor, int fill, Insets insets, int ipadx, int ipady)`

  Создает объект `GridBagConstraints`, заполняя все его поля указанными значениями. Компания Sun рекомендует использовать этот конструктор только в программах автоматической генерации интерфейса, поскольку полученный в результате код труден для восприятия.

Диспетчер компоновки `GroupLayout`

Прежде чем мы приступим к изучению API класса `GroupLayout`, стоит сначала вкратце ознакомиться с построителем пользовательского интерфейса Mattisse GUI builder из состава NetBeans. Мы не станем приводить здесь полное руководство по Mattise. При желании вы найдете его по адресу `http://www.netbeans.org/kb/articles/matisse.html`.

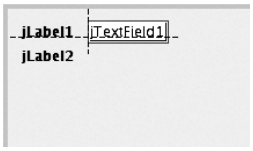
Опишем последовательность компоновки верхней части диалога, представленного на рис. 9.13. Сначала нужно создать новый проект и добавить к нему новую форму JFrame. Перетащите метку в угол, пока не появятся две линии, отделяющие ее от границ контейнера:



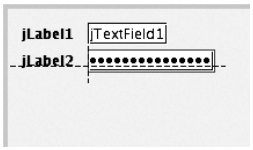
Поместите другую метку под первой строкой:



Перетащите текстовое поле так, чтобы его базовая линия была выровнена с базовой линией первой метки. Опять-таки, обращайте внимание на управляющие линии:



И, наконец, выровняйте поле пароля по метке слева и находящемуся выше полю.



Matisse транслирует эти действия в следующий код Java:

```
layout.setHorizontalGroup(
    layout.createParallelGroup(GroupLayout.Alignment.LEADING)
    .addGroup(layout.createSequentialGroup()
      .addContainerGap()
      .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
      .addGroup(layout.createSequentialGroup()
               .addComponent(jLabel1)
               .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
               .addComponent(jTextField1))
            .addGroup(layout.createSequentialGroup()
               .addComponent(jLabel2)
               .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
               .addComponent(jPasswordField1)))
            .addContainerGap(222, Short.MAX_VALUE)));
layout.setVerticalGroup(
    layout.createParallelGroup(GroupLayout.Alignment.LEADING)
    .addGroup(layout.createSequentialGroup()
      .addContainerGap()
      .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
            .addComponent(jLabel1)
            .addComponent(jTextField1))
            .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
      .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
            .addComponent(jLabel2)
            .addComponent(jPasswordField1))
            .addContainerGap(244, Short.MAX_VALUE)));
```

Выглядит довольно угрожающе, но к счастью, вам не нужно писать этот код. Однако полезно иметь базовое представление о действиях компоновки, чтобы уметь находить ошибки. Проанализируем базовую структуру этого кода. Врезки по API в конце этого раздела объяснят в подробностях каждый из этих методов и классов.

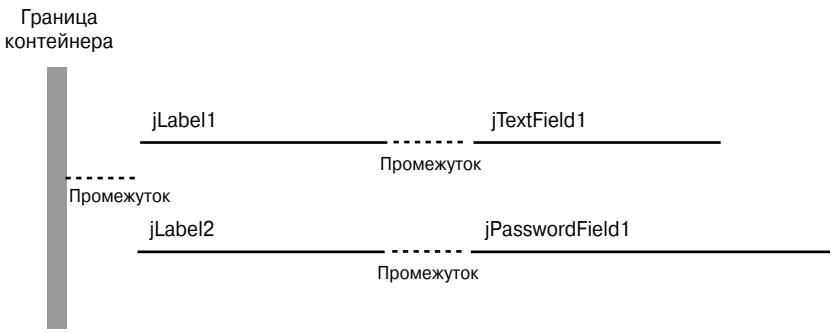
Компоненты организованы помещением их в объекты типа `GroupLayout.SequentialGroup` или `GroupLayout.ParallelGroup`. Эти классы являются наследниками `GroupLayout.Group`. Группы могут содержать компоненты, пробелы и вложенные группы.

Разнообразные методы add групповых классов возвращают объект группы, так что вызовы методов могут быть вложены примерно так:

```
group.addComponent(...).addPreferredGap(...).addComponent(...);
```

Как видно из кода примера, компоновка групп разделяет вычисления позиций по горизонтали и вертикали.

Чтобы визуализировать горизонтальные вычисления, предположим, что компоненты "сплющены" до нулевой высоты, как здесь:



Здесь присутствует две параллельные последовательности компонентов, описанные следующим (слегка упрощенным) кодом:

```
.addContainerGap()
.addGroup(layout.createParallelGroup()
    .addGroup(layout.createSequentialGroup()
            .addComponent(jLabel1)
            .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
            .addComponent(jTextField1))
    .addGroup(layout.createSequentialGroup()
            .addComponent(jLabel2)
            .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
            .addComponent(jPasswordField1)))
```

Минуточку! Это же не может быть правильным. Если метки имеют разную длину, то текстовое поле и поле пароля не будут выровнены.

Нужно как-то сообщить Matisse о том, что мы хотим выровнять эти поля. Выберем оба поля, выполним щелчок правой кнопкой мыши и выберем в контекстном меню пункт Align⇨Left to Column (Выровнять⇨Влево внутри колонки). Так же выровняем метки (рис. 9.32).

Это автоматически изменит код компоновки:

```
.addGroup(layout.createSequentialGroup()
    .addContainerGap()
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
            .addComponent(jLabel1, GroupLayout.Alignment.TRAILING)
            .addComponent(jLabel2, GroupLayout.Alignment.TRAILING))
    .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
            .addComponent(jTextField1)
            .addComponent(jPasswordField1))
```

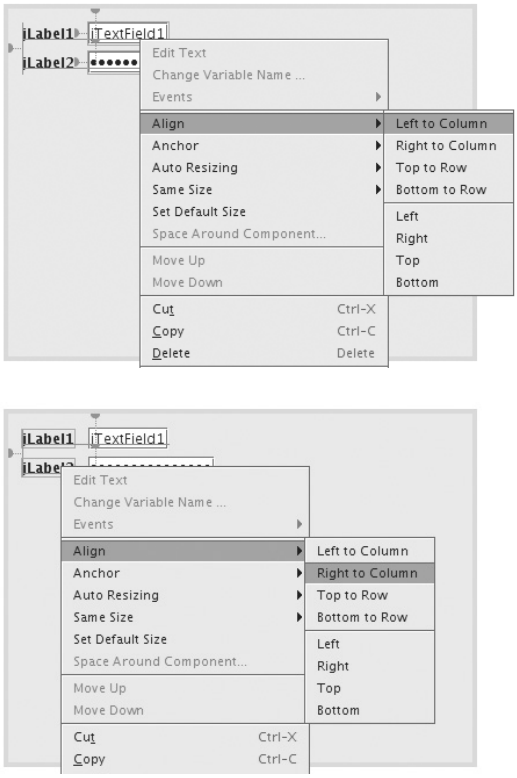
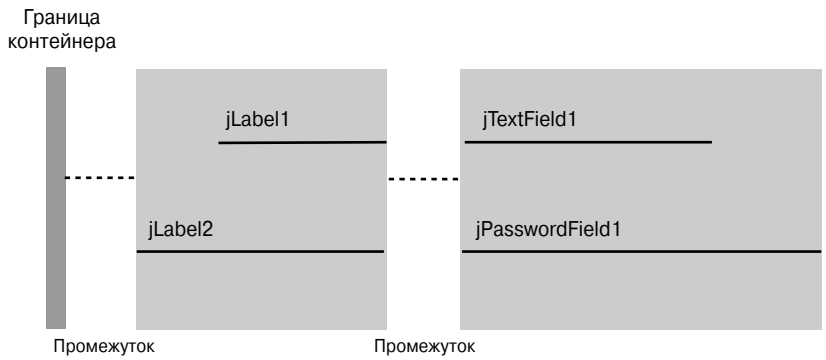


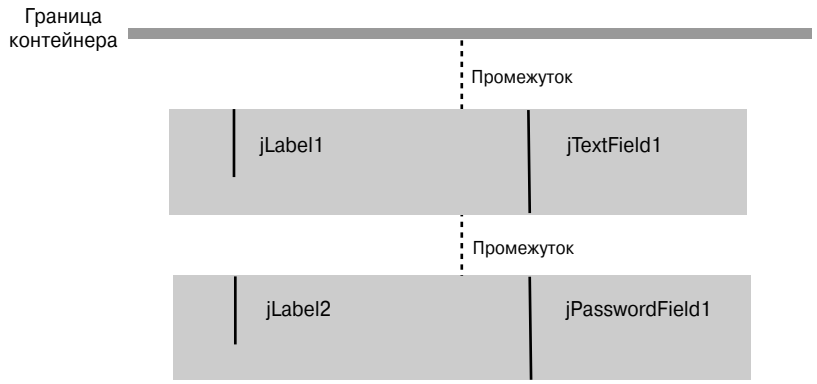
Рис. 9.32. Выравнивание меток и текстовых полей в Matisse

Теперь метки и поля помещены в две параллельные группы. Первая группа имеет выравнивание `TRAILING` (что означает выравнивание вправо при направлении текста слева направо):



Умение Matisse транслировать инструкции дизайнера во вложенные группы кажется волшебством, но Артур Кларк говорил, что любая достаточно развитая технология неотличима от волшебства.

Для полноты картины рассмотрим вычисления вертикального расположения. Теперь вам нужно представить компоненты как не имеющие ширины. У нас есть последовательная группа, состоящая из двух параллельных групп, разделенных пробелом:



Соответствующий код выглядит так:

```
layout.createSequentialGroup()
    .addContainerGap()
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
           .addComponent(jLabel1)
           .addComponent(jTextField1))
    .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
           .addComponent(jLabel2)
           .addComponent(jPasswordField1))
```

Как видите, здесь компоненты выравниваются по их базовым линиям. (Базовая линия — это линия, по которой выравнивается текст компонентов.)



На заметку! Аккуратное выравнивание по базовой линии было невозможно в ранних версиях Java. Но, наконец, в Java SE 6 появился метод `getBaseline` класса `Component`, предназначенный для точного определения базовой линии компонента, содержащего текст.

Вы можете принудительно установить набору компонентов одинаковый размер. Например, мы можем пожелать, чтобы текстовое поле и поле пароля точно совпадали по размеру. Выберите в Matisse оба поля, щелкните правой кнопкой мыши и затем выберите пункт меню Same Size⇨Same Width (Одинаковый размер⇨Одинаковая ширина), как показано на рис. 9.33.

Matisse добавляет следующий оператор в код компоновки:

```
layout.linkSize(SwingConstants.HORIZONTAL, new Component[]
  {jPasswordField1, jTextField1});
```

Код в листинге 9.12 демонстрирует, как разместить компоненты селектора шрифтов из предыдущего раздела, используя `GroupLayout` вместо `GridBagLayout`. Может показаться, что код не стал проще, чем в листинге 9.10, но нам не пришлось писать его вручную. Мы использовали Matisse для осуществления компоновки и затем немного подчистили сгенерированный код.

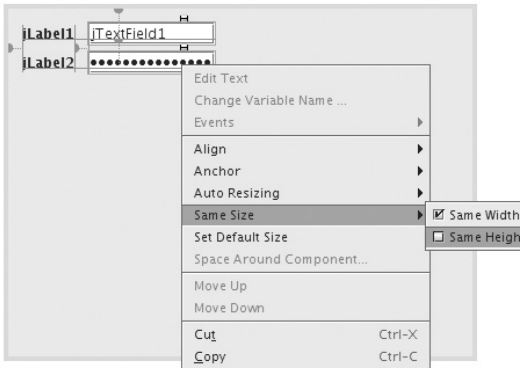


Рис. 9.33. Установка одинаковой ширины для двух компонентов

Листинг 9.12. Содержимое файла `GroupLayoutTest.java`

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6. * @version 1.0 2007-04-27
7. * @author Cay Horstmann
8. */
9. public class GroupLayoutTest
10. {
11.     public static void main(String[] args)
12.     {
13.       EventQueue.invokeLater(new Runnable()
14.       {
15.             public void run()
16.             {
17.                FontFrame frame = new FontFrame();
18.                frame.setDefaultCloseOperation(
19.                      JFrame.EXIT_ON_CLOSE);
20.                frame.setVisible(true);
21.             }
22.     });
23. }
24. /**
25.  * Фрейм, использующий групповую компоновку
26.  * для размещения компонентов селектора шрифта.
27.  */
28. class FontFrame extends JFrame
29. {
30.     public FontFrame()
31.     {
32.             setTitle("GroupLayoutTest");
33.             setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34.
35.             ActionListener listener = new FontAction();
36.
```

```
37.            // Конструирование компонентов.
38.            JLabel faceLabel = new JLabel("Face: ");
39.
40.            face = new JComboBox(new String[] { "Serif", "SansSerif",
41.            "Monospaced", "Dialog",
42.            "DialogInput" });
43.
44.            face.addActionListener(listener);
45.
46.            JLabel sizeLabel = new JLabel("Size: ");
47.
48.            size = new JComboBox(new String[] { "8", "10", "12", "15", "18",
49.            "24", "36", "48" });
50.
51.            size.addActionListener(listener);
52.
53.            bold = new JCheckBox("Bold");
54.            bold.addActionListener(listener);
55.
56.            italic = new JCheckBox("Italic");
57.            italic.addActionListener(listener);
58.
59.            sample = new JTextArea();
60.            sample.setText("The quick brown fox jumps over the lazy dog");
61.            sample.setEditable(false);
62.            sample.setLineWrap(true);
63.            sample.setBorder(BorderFactory.createEtchedBorder());
64.
65.            pane = new JScrollPane(sample);
66.
67.            GroupLayout layout = new GroupLayout(getContentPane());
68.            setLayout(layout);
69.            layout.setHorizontalGroup(layout.createParallelGroup(GroupLayout.
70.            Alignment.LEADING).addGroup(
71.            layout.createSequentialGroup().addContainerGap().addGroup(
72.            layout.createParallelGroup(GroupLayout.Alignment.LEADING).addGroup(
73.            GroupLayout.Alignment.TRAILING,
74.            layout.createSequentialGroup().addGroup(
75.            layout.createParallelGroup(GroupLayout.Alignment.TRAILING)
76.            .addComponent(faceLabel).addComponent(sizeLabel))
77.            .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
78.            .addGroup(
79.            layout.createParallelGroup(
80.            GroupLayout.Alignment.LEADING, false)
81.            .addComponent(size).addComponent(face)))
82.            .addComponent(italic).addComponent(bold)).addPreferredGap(
83.            LayoutStyle.ComponentPlacement.RELATED).addComponent(pane)
84.            .addContainerGap()));
85.
86.            layout.linkSize(SwingConstants.HORIZONTAL,
87.            new java.awt.Component[] { face, size });
88.
89.            layout.setVerticalGroup(layout.createParallelGroup(GroupLayout.
90.            Alignment.LEADING).addGroup(
91.            layout.createSequentialGroup().addContainerGap().addGroup(
```

```
92.             layout.createParallelGroup(GroupLayout.Alignment.LEADING)
93.             .addComponent(
94.             pane, GroupLayout.Alignment.TRAILING).addGroup(
95.             layout.createSequentialGroup().addGroup(
96.             layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
97.             .addComponent(face).addComponent(faceLabel))
98.             .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
99.             .addGroup(
100.            layout.createParallelGroup(
101.            GroupLayout.Alignment.BASELINE).addComponent(size)
102.            .addComponent(sizeLabel)).addPreferredGap(
103.            LayoutStyle.ComponentPlacement.RELATED).addComponent(
104.            italic, GroupLayout.DEFAULT_SIZE,
105.            GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
106.            .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
107.            .addComponent(bold, GroupLayout.DEFAULT_SIZE,
108.            GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)))
109.            .addContainerGap()));
110.       }
111.
112.    public static final int DEFAULT_WIDTH = 300;
113.    public static final int DEFAULT_HEIGHT = 200;
114.
115.    private JComboBox face;
116.    private JComboBox size;
117.    private JCheckBox bold;
118.    private JCheckBox italic;
119.    private JScrollPane pane;
120.    private JTextArea sample;
121.
122.    /**
123.     * Обработчик, изменяющий шрифт текста примера.
124.     */
125.    private class FontAction implements ActionListener
126.    {
127.      public void actionPerformed(ActionEvent event)
128.      {
129.            String fontFace = (String) face.getSelectedItem();
130.            int fontStyle = (bold.isSelected()?Font.BOLD:0)
131.                 + (italic.isSelected() ? Font.ITALIC : 0);
132.            int fontSize = Integer.parseInt((String) size.getSelectedItem());
133.            Font font = new Font(fontFace, fontStyle, fontSize);
134.            sample.setFont(font);
135.            sample.repaint();
136.      }
137.    }
138. }
```

API java.swing.GroupLayout 1.0

- `GroupLayout(Container host)`

  Конструирует `GroupLayout` для компоновки компонентов в контейнере `host`. (Обратите внимание, что вам все равно нужно вызвать `setLayout` для объекта `host`.)

- `void setHorizontalGroup(GroupLayout.Group g)`
- `void setVerticalGroup(GroupLayout.Group g)`

Устанавливает группу, управляющую горизонтальной или вертикальной раскладкой.

- `void linkSize(Component... components)`
- `void linkSize(int axis, Component... component)`

Устанавливает для двух компонентов одинаковый размер или одинаковый размер только по заданным осям (либо `SwingConstants.HORIZONTAL`, либо `SwingConstants.VERTICAL`).

- `GroupLayout.SequentialGroup createSequentialGroup()`

Создает группу, последовательно размещающую свои члены.

- `GroupLayout.ParallelGroup createParallelGroup()`
- `GroupLayout.ParallelGroup createParallelGroup(GroupLayout.Alignment align)`
- `GroupLayout.ParallelGroup createParallelGroup(GroupLayout.Alignment align, boolean resizable)`

Создает группу, раскладывающую свои члены параллельно.

Параметры `align` Одно из значений: `BASELINE`, `LEADING` (по умолчанию), `TRAILING` или `CENTER`.

`resizabke` `true` (по умолчанию), когда группа может изменять размеры; `false` — если предпочтительный размер совпадает с минимальным и максимальным.

- `boolean getHonorsVisibility()`
- `void setHonorsVisibility(boolean b)`

Извлекает или устанавливает свойство `honorsVisibility`. Когда равно `true` (по умолчанию), то невидимые компоненты не участвуют в компоновке. Когда равно `false`, в компоновке участвуют только видимые компоненты. Это удобно, когда вы временно скрываете некоторые компоненты, но не хотите, чтобы при этом компоновка изменилась.

- `boolean getAutoCreateGaps()`
- `void setAutoCreateGaps(boolean b)`
- `boolean getAutoCreateContainerGaps()`
- `void setAutoCreateContainerGaps(boolean b)`

Извлекает или устанавливает свойства `autoCreateGaps` и `autoCreateContainerGaps`. Когда равно `true`, то автоматически добавляются отступы между компонентами или на границах контейнера. По умолчанию равно `false`. Значение `true` удобно, когда вы вручную создаете `GroupLayout`.

API java.swing.GroupLayout.Group

- `GroupLayout.Group addComponent(Component c)`
- `GroupLayout.Group addComponent(Component c, int minimumSize, int preferredSize, int maximumSize)`

Добавляет компонент в группу. Параметры `size` могут быть действительными (не отрицательными) значениями либо специальными константами `GroupLayout.DEFAULT_SIZE` или `GroupLayout.PREFERRED_SIZE`. Когда используется `DEFAULT_SIZE`, вызываются методы компонента `getMinimumSize()`, `getPreferredSize()`

или `getMaximumSize()`. При использовании `PREFERRED_SIZE` вызывается метод компонента `getPreferredSize()`.

- `GroupLayout.Group addGap(int size)`
- `GroupLayout.Group addGap(int minimumSize, int preferredSize, int maximumSize)`

  Добавляет отступ с жестко заданным размером.

API java.swing.GroupLayout.ParallelGroup

- `GroupLayout.ParallelGroup addComponent(Component c, GroupLayout.Alignment align)`
- `GroupLayout.ParallelGroup addComponent(Component c, GroupLayout.Alignment align, int minimumSize, int preferredSize, int maximumSize)`
- `GroupLayout.ParallelGroup addComponent(GroupLayout.Group g, GroupLayout.Alignment align)`

  Добавляет компонент или группу в данную группу, используя заданное выравнивание (одно из значений `BASELINE`, `LEADING`, `TRAILING` или `CENTER`).

API java.swing.GroupLayout.SequentalGroup

- `GroupLayout.SequentalGroup addContainerGap()`
- `GroupLayout.SequentalGroup addContainerGap(int preferredSize, int maximumSize)`

  Добавляет отступ для отделения компонентов от края контейнера.

- `GroupLayout.SequentalGroup addPrefferedGap(LayoutStyle.ComponentPlacement type)`

  Добавляет отступ для разделения компонентов. Типом является `LayoutStyle.ComponentPlacement.RELATED` или `LayoutStyle.ComponentPlacement.UNRELATED`.

Компоновка без использования диспетчера

Иногда бывает нужно просто поместить какой-то компонент в требуемую позицию, не прибегая к помощи диспетчера компоновки. Такой процесс называют *абсолютным позиционированием*. Если приложение задумывается как платформенно-независимое, такой подход не годится, но если вы хотите быстро создать прототип интерфейса, он вполне приемлем.

Чтобы разместить компонент в требуемой позиции, надо выполнить следующие действия.

1. Вместо диспетчера компоновки задать значение `null`.
2. Добавить компонент в контейнер.
3. Задать координаты и размер.

```
setLayout(null);
JButton ok = new JButton("OK");
add(ok);
ok.setBounds(10, 10, 30, 15);
```

API `java.awt.Component 1.0`

- `void setBounds(int x, int y, int width, int height)`

  Перемещает компонент и изменяет его размеры.

  Параметры `x,y` Координаты левого верхнего угла компонента.

  `width,height` Новые размеры компонента.

Диспетчер компоновки, определяемый разработчиком

В принципе, можно разработать собственный класс `LayoutManager`, управляющий компонентами по-своему. В качестве забавного примера мы продемонстрируем, как можно расставить все компоненты в контейнере по кругу (рис. 9.34).

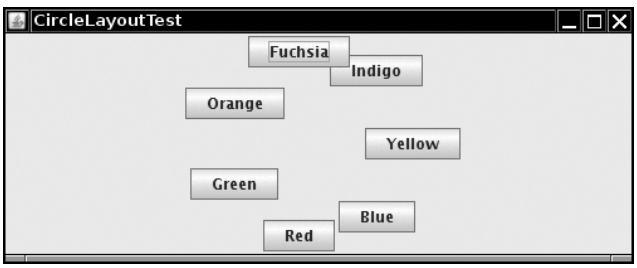


Рис. 9.34. Размещение компонентов по кругу

Ваш собственный класс диспетчера компоновки должен реализовывать интерфейс `LayoutManager`. Вам придется переопределить следующие пять методов:

```
void addLayoutComponent(String s, Component c);
void removeLayoutComponent(Component c);
Dimension preferredLayoutSize(Container parent);
Dimension minimumLayoutSize(Container parent);
void layoutContainer(Container parent);
```

Первые два метода вызываются при добавлении или удалении компонента. Если никакой дополнительной информации о компоненте нет, можно оставить эти методы пустыми. Следующие два метода вычисляют объем пространства, необходимого при минимальных и рекомендуемых размерах компонентов. Обычно эти величины равны. Пятый метод вызывает `setBounds()` для всех компонентов и выполняет основную работу.

На заметку! В библиотеке AWT есть второй интерфейс, который называется `LayoutManager2`. В нем содержатся 10 методов, а не 5. Главная особенность этого интерфейса состоит в том, что с его помощью можно использовать метод `add()` без каких-либо ограничений. Например, классы `BorderLayout` и `GridBagLayout` реализуют именно интерфейс `LayoutManager2`.

В листинге 9.13 приведен код программы, реализующей диспетчер компоновки `CircleLayout`, который размещает компоненты вдоль окружности внутри контейнера. Это довольно забавно, правда, совершенно бесполезно.

Листинг 9.13. Содержимое файла `CircleLayoutTest.java`

```
1.  import java.awt.*;
1.   import javax.swing.*;
2.
3.   /**
4.    * @version 1.32 2007-06-12
5.    * @author Cay Horstmann
6.    */
7.   public class CircleLayoutTest
8.   {
9.      public static void main(String[] args)
10.    {
11.       EventQueue.invokeLater(new Runnable()
12.          {
13.             public void run()
14.             {
15.                CircleLayoutFrame frame = new CircleLayoutFrame();
16.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
17.                frame.setVisible(true);
18.             }
19.          });
20.    }
21.  }
22.
23.  /**
24.   * Фрейм, в котором кнопки размещаются по окружности.
25.   */
26.  class CircleLayoutFrame extends JFrame
27.  {
28.    public CircleLayoutFrame()
29.    {
30.       setTitle("CircleLayoutTest");
31.
32.       setLayout(new CircleLayout());
33.       add(new JButton("Yellow"));
34.       add(new JButton("Blue"));
35.       add(new JButton("Red"));
36.       add(new JButton("Green"));
37.       add(new JButton("Orange"));
38.       add(new JButton("Fuchsia"));
39.       add(new JButton("Indigo"));
40.       pack();
41.    }
42.  }
43.
44.  /**
45.   * Диспетчер компоновки, обеспечивающий размещение
46.   * компонентов по окружности.
47.   */
48.  class CircleLayout implements LayoutManager
49.  {
50.    public void addLayoutComponent(String name, Component comp)
51.    {
52.    }
53.
```

```
54.    public void removeLayoutComponent(Component comp)
55.    {
56.    }
57.
58.    public void setSizes(Container parent)
59.    {
60.       if (sizesSet) return;
61.       int n = parent.getComponentCount();
62.
63.       preferredWidth = 0;
64.       preferredHeight = 0;
65.       minWidth = 0;
66.       minHeight = 0;
67.       maxComponentWidth = 0;
68.       maxComponentHeight = 0;
69.
70.       // Вычисление максимальной ширины и высоты компонента.
71.       // Рекомендуемый размер устанавливается как сумма размеров.
72.       for (int i = 0; i < n; i++)
73.       {
74.          Component c = parent.getComponent(i);
75.          if (c.isVisible())
76.          {
77.             Dimension d = c.getPreferredSize();
78.             maxComponentWidth = Math.max(maxComponentWidth, d.width);
79.             maxComponentHeight = Math.max(maxComponentHeight, d.height);
80.             preferredWidth += d.width;
81.             preferredHeight += d.height;
82.          }
83.       }
84.       minWidth = preferredWidth / 2;
85.       minHeight = preferredHeight / 2;
86.       sizesSet = true;
87.    }
88.
89.    public Dimension preferredLayoutSize(Container parent)
90.    {
91.       setSizes(parent);
92.       Insets insets = parent.getInsets();
93.       int width = preferredWidth + insets.left + insets.right;
94.       int height = preferredHeight + insets.top + insets.bottom;
95.       return new Dimension(width, height);
96.    }
97.
98.    public Dimension minimumLayoutSize(Container parent)
99.    {
100.      setSizes(parent);
101.      Insets insets = parent.getInsets();
102.      int width = minWidth + insets.left + insets.right;
103.      int height = minHeight + insets.top + insets.bottom;
104.      return new Dimension(width, height);
105.   }
106.
107.   public void layoutContainer(Container parent)
108.   {
109.      setSizes(parent);
```

```
110.
111.       // Определение центра окружности.
112.
113.       Insets insets = parent.getInsets();
114.       int containerWidth = parent.getSize().width - insets.left - insets.right;
115.       int containerHeight = parent.getSize().height - insets.top - insets.bottom;
116.
117.       int xcenter = insets.left + containerWidth / 2;
118.       int ycenter = insets.top + containerHeight / 2;
119.
120.       // Определение радиуса окружности.
121.
122.       int xradius = (containerWidth - maxComponentWidth) / 2;
123.       int yradius = (containerHeight - maxComponentHeight) / 2;
124.       int radius = Math.min(xradius, yradius);
125.
126.       // Размещение компонентов по окружности.
127.
128.       int n = parent.getComponentCount();
129.       for (int i = 0; i < n; i++)
130.       {
131.          Component c = parent.getComponent(i);
132.          if (c.isVisible())
133.          {
134.             double angle = 2 * Math.PI * i / n;
135.
136.             // Центр компонента.
137.             int x = xcenter + (int) (Math.cos(angle) * radius);
138.             int y = ycenter + (int) (Math.sin(angle) * radius);
139.
140.             // Компонент располагается так, чтобы его
141.             // центр находился в точке (x, y), а
142.             // размеры равны рекомендуемым.
143.             Dimension d = c.getPreferredSize();
144.             c.setBounds(x - d.width / 2, y - d.height / 2, d.width, d.height);
145.          }
146.       }
147.    }
148.
149.    private int minWidth = 0;
150.    private int minHeight = 0;
151.    private int preferredWidth = 0;
152.    private int preferredHeight = 0;
153.    private boolean sizesSet = false;
154.    private int maxComponentWidth = 0;
155.    private int maxComponentHeight = 0;
156. }
```

API java.awt.LayoutManager 1.0

- `void addLayoutComponent(String name, Component comp)`

  Добавляет компонент в схему компоновки.

  Параметры `name` Идентификатор, задающий координаты компонента.

  `comp` Добавляемый компонент.

- `void removeLayoutComponent(Component comp)`

  Удаляет компонент из схемы компоновки.

- `Dimension preferredLayoutSize(Container parent)`

  Возвращает рекомендуемые размеры контейнера, в котором выполняется компоновка.

- `Dimension minimumLayoutSize(Container parent)`

  Возвращает минимальные размеры контейнера, в котором выполняется компоновка.

- `void layoputContainer(Container parent)`

  Выполняет компоновку внутри контейнера.

Порядок обхода компонентов

Если в окне содержится много компонентов, нужно подумать о *порядке их обхода*. Когда окно открывается в первый раз, фокус ввода находится на том компоненте, который считается первым при заданном порядке обхода. Каждый раз, когда пользователь нажимает клавишу <Tab>, фокус перемещается на следующий компонент. (Напомним, что с компонентом, обладающим фокусом ввода, можно работать с помощью клавиш. Например, на кнопке, владеющей фокусом, можно "щелкнуть" с помощью клавиши пробела.) Вы лично можете оставаться безразличным к такой возможности, однако есть много пользователей, которые предпочитают работать именно так. Среди них есть "ненавистники" мыши, а также те, кто просто не могут ею пользоваться — например, если они выполняют навигацию по интерфейсу голосом. По этой причине вам стоит знать, как Swing обрабатывает порядок обхода.

Этот порядок обхода достаточно разумен: сначала — слева направо, затем — сверху вниз. Например, в диалоговом окне для выбора шрифта обход компонентов осуществлялся в следующем порядке (рис. 9.35).

1. Выпадающий список, предназначенный для выбора начертаний.
2. Текстовая область с фрагментом текста, используемым в качестве образца. (В нем для перехода к следующему полю нажимайте <Ctrl+Tab>, потому что символ <Tab> трактуется как часть вводимого текста.)
3. Выпадающий список, предназначенный для выбора размеров.
4. Флажок опции для установки полужирного шрифта.
5. Флажок опции для установки курсива.



Рис. 9.35. Геометрический порядок обхода компонентов



На заметку! В библиотеке AWT порядок обхода определялся очередностью, в которой компоненты вставлялись в контейнер. В библиотеке Swing порядок включения компонентов не имеет значения — учитывается только вид компоновки.

Если контейнеры вложены друг в друга, ситуация усложняется. Когда фокус передается вложенному контейнеру, его автоматически получает левый верхний компонент, а затем начинается обход остальных компонентов. В конце концов, фокус передается компоненту, следующему за контейнером. Это обстоятельство можно с выгодой использовать, помещая группу связанных между собой элементов в другой контейнер, например панель.



На заметку! Начиная с Java SE 1.4, можно указать, чтобы некоторый компонент при обходе не получал фокус ввода. Ранее для этого нужно было переопределить метод `isFocusTraversable`, но теперь этот метод не рекомендован к применению

Подводя итог, обратим внимание на две стандартные стратегии обхода, принятые в Java SE 1.4.

- Приложения, использующие исключительно AWT, применяют политику `DefaultFocusTraversablePolicy`. Компоненты получают при обходе фокус ввода, если они являются видимыми, отображаются на экране, доступны и передача им фокуса не запрещена. Кроме того, требуется, чтобы их платформенно-зависимые эквиваленты также могли иметь фокус. Обход компонентов осуществляется в том же порядке, в котором они были включены в контейнер.
- Приложения Swing, используют политику `LayoutFocusTraversablePolicy`. Компоненты получают фокус, если они являются видимыми, отображаются на экране, доступны и передача им фокуса не запрещена. Обход производится в геометрическом порядке: слева направо и сверху вниз. Однако вложенный контейнер создает новый "цикл обхода" — сначала перебираются содержащиеся в нем компоненты, а лишь затем фокус передается компоненту, следующему за контейнером.



На заметку! Термин "цикл" может привести к недоразумениям. По достижении последнего элемента дочернего контейнера фокус передается не его первому элементу, а следующему компоненту родительского контейнера. В API реализованы настоящие циклы, включая поддержку клавиш, предназначенных для перехода вверх и вниз по иерархии циклов. Однако стандартная стратегия обхода не использует иерархических циклов, она реализует порядок перебора элементов "в глубину".



На заметку! В Java SE 1.3 существовала возможность изменить порядок обхода, принятый по умолчанию. Для этого надо было вызвать метод `setNextFocusableComponent()` класса `JComponent`. Теперь этот метод не рекомендован к применению. Чтобы изменить порядок перебора компонентов, попробуйте сгруппировать связанные между собой компоненты на отдельных панелях так, чтобы они образовали циклы. Если этот способ не даст требуемого результата, надо либо установить средства переупорядочения компонентов, либо полностью заменить стратегию обхода. Ни один из этих подходов не внушает доверия — детали можно найти в документации по API.

Диалоговые окна

До сих пор мы рассматривали только компоненты пользовательского интерфейса, которые находились в окне фрейма, создаваемого в приложении. Однако при создании приложений возникает необходимость в отдельных всплывающих диалоговых окнах, которые отображались бы на экране и обеспечивали обмен информацией с пользователем.

Как и в большинстве оконных систем, в AWT различаются *модальные* и *немодальные* диалоговые окна. Модальные диалоговые окна не позволяют пользователю одновременно работать с другими окнами приложения. Такие окна необходимы, если пользователь должен ввести информацию, от которой зависит дальнейшая работа приложения. Например, при считывании файла пользователь должен сначала указать его имя. Только когда пользователь закроет модальное диалоговое окно, программа приступит к выполнению операции считывания.

Немодальное диалоговое окно позволяет пользователю одновременно вводить информацию как в нем, так и в других окнах приложения. Примером такого окна является панель инструментов. Она остается на месте сколь угодно долго, и пользователь может одновременно взаимодействовать как с ней, так и с другими окнами.

Начнем этот раздел с простейшего модального диалогового окна, содержащего строку сообщения. В пакете Swing есть удобный класс `JOptionPane`, позволяющий выводить на экран модальное диалоговое окно, не создавая специального кода для его поддержки. Далее мы расскажем, как можно реализовать свой собственный диалог. А в заключение покажем, как передавать информацию из приложения в диалоговое окно и обратно.

Этот раздел завершается анализом двух стандартных диалоговых окон, предназначенных для работы с файлами и установки цветов. Диалоговое окно для работы с файлами устроено достаточно сложно. Для работы с ним необходимо хорошо знать класс `JFileChooser` из библиотеки Swing. Для выбора цветов из палитры вам может пригодиться диалоговое окно `JColorChooser`.

Диалоговые окна для выбора опций

В библиотеке Swing содержится много готовых диалоговых окон, которые позволяют вводить небольшие объемы информации. В классе `JOptionPane` есть четыре статических метода:

`showMessageDialog`	Выводит на экран сообщение и ожидает, пока пользователь не щелкнет на кнопке OK.
`showConfirmDialog`	Выводит на экран сообщение и ждет подтверждения (OK или Cancel (Отмена)).
`showOptionDialog`	Выводит на экран сообщение и предоставляет пользователю выбор из нескольких вариантов.
`showInputDialog`	Выводит на экран сообщение и поле редактирования, в котором пользователь должен ввести информацию.

На рис. 9.36 показано типичное диалоговое окно. Как видите, в нем содержатся следующие компоненты:

- пиктограмма;
- сообщение;
- одна или несколько кнопок.

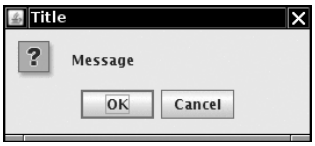


Рис. 9.36. Диалоговое окно, содержащее пиктограмму, сообщение и две кнопки

Диалоговое окно может содержать компонент для ввода данных. Этим компонентом может быть поле редактирования, в котором пользователь вводит произвольную строку, или выпадающий список, один из пунктов которого пользователь должен выбрать.

Компоновка диалоговых окон и выбор пиктограмм для стандартных сообщений зависит от стиля интерфейса.

Пиктограмма в левой части окна выбирается в зависимости от типа сообщения. Существуют пять типов сообщений:

```
ERROR_MESSAGE
INFORMATION_MESSAGE
WARNING_MESSAGE
QUESTION_MESSAGE
PLAIN_MESSAGE
```

Для сообщения `PLAIN_MESSAGE` пиктограмма не предусмотрена. Для каждого из типов диалоговых окон существует также метод, позволяющий использовать свою собственную пиктограмму.

С каждым из типов диалоговых окон можно связать определенное сообщение, которое может быть строкой, пиктограммой, компонентом пользовательского интерфейса или любым другим объектом. Ниже показано, как эти сообщения отображаются на экране.

`String`	Выводит строку.
`Icon`	Отображает пиктограмму.
`Component`	Отображает компонент.
`Object[]`	Выводит все объекты, отображая их один поверх другого.
Любой другой объект	Применяет метод `toString` и выводит на экран строку, полученную в результате этого преобразования.

Все эти возможности продемонстрированы в программе, приведенной в листинге 9.14.

Разумеется, в большинстве случаев на экран выводится именно строка сообщения. В то же время возможность отображать в диалоговом окне объекты класса `Component` обеспечивает бо́льшую гибкость.

Внешний вид кнопок, размещенных в нижней части окна, зависит от типа диалога и типа опций. Вызывая методы `showMessageDialog()` или `showInputDialog()`, вы ограничиваетесь стандартным набором кнопок (OK либо OK и Cancel). Вызывая метод `showConfirmDialog()`, можно выбрать один из четырех типов опций:

```
DEFAULT_OPTION
YES_NO_OPTION
YES_NO_CANCEL_OPTION
OK_CANCEL_OPTION
```

С помощью метода `showOptionDialog()` можно указать произвольный набор вариантов, задав массив объектов, соответствующих каждому из них. Элементы этого массива отображаются на экране следующим образом.

`String`	Создает кнопку, меткой которой является заданная строка.
`Icon`	Создает кнопку, меткой которой является заданная пиктограмма.
`Component`	Отображает компонент.
Любой другой объект	Применяет метод `toString` и создает кнопку, меткой которой является полученная в результате строка.

Статические методы, предназначенные для создания диалоговых окон, возвращают следующие значения.

`showMessageDialog`	Возвращаемое значение отсутствует.
`showConfirmDialog`	Целое число, соответствующее выбранному варианту.
`showOptionDialog`	Целое число, соответствующее выбранному варианту.
`showInputDialog`	Строка, введенная или выбранная пользователем.

Методы `showConfirmDialog()` и `showOptionDialog()` возвращают целое число, обозначающее кнопку, на которой щелкнул пользователь. В диалоговом окне для выбора опций это число является порядковым номером. Если вместо выбора опции пользователь закрыл диалоговое окно, возвращается значение `CLOSED_OPTION`,. Константы, используемые в качестве возвращаемых значений, приведены ниже.

```
OK_OPTION
CANCEL_OPTION
YES_OPTION
NO_OPTION
CLOSED_OPTION
```

Несмотря на обилие мнемонических обозначений, создать диалоговое окно достаточно просто.

1. Выберите тип диалогового окна (сообщение, подтверждение, выбор или ввод).
2. Выберите пиктограмму (ошибка, информация, предупреждение, вопрос, своя собственная или никакой).
3. Выберите сообщение (строка, пиктограмма, пользовательский компонент или массив компонентов).
4. Если вы выбрали диалоговое окно для подтверждения выбора, задайте тип вариантов (по умолчанию, Yes/No, No/Cancel или OK/Cancel).
5. Если вы выбрали диалоговое окно для выбора опций, задайте варианты (строки, пиктограммы или собственные компоненты), а также вариант, выбираемый по умолчанию.
6. Если вы выбрали диалоговое окно для ввода данных, выберите поле редактирования или выпадающий список.
7. Найдите подходящий метод в классе `JOptionPane`.

Предположим, что вы хотите отобразить на экране диалоговое окно, показанное на рис. 9.36. В этом окне выводится сообщение, а пользователю предлагается подтвердить его или отклонить. Следовательно, это — диалоговое окно для подтверждения выбора. Пиктограмма, отображаемая в этом окне, относится к предупреждениям.

Сообщение является строкой. Набор вариантов задается константой OK_CANCEL_OPTION. Итак, для создания такого диалогового окна выполните следующее:

```
int selection = JOptionPane.showConfirmDialog(parent,
  "Message", "Title",
  JOptionPane.OK_CANCEL_OPTION,
  JOprionPane.WARNING_MESSAGE);
if (selection == JOptionPane.OK_OPTION) ...
```

Совет. Строка сообщения может содержать символ перевода строки ('\n'). В этом случае сообщение выводится в виде нескольких строк.

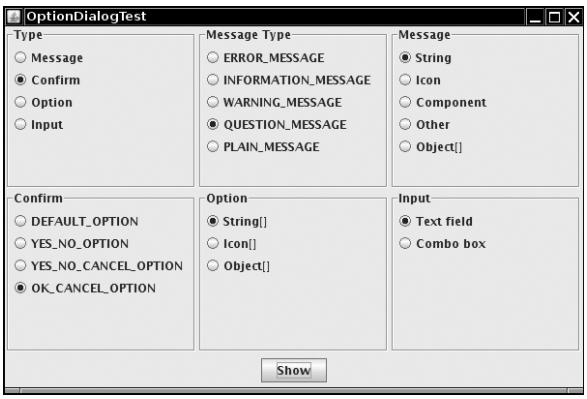


Рис. 9.37. Окно программы OptionDialogTest

Программа, приведенная в листинге 9.14, позволяет с помощью окна, показанного на рис. 9.37, сделать требуемые установки, генерирует с учетом установок диалоговое окно и отображает его.

Листинг 9.14. Содержимое файла OptionDialogTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.awt.geom.*;
4. import java.util.*;
5. import javax.swing.*;
6.
7. /**
8.  * @version 1.33 2007-04-28
9.  * @author Cay Horstmann
10. */
11.public class OptionDialogTest
12.{
13.   public static void main(String[] args)
14.   {
15.      EventQueue.invokeLater(new Runnable()
16.         {
17.            public void run()
18.            {
19.               OptionDialogFrame frame = new OptionDialogFrame();
```

```
20.             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21.             frame.setVisible(true);
22.          }
23.       });
24.   }
25. }
26.
27. /**
28.  * Панель с переключателями.
29.  */
30. class ButtonPanel extends JPanel
31. {
32.   /**
33.    * Конструктор панели.
34.    * @param title Заголовок на рамке
35.    * @param options Массив меток переключателей
36.    */
37.   public ButtonPanel(String title, String... options)
38.   {
39.     setBorder(BorderFactory.createTitledBorder(
40.         BorderFactory.createEtchedBorder(), title));
41.     setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
42.     group = new ButtonGroup();
43.
44.     // Для каждого заголовка создается переключатель.
45.     for (String option : options)
46.     {
47.       JRadioButton b = new JRadioButton(option);
48.       b.setActionCommand(option);
49.       add(b);
50.       group.add(b);
51.       b.setSelected(option == options[0]);
52.     }
53.   }
54.
55.   /**
56.    * Определение текущего выбора.
57.    * @return Метка выбранного переключателя
58.    */
59.   public String getSelection()
60.   {
61.     return group.getSelection().getActionCommand();
62.   }
63.
64.   private ButtonGroup group;
65. }
66.
67. /**
68.  * Фрейм, позволяющий выбирать характеристики диалогового окна.
69.  */
70. class OptionDialogFrame extends JFrame
71. {
72.   public OptionDialogFrame()
73.   {
74.     setTitle("OptionDialogTest");
75.     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
```

```
76.
77.     JPanel gridPanel = new JPanel();
78.     gridPanel.setLayout(new GridLayout(2, 3));
79.
80.     typePanel = new ButtonPanel("Type", "Message", "Confirm", "Option", "Input");
81.     messageTypePanel = new ButtonPanel("Message Type", "ERROR_MESSAGE",
82.        "INFORMATION_MESSAGE", "WARNING_MESSAGE", "QUESTION_MESSAGE",
83.        "PLAIN_MESSAGE");
84.     messagePanel = new ButtonPanel("Message", "String", "Icon",
85.        "Component", "Other", "Object[]");
86.     optionTypePanel = new ButtonPanel("Confirm", "DEFAULT_OPTION", "YES_NO_OPTION",
87.        "YES_NO_CANCEL_OPTION", "OK_CANCEL_OPTION");
88.     optionsPanel = new ButtonPanel("Option", "String[]", "Icon[]", "Object[]");
89.     inputPanel = new ButtonPanel("Input", "Text field", "Combo box");
90.
91.     gridPanel.add(typePanel);
92.     gridPanel.add(messageTypePanel);
93.     gridPanel.add(messagePanel);
94.     gridPanel.add(optionTypePanel);
95.     gridPanel.add(optionsPanel);
96.     gridPanel.add(inputPanel);
97.
98.     // Добавление панели с кнопкой Show.
99.
100.    JPanel showPanel = new JPanel();
101.    JButton showButton = new JButton("Show");
102.    showButton.addActionListener(new ShowAction());
103.    showPanel.add(showButton);
104.
105.    add(gridPanel, BorderLayout.CENTER);
106.    add(showPanel, BorderLayout.SOUTH);
107. }
108.
109. /**
110.  * Получение выбранного сообщения.
111.  * @return Строка, пиктограмма, компонент или массив объектов
112.  */
113. public Object getMessage()
114. {
115.    String s = messagePanel.getSelection();
116.    if (s.equals("String")) return messageString;
117.    else if (s.equals("Icon")) return messageIcon;
118.    else if (s.equals("Component")) return messageComponent;
119.    else if (s.equals("Object[]")) return new Object[]
120.      { messageString, messageIcon, messageComponent, messageObject };
121.    else if (s.equals("Other")) return messageObject;
122.    else return null;
123. }
124.
125. /**
126.  * Получение выбранной опции.
127.  * @return Массив строк, пиктограмм или объектов,
128.  * в зависимости от выбора в панели Option
129.  */
```

```
130.  public Object[] getOptions()
131.  {
132.     String s = optionsPanel.getSelection();
133.     if (s.equals("String[]")) return new String[] { "Yellow", "Blue", "Red" };
134.     else if (s.equals("Icon[]")) return new Icon[]
135.      { new ImageIcon("yellow-ball.gif"),
136.        new ImageIcon("blue-ball.gif"), new ImageIcon("red-ball.gif") };
137.     else if (s.equals("Object[]")) return new Object[]
138.      { messageString, messageIcon, messageComponent, messageObject };
139.     else return null;
140.  }
141.
142.  /**
143.   * Получение типа сообщения или набора вариантов.
144.   * @param panel Панель Message Type или Confirm
145.   * @return Константа XXX_MESSAGE или XXX_OPTION,
146.     выбранная в классе JOptionPane
147.   */
148.  public int getType(ButtonPanel panel)
149.  {
150.     String s = panel.getSelection();
151.     try
152.     {
153.        return JOptionPane.class.getField(s).getInt(null);
154.     }
155.     catch (Exception e)
156.     {
157.        return -1;
158.     }
159.  }
160.
161.  /**
162.   * Слушатель действия для кнопки Show. Отображает
163.   * диалоговое окно, тип которого зависит от выбора
164.   * в панели Type.
165.   */
166.  private class ShowAction implements ActionListener
167.  {
168.     public void actionPerformed(ActionEvent event)
169.     {
170.        if (typePanel.getSelection().equals("Confirm"))
171.          JOptionPane.showConfirmDialog(
172.           OptionDialogFrame.this, getMessage(), "Title",
173.              getType(optionTypePanel),
174.              getType(messageTypePanel));
175.        else if (typePanel.getSelection().equals("Input"))
176.        {
177.           if (inputPanel.getSelection().equals("Text field"))
178.             JOptionPane.showInputDialog(
179.              OptionDialogFrame.this, getMessage(), "Title",
180.                getType(messageTypePanel));
181.           else JOptionPane.showInputDialog(OptionDialogFrame.this,
182.              getMessage(), "Title", getType(messageTypePanel),
183.              null, new String[] { "Yellow", "Blue", "Red" }, "Blue");
184.        }
```

```
185.       else if (typePanel.getSelection().equals("Message"))
186.         JOptionPane.showMessageDialog(
187.          OptionDialogFrame.this, getMessage(),
188.          "Title", getType(messageTypePanel));
189.       else if (typePanel.getSelection().equals("Option"))
190.         JOptionPane.showOptionDialog(
191.          OptionDialogFrame.this, getMessage(), "Title", getType(optionTypePanel),
192.             getType(messageTypePanel), null, getOptions(), getOptions()[0]);
193.     }
194.  }
195.
196.   public static final int DEFAULT_WIDTH = 600;
197.   public static final int DEFAULT_HEIGHT = 400;
198.
199.   private ButtonPanel typePanel;
200.   private ButtonPanel messagePanel;
201.   private ButtonPanel messageTypePanel;
202.   private ButtonPanel optionTypePanel;
203.   private ButtonPanel optionsPanel;
204.   private ButtonPanel inputPanel;
205.
206.   private String messageString = "Message";
207.   private Icon messageIcon = new ImageIcon("blue-ball.gif");
208.   private Object messageObject = new Date();
209.   private Component messageComponent = new SampleComponent();
210.}
211.
212./**
213. * Компонент с графическим изображением.
214. */
215.
216.class SampleComponent extends JComponent
217.{
218.   public void paintComponent(Graphics g)
219.   {
220.      Graphics2D g2 = (Graphics2D) g;
221.      Rectangle2D rect = new Rectangle2D.Double(0, 0, getWidth() - 1,
222.                           getHeight() - 1);
223.      g2.setPaint(Color.YELLOW);
224.      g2.fill(rect);
225.      g2.setPaint(Color.BLUE);
226.      g2.draw(rect);
227.   }
228.
229.   public Dimension getPreferredSize()
230.   {
231.      return new Dimension(10, 10);
232.   }
233.}
```

API javax.swing.JOptionPane 1.2

- `static void showMessageDialog(Component parent, Object message, String title, int messageType, Icon icon)`
- `static void showMessageDialog(Component parent, Object message, String title, int messageType)`
- `static void showMessageDialog(Component parent, Object message)`
- `static void showInternalMessageDialog(Component parent, Object message, String title, int messageType, Icon icon)`
- `static void showInternalMessageDialog(Component parent, Object message, String title, int messageType)`
- `static void showInternalMessageDialog(Component parent, Object message)`

Выводит на экран обычное диалоговое окно или внутреннее диалоговое окно с сообщением. (Внутреннее диалоговое окно целиком содержится внутри фрейма.)

Параметры		
	`parent`	Родительский компонент (значение этого параметра может быть равно `null`).
	`message`	Сообщение, которое выводится в диалоговом окне (может быть строкой, пиктограммой, компонентом или массивом компонентов).
	`title`	Строка заголовка диалогового окна.
	`messageType`	Одна из следующих констант: `ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `WARNING_MESSAGE`, `QUESTION_MESSAGE` или `PLAIN_MESSAGE`.
	`icon`	Пиктограмма, которая выводится вместо стандартной.

- `static int showConfirmDialog(Component parent, Object message, String title, int optionType, Icon icon)`
- `static int showConfirmDialog(Component parent, Object message, String title, int optionType, int messageType)`
- `static int showConfirmDialog(Component parent, Object message, String title, int optionType)`
- `static int showConfirmDialog(Component parent, Object message)`
- `static int showInternalConfirmDialog(Component parent, Object message, String title, int optionType, Icon icon)`
- `static int showInternalConfirmDialog(Component parent, Object message, String title, int optionType, int messageType)`
- `static int showInternalConfirmDialog(Component parent, Object message, String title, int optionType)`
- `static int showInternalConfirmDialog(Component parent, Object message)`

Выводит на экран диалоговое или внутреннее диалоговое окно для подтверждения выбора. (Внутреннее диалоговое окно целиком отображается внутри собственного фрейма.) Возвращает опцию, выбранную пользователем (одну из следующих констант: `OK_OPTION`, `YES_OPTION` или `NO_OPTION`), либо `CLOSED_OPTION`, если пользователь закрыл диалоговое окно.

Параметры

`parent`	Родительский компонент (значение этого параметра может быть равно `null`).
`message`	Сообщение, которое выводится в диалоговом окне (может быть строкой, пиктограммой, компонентом или массивом компонентов).
`title`	Строка заголовка диалогового окна.
`messageType`	Одна из следующих констант: `ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `WARNING_MESSAGE`, `QUESTION_MESSAGE` или `PLAIN_MESSAGE`.
`optionType`	Одна из следующих констант: `DEFAULT_OPTION`, `YES_NO_OPTION`, `YES_NO_CANCEL_OPTION` или `OK_CANCEL_OPTION`.
`icon`	Пиктограмма, которая выводится вместо стандартной.

- `static int showOptionDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon, Object[] options, Object default)`
- `static int showInternalOptionDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon, Object[] options, Object default)`

Выводит на экран диалоговое окно или внутреннее диалоговое окно для выбора варианта. (Внутреннее диалоговое окно целиком отображается внутри собственного фрейма.) Возвращает номер варианта, выбранного пользователем, или константу `CLOSED_OPTION`, если пользователь закрыл диалоговое окно.

Параметры

`parent`	Родительский компонент (значение этого параметра может быть равно `null`).
`message`	Сообщение, которое выводится в диалоговом окне (может быть строкой, пиктограммой, компонентом или массивом компонентов).
`title`	Строка заголовка диалогового окна.
`messageType`	Одна из следующих констант: `ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `WARNING_MESSAGE`, `QUESTION_MESSAGE` или `PLAIN_MESSAGE`.
`optionType`	Одна из следующих констант: `DEFAULT_OPTION`, `YES_NO_OPTION`, `YES_NO_CANCEL_OPTION` или `OK_CANCEL_OPTION`.
`icon`	Пиктограмма, которая выводится вместо стандартной.
`options`	Массив опций, элементами которого могут быть строки, пиктограммы или компоненты.
`default`	Опция, предлагаемая по умолчанию.

- `static Object showInputDialog(Component parent, Object message, String title, int messageType, Icon icon, Object[] values, Object default)`
- `static String showInputDialog(Component parent, Object message, String title, int messageType)`

- `static String showInputDialog(Component parent, Object message)`
- `static String showInputDialog(Object message)`
- `static String showInputDialog(Component parent, Object message, Object default)` **1.4**
- `static String showInputDialog(Object message, Object default)` **1.4**
- `static Object showInternalInputDialog(Component parent, Object message, String title, int messageType, Icon icon, Object[] values, Object default)`
- `static String showInternalInputDialog(Component parent, Object message, String title, int messageType)`
- `static Object showInternalInputDialog(Component parent, Object message)`

Выводит на экран диалоговое окно или внутреннее диалоговое окно для ввода данных. (Внутреннее диалоговое окно целиком отображается внутри собственного фрейма.) Возвращает строку, введенную пользователем, или значение `null`, если пользователь закрыл окно.

Параметры		
	`parent`	Родительский компонент (значение этого параметра может быть равно `null`).
	`message`	Сообщение, которое выводится в диалоговом окне (может быть строкой, пиктограммой, компонентом или массивом компонентов).
	`title`	Строка заголовка диалогового окна.
	`messageType`	Одна из следующих констант: `ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `WARNING_MESSAGE`, `QUESTION_MESSAGE` или `PLAIN_MESSAGE`.
	`optionType`	Одна из следующих констант: `DEFAULT_OPTION`, `YES_NO_OPTION`, `YES_NO_CANCEL_OPTION` или `OK_CANCEL_OPTION`.
	`icon`	Пиктограмма, которая выводится вместо стандартной.
	`values`	Массив значений, предлагаемых в выпадающем списке.
	`default`	Значение, предлагаемое пользователю по умолчанию.

Создание диалоговых окон

Как было сказано ранее, для использования предопределенных диалоговых окон применяется класс `JOptionPane`. Теперь мы опишем, как "вручную" создать свое собственное диалоговое окно.

На рис. 9.38 показано типичное модальное диалоговое окно, содержащее сообщение. Подобное окно отображается на экране после того, как пользователь выбрал пункт меню About.

Для того чтобы реализовать такое окно, необходимо создать подкласс класса `JDialog`. По сути, это тот же процесс, которым мы воспользовались для создания главного окна приложения, расширяя класс `JFrame`.

Точнее говоря, нужно сделать следующее.

1. В конструкторе вашего диалогового окна вызовите конструктор суперкласса `JDialog`. Добавьте в диалоговое окно компоненты пользовательского интерфейса.
2. Добавьте обработчики событий.
3. Задайте размер диалогового окна.

При вызове конструктора суперкласса нужно указать фрейм-владелец, заголовок окна и признак модальности.

Фрейм-владелец управляет местом отображения диалога. Вы можете указать null вместо владельца — тогда диалог будет принадлежать скрытому фрейму.

Признак модальности указывает на то, должны ли блокироваться другие окна вашего приложения, пока отображается диалог. Немодальные окна не блокируют другие окна. Модальный диалог блокирует все остальные окна приложения (за исключением дочерних по отношению к этому диалогу). Вы используете немодальные диалоги для создания панелей инструментов, к которым постоянно открыт доступ. С другой стороны, модальные диалоги вы будете использовать, чтобы вынудить пользователя ввести необходимую информацию, прежде чем продолжить работу приложения.



На заметку! Начиная с Java SE 6, добавлены два дополнительных типа модальности. Документно-модальный диалог блокирует все окна, относящиеся к одному "документу", или, точнее говоря, все окна с тем же родительским корневым окном, что и у данного диалога. Это решает проблему систем помощи. В более старых версиях пользователи не могли взаимодействовать с окнами справки, когда всплывал модальный диалог. Toolkit-модальный диалог блокирует все окна из одного и того же "toolkit" (набора инструментов). Toolkit — это программа Java, запускающая несколько приложений, такая как механизм аплетов в браузере. Дополнительную информацию на эту тему можно найти по адресу `http://java.sun.com/developer/technicalArticles/J2SE/Desktop/javase6/modality`.



Рис. 9.38. Диалоговое окно About

Ниже приведен фрагмент программы, создающий диалоговое окно.

```
public AboutDialog extends JDialog
{
   public AboutDialog(JFrame owner)
   {
      super(owner, "About DialogTest", true);
      add(new JLabel(
         "<html><h1><i>Core Java</i></h1><hr>
         By Cay Horstmann and Gary Cornell</html>"),
         BorderLayout.CENTER);

      JPanel panel = new JPanel();
      JButton ok = new JButton("OK");
```

```
      ok.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               setVisible(false);
            }
         });

      panel.add(ok);
      add(panel, BorderLayout.SOUTH);

      setSize(250, 150);
   }
}
```

Как видите, конструктор добавляет элементы пользовательского интерфейса, в данном случае — метки и кнопку. Кроме того, он добавляет обработчик кнопки и задает размер окна.

Для того чтобы отобразить диалоговое окно на экране, необходимо создать новый объект `JDialog` и вызвать метод `setVisible()`:

```
JDialog dialog = new AboutDialog(this);
dialog.setVisible(true);
```

Фактически в программе, рассматриваемой здесь в качестве примера, диалоговое окно создается только единожды, а затем мы используем его повторно каждый раз, когда пользователь щелкает на кнопке About:

```
if (dialog == null) // Первый раз.
   dialog == new AboutDialog(this);
dialog.setVisible(true);
```

Когда пользователь щелкает на кнопке OK, диалоговое окно должно закрываться. Эта реакция определяется в обработчике событий, связанных с кнопкой OK:

```
ok.addActionListener(new
   ActionListener()
   {
      public void actionPerformed(ActionEvent evt)
      {
         setVisible(false);
      }
   });
```

Когда пользователь закрывает окно диалога, щелкая на кнопке Close, оно исчезает с экрана. Как и в классе `JFrame`, программист может изменить эту реакцию с помощью метода `setDefaultCloseOperation()`.

Программа, демонстрирующая использование диалогового окна About, приведена в листинге 9.15.

Листинг 9.15. Содержимое файла `DialogTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import javax.swing.*;
4.
```

```
5.  /**
6.   * @version 1.33 2007-06-12
7.   * @author Cay Horstmann
8.   */
9.  public class DialogTest
10. {
11.   public static void main(String[] args)
12.   {
13.      EventQueue.invokeLater(new Runnable()
14.      {
15.         public void run()
16.         {
17.            DialogFrame frame = new DialogFrame();
18.            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.            frame.setVisible(true);
20.         }
21.      });
22.   }
23. }
24.
25. /**
26.  * Фрейм содержит меню. При выборе пункта File->About
27.  * отображается диалоговое окно.
28.  */
29. class DialogFrame extends JFrame
30. {
31.   public DialogFrame()
32.   {
33.      setTitle("DialogTest");
34.      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
35.
36.      // Создание меню File.
37.
38.      JMenuBar menuBar = new JMenuBar();
39.      setJMenuBar(menuBar);
40.      JMenu fileMenu = new JMenu("File");
41.      menuBar.add(fileMenu);
42.
43.      // Добавление пунктов меню About и Exit.
44.
45.      // При выборе пункта About отображается
46.      // диалоговое окно About.
47.
48.      JMenuItem aboutItem = new JMenuItem("About");
49.      aboutItem.addActionListener(new ActionListener()
50.      {
51.         public void actionPerformed(ActionEvent event)
52.         {
53.            if (dialog == null) // в первый раз
54.               dialog = new AboutDialog(DialogFrame.this);
55.            dialog.setVisible(true); // отобразить диалог
56.         }
57.      });
58.      fileMenu.add(aboutItem);
59.
60.      // При активизации пункта Exit программа завершается.
```

```
61.
62.       JMenuItem exitItem = new JMenuItem("Exit");
63.       exitItem.addActionListener(new ActionListener()
64.       {
65.          public void actionPerformed(ActionEvent event)
66.          {
67.             System.exit(0);
68.          }
69.       });
70.       fileMenu.add(exitItem);
71.    }
72.
73.    public static final int DEFAULT_WIDTH = 300;
74.    public static final int DEFAULT_HEIGHT = 200;
75.
76.    private AboutDialog dialog;
77. }
78.
79. /**
80.  * Модальное диалоговое окно отображает сообщение
81.  * и ожидает щелчка на кнопке OK.
82.  */
83. class AboutDialog extends JDialog
84. {
85.    public AboutDialog(JFrame owner)
86.    {
87.       super(owner, "About DialogTest", true);
88.
89.       // Метка с HTML-форматированием выравнивается по центру.
90.
91.       add(new JLabel(
92.          "<html><h1><i>Core Java</i></h1><hr>
93.          By Cay Horstmann and Gary Cornell</html>"),
94.          BorderLayout.CENTER);
95.
96.       // При активизации кнопки OK диалоговое окно закрывается.
97.
98.       JButton ok = new JButton("Ok");
99.       ok.addActionListener(new ActionListener()
100.      {
101.         public void actionPerformed(ActionEvent event)
102.         {
103.            setVisible(false);
104.         }
105.      });
106.
107.      // Кнопка OK помещается в нижнюю часть окна.
108.
109.      JPanel panel = new JPanel();
110.      panel.add(ok);
111.      add(panel, BorderLayout.SOUTH);
112.
113.      setSize(250, 150);
114.   }
115.}
```

API javax.swing.JDialog 1.2

- `public JDialog(JFrame parent, String title, boolean modal)`

  Создает диалоговое окно, которое является невидимым, пока не будет принудительно выведено на экран.

  Параметры `parent` Родительский компонент (значение этого параметра может быть равно `null`).

  `title` Строка заголовка диалогового окна.

  `modal` Если значение этого параметра равно `true`, диалоговое окно является модальным (блокирует доступ к остальным окнам).

Передача данных

Чаще всего диалоговые окна создаются для того, чтобы получить информацию от пользователя. Выше было показано, насколько легко создавать объекты класса `JDialog`: задайте исходные данные, а затем вызывайте метод `setVisible(true)`, который выведет окно на экран. Теперь посмотрим, как можно вводить данные с помощью диалогового окна.

Обратите внимание на окно, показанное на рис. 9.39. Его можно использовать для получения имени пользователя и пароля.



Рис. 9.39. Диалоговое окно для ввода пароля

В диалоговом окне должны быть предусмотрены методы, генерирующие данные по умолчанию. Например, в классе `PasswordChooser` есть метод `setUser` для помещения значений по умолчанию в следующие поля:

```
public void setUser(User u)
{
   userName.setText(u.getName());
}
```

Установив при необходимости значения по умолчанию, окно выводится на экран. Для этого вызывается метод `setVisible(true)`.

Пользователь должен ввести информацию и щелкнуть на кнопке OK или Cancel. Обработчики событий, связанных с обеими кнопками, вызывают метод `setVisible(false)`. Этот вызов выполняет действия, обратные `setVisible(true)`. В качестве альтернативы пользователь может закрыть диалог. Если слушатель событий, связанных с диалоговым окном, не установлен, то выполняются обычные операции по закрытию окон: диалоговое окно становится невидимым.

Обратите внимание, что вызов метода `setVisible(true)` блокирует выполнение программы до тех пор, пока пользователь не выполнит действие, приводящее к за-

крытию окна. Такой подход существенно упрощает создание и использование модальных диалоговых окон.

В большинстве случаев разработчику программы необходима информация о том, подтвердил ли пользователь введенную им информацию. В программе, рассматриваемой в качестве примера, используется следующий подход. Перед обращением к диалоговому окну устанавливается значение `false` переменной `ok`. Значение `true` присваивается этой переменной только в обработчике, связанном с кнопкой OK. В этом случае введенные пользователем данные могут быть использованы программой.



На заметку! Передача данных из немодального диалога не так проста. При отображении такого окна вызов `setVisible(true)` не блокирует выполнение программы. Если пользователь выполнит некоторые установки в немодальном диалоговом окне и щелкнет на кнопке OK, это окно должно оповестить некоторый слушатель в программе.

В рассматриваемой здесь программе присутствуют и другие важные усовершенствования. При создании объекта класса `JDialog` должен быть указан фрейм-владелец. Однако зачастую приходится отображать на экране одно и то же диалоговое окно с разными фреймами. Гораздо удобнее, если фрейм-владелец задается, *когда диалоговое окно готово к выводу на экран*, а не во время создания объекта класса `PasswordChooser`.

Этого можно добиться, если класс `PasswordChooser` будет расширять класс `JPanel`, а не класс `JDialog`. Попробуем создать объект `JDialog` при выполнении метода `showDialog()`.

```
public boolean showDialog(Frame owner, String title)
{
   ok = false;

   if (dialog == null || dialog.getOwner() != owner)
   {
      dialog = new JDialog(owner, true);
      dialog.add(this);
      dialog.pack();
   }

   dialog.setTitle(title);
   dialog.setVisible(true);
   return ok;
}
```

Отметим, что для большей безопасности значение ссылки `owner` должно быть равно `null`.

Иногда фрейм-владелец бывает недоступен. Однако его можно вычислить с помощью объекта `parent`:

```
Frame owner;
if (parent instanceof Frame)
   owner = (Frame) parent;
else
   owner = (Frame) SwingUtilities.getAncestorOfClass
      (Frame.class, parent);
```

Этот подход применяется в нашей программе. Класс `JOptionPane` также использует этот механизм.

Многие диалоговые окна имеют *кнопку по умолчанию*, которая выбирается автоматически, если пользователь нажимает клавишу ввода (в большинстве стилей этой клавишей является <Enter>). Кнопка, заданная по умолчанию, выделяется среди остальных, чаще всего контуром.

Для установки кнопки по умолчанию на *корневой панели* диалогового окна нужно вызвать следующий метод:

```
dialog.getRootPane().setDefaultButton(okButton);
```

Если вы собираетесь разместить интерфейсные элементы в панели, будьте осторожны: устанавливайте кнопку по умолчанию только после оформления панели в виде диалогового окна. Панель сама по себе не имеет корневой панели.

В листинге 9.16 приведен полный текст программы, иллюстрирующей обмен данными с диалоговым окном.

Листинг 9.16. Содержимое файла `DataExchangeTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import javax.swing.*;
4.
5.  /**
6.   * @version 1.33 2007-06-12
7.   * @author Cay Horstmann
8.   */
9.  public class DataExchangeTest
10. {
11.   public static void main(String[] args)
12.   {
13.      EventQueue.invokeLater(new Runnable()
14.         {
15.            public void run()
16.            {
17.               DataExchangeFrame frame = new DataExchangeFrame();
18.               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.               frame.setVisible(true);
20.            }
21.         });
22.   }
23. }
24.
25. /**
26.  * Фрейм с меню. При выборе пункта File->Connect
27.  * отображается диалоговое окно для ввода пароля.
28.  */
29. class DataExchangeFrame extends JFrame
30. {
31.   public DataExchangeFrame()
32.   {
33.      setTitle("DataExchangeTest");
34.      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
35.
36.      // Создание меню File.
37.
38.      JMenuBar mbar = new JMenuBar();
39.      setJMenuBar(mbar);
40.      JMenu fileMenu = new JMenu("File");
```

```
41.       mbar.add(fileMenu);
42.
43.       // Создание пунктов Connect и Exit.
44.
45.       JMenuItem connectItem = new JMenuItem("Connect");
46.       connectItem.addActionListener(new ConnectAction());
47.       fileMenu.add(connectItem);
48.
49.       // При активизации пункта меню Exit программа завершается.
50.
51.       JMenuItem exitItem = new JMenuItem("Exit");
52.       exitItem.addActionListener(new ActionListener()
53.       {
54.          public void actionPerformed(ActionEvent event)
55.          {
56.             System.exit(0);
57.          }
58.       });
59.       fileMenu.add(exitItem);
60.
61.       textArea = new JTextArea();
62.       add(new JScrollPane(textArea), BorderLayout.CENTER);
63.    }
64.
65.    public static final int DEFAULT_WIDTH = 300;
66.    public static final int DEFAULT_HEIGHT = 200;
67.
68.    private PasswordChooser dialog = null;
69.    private JTextArea textArea;
70.
71.    /**
72.     * Действие Connect отображает диалоговое окно для ввода пароля.
73.     */
74.
75.    private class ConnectAction implements ActionListener
76.    {
77.       public void actionPerformed(ActionEvent event)
78.       {
79.          // При первом обращении создается диалоговое окно.
80.
81.          if (dialog == null) dialog = new PasswordChooser();
82.
83.          // Установка значений по умолчанию.
84.          dialog.setUser(new User("yourname", null));
85.
86.          // Вывод диалогового окна.
87.          if (dialog.showDialog(DataExchangeFrame.this, "Connect"))
88.          {
89.             // Если пользователь подтвердил введенные
90.             // данные, они используются в программе.
91.             User u = dialog.getUser();
92.             textArea.append("user name = " + u.getName() + ", password = "
93.                + (new String(u.getPassword())) + "\n");
94.          }
95.       }
96.    }
97. }
```

```
98.
99. /**
100. * Окно для ввода пароля, отображаемое в диалоговом окне.
101. */
102.class PasswordChooser extends JPanel
103.{
104.   public PasswordChooser()
105.   {
106.      setLayout(new BorderLayout());
107.
108.      // Создание панели с полями для ввода
109.      // имени пользователя и пароля.
110.
111.      JPanel panel = new JPanel();
112.      panel.setLayout(new GridLayout(2, 2));
113.      panel.add(new JLabel("User name:"));
114.      panel.add(username = new JTextField(""));
115.      panel.add(new JLabel("Password:"));
116.      panel.add(password = new JPasswordField(""));
117.      add(panel, BorderLayout.CENTER);
118.
119.      // Создание кнопок OK и Cancel.
120.
121.      okButton = new JButton("Ok");
122.      okButton.addActionListener(new ActionListener()
123.      {
124.        public void actionPerformed(ActionEvent event)
125.        {
126.          ok = true;
127.          dialog.setVisible(false);
128.        }
129.      });
130.
131.      JButton cancelButton = new JButton("Cancel");
132.      cancelButton.addActionListener(new ActionListener()
133.      {
134.        public void actionPerformed(ActionEvent event)
135.        {
136.          dialog.setVisible(false);
137.        }
138.      });
139.
140.      // Включение кнопок в нижнюю часть окна.
141.
142.      JPanel buttonPanel = new JPanel();
143.      buttonPanel.add(okButton);
144.      buttonPanel.add(cancelButton);
145.      add(buttonPanel, BorderLayout.SOUTH);
146.   }
147.
148.   /**
149.    * Установка значений по умолчанию.
150.    * @param u Информация о пользователе, принятая по умолчанию
151.    */
```

```
152.   public void setUser(User u)
153.   {
154.      username.setText(u.getName());
155.   }
156.
157.   /**
158.    * Получение введенной информации.
159.    * @return Объект User, состояние которого отражает
160.    *         введенные данные
161.    */
162.   public User getUser()
163.   {
164.      return new User(username.getText(), password.getPassword());
165.   }
166.
167.   /**
168.    * Отображение панели для ввода данных в диалоговом окне.
169.    * @param parent Компонент фрейма-владельца или null
170.    * @param title Заголовок диалогового окна
171.    */
172.   public boolean showDialog(Component parent, String title)
173.   {
174.      ok = false;
175.
176.      // Обнаружение фрейма-владельца.
177.
178.      Frame owner = null;
179.      if (parent instanceof Frame) owner = (Frame) parent;
180.      else owner = (Frame) SwingUtilities.getAncestorOfClass(Frame.class, parent);
181.
182.      // При первом обращении или при изменении фрейма-владельца
183.      // создается новое диалоговое окно.
184.
185.      if (dialog == null || dialog.getOwner() != owner)
186.      {
187.         dialog = new JDialog(owner, true);
188.         dialog.add(this);
189.         dialog.getRootPane().setDefaultButton(okButton);
190.         dialog.pack();
191.      }
192.
193.      // Установка заголовка и отображение диалогового окна.
194.
195.      dialog.setTitle(title);
196.      dialog.setVisible(true);
197.      return ok;
198.   }
199.
200.   private JTextField username;
201.   private JPasswordField password;
202.   private JButton okButton;
203.   private boolean ok;
204.   private JDialog dialog;
205.}
206.
```

```
/**
 * Для пользователя определены имя и пароль. Из соображений
 * безопасности пароль содержится не в объекте String,
 * а в массиве char[].
 */
class User
{
   public User(String aName, char[] aPassword)
   {
      name = aName;
      password = aPassword;
   }

   public String getName()
   {
      return name;
   }

   public char[] getPassword()
   {
      return password;
   }

   public void setName(String aName)
   {
      name = aName;
   }

   public void setPassword(char[] aPassword)
   {
      password = aPassword;
   }

   private String name;
   private char[] password;
}
```

API javax.swing.SwingUtilities 1.2

- `Container getAncestorOfClass(Class c, Component comp)`

  Возвращает наиболее глубоко вложенный родительский контейнер указанного компонента, принадлежащего заданному классу или одному из его подклассов.

API javax.swing.JComponent 1.2

- `JRootPane getRootPane()`

  Определяет корневую панель, содержащую заданный компонент. Если компонент не имеет предка с корневой панелью, возвращает ссылку `null`.

API javax.swing.JRootPane 1.2

- `void setDefaultButton(JButton button)`

Устанавливает на корневой панели кнопку по умолчанию. Чтобы заблокировать кнопку по умолчанию, этот метод вызывается с параметром `null`.

API **javax.swing.JButton 1.2**

- `boolean isDefaultButton()`

  Возвращает значение `true`, если данная кнопка является кнопкой по умолчанию в корневой панели.

Файловые диалоги

Во многих приложениях нужно открывать и сохранять файлы. Хорошее диалоговое окно, позволяющее пользователю свободно перемещаться в рамках файловой системы, создать довольно трудно. Да это и не нужно — ведь вы не собираетесь изобретать колесо! В библиотеке Swing есть класс `JFileChooser`, позволяющий отображать на экране диалоговое окно для выбора файлов, которое удовлетворяет потребностям большинства приложений. Это диалоговое окно всегда является модальным. Обратите внимание: класс `JFileChooser` не расширяет класс `JDialog`. Вместо вызова метода `setVisible(true)` для отображения диалогового окна, предназначенного для открытия файлов, вызывается метод `showOpenDialog()`, а для сохранения файлов — метод `showSaveDialog()`. Кнопка, подтверждающая выбор файла, автоматически помечается как Open или Save. С помощью метода `showDialog()` можно задать свою собственную метку кнопки. На рис. 9.40 показан пример диалогового окна для выбора файлов.

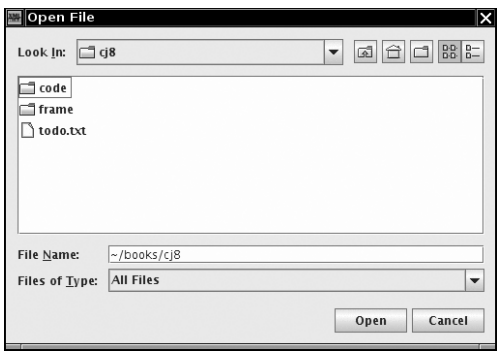


Рис. 9.40. Диалоговое окно для выбора файлов

Чтобы отобразить на экране диалоговое окно для выбора файлов и определить, какой выбор сделал пользователь, выполните описанные ниже действия.

1. Создайте объект класса `JFileChooser`. В отличие от конструктора класса `JDialog`, родительский компонент указывать не нужно. Такое решение позволяет повторно использовать диалоговое окно для выбора файлов при работе с различными фреймами.

   ```
   JFileChooser chooser = new JFileChooser();
   ```

! **Совет.** Повторное использование диалогового окна для выбора файлов — прекрасная идея, поскольку конструктор класса `JFileChooser` иногда работает довольно медленно. Особенно это заметно в среде Windows, когда пользователю доступно много сетевых дисков.

2. Укажите каталог, вызвав метод `setCurrentDirectory()`. Например, чтобы задать текущий каталог, нужно использовать следующее выражение:

   ```
   chooser.setCurrentDirectory(new File("."));
   ```

   При этом необходимо задать объект `File`. Детально класс `File` описывается в главе 12, а пока вам достаточно знать, что в этом классе есть конструктор `File(String filename)`, преобразовывающий строку объекта `File`.

3. Если нужно предложить пользователю имя файла, выбираемого по умолчанию, задайте его с помощью метода `setSelectedFile()`.

   ```
   chooser.setSelectedFile(new File(filename));
   ```

4. Чтобы дать пользователю возможность выбрать одновременно несколько файлов, вызовите метод `setMultiSelectionEnabled()`. Разумеется, делать это совершенно не обязательно, поскольку подобная возможность нужна далеко не всегда.

   ```
   chooser.setMultiSelectionEnabled(true);
   ```

5. Для того чтобы ограничить выбор пользователя файлами, имеющими определенное расширение (например, расширение `.gif`), нужно задать *фильтр файлов*. Этот вопрос мы обсудим немного позже.

6. По умолчанию пользователь может выбирать в диалоговом окне `JFileChooser` только файлы. Если вы хотите предоставить пользователю возможность выбирать целые каталоги, используйте метод `setFileSelectionMode()` и вызывайте его со следующими параметрами:

   `JFileChooser.FILES_ONLY` (по умолчанию)
   `JFileChooser.DIRECTORIES_ONLY` или
   `JFileChooser.FILES_AND_DIRECTORIES`

7. Диалоговое окно выводится на экран с помощью методов `showOpenDialog()` или `showSaveDialog()`. При вызове этих методов нужно указывать родительский компонент.

   ```
   int result = chooser.showOpenDialog(parent);
   int result = chooser.showSaveDialog(parent);
   ```

   Единственное отличие между этими вызовами заключается в метке “кнопки выбора”, т.е. кнопки, на которой пользователь щелкает, выбирая файл. Можно также вызвать метод `showDialog` и явно задать текст на кнопке.

   ```
   int result = chooser.showDialog(parent, "Select");
   ```

   Эти вызовы возвращают управление только тогда, когда пользователь сделал выбор или прекратил диалог, закрыв окно. Возвращаемое значение может быть одной из следующих констант: `JFileChooser.APPROVE_OPTION`, `JFileChooser.CANCEL_OPTION` или `JFileChooser.ERROR_OPTION`.

8. Выбранный файл можно получить с помощью методов `getSelectedFile()` или `getSelectedFiles()`. Эти методы возвращают либо один объект класса `File`, либо массив таких объектов. Если пользователю нужно знать лишь имя файла, вызывайте метод `getPath()`. Например:

   ```
   String filename = chooser.getSelectedFile().getPath();
   ```

Большинство этих действий довольно просты. Основная трудность связана с использованием диалогового окна для выбора файлов, ограничивающего выбор пользователя. Предположим, что пользователь должен выбирать только графические файлы в форма-

те GIF. Следовательно, в диалоговом окне должны отображаться только файлы, имеющие расширение .gif. Кроме того, пользователю нужно подсказать, что это за файлы. Ситуация может оказаться еще сложнее. Графические файлы в формате JPEG могут иметь расширения .jpg или .jpeg. Для преодоления этих трудностей разработчики предложили элегантный механизм: чтобы ограничить спектр отображаемых файлов, нужно создать подкласс абстрактного класса javax.swing.filechooser.FileFilter. Диалоговое окно передает фильтру каждый файл и отображает только файлы, прошедшие его.

На момент написания этой книги был известен только один такой подкласс: фильтр, задаваемый по умолчанию и пропускающий все файлы.

Создать свой собственный фильтр довольно легко. Достаточно реализовать два метода, которые в классе FileFilter объявлены как абстрактные:

```
public boolean accept(File f)
public String getDescription();
```

Первый из приведенных методов проверяет, соответствует ли файл наложенным ограничениям. Второй метод возвращает описание типа файла, который может отображаться в диалоговом окне.



На заметку! В пакете java.io также существует интерфейс FileFilter (совершенно независимый от описанного выше абстрактного класса). В этом интерфейсе объявлен только один метод — boolean accept(File f). Он используется в методе listFiles() класса File для вывода списка файлов, находящихся в каталоге. Совершенно непонятно, почему разработчики библиотеки Swing не стали расширять этот интерфейс, — возможно, библиотека классов языка Java настолько сложна, что даже программисты компании Sun не знают обо всех стандартных классах и интерфейсах.

Если в программу одновременно импортируются пакеты java.io и javax.swing.filterchooser, вам снова придется разрешать конфликт имен Проще всего вместо пакета javax.swing.filechooser.* импортировать класс javax.swing.filechooser.FileFilter.

Имея объект фильтра, вы используете метод setFileFilter класса JFileChooser для инсталляции его в объект этого класса:

```
chooser.setFileFilter(new FileNameExtensionFilter("Image files", "gif", "jpg");
```

Можно установить несколько фильтров последовательностью вызовов:

```
chooser.addChoosableFileFilter(filter1);
chooser.addChoosableFileFilter(filter2);
. . .
```

Пользователь выбирает фильтр из выпадающего списка в нижней части файлового диалога. По умолчанию в нем всегда присутствует фильтр "All files" (Все файлы). Это хорошая идея, особенно в случае, когда пользователю вашей программы нужно выбрать файл с нестандартным расширением. Однако если вы хотите подавить фильтр "All files", вызовите chooser.setAcceptAllFileFilterUsed(false).



Внимание! Если вы повторно используете один и тот же JFileChooser для загрузки и сохранения файлов различных типов, вызовите chooser.resetChoosableFilters(), чтобы очистить старые фильтры перед установкой новых

И, наконец, в диалоговом окне выбора файла каждый файл можно сопроводить специальной пиктограммой и кратким описанием. Для этого нужно предусмотреть экземпляр класса, расширяющего класс `FileView` из пакета `javax.swing.filechooser`. Это довольно сложно и не всегда оправдано. Обычно внешний вид файла программиста не интересует — настраиваемые стили интерфейса сами решают эту задачу. Однако если вы захотите сопроводить файлы определенного типа специальными пиктограммами, можно задать свой собственный стиль отображения файлов. Для этого нужно расширить класс `FileView` и реализовать пять методов.

```
Icon getIcon(File f);
String getName(File f);
String getDescription(File f);
String getTypeDescription(File f);
Boolean isTraversable(File f);
```

Затем следует вызвать метод `setFileView()`, чтобы связать представление с диалоговым окном выбора файлов.

Диалоговое окно вызывает эти методы для каждого файла или каталога, подлежащего выводу на экран. Если для пиктограммы, имени или описания файла метод возвращает ссылку `null`, диалоговое окно использует стиль, заданный по умолчанию. Это правильно, поскольку такой подход позволяет применять особый стиль только к отдельным типам файлов.

Для того чтобы решить, открывать ли каталог, выбранный пользователем, диалоговое окно вызывает метод `isTraversable`. Обратите внимание на то, что этот метод возвращает объект класса `Boolean`, а не значение типа `boolean`! Это выглядит довольно загадочно, но удобно — если у вас нет особых требований к стилю, возвращайте нулевую ссылку `null`. Затем диалоговое окно использует стиль отображения файлов, заданный по умолчанию. Иными словами, метод возвращает объект класса `Boolean`, чтобы дать вам возможность выбрать одно из трех: да (`Boolean.TRUE`), нет (`Boolean.FALSE`) и не важно (`null`).

Программа, рассматриваемая здесь в качестве примера, содержит простой класс, задающий стиль отображения файлов. Этот класс выводит на экран определенную пиктограмму каждый раз, когда файл проходит фильтр. Мы используем этот класс для отображения на экране палитры пиктограмм для всех графических файлов.

```
class FileIconView extends FileView
{
   public FileIconView(FileFilter aFilter, Icon anIcon)
   {
      filter = aFilter;
      icon = anIcon;
   }

   public Icon getIcon(File f)
   {
      if (!f.isDirectory() && filter.accept(f))
         return icon;
      else return null;
   }

   private FileFilter filter;
   private Icon icon;
}
```

Чтобы связать объект класса `FileView` с диалоговым окном, применяется метод `setFileView()`:

```
chooser.setFileView(new FileIconView(filter, new ImageIcon("palette.gif");));
```

Если файл прошел фильтр, на экран будет выведена соответствующая пиктограмма палитры, а для всех остальных файлов будет использоваться стиль, заданный по умолчанию.



Совет. В каталоге `demo/jfc/FileChooserDemo` пакета JDK можно найти более полезный класс `ExampleFileView`. Этот класс позволяет связывать пиктограммы и описания с любыми расширениями файлов.

Отметим также, что диалоговое окно для выбора файлов можно снабдить *компонентом доступа* (accessory component). Например, на рис. 9.41 показан подобный компонент, позволяющий отображать наряду со списком файлов содержимое файла, выбранного в текущий момент.



Рис. 9.41. Диалоговое окно для выбора файлов с возможностью предварительного просмотра

Компонентом доступа может быть любой компонент библиотеки Swing. В данном случае мы расширяем класс `JLabel` и используем в качестве пиктограммы уменьшенную копию графического изображения.

```
class ImagePreviewer extends JLabel
{
   public ImagePreviewer(JFileChooser chooser)
   {
      setPreferredSize(new Dimension(100, 100));
      setBorder(BorderFactory.createEtchedBorder());
   }

   public void loadImage(File f)
   {
      ImafeIcon icon = new ImageIcon(f.getPath());
      if(icon.getIconWidth() > getWidth())
         icon = new ImageIcon(icon.getImage().getScaledInstance
               (getWidth(), -1, Image.SCALE_DEFAULT));
      setIcon(icon);
      repaint();
   }
}
```

Осталась лишь одна проблема. Мы хотим обновлять изображение предварительного просмотра всякий раз, когда пользователь выбирает новый файл. Для этого диалоговое окно использует механизм JavaBeans, извещая обработчиков об изменениях своих свойств. Выбранный файл — это свойство, которое можно отслеживать с помощью обработчика `PropertyChangeListener`. Более детально этот механизм обсуждается во втором томе. Ниже показано, каким образом происходит оповещение обработчиков.

```
chooser.addPropertyChangeListener( new PropertyChangeListener()
   {
      public void propertyChange(PropertyChangeEvent event)
      {
         if (event.getPropertyName(0) ==
            JFileChooser.SELECTED_FILE_CHANGED_PROPERTY)
         {
            File newfile = (File)event.getNewValue()
            // Обновление компонента доступа.
         }
      }
  });
```

В нашем примере этот код находится внутри конструктора класса `ImagePreviewer`.

В листинге 9.17 содержится модифицированная программа `ImageViewer` из главы 2, в которой диалоговое окно для выбора файлов дополнено стилем представления файлов и компонентом для предварительного просмотра их содержимого.

Листинг 9.17. Содержимое файла `FileChooserTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.beans.*;
4.  import java.util.*;
5.  import java.io.*;
6.  import javax.swing.*;
7.  import javax.swing.filechooser.*;
8.  import javax.swing.filechooser.FileFilter;
9.
10. /**
11.  * @version 1.23 2007-06-12
12.  * @author Cay Horstmann
13.  */
14. public class FileChooserTest
15. {
16.    public static void main(String[] args)
17.    {
18.       EventQueue.invokeLater(new Runnable()
19.          {
20.             public void run()
21.             {
22.                ImageViewerFrame frame = new ImageViewerFrame();
23.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24.                frame.setVisible(true);
25.             }
26.          });
27.    }
28. }
29.
```

```
30. /**
31.  * Фрейм, содержащий меню для загрузки графических файлов
32.  * и область для отображения загруженных изображений.
33.  */
34. class ImageViewerFrame extends JFrame
35. {
36.    public ImageViewerFrame()
37.    {
38.       setTitle("FileChooserTest");
39.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
40.
41.       // Установка строки меню.
42.       JMenuBar menuBar = new JMenuBar();
43.       setJMenuBar(menuBar);
44.
45.       JMenu menu = new JMenu("File");
46.       menuBar.add(menu);
47.
48.       JMenuItem openItem = new JMenuItem("Open");
49.       menu.add(openItem);
50.       openItem.addActionListener(new FileOpenListener());
51.
52.       JMenuItem exitItem = new JMenuItem("Exit");
53.       menu.add(exitItem);
54.       exitItem.addActionListener(new ActionListener()
55.       {
56.          public void actionPerformed(ActionEvent event)
57.          {
58.             System.exit(0);
59.          }
60.       });
61.
62.       // Для вывода изображений используется метка.
63.       label = new JLabel();
64.       add(label);
65.
66.       // Создание диалогового окна для выбора файла.
67.       chooser = new JFileChooser();
68.
69.       // Принимаются только графические файлы
70.       // с расширениями .jpg, .jpeg, .gif
71.       /*
72.       final ExtensionFileFilter filter = new ExtensionFileFilter();
73.       filter.addExtension("jpg");
74.       filter.addExtension("jpeg");
75.       filter.addExtension("gif");
76.       filter.setDescription("Image files");
77.       */
78.       FileNameExtensionFilter filter = new FileNameExtensionFilter(
79.          "Image files", "jpg", "jpeg", "gif");
80.       chooser.setFileFilter(filter);
81.
82.       chooser.setAccessory(new ImagePreviewer(chooser));
83.
84.       chooser.setFileView(new FileIconView(filter, new ImageIcon("palette.gif")));
85.    }
```

```
86.
87.    /**
88.     * Слушатель событий, связанных с пунктом меню File->Open.
89.     */
90.    private class FileOpenListener implements ActionListener
91.    {
92.       public void actionPerformed(ActionEvent event)
93.       {
94.          chooser.setCurrentDirectory(new File("."));
95.
96.          // Отображение диалогового окна для выбора файлов.
97.          int result = chooser.showOpenDialog(ImageViewerFrame.this);
98.
99.          // Если файл является допустимым, он используется
100.         // в качестве пиктограммы для метки.
101.         if (result == JFileChooser.APPROVE_OPTION)
102.         {
103.            String name = chooser.getSelectedFile().getPath();
104.            label.setIcon(new ImageIcon(name));
105.         }
106.      }
107.   }
108.
109.   public static final int DEFAULT_WIDTH = 300;
110.   public static final int DEFAULT_HEIGHT = 400;
111.
112.   private JLabel label;
113.   private JFileChooser chooser;
114.}
115.
116./**
117. * Средства отображения, предоставляющие пиктограмму
118. * для каждого файла, распознаваемого фильтром
119. */
120.class FileIconView extends FileView
121.{
122.   /**
123.    * Конструирует FileIconView.
124.    * @param aFilter все файлы, которые пропускает этот
125.    *             фильтр, помечаются специальной пиктограммой
126.    * icon. @param anIcon Пиктограмма для файлов, распознаваемых фильтром
127.    */
128.   public FileIconView(FileFilter aFilter, Icon anIcon)
129.   {
130.      filter = aFilter;
131.      icon = anIcon;
132.   }
133.
134.   public Icon getIcon(File f)
135.   {
136.      if (!f.isDirectory() && filter.accept(f)) return icon;
137.      else return null;
138.   }
139.
```

```
140.   private FileFilter filter;
141.   private Icon icon;
142.}
143.
144. /**
145.  * Компонент доступа, используемый для предварительного
146.  * просмотра изображений.
147.  */
148. class ImagePreviewer extends JLabel
149. {
150.   /**
151.    * Конструирует ImagePreviewer.
152.    * @param chooser Диалоговое окно для выбора файлов.
153.    *              Изменение свойства приводит к изменению
154.    *              предварительного просмотра
155.    */
156.   public ImagePreviewer(JFileChooser chooser)
157.   {
158.     setPreferredSize(new Dimension(100, 100));
159.     setBorder(BorderFactory.createEtchedBorder());
160.
161.     chooser.addPropertyChangeListener(new PropertyChangeListener()
162.     {
163.       public void propertyChange(PropertyChangeEvent event)
164.       {
165.         if (event.getPropertyName() ==
166.             JFileChooser.SELECTED_FILE_CHANGED_PROPERTY)
167.         {
168.           // Пользователь выбрал новый файл.
169.           File f = (File) event.getNewValue();
170.           if (f == null)
171.           {
172.             setIcon(null);
173.             return;
174.           }
175.
176.           // Чтение изображения для использования
177.           // в качестве пиктограммы.
178.           ImageIcon icon = new ImageIcon(f.getPath());
179.
180.           // Если пиктограмма слишком велика,
181.           // выполняется масштабирование.
182.           if (icon.getIconWidth() > getWidth()) icon =
183.             new ImageIcon(icon.getImage()
184.             .getScaledInstance(getWidth(), -1, Image.SCALE_DEFAULT));
185.
186.           setIcon(icon);
187.         }
188.       }
189.     });
190.   }
191. }
```

API javax.swing.JFileChooser 1.2

- `JFileChooser()`

  Создает диалоговое окно для выбора файлов, которое можно использовать с несколькими фреймами.

- `void setCurrentDirectory(File dir)`

  Задает исходный каталог, содержимое которого отображается в диалоговом окне для выбора файлов.

- `void setSelectedFile(File file)`
- `void setSelectedFiles(File[] file)`

  Задает файл, выбранный в окне по умолчанию.

- `void setMultiSelectionEnabled(boolean b)`

  Устанавливает или отменяет режим выбора нескольких файлов.

- `void setFileSelectionMode(int mode)`

  Позволяет пользователю выбирать только файлы (по умолчанию), только каталоги или и файлы, и каталоги одновременно. Параметр `mode` может принимать значения `JFileChooser.FILES_ONLY`, `JFileChooser.DIRECTORIES_ONLY` и `JFileChooser.FILES_AND_DIRECTORIES`.

- `int showOpenDialog(Component parent)`
- `int showSaveDialog(Component parent)`
- `int showDialog(Component parent, String approveButtonText)`

  Выводит на экран диалоговое окно, в котором кнопка выбора имеет надпись Open, Save или произвольное имя, заданное строкой `approveButtonText`. Возвращает значения `APPROVE_OPTION`, `CANCEL_OPTION` (пользователь щелкнул на кнопке Cancel или закрыл диалоговое окно) или `ERROR_OPTION` (в случае возникновения ошибки).

- `File getSelectedFile()`
- `File[] getSelectedFiles()`

  Возвращает файл или файлы, выбранные пользователем (или возвращает значение `null`, если пользователь ничего не выбрал).

- `void setFileFilter(FileFilter filter)`

  Устанавливает маску файлов для диалогового окна. В окне отображаются только те файлы, для которых метод `filter.accept()` возвращает значение `true`. Также добавляет фильтр к списку выбираемых фильтров.

- `void addChoosableFileFilter(FileFilter filter)`

  Добавляет фильтр к списку выбираемых фильтров.

- `void setAcceptAllFileFilterUsed(boolean b)`

  Включает фильтр All files в выпадающий список или удаляет соответствующий пункт из списка.

- `void resetChoosableFileFilters()`

  Очищает список фильтров. Если фильтр All files специально не отменен, он остается в списке.

- `void setFileView(FileView view)`

  Устанавливает средства просмотра, предоставляющие информацию о файлах, отображаемых в окне.

- `void setAccessory(JComponent component)`

  Задает компонент доступа.

API javax.swing.filechooser.FileFilter 1.2

- `boolean accept(File f)`

  Возвращает значение `true`, если указанный файл должен отображаться в диалоговом окне.

- `String getDescription()`

  Возвращает описание указанного фильтра, например, `Image files (*.gif, *.jpeg)`.

API javax.swing.filechooser.FileNameExtensionFilter 6

- `FileNameExtensionFilter(String description, String ... extensions)`

  Конструирует фильтр файлов с заданным описанием, принимающий все каталоги и все файлы, чьи имена заканчиваются точкой, за которой следуют строки заданных расширений.

API javax.swing.fileChooser.FileView 1.2

- `String getName(File f)`

  Возвращает имя файла `f` или значение `null`. Обычно этот метод возвращает значение `f.getName()`.

- `String getDescription(File f)`

  Возвращает описание файла `f`, ориентированное на пользователя, или значение `null`. Например, если файл `f` представляет собой документ в формате HTML, этот метод может вернуть его заголовок.

- `String getTypeDescription(File f)`

  Возвращает описание типа файла `f`, ориентированное на пользователя, или значение `null`. Например, если файл `f` представляет собой документ в формате HTML, этот метод может вернуть строку `Hypertext document`.

- `Icon getIcon(File f)`

  Возвращает пиктограмму, назначенную для файла `f`, или значение `null`. Например, если `f` — файл JPEG, этот метод может вернуть пиктограмму-миниатюру его содержимого.

- `Boolean isTraversable(File f)`

  Если пользователь может открыть указанный каталог, возвращается значение `Boolean.TRUE`. Если каталог представляет собой составной документ, может возвращать значение `false`. Подобно методам `FileView`, этот метод может возвращать значение `null`, отмечая тот факт, что диалоговое окно должно использовать обычный стиль отображения файлов.

Диалоговое окно для выбора цвета

Очевидно, что качественное диалоговое окно для выбора файлов – довольно сложный элемент пользовательского интерфейса, который вряд ли стоит реализовывать самостоятельно. Многие инструментальные средства для создания пользовательского интерфейса дают возможность формировать диалоговые окна для выбора даты или времени, валюты, шрифтов, цвета и т.п. Это дает двойную выгоду. В результате программисты могут использовать готовые высококачественные компоненты, а пользователи получают удобный интерфейс.

На данный момент в пакете Swing, кроме окна для выбора файлов, предусмотрено лишь одно подобное диалоговое окно, позволяющее выбирать цвет. Это окно реализовано с помощью класса `JColorChooser`. Внешний вид различных вкладок окна для выбора цвета показан на рис. 9.42–9.44. Как и класс `JFileChooser`, класс `JColorChooser` является компонентом, а не диалоговым окном, тем не менее, он содержит удобные методы, позволяющие создавать диалоговые окна.

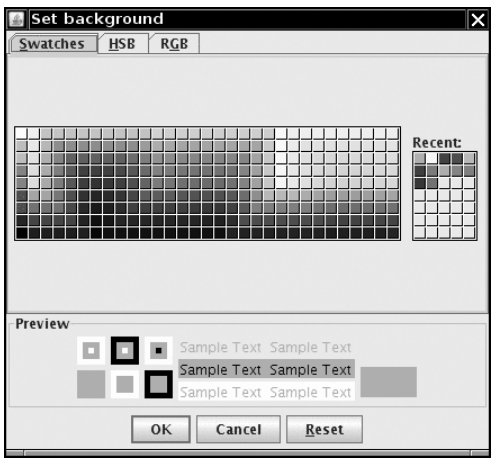


Рис. 9.42. Панель Swatches диалогового окна для выбора цвета

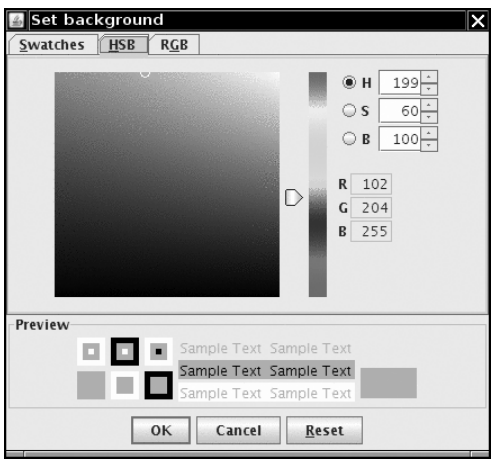


Рис. 9.43. Панель HSB диалогового окна для выбора цвета

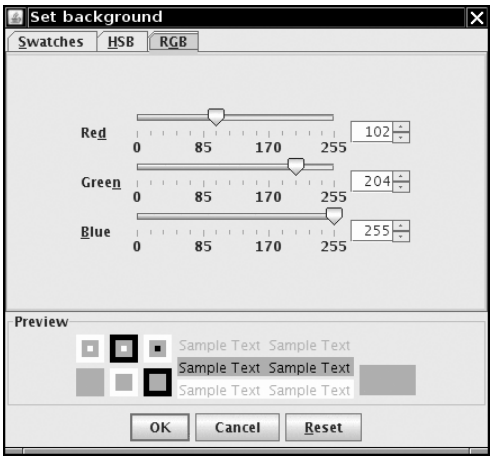


Рис. 9.44. Панель RGB диалогового окна для выбора цвета

Модальное диалоговое окно для выбора цвета создается с помощью следующего выражения:

```
Color selectedColor = JColorChooser.showDialog(parent, title, initialColor);
```

Выбор цвета можно также обеспечить с помощью немодального диалогового окна. Для этого нужно указать следующие элементы.

- Родительский компонент.
- Заголовок диалогового окна.
- Флаг, управляющий выбором модального или немодального диалогового окна.
- Компонент для выбора цвета.
- Обработчик событий, связанных с кнопками OK и Cancel (или параметр `null`, если обработчики не нужны).

Попробуем создать немодальное диалоговое окно для установки цвета фона. Установка цвета производится после щелчка на кнопке OK.

```
chooser = new JColorChooser();
dialog = JColorChooser.createDialog(
  parent,
  "Background Color",
  false               /* немодальное */
  chooser,
  new ActionListener() // слушатель кнопки OK
    {
      public void actionPerformed(ActionEvent event)
      {
        setBackground(chooser.getColor());
      }
    },
  null    /* слушатель кнопки Cancel не задается */);
```

Еще лучше предоставить пользователю возможность сразу увидеть эффект выбора. Для отслеживания нужна модель выбора и слушатель изменений.

```
chooser.getSelectionMode().addChangeListener (new
   ChangeListener()
   {
      public void stateChanged(ChangeEvent event)
      {
         // Действия с методом chooser.getColor();
      }
   });
```

В данном случае кнопки OK и Cancel становятся лишними. Можно просто добавить компонент для выбора цвета в немодальное диалоговое окно.

```
dialog = new JDialog(parent, false /* немодальное */);
dialog.add(chooser);
dialog.pack();
```

Программа, приведенная в листинге 9.18, показывает три типа диалога. Щелкнув на кнопке Modal, пользователь должен будет выбрать цвет и лишь затем сможет продолжить работу. Щелкнув на кнопке Modeless, пользователь создаст немодальное окно, но изменение цвета произойдет только после того, как он щелкнет на кнопке OK. И, наконец, щелкнув на кнопке Immediate, пользователь создаст немодальное диалоговое окно без кнопок; как только он выберет новый цвет, цвет фона панели немедленно изменится.

Листинг 9.18. Содержимое файла `ColorChooserText.java`

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4. import javax.swing.event.*;
5.
6. /**
7.  * @version 1.03 2007-06-12
8.  * @author Cay Horstmann
9.  */
10. public class ColorChooserTest
11. {
12.    public static void main(String[] args)
13.    {
14.       EventQueue.invokeLater(new Runnable()
15.          {
16.             public void run()
17.             {
18.                ColorChooserFrame frame = new ColorChooserFrame();
19.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20.                frame.setVisible(true);
21.             }
22.          });
23.    }
24. }
25.
26. /**
27.  * Фрейм с панелью для выбора цвета.
28.  */
29. class ColorChooserFrame extends JFrame
30. {
```

```
31.   public ColorChooserFrame()
32.   {
33.      setTitle("ColorChooserTest");
34.      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
35.
36.      // Добавление панели для выбора цвета к фрейму.
37.
38.      ColorChooserPanel panel = new ColorChooserPanel();
39.      add(panel);
40.   }
41.
42.   public static final int DEFAULT_WIDTH = 300;
43.   public static final int DEFAULT_HEIGHT = 200;
44. }
45.
46. /**
47.  * Панель с кнопками для выбора трех типов компонентов.
48.  */
49. class ColorChooserPanel extends JPanel
50. {
51.   public ColorChooserPanel()
52.   {
53.      JButton modalButton = new JButton("Modal");
54.      modalButton.addActionListener(new ModalListener());
55.      add(modalButton);
56.
57.      JButton modelessButton = new JButton("Modeless");
58.      modelessButton.addActionListener(new ModelessListener());
59.      add(modelessButton);
60.
61.      JButton immediateButton = new JButton("Immediate");
62.      immediateButton.addActionListener(new ImmediateListener());
63.      add(immediateButton);
64.   }
65.
66.   /**
67.    * Данный слушатель отображает модальное окно для выбора цвета.
68.    */
69.   private class ModalListener implements ActionListener
70.   {
71.      public void actionPerformed(ActionEvent event)
72.      {
73.         Color defaultColor = getBackground();
74.         Color selected = JColorChooser.showDialog(ColorChooserPanel.this,
75.                           "Set background", defaultColor);
76.         if (selected != null) setBackground(selected);
77.      }
78.   }
79.
80.   /**
81.    * Данный слушатель отображает немодальное окно
82.    * для выбора цвета. Цвет панели изменяется после
83.    * щелчка на кнопке OK.
84.    */
85.   private class ModelessListener implements ActionListener
86.   {
```

```
87.      public ModelessListener()
88.      {
89.         chooser = new JColorChooser();
90.         dialog = JColorChooser.createDialog(ColorChooserPanel.this,
91.            "Background Color",
92.            false /* немодальное */,
93.            chooser, new ActionListener() // слушатель кнопки OK
94.         {
95.            public void actionPerformed(ActionEvent event)
96.            {
97.               setBackground(chooser.getColor());
98.            }
99.         }, null /* слушатель кнопки Cancel не задается */);
100.     }
101.
102.     public void actionPerformed(ActionEvent event)
103.     {
104.        chooser.setColor(getBackground());
105.        dialog.setVisible(true);
106.     }
107.
108.     private JDialog dialog;
109.     private JColorChooser chooser;
110.  }
111.
112.  /**
113.   * Данный слушатель отображает немодальное окно
114.   * для выбора цвета. Цвет панели изменяется сразу после выбора.
115.   */
116.  private class ImmediateListener implements ActionListener
117.  {
118.     public ImmediateListener()
119.     {
120.        chooser = new JColorChooser();
121.        chooser.getSelectionModel().addChangeListener(new ChangeListener()
122.        {
123.           public void stateChanged(ChangeEvent event)
124.           {
125.              setBackground(chooser.getColor());
126.           }
127.        });
128.
129.        dialog = new JDialog((Frame) null, false /* немодальное */);
130.        dialog.add(chooser);
131.        dialog.pack();
132.     }
133.
134.     public void actionPerformed(ActionEvent event)
135.     {
136.        chooser.setColor(getBackground());
137.        dialog.setVisible(true);
138.     }
139.
140.     private JDialog dialog;
141.     private JColorChooser chooser;
142.  }
143.}
```

API java.swing.JColorChooser 1.2

- `JColorChooser()`

  Создает компонент для выбора цвета, у которого исходным цветом считается белый.

- `Color getColor()`
- `void setColor(Color c)`

  Возвращает и устанавливает текущий цвет для компонента.

- `static Color showDialog(Component parent, String title, Color initialColor)`

  Отображает на экране модальное диалоговое окно, содержащее компонент для выбора цвета.

Параметры		
	`parent`	Компонент, поверх которого отображается диалоговое окно.
	`title`	Заголовок фрейма.
	`initialColor`	Исходный цвет в компоненте.

- `static JDialog createDialog(Component parent, String title, boolean modal, JColorChooser chooser, ActionListener okListener, ActionListener cancelListener)`

  Создает диалоговое окно, содержащее компонент для выбора цвета.

Параметры		
	`parent`	Компонент, поверх которого отображается диалоговое окно.
	`title`	Заголовок фрейма.
	`modal`	Принимает значение `true`, если выполнение программы должно блокироваться, пока диалоговое окно не будет закрыто.
	`chooser`	Компонент для выбора цвета, который следует добавить в диалоговое окно.
	`OkListener`, `cancelListener`	Обработчики для кнопок OK и Cancel.

На этом обсуждение компонентов пользовательского интерфейса завершено. Материал, изложенный в главах 7–9, поможет вам самостоятельно создавать простые графические интерфейсы с помощью библиотеки Swing. Более подробно соответствующие вопросы обсуждаются во втором томе.

Глава 10

РАЗВЕРТЫВАНИЕ ПРИЛОЖЕНИЙ И АПЛЕТОВ

В ЭТОЙ ГЛАВЕ...

- Файлы JAR
- Технология Java Web Start
- Аплеты
- Хранение параметров настройки

К этому моменту вы уже должны чувствовать себя уверенно при использовании большинства средств языка Java. К тому же вы уже получили достаточное представление основ программирования графики в Java. Теперь, когда вы готовы к созданию приложений для ваших пользователей, вам понадобится знать, как подготавливать их к поставке на пользовательские компьютеры. Традиционный способ развертывания, который шумно рекламировался в первые годы существования Java, заключался в использовании *аплетов*. Аплет — это особого рода Java-программа, которую воспринимающий Java браузер может загружать из Internet и затем выполнять. Они предназначались для того, чтобы освободить пользователей от утомительного процесса инсталляции программного обеспечения, предоставив им доступ к программам с любого компьютера, поддерживающего язык Java, или устройства, имеющего выход во всемирную сеть.

По разным причинам аплеты так и не оправдали ожидания. Недавно компания Sun разработала альтернативный подход к доставке приложений, основанных на использовании сети. Эта технология называется *Java Web Start*. В ней исправлены некоторые из недостатков, характерные для аплетов.

В этой главе показано, как создавать и доставлять аплеты, как запаковывать аплеты и приложения для доставки пользователям. Здесь также будут обсуждаться вопросы хранения информации о конфигурации и доступа к ней.

Файлы JAR

Когда вы упаковываете приложение, то хотите предоставить пользователю единственный файл, а не целую структуру каталогов, полную файлов классов. Файлы Java Archive (JAR) были разработаны специально для этой цели. Файл JAR может содержать наряду с файлами классов и файлы других типов, такие как файлы графических изображений и звуковые файлы. Более того, файлы JAR сжимаются с применением известного формата ZIP.



Совет. В Java SE 5.0 была представлена новая схема сжатия, называемая “pack200”, специально оптимизированная для более эффективного сжатия файлов классов, чем та, что обеспечивается обобщенным алгоритмом ZIP. Sun обещает степень сжатия файлов классов, достигающую 90%. Подробно об этом описывается в `http://java.sun.com/javase/6/docs/technotes/guide/deployment/deployment-guide/pack200.html`.

Для создания JAR-файлов используется инструмент `jar`. (В инсталляции JDK по умолчанию утилита `jar` располагается в каталоге `jdk/bin`.) Новый JAR-файл создается с помощью выражения, подобного следующему:

```
jar cvf имя_JAR-файла файл_1 файл_2 ...
```

Например:

```
jar cvf CalculatorClasses.jar *.java icon.gif
```

Общий формат команды `jar` выглядит следующим образом:

```
jar опции файл_1 файл_2 ...
```

В табл. 10.1 перечислены все опции программы `jar`. Они напоминают опции программы `tar`, хорошо известной пользователям операционной системы Unix.

Вы можете паковать в файлы JAR прикладные приложения, программные компоненты (иногда называемые bean-компонентами — см. второй том) и библиотеки кода. Например, библиотека исполняющей системы JDK содержит очень большой файл `rt.jar`.

Таблица 10.1. Опции программы `jar`

Опция	Описание
`c`	Создает новый или пустой архив и добавляет в него файлы. Если в качестве имени файла указано имя каталога, программа `jar` обрабатывает его рекурсивно.
`C`	Временно изменяет каталог. Например, команда `jar cvf JARFileName.jar -C classes *.class` направляет файлы в подкаталог `classes`.
`e`	Создает точку входа в манифесте (см. раздел “Исполняемые JAR-файлы” далее в главе).
`f`	Задает имя JAR-файла в качестве второго параметра командной строки. Если этот параметр пропущен, то программа `jar` запишет результат в стандартный поток вывода (при создании JAR-файла) или считает его из стандартного потока ввода (при извлечении или просмотре содержимого JAR-файла).
`i`	Создает индексный файл.
`m`	Добавляет в JAR-архив *файл манифеста* (manifest), представляющий собой описание содержимого архива и его происхождения. Файл манифеста создается по умолчанию для каждого архива, но для подробного описания содержимого JAR-файла можно создать свой собственный манифест. Эта тема подробно обсуждается во втором томе.
`M`	Отменяет создание манифеста.
`t`	Отображает на экране содержимое архива.
`u`	Обновляет существующий JAR-файл.
`v`	Выводит подробные данные.
`x`	Извлекает файлы. Если указано несколько имен файлов, извлекаются только они. В противном случае извлекаются все файлы.
`0`	Помещает данные в архив, не упаковывая их в формате ZIP.

Файл манифеста

В дополнение к файлам классов, изображениям и другим ресурсам, каждый JAR-файл содержит также *файл манифеста*, описывающий специальные средства данного архива.

Файл манифеста называется `MANIFEST.MF` и находится в специальном подкаталоге `META-INF`. Минимально допустимый манифест не особенно интересен:

```
Manifest-Version: 1.0
```

Сложные манифесты содержат гораздо более объемные описания, которые группируются в разделы. Первый раздел манифеста называется *главным*. Он относится ко всему JAR-файлу. Остальные описания относятся к отдельным файлам, пакетам или URL. Эти записи должны начинаться словом `Name`. Разделы отделяются друг от друга пустыми строками.

```
Manifest-Version: 1.0
Описание архива.

Name: Woozle.class
Описание файла.

Name: com/mycompany/mypkg/
Описание пакета.
```

Для того чтобы отредактировать этот манифест, поместите в него нужные строки. Затем выполните команду

```
jar cfm имя_JAR_файла имя_файла_манифеста
```

Например, для того чтобы создать новый JAR-файл с манифестом, вызовите утилиту `jar` следующим образом:

```
jar cfm MyArchive.jar manifest.mf com/mycompany/mypkg/*.class
```

Чтобы добавить в манифест существующего JAR-файла новые записи, запишите эти строки в текстовый файл, а затем выполните команду

```
jar ufm MyArchive.jar manfest-additions.mf
```



На заметку! Более подробная информация о форматах JAR-файлов и файлов манифеста доступна по адресу `http://java.sun.com/javase/6/docs/technotes/guides/jar`.

Исполняемые JAR-файлы

Начиная с Java SE 6, вы можете использовать опцию `e` утилиты `jar`, чтобы специфицировать *точку входа* вашей программы — класс, который будет обычно указываться при запуске программы `java`:

```
jar cvfe MyProgram.jar com.mycompany.mypkg.MainAppClass добавляемые файлы
```

Пользователи могут запускать программу следующим образом:

```
java -jar MyProgram.jar
```

В более старых версиях JDK необходимо было специфицировать *главный класс* вашей программы следующим образом:

```
Main-Class: com.mycompany.mypkg.MainAppClass
```

Не добавляйте расширение `.class` к имени главного класса. Затем запустите `jar`:

```
jar cvfm MyProgram.jar mainclass.mf files to add
```



Внимание! Последняя строка в манифесте должна заканчиваться символом перевода строки. В противном случае манифест не будет прочитан правильно. Это распространенная ошибка — когда создается текстовый файл с единственной строкой `Main-Class`, но без ограничителя строки.

В зависимости от конфигурации операционной системы, вы можете запускать приложения двойным щелчком по пиктограмме JAR-файла. Приведем описания поведения для разных операционных систем.

- В среде Windows инсталлятор исполняющей системы Java создает ассоциацию расширения файлов `.jar`, которая запускает такой файл командой `javaw -jar`. (В отличие от команды `java`, `javaw` не открывает окна оболочки.)
- В среде Solaris операционная система распознает “магическое число” JAR-файла и запускает его командой `java -jar`.
- В среде Mac OS операционная система распознает расширение файла `.jar` и выполняет программу Java при двойном щелчке на JAR-файле.

Однако Java-программа, находящаяся в JAR-файле, имеет тот же вид, что и "родное" приложение операционной системы. На Windows вы можете использовать утилиты-оболочки от независимых поставщиков, которые превращают JAR-файлы в исполняемые файлы Windows. Такая оболочка представляет собой программу Windows со знакомым расширением `.exe`, которая запускает виртуальную машину Java (JVM) или же сообщает пользователю, что делать, если JVM не найдено. Существует множество коммерческих продуктов и продуктов с открытым кодом наподобие JSmooth (`http://jsmooth.sourceforge.net`) и Launch4J (`http://launch4j.sourceforge.net`). Генератор инсталляций с открытым кодом IzPack (`http://izpack.org`) также содержит "родную" систему запуска программ. Подробнее об этом читайте в `http://www.javalobby.org/articles/java2exe`.

На платформе Macintosh ситуация немного проще. Утилита прикладных пакетов `MRJAppBuilder` позволяет превратить JAR-файл в первоклассное приложение Mac. Подробнее об этом читайте в `http://java.sun.com/developer/technicalArticles/JavaLP/JavaToMac3`.

Ресурсы

Классы, используемые в аплетах и приложениях, часто ассоциируются с файлами, данных. Примеры таких файлов перечислены ниже.

- Графические и аудиофайлы.
- Текстовые файлы, содержащие строки сообщений и метки кнопок.
- Файлы, содержащие двоичные данные, например, предназначенные для описания компоновки карты.

В языке Java все эти файлы называются *ресурсами*.



На заметку! В операционной системе Windows слово "ресурсы" имеет более узкий смысл. В этой системе к ресурсам также относятся пиктограммы, метки кнопок и т.п., но они должны быть связаны с исполняемыми файлами, а доступ к ним должен обеспечивать стандартный программный интерфейс. В противоположность этому ресурсы в языке Java находятся в отдельных файлах, а не в файлах, содержащих классы, и доступ к ним имеет любой класс.

Рассмотрим класс `AboutPanel`, выводящий на экран сообщение (рис. 10.1).



Рис. 10.1. Отображение ресурса из JAR-файла

Конечно, название книги и год выпуска, указанные в панели, в следующем издании книги изменятся. Для того чтобы легко отслеживать такие изменения, текст нужно поместить в отдельный файл, а не встраивать в программу в виде строки.

Но где же следует разместить файл `about.txt`? Разумеется, было бы удобно поместить его вместе с остальными файлами программы, например, в JAR-файле.

Загрузчик классов знает, где находятся файлы классов, если они содержатся в каталогах, указанных в переменной окружения `CLASSPATH` или в архиве или на Web-сервере. Механизм ресурсов дает программисту такие же удобства при работе с файлами, которые не являются файлами классов. Для этого необходимо выполнить описанные ниже действия.

1. Получить объект `Class`, имеющий ресурс, например, `AboutPanel.class`.
2. Если ресурс представляет собой графический или звуковой файл, вызвать метод `getResource(filename)` и определить URL ресурса. Затем прочитать его непосредственно с помощью методов `getImage()` или `getAudioClip()`.
3. В противном случае применить метод `getResourceAsStream()` к URL и прочитать данные из файла.

Дело в том, что загрузчик класса помнит, где располагается класс, и может обнаружить там же файлы ресурсов.

Например, чтобы создать пиктограмму, хранящуюся в файле `about.gif`, нужно использовать следующий фрагмент кода:

```
URL url = AboutPanel.class.getResource("about.gif");
ImageIcon icon = new ImageIcon(url);
```

Эти выражения означают: “искать файл `about.gif` там же, где находился файл `AboutPanel.class`”.

Чтобы прочитать содержимое файла `about.txt`, можно использовать аналогичные команды:

```
InputStream stream = ResourceTest.class.getResourceAsStream("about.txt");
Scanner in = new Scanner(stream);
```

Ресурсы не обязательно помещать в один каталог с файлом классов. Вместо этого их можно записать в подкаталог. Имя ресурса с учетом подкаталогов выглядит так:

```
data/text/about.txt
```

Это — относительное имя ресурса, поэтому оно определяется относительно пакета, которому принадлежит класс, загружающий этот ресурс. Обратите внимание на то, что в этом имени всегда нужно использовать разделитель /, независимо от того, какие разделители применяются в операционной системе. Например, на платформе Windows загрузчик ресурсов автоматически преобразует символ / в разделитель \.

Имя ресурса, начинающееся с символа /, называется *абсолютным*. По этому имени ресурс обнаруживается точно так же, как и класс. Например, приведенный ниже ресурс находится в каталоге `corejava`.

```
/corejava/title.txt
```

Этот каталог может быть подкаталогом одного из каталогов, указанных в переменной окружения `CLASSPATH`, находиться в JAR-файле или, в случае аплета, на Web-сервере.

Несмотря на то что ресурс загружается автоматически, стандартных методов интерпретации содержимого файла ресурсов не существует. Каждый аплет по-своему интерпретирует данные, находящиеся в подобных файлах.

Кроме того, ресурсы применяются для интернационализации программ. В файлах ресурсов, например, содержатся сообщения и метки на разных языках. Каждый такой файл соответствует отдельному языку. API интернационализации, который будет обсуждаться во втором томе, поддерживает стандартный способ организации и доступа к этим файлам.

В листинге 10.1 демонстрируется исходный код программы, использующей загрузку ресурсов. Скомпилируйте, соберите JAR-файл и запустите его на выполнение:

```
javac ResourceTest.java
jar cvfm ResourceTest.jar ResourceTest.mf *.class *.gif *.txt
java -jar ResourceTest.jar
```

Переместите JAR-файл в другой каталог и запустите его снова, чтобы убедиться, что программа читает все, что ей нужно, из JAR-файла, а не текущего каталога.

Листинг 10.1. Содержимое файла `ResourceTest.java`

```
1. import java.awt.*;
2. import java.io.*;
3. import java.net.*;
4. import java.util.*;
5. import javax.swing.*;
6.
7. /**
8.  * @version 1.4 2007-04-30
9.  * @author Cay Horstmann
10. */
11. public class ResourceTest
12. {
13.       public static void main(String[] args)
14.       {
15.                EventQueue.invokeLater(new Runnable()
16.                {
17.                         public void run()
18.                         {
19.                                  ResourceTestFrame frame =
20.                                  new ResourceTestFrame();
21.                                  frame.setDefaultCloseOperation(
22.                                  JFrame.EXIT_ON_CLOSE);
23.                                  frame.setVisible(true);
24.                         }
25.                });
26.       }
27. }
28.
29. /**
30.  * Фрейм, загружающий ресурсы изображение и текста
31.  */
32. class ResourceTestFrame extends JFrame
33. {
34.       public ResourceTestFrame()
35.       {
36.                setTitle("ResourceTest");
37.                setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
38.                URL aboutURL = getClass().getResource("about.gif");
39.                Image img = Toolkit.getDefaultToolkit().getImage(aboutURL);
```

```
40.                setIconImage(img);
41.
42.                JTextArea textArea = new JTextArea();
43.                InputStream stream = getClass().getResourceAsStream("about.txt");
44.                Scanner in = new Scanner(stream);
45.                while (in.hasNext())
46.                   textArea.append(in.nextLine() + "\n");
47.                add(textArea);
48.       }
49.
50.       public static final int DEFAULT_WIDTH = 300;
51.       public static final int DEFAULT_HEIGHT = 300;
52. }
```

API **java.lang.Class 1.0**

- `URL getResource(String name)`
- `InputStream getResourceAsStream(String name)` 1.1

  Находит ресурс там же, где расположен класс, и возвращает его URL или поток ввода, который можно использовать для загрузки. Если ресурс не найден, методы возвращают ссылку `null`, не генерируя исключений и не сообщая об ошибках ввода-вывода.

Герметизация пакета

В главе 4 уже упоминалось, что Java-пакет можно "герметизировать", чтобы в него больше нельзя было добавить новые классы. Разработчики обычно применяют герметизацию в том случае, если они используют классы, методы и поля, имеющие область видимости, ограниченную пакетом.

Например, если вы герметизируете пакет `com.mycompany.util`, то приведенный ниже оператор нельзя будет применить ни для одного класса вне герметизированного архива:

```
package com.mycompany.util;
```

По умолчанию пакеты в JAR-файле не герметизированы. Изменить данные установки можно, поместив в главный раздел файла манифеста следующую строку:

```
Sealed: true
```

Для каждого пакета можно отдельно указать, герметизировать его или нет, добавив в файл манифеста новый раздел.

```
Name: com/mycompany/util/
Sealed: true

Name: com/mycompany/misc/
Sealed: false
```

Для того чтобы герметизировать пакет, необходимо создать текстовый файл с инструкциями манифеста, а затем выполнить команду

```
jar cvmf MyArchive.jar manifest.mf добавляемые файлы
```

Технология Java Web Start

Java Web Start — это технология, предназначенная для доставки приложений по Internet. Приложения Java Web Start имеет следующие характеристики.

- Обычно они доставляются с помощью браузера. Как только приложение Java Web Start загружено, оно может быть запущено без браузера.
- Они не "живут" в окне браузера. Приложение отображается в своем собственном окне, вне браузера.
- Они не используют реализации Java браузера. Браузер запускает внешнюю программу при получении дескриптора приложения. Для этого используется механизм, подобный тому, который применяется в Adobe Acrobat или RealAudio.
- Приложениям, снабженным цифровой подписью, могут быть предоставлены права свободного доступа к локальной машине. Приложения, не имеющие подписи, запускаются в "песочнице" (sandbox), что запрещает потенциально опасные операции.

Для того чтобы подготовить приложение к доставке с помощью технологии Java Web Start, необходимо запаковать его в один или несколько JAR-файлов. Затем нужно подготовить файл описания в формате JNLP (Java Network Launch Protocol — сетевой протокол запуска Java-приложений). Теперь разместите файлы на Web-сервере. После этого необходимо убедиться, что на Web-сервере MIME-тип `application/x-java-jnlp-file` связан с расширением `.jnlp`. (Браузеры используют MIME-тип для того, чтобы определить, какую программу следует запустить для поддержки документа.) Детали можно найти в документации, сопровождающей ваш Web-сервер.



Совет. Чтобы поэкспериментировать с технологией Java Web Start, инсталлируйте программу Tomcat. Найти ее можно по адресу `jakarta.apache.org/tomcat`. Tomcat — это контейнер для сервлетов и JSP-документов, но он также поддерживает и Web-страницы. Текущая версия этого контейнера настроена на обработку JNLP-файлов.

Попробуйте доставить с помощью технологии Java Web Start приложение, реализующее калькулятор, которое было описано в главе 9. Для этого выполните следующие действия.

1. Скомпилируйте программу `Calculator.java`.
2. Подготовьте файл манифеста `Calculator.mf`, содержащий следующую строку.

   ```
   Main-Class: Calculator
   ```

3. Создайте JAR-файл с помощью команды

   ```
   jar cvfm Calculator.jar Calculator.mf *.class
   ```

4. Подготовьте файл `Calculator.jnlp`, содержимое которого приведено ниже.

   ```
   <?xml version="1.0" encoding="utf-8"?>
   <jnlp spec="1.0+"codebase="http://localhost:8080/calculator/"
       href="Calculator.jnlp">
     <information>
       <title>Calculator Demo Application</title>
       <vendor>Cay S. Horstmann</vendor>
       <description>A Calculator</description>
       <offline-allowed/>
     </information>
   ```

```
  <resources>
    <j2se version="5.0+"/>
    <jar href="Calculator.jar"/>
  </resources>
  <application-desc/>
</jnlp>
```

(Отметим, что номером версии должен быть `1.5.0`, а не `5.0`. Начиная с Java SE 6, вы можете использовать `java` вместо `j2se` в качестве имени дескриптора.)

Формат файла очевиден. Его спецификация содержится на Web-странице `http://java.sun.com/products/javawebstart/docs/developersguide.html`.

5. Если вы используете контейнер Tomcat, создайте каталог *`tomcat`*`/webapps/calculator`, где *`tomcat`* — каталог, в котором инсталлирован продукт Tomcat. Создайте подкаталог `tomcat/webapps/calculator/WEB-INF` и поместите в него минимальную версию файла `web.xml`.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE web-app PUBLIC
   "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
   "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
</web-app>
```

6. Разместите JAR- и JNLP-файл на Web-сервере так, чтобы URL соответствовал записи `codebase` в JNLP-файле. Если вы используете сервер Tomcat, это будет каталог *`tomcat`*`/webapps/calculator`.
7. Убедитесь, что браузер правильно настроен для использования средств Java Web Start, проверив, что MIME-тип `application/x-java-jnlp-file` связан с приложением `javaw`. Если на вашем компьютере инсталлирован пакет JDK, настройка выполняется автоматически.
8. Запустите Tomcat.
9. Обратитесь с помощью браузера к файлу `http://localhost:8080/calculator/Calculator.jnlp`.
10. Вы должны увидеть окно, подобное тому, которое показано на рис. 10.2.
11. Затем на экране должен появиться калькулятор, обрамление которого свидетельствует о том, что он является приложением Java Web Start (рис. 10.3).

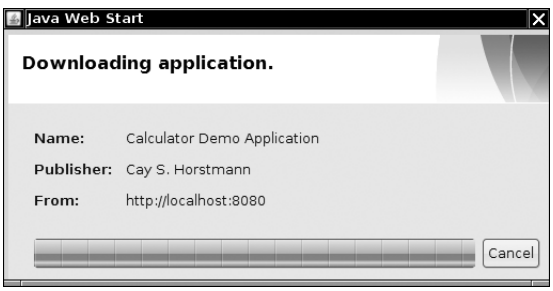


Рис. 10.2. Запуск Java Web Start

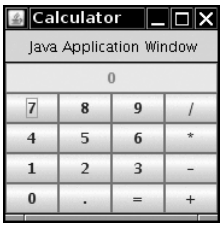


Рис. 10.3. Приложение Calculator, доставленное с помощью Java Web Start

12. Когда вы снова обратитесь к файлу JNLP, приложение будет извлечено из кэша. Просмотреть содержимое кэша можно с помощью панели управления Java Plug-in (рис. 10.4). Начиная с версии Java SE 5.0, панель управления используется как для аплетов, так и для приложений Java Web Start. Если вы работаете в системе Windows, панель управления Java Plug-in можно найти на системной панели управления. В системе Linux следует запустить `jdk/jre/bin/ControlPanel`.

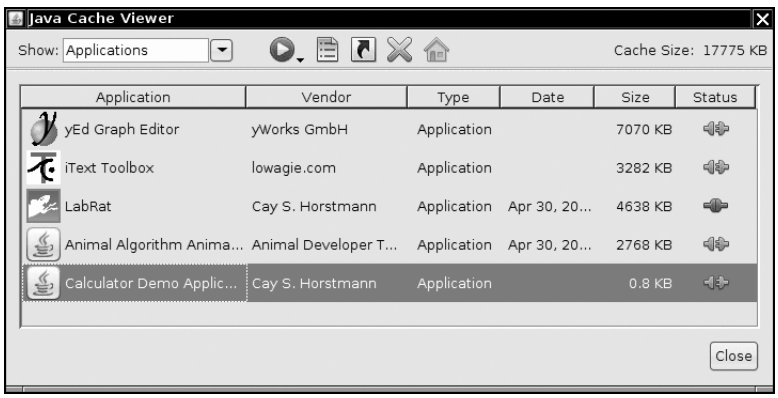


Рис. 10.4. Просмотр кэша



Совет. Если вы не хотите запускать Web-сервер до того, как протестируете конфигурацию JNLP, то можете временно переопределить URL `codebase` в файле запуска:

```
javaws -codebase file://каталог_программы JNLP-файл
```

Например, в UNIX вы можете просто дать следующую команду из каталога, содержащего файл JNLP:

```
javaws -codebase file://`pwd` WebStartCalculator.jnlp
```

Конечно, не нужно заставлять своих пользователей запускать средство просмотра кэша всякий раз, когда требуется запустить ваше приложение. Вы можете поручить инсталлятору установить ярлыки рабочего стола и меню. Добавьте следующие строки в файл JNLP:

```
<shortcut>
      <desktop/>
      <menu submenu="Accessories"/>
</shortcut>
```

Когда пользователь впервые загружает приложение, отображается “предупреждение интеграции рабочего стола” (рис. 10.5).

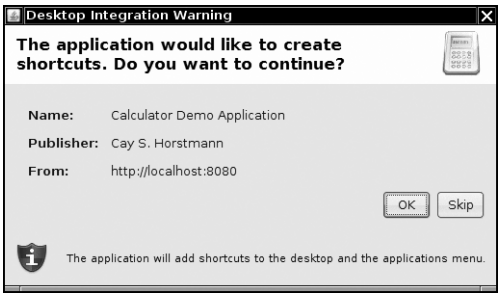


Рис. 10.5. Предупреждение интеграции рабочего стола

Вы должны также применить пиктограмму для ярлыка меню и экрана запуска. Sun рекомендует использовать пиктограммы 32×32 и 64×64. Поместите файлы пиктограмм на Web-сервер вместе с файлами JNLP и JAR. Добавьте следующие строки в раздел `information` файла JNLP:

```
<icon href="calc_icon32.png" width="32" height="32" />
<icon href="calc_icon64.png" width="64" height="64" />
```

Отметим, что эти пиктограммы не имеют отношения к пиктограмме приложения. Если вы хотите, чтобы приложение имело пиктограмму, то вам придется добавить отдельную пиктограмму в JAR-файл и вызвать метод `setIconImage()` в классе фрейма (для примера см. листинг 10.1).

“Песочница”

Всякий раз при загрузке кода с удаленного сайта и локальном его выполнении на первый план выходит вопрос безопасности. Щелчок на единственной ссылке может запустить приложение Java Web Start. Посещение Web-сайта автоматически запускает все аплеты на странице. При щелчке по ссылке или посещении Web-страницы на пользовательский компьютер может быть инсталлирован произвольный код, злоумышленники получают возможность хищения конфиденциальной информации, доступа к финансовой информации либо применения пользовательских машин для рассылки спама.

Чтобы исключить злоупотребление технологией, пришлось обзавестись моделью безопасности, которую мы детально рассмотрим во втором томе. Доступ ко всем системным ресурсам контролирует *диспетчер безопасности*. По умолчанию он разрешает только безвредные операции. Чтобы позволить дополнительные операции, код должен быть снабжен цифровой подписью, и пользователь должен утвердить подписанный сертификат.

Что же может делать удаленный код на всех платформах? Всегда разрешается показывать графические изображения, воспроизводить звук, принимать от пользователя нажатия клавиш и щелчки мыши, а также посылать пользовательский ввод обратно на хост, с которого был загружен код. Этой функциональности достаточно для отображения фактов и фигур или для принятия ввода пользователя для дальнейшего сохранения. Ограниченная среда выполнения часто называется “песочницей” (sandbox). Код, работающий в пределах “песочницы”, не может никак повлиять на пользовательскую систему или “шпионить” за ней.

В частности, программы в "песочнице" имеют перечисленные ниже ограничения.

- Они *никогда* не могут запускать никаких локальных исполняемых программ.
- Они не могут читать или писать в локальной файловой системе.
- Они не могут извлекать никакой информации о локальном компьютере, за исключением установленной версии Java и некоторых безвредных деталей, касающихся операционной системы. В частности, код в "песочнице" не может получить имени пользователя, адреса его электронной почты и т.п.
- Удаленно загруженные программы не могут взаимодействовать ни с каким хостом, за исключением сервера, с которого они были загружены; этот сервер называется *исходящим хостом* (originating host). Это правило часто называют "удаленный код может звонить только домой". Данное правило защищает пользователей от кода, который может попытаться шпионить за ресурсами Internet. (Начиная с Java SE 6, приложение Java Web Start может устанавливать и другие сетевые подключения, но только с согласия пользователя.)
- Все всплывающие окна выдают предупреждающее сообщение. Это сообщение — средство безопасности, исключающее вероятность того, что пользователь спутает его с окном локального приложения. Такой прием исключает ситуацию, когда нежелательный пользователь посещает Web-страницу, взламывает работающий удаленный код и затем вводит пароль и номер кредитной карты, которые могут быть отправлены обратно на Web-сервер. В ранних версиях JDK это сообщение было весьма угрожающим: "Untrusted Java Applet Window" (Окно недоверяемого Java-аплета). Каждая последующая версия немного смягчала формулировку: сначала "Unauthenticated Java Applet Window" (Окно неаутентифицированного Java-аплета), затем "Warning: Java Applet Window" (Предупреждение: окно Java-аплета). А теперь — просто "Java Applet Window" (Окно Java-аплета) или "Java Application Window" (Окно Java-приложения).

Подписанный код

Ограничения "песочницы" во многих ситуациях оказываются чересчур строгими. Например, к корпоративной сети можно легко вообразить себе приложение Java Web Start или аплет, которому нужен доступ к локальным файлам. Существует возможность весьма детального контроля прав, выдаваемых конкретному приложению; мы будем говорить об этом во втором томе. Конечно, приложение может просто запросить все права настольного приложения, и довольно небольшая часть приложений Java Web Start так и делают. Это достигается добавлением следующих дескрипторов в файл JNLP:

```
<security>
  <all-permissions/>
</security>
```

Чтобы запускаться вне "песочницы", JAR-файлы приложения Java Web Start должны быть снабжены *цифровой подписью*. Подписанный JAR-файл удостоверяется сертификатом, подтверждающим идентичность того, кто его подписал. Криптографическая техника гарантирует, что такой сертификат не может быть подделан, и что любые попытки проникновения в подписанный файл будут обнаружены.

Например, предположим, что вы получили приложение, созданное и подписанное yWorks GmbH, с применением сертификата, изданного Thawte (рис. 10.6). Когда вы получаете приложение, вы можете быть уверены в следующем.

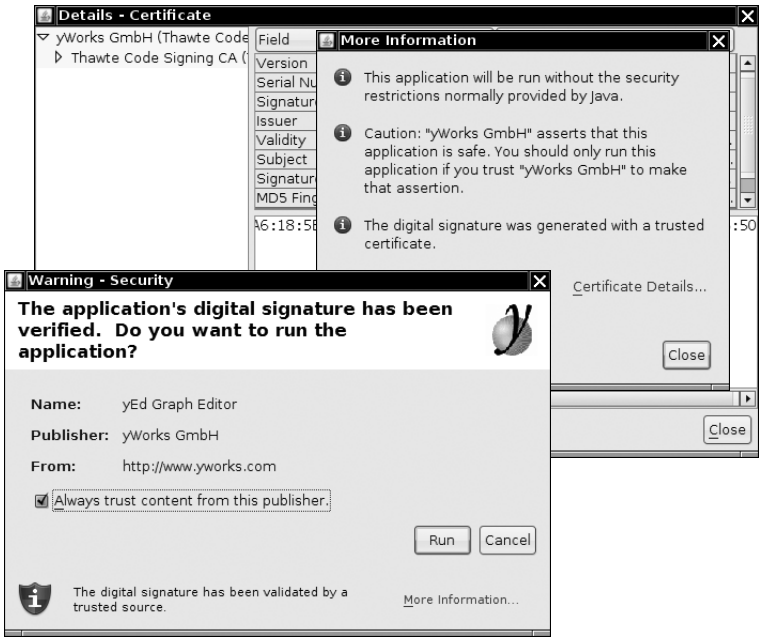


Рис. 10.6. Сертификат безопасности

1. Код в точности таков, каким он был на момент подписания; никакой посторонний не подделал его.
2. Сигнатура действительно принадлежит yWorks.
3. Сертификат действительно издан Thawte (Java Web Start знает, как проверять сертификаты Thawte, а также некоторых других поставщиков).

К сожалению, это все, что вы знаете. Вы не уверены, что код в действительности безопасен. Фактически, если вы щелкаете на ссылке More Information (Дополнительная информация), то узнаете, что приложение будет запущено без ограничений безопасности, обычно предоставляемых Java. Нужно ли инсталлировать и запускать такое приложение? Это в действительности зависит от вашего доверия yWorks GmbH.

Получение сертификата от одного из поддерживаемых поставщиков стоит несколько сотен долларов в год. Многие разработчики просто генерируют свой собственный сертификат и используют его для подписи кода. Конечно, Java Web Start не имеет возможности проверить подлинности таких сертификатов. Когда вы получаете такое приложение, вы знаете следующее.

1. Код в точности таков, каким был на момент подписи; никто не подделал его.
2. Кто-то подписал этот код, но Java Web Start не может проверить, кто это был.

Это все достаточно бесполезно; любой может подделать код, а затем подписать его, прикинувшись его автором. Тем не менее, Java Web Start с удовольствием предоставляет сертификат для вашего подтверждения (рис. 10.7). Теоретически возможно верифицировать сертификат другим способом, но немногие пользователи обладают достаточной сообразительностью, чтобы сделать это.

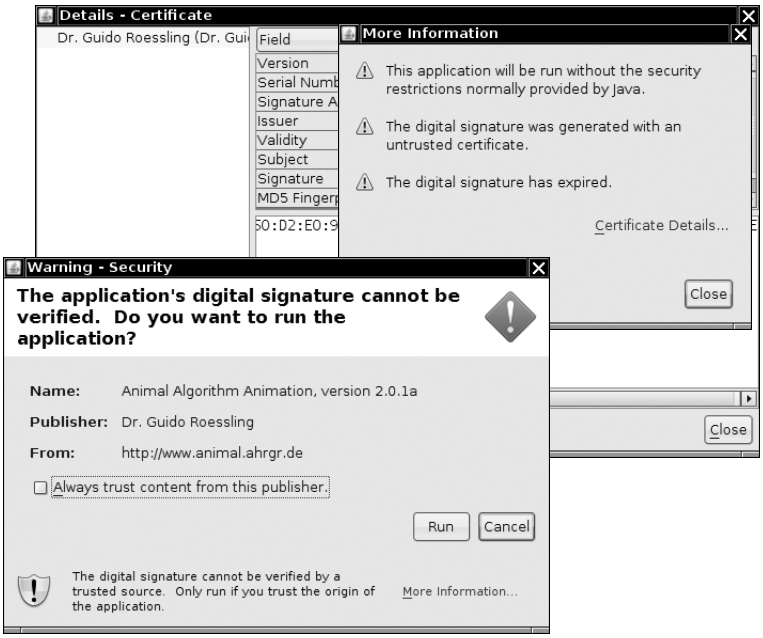


Рис. 10.7. Небезопасный сертификат

Конечно, многие люди каждый день загружают и запускают приложения из Internet. Если вы обнаружили, что ваши пользователи доверяют вашим приложениям и вашей Web-инфраструктуре, то вперед! Используйте ваш собственный сертификат. (Подробности доступны по адресу `http://java.sun.com/javase/6/docs/technotes/guides/javaws/developersguide/development.html`). Если же нет — обеспечьте вашим пользователям безопасность, оставаясь в границах "песочницы". С помощью JNLP API (о которой речь пойдет в следующем разделе), вы можете разрешать вашей программе избирательно обращаться к ресурсам, требуя согласия пользователей.

JNLP API

JNLP API позволяет неподписанному приложению запускаться в "песочнице" и в то же время получать безопасный доступ к локальным ресурсам. Например, в нем предусмотрены службы для загрузки и сохранения файлов. Приложение не видит файловой системы и не может специфицировать имена файлов. Вместо этого отображается файловый диалог, и пользователь программы выбирает файл. Перед появлением диалогового окна для выбора файлов пользователь получает предупреждение и должен выразить согласие на продолжение работы (рис. 10.8). Более того, на самом деле API не предоставляет программе доступа к объекту `File`. В частности, приложение не может самостоятельно обнаружить файл. Поэтому программистам предоставляются инструменты для реализации действий открытия и сохранения файла, но системная информация максимально скрыта от сомнительных приложений.

API предоставляет разработчику описанные ниже возможности.

- Загрузка и сохранение файлов.
- Доступ к буферу обмена.
- Печать.

- Отображение документа в стандартном браузере.
- Хранение и извлечение конфигурационных данных.
- Средства, позволяющие убедиться в том, что выполняется только один экземпляр приложения (данная возможность была реализована в Java SE 5.0).

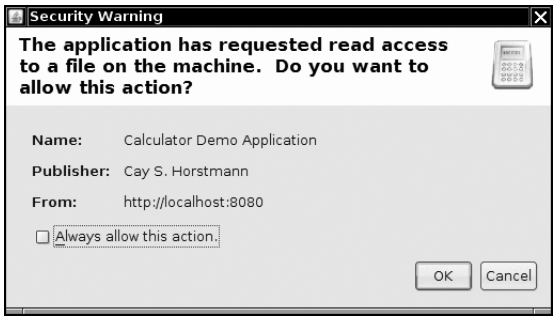


Рис. 10.8. Окно предупреждения о безопасности Java Web Start

Чтобы воспользоваться этими возможностями, необходимо применить класс `ServiceManager`, примерно так:

```
FileSaveService service = (FileSaveService)
   ServiceManager.lookup("javax.jnlp.FileSaveService");
```

Если соответствующая функция оказалась недоступной, генерируется исключение `UnavailableServiceException`.



На заметку! Для компиляции программы, использующей интерфейс JNLP API, необходимо указать в составе пути для поиска классов файл `javaws.jar`. Этот файл находится в подкаталоге `jre/lib` каталога JDK.

Теперь перейдем к обсуждению наиболее полезных возможностей JNLP. Чтобы сохранить файл, следует указать в диалоговом окне предположения о первоначальном пути и расширения файлов, а также сохраняемые данные и предполагаемое имя файла. Рассмотрим пример:

```
service.saveFileDialog(".", new String[] { "txt" }, data, "calc.txt");
```

Данные должны быть доставлены в поток `InputStream`. Иногда эта задача оказывается довольно сложной. Программа, приведенная в листинге 10.2, использует описанную ниже стратегию.

1. Создается экземпляр класса `ByteArrayOutputStream`, предназначенный для хранения байтов, подлежащих записи на диск.
2. Создается экземпляр класса `PrintStream`, посылающий эти данные в поток `ByteArrayOutputStream`.
3. Информация, подлежащая сохранению в потоке печати, выводится в `PrintStream`.
4. Создается экземпляр класса `ByteArrayInputStream`, считывающий сохраненные байты.
5. Поток передается методу `saveFileDialog()`.

Более подробно потоки описываются во втором томе. Пока же мы можем не обращать внимания на детали, указанные в демонстрационной программе.

Чтобы считать данные из файла, используется класс `FileOpenService`. Его метод `openFileDialog()` получает первоначальный путь и расширения файлов и возвращает объект класса `FileContents`. Затем можно вызвать метод `getInputStream()` и считать данные из файла. Если пользователь не выбрал файл, метод `openFileDialog()` возвращает значение `null`.

```
FileOpenService service = (FileOpenService)
   ServiceManager.lookup("javax.jnlp.FileOpenService");
FileContents contents =  service.openFileDialog(".",
   new String[] { "txt" });
if (contents != null)
{
   InputStream in = contents.getInputStream();
   . . .
}
```

Обратите внимание, что ваше приложение не знает имени и места расположения файла. В противоположность этому, если вы хотите открыть определенный файл, то используете `ExtendedService`:

```
ExtendedService service = (ExtendedService)
  ServiceManager.lookup("javax.jnlp.ExtendedService");
FileContents contents = service.openFile(new File("c:\\autoexec.bat"));
if (contents != null)
{
   OutputStream out = contents.getOutputStream();
   . . .
}
```

Пользователь вашей программы должен разрешить доступ к файлу (рис. 10.9).

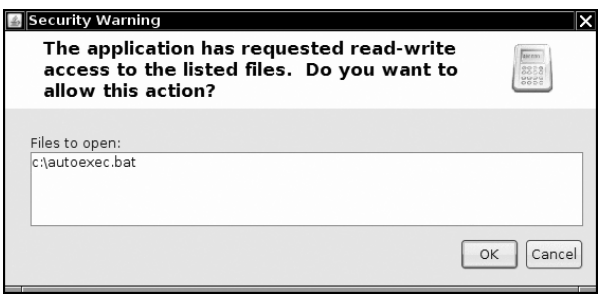


Рис. 10.9. Предупреждение о доступе к файлу

Для того чтобы отобразить документ в стандартном браузере (аналогично методу `showDocument()` для аплетов), применяется интерфейс `BasicService`. Учтите, что в некоторых системах стандартный браузер не установлен.

```
BasicService service = (BasicService)
   ServiceManager.lookup("javax.jnlp.BasicService");
if (service.isWebBrowserSupported())
   service.showDocument(url);
else ...
```

Существует устаревший метод `PersistentService()`, позволяющий приложению сохранять небольшие объемы информации о настройках и извлекать ее, если приложение запускается вновь. Эта возможность напоминает механизм cookie, применяемый при работе с протоколом HTTP. В качестве ключей используются URL. Эти URL не обязательно должны ссылаться на реальные ресурсы. Они лишь представляют собой удобную иерархическую схему именования файлов. По каждому URL-ключу приложение может записать произвольные двоичные данные. (Объем записи может быть ограничен размером блока.)

Для того чтобы изолировать приложения друг от друга, каждое приложение должно использовать URL, начинающиеся с имени сервера, содержащего его код (указанный в JNLP-файле). Например, если приложение загружено с Web-страницы `http://myserver.com/apps`, то оно может использовать лишь ключи, имеющие вид `http://myserver.com/apps/subkey1/subkey2/...` Попытка получить доступ к другим ключам будет заведомо неудачной.

Чтобы определить адрес сервера, на котором хранится код, приложение может вызвать метод `getCodebase()` класса `BasicService`.

Новый ключ создается с помощью метода `create()` класса `PersistenceService`:

```
URL url = new URL(codebase, "mykey");
servise.create(url, maxSize);
```

Для доступа к информации, связанной с конкретным ключом, вызывается метод `get()`. Этот метод возвращает экземпляр класса `FileContents`, с помощью которого можно читать и записывать данные, соответствующие указанному ключу.

```
FileContents contents = service.get(url);
InputStream in = contents.getInputStream();
OutputStream out = contents.getOutputStream(true);
   // true = перезапись
```

К сожалению, определить, существует указанный ключ или его нужно создать заново, довольно сложно. Мы можем лишь надеяться, что ключ существует, и вызывать метод `get`. Если при этом генерируется исключение `FileNotFoundException`, необходимо создать новый ключ.



На заметку! Начиная с Java SE 5.0, и приложения Java Web Start, и аплеты могут выводить данные на печать, используя обычные средства API. При этом отображается диалоговое окно, запрашивающее у пользователя согласие на доступ к принтеру. Подробно средства API для работы с принтером будут рассматриваться во втором томе.

Программа, код которой приведен в листинге 10.2, представляет собой модификацию приложения, реализующего калькулятор. Этот калькулятор имеет виртуальную бумажную ленту, на которую записываются результаты всех вычислений. Список всех предыдущих вычислений можно сохранять и считывать из файла. Для демонстрации постоянного хранения данных приложение позволяет задавать заголовок фрейма. Если программу запустить вновь, она извлечет этот заголовок из постоянного хранилища (рис. 10.10).

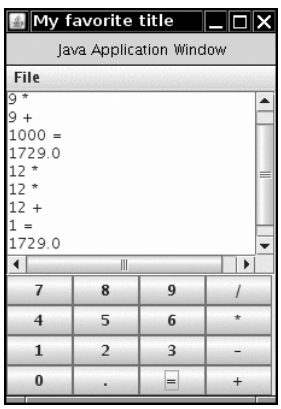


Рис. 10.10. Приложение `WebStartCalculator`

Листинг 10.2. Содержимое файла `WebStartCalculator.java`

```
1.  import java.awt.EventQueue;
2.  import java.awt.event.*;
3.  import java.io.*;
4.  import java.net.*;
5.  import javax.swing.*;
6.  import javax.jnlp.*;
7.
8.  /**
9.   * Калькулятор с поддержкой хронологии. Может
10.  * быть доставлен средствами Java Web Start.
11.  * @version 1.02 2007-06-12
12.  * @author Cay Horstmann
13.  */
14. public class WebStartCalculator
15. {
16.    public static void main(String[] args)
17.    {
18.       EventQueue.invokeLater(new Runnable()
19.          {
20.             public void run()
21.             {
22.                CalculatorFrame frame = new CalculatorFrame();
23.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24.                frame.setVisible(true);
25.             }
26.          });
27.    }
28. }
29.
30. /**
31.  * Фрейм, содержащий панель калькулятора и меню
32.  * для чтения и записи предыдущих результатов.
33.  */
34. class CalculatorFrame extends JFrame
35. {
36.    public CalculatorFrame()
37.    {
38.       setTitle();
39.       panel = new CalculatorPanel();
40.       add(panel);
41.
42.       JMenu fileMenu = new JMenu("File");
43.
44.       JMenuItem openItem = fileMenu.add("Open");
45.       openItem.addActionListener(new ActionListener()
46.       {
47.          public void actionPerformed(ActionEvent event)
48.          {
49.             open();
50.          }
51.       });
52.
53.       JMenuItem saveItem = fileMenu.add("Save");
```

```
54.       saveItem.addActionListener(new ActionListener()
55.       {
56.          public void actionPerformed(ActionEvent event)
57.          {
58.             save();
59.          }
60.       });
61.       JMenuBar menuBar = new JMenuBar();
62.       menuBar.add(fileMenu);
63.       setJMenuBar(menuBar);
64.
65.       pack();
66.    }
67.
68.    /**
69.     * Извлечение заголовка из постоянного хранилища.
70.     * Если ранее заголовок не был задан, он запрашивается у пользователя.
71.     */
72.    public void setTitle()
73.    {
74.       try
75.       {
76.          String title = null;
77.
78.          BasicService basic = (BasicService)
79.             ServiceManager.lookup("javax.jnlp.BasicService");
80.          URL codeBase = basic.getCodeBase();
81.
82.          PersistenceService service = (PersistenceService) ServiceManager
83.             .lookup("javax.jnlp.PersistenceService");
84.          URL key = new URL(codeBase, "title");
85.
86.          try
87.          {
88.             FileContents contents = service.get(key);
89.             InputStream in = contents.getInputStream();
90.             BufferedReader reader = new BufferedReader(new InputStreamReader(in));
91.             title = reader.readLine();
92.          }
93.          catch (FileNotFoundException e)
94.          {
95.             title = JOptionPane.showInputDialog("Please supply a frame title:");
96.             if (title == null) return;
97.
98.             service.create(key, 100);
99.             FileContents contents = service.get(key);
100.            OutputStream out = contents.getOutputStream(true);
101.            PrintStream printOut = new PrintStream(out);
102.            printOut.print(title);
103.         }
104.         setTitle(title);
105.      }
106.      catch (UnavailableServiceException e)
107.      {
108.         JOptionPane.showMessageDialog(this, e);
109.      }
```

```
110.     catch (MalformedURLException e)
111.     {
112.        JOptionPane.showMessageDialog(this, e);
113.     }
114.     catch (IOException e)
115.     {
116.        JOptionPane.showMessageDialog(this, e);
117.     }
118.  }
119.
120.  /**
121.   * Открытие файла с хронологией и обновление
122.   * отображаемых данных.
123.   */
124.  public void open()
125.  {
126.     try
127.     {
128.        FileOpenService service = (FileOpenService) ServiceManager
129.           .lookup("javax.jnlp.FileOpenService");
130.        FileContents contents = service.openFileDialog(".", new String[] { "txt" });
131.
132.        JOptionPane.showMessageDialog(this, contents.getName());
133.        if (contents != null)
134.        {
135.           InputStream in = contents.getInputStream();
136.           BufferedReader reader = new BufferedReader(new InputStreamReader(in));
137.           String line;
138.           while ((line = reader.readLine()) != null)
139.           {
140.              panel.append(line);
141.              panel.append("\n");
142.           }
143.        }
144.     }
145.     catch (UnavailableServiceException e)
146.     {
147.        JOptionPane.showMessageDialog(this, e);
148.     }
149.     catch (IOException e)
150.     {
151.        JOptionPane.showMessageDialog(this, e);
152.     }
153.  }
154.
155.  /**
156.   * Сохранение хронологии в файле.
157.   */
158.  public void save()
159.  {
160.     try
161.     {
162.        ByteArrayOutputStream out = new ByteArrayOutputStream();
163.        PrintStream printOut = new PrintStream(out);
164.        printOut.print(panel.getText());
165.        InputStream data = new ByteArrayInputStream(out.toByteArray());
```

```
166.        FileSaveService service = (FileSaveService) ServiceManager
167.           .lookup("javax.jnlp.FileSaveService");
168.        service.saveFileDialog(".", new String[] { "txt" }, data, "calc.txt");
169.     }
170.     catch (UnavailableServiceException e)
171.     {
172.        JOptionPane.showMessageDialog(this, e);
173.     }
174.     catch (IOException e)
175.     {
176.        JOptionPane.showMessageDialog(this, e);
177.     }
178.  }
179.
180.  private CalculatorPanel panel;
181.}
```

API javax.jnlp.ServiceManager

- `static String[] getServiceNames()`

  Возвращает имена объектов, обеспечивающих все доступные функциональные возможности.

- `static Object lookup(String name)`

  Возвращает объект, обеспечивающий одну из функциональных возможностей и имеющий заданное имя.

API javax.jnlp.BasicService

- `URL getCodeBase()`

  Возвращает имя каталога, содержащего код данного приложения.

- `boolean isWebBrowserSupported()`

  Возвращает значение `true`, если Java Web Start может запустить Web-браузер.

- `boolean showDocument(URL url)`

  Пытается отобразить заданный ресурс с помощью браузера. Если запрос выполнен успешно, возвращает значение `true`.

API javax.jnlp.FileContents

- `inputStream getInputStream()`

  Возвращает входной поток, предназначенный для чтения содержимого файла.

- `OutputStream getOutputStream(boolean overwrite)`

  Возвращает выходной поток, предназначенный для записи в файл; если значение параметра `overwrite` равно `true`, содержимое файла перезаписывается.

- `String getName()`

  Возвращает имя файла (но не полный путь).

- `boolean canRead()`
- `boolean canWrite()`

  Эти методы возвращают значение `true`, если соответствующий файл можно прочитать или записать.

API javax.jnlp.FileOpenService

- `FileContents openFileDialog(String pathHint, String[] extensions)`
- `FileContents openMultiDialog(String pathHint, String[] extensions)`

  Эти методы выводят предупреждающее сообщение и диалоговое окно для выбора файлов. Они возвращают дескрипторы файлов, выбранных пользователем, или значение `null`, если пользователь не выбрал ни одного файла.

API javax.jnlp.FileSaveService

- `FileContents saveFileDialog(String pathHint, String[] extensions, String nameHint)`
- `FileContents saveFileDialog(String pathHint, String[] extensions, FileContents data)`

  Эти методы выводят предупреждающее сообщение и диалоговое окно для выбора файлов. Они записывают данные и возвращают дескрипторы файлов, выбранных пользователем, или значение `null`, если пользователь не выбрал ни одного файла.

API javax.jnlp.PersistenceService

- `long create(URL key, long maxsize)`

  Создает в постоянном хранилище запись, идентифицируемую заданным ключом. Возвращает максимальный размер памяти, предоставленной элементу.

- `void delete(URL key)`

  Удаляет запись, соответствующую заданному ключу.

- `String[] getNames(URL url)`

  Возвращает относительные имена всех ключей, начинающихся с заданного URL.

- `FileContents get(URL key)`

  Определяет дескриптор, с помощью которого можно модифицировать данные, связанные с заданным ключом. Если заданному ключу ничего не соответствует, генерируется исключение `FileNotFoundException`.

Аплеты

Аплеты — это программы Java, включаемые в страницы HTML. Страница HTML должна сообщать браузеру, какой аплет нужно загрузить, и куда поместить каждый аплет на странице. Как вы и могли ожидать, для использования аплета необходим специальный дескриптор, который сообщает браузеру, где взять файлы классов, и как аплет позиционирован на Web-странице (размер, местоположение и т.п.). Затем браузер извлекает необходимые файлы классов из Internet (или из каталога на пользовательской машине) и автоматически запускает аплет.

Когда появились аплеты, программисты должны были использовать для просмотра Web-страниц браузер HotJava, специально созданный компанией Sun для поддержки аплетов. Популярность аплетов существенно возросла, когда компания Netscape включила виртуальную машину языка Java в свой браузер. Вскоре такая возможность появилась и в Internet Explorer. К сожалению, возникла проблема. Компания Netscape не стала поддерживать более современные версии языка Java, а компания Microsoft склонялась к решению вовсе отказаться от Java.

Для того чтобы решить эту проблему, компания Sun разработала и распространила инструмент под названием Java Plug-In. Используя различные механизмы расширения возможностей Internet Explorer и Netscape, этот инструмент можно совершенно свободно встроить в оба браузера. Это позволяет использовать их для запуска аплетов с помощью среды поддержки Java (Java Runtime Environment), разработанной компанией Sun. Предполагалось, что Java Plug-In позволит воспользоваться преимуществами последних версий виртуальных машин.



На заметку! Чтобы запустить аплеты, код которых приведен в настоящей главе, с помощью своего браузера, необходимо инсталлировать текущую версию программы Java Plug-In и убедиться в том, что браузер связан с ней. Java Plug-In вместе с информацией по конфигурированию можно найти на сайте `http://java.com`.

Простой аплет

По традиции начнем с простейшего аплета. Перепишем в виде аплета программу `NotHelloWord`. В рамках этой книги для реализации аплетов мы будем пользоваться библиотекой Swing. Все наши аплеты будут расширять класс `JApplet`, являющийся суперклассом всех аплетов, создаваемых с помощью библиотеки Swing. Как показано на рис. 10.11, класс `JApplet` является подклассом обычного класса `Applet`.



На заметку! Если аплет содержит компоненты библиотеки Swing, следует расширять класс `JApplet`. Компоненты Swing внутри объекта обычного класса `Applet` будут отображаться неправильно.

В листинге 10.3 показан код аплета, выполняющего те же действия, что и программа Not Hello World.

Обратите внимание на сходство этого аплета с программой, приведенной в главе 7. Однако, поскольку аплет существует внутри Web-страницы, здесь нет необходимости указывать метод для выхода из аплета.

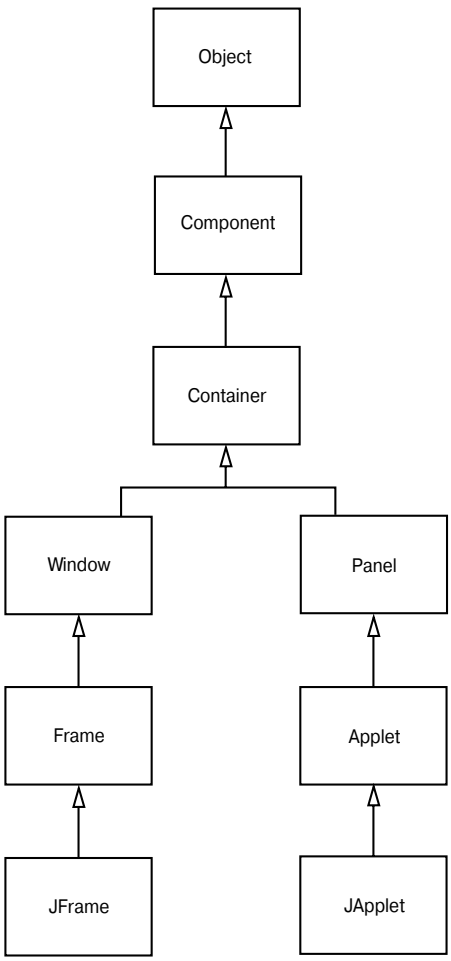


Рис. 10.11. Диаграмма наследования аплетов

Листинг 10.3. Содержимое файла NotHelloWorldApplet.java

```
1.  /*
2.   * Для выполнения аплета в среде браузера нужен
3.   * следующий HTML-элемент:
4.   * <applet code="NotHelloWorldApplet.class"
5.   *   width="300" height="100">
6.   * </applet>
7.   */
8.
9.  import java.awt.*;
10. import javax.swing.*;
11.
12. /**
13.  * @version 1.22 2007-06-12
14.  * @author Cay Horstmann
15.  */
16. public class NotHelloWorldApplet extends JApplet
17. {
18.    public void init()
19.    {
20.       EventQueue.invokeLater(new Runnable()
21.          {
22.             public void run()
23.             {
24.                JLabel label = new JLabel("Not a Hello, World applet",
25.                                          SwingConstants.CENTER);
26.                add(label);
27.             }
28.          });
29.    }
30. }
```

Для того чтобы выполнить аплет, необходимо сделать следующее.

1. Скомпилировать исходные файлы в файлы классов.
2. Создать HTML-файл, сообщающий браузеру, какой файл класса следует загрузить первым, и какие размеры имеет аплет.

Файл в формате HTML принято (но не обязательно) называть так же, как и класс аплета, содержащийся внутри него. Следуя этой традиции, дадим нашему файлу имя `NotHelloWorldApplet.html`. Ниже приводится его содержимое.

```
<applet code="NotHelloWorldApplet.class" width=300 height=300>
</applet>
```

Перед тем как просмотреть аплет в браузере, желательно проверить его с помощью программы `appletviewer`, предназначенной для просмотра аплетов. Эта программа входит в пакет JDK. Чтобы запустить аплет с помощью `appletviewer`, нужно выполнить команду

```
appletviewer NotHelloWorldApplet.html
```

В командной строке указывается имя HTML-файла, а не файла класса. На рис. 10.12 показано окно программы `appletviewer`, в котором отображается аплет `NotHelloWorldApplet`.

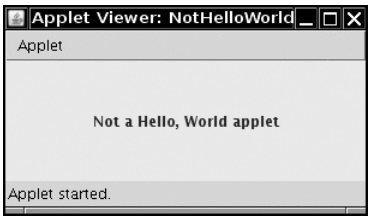


Рис. 10.12. Аплет в программе просмотра аплетов



Совет. Избежать создания дополнительного файла в формате HTML можно с помощью следующего приема. Добавьте дескриптор аплета в исходный файл *в виде комментариев*:

```
/*
   <applet code="MyApplet.class" width=300 height=300>
   </applet>
*/
public class MyApplet extends JApplet
...
```

Затем запустите программу `appletviewer`, задав в командной строке в качестве параметра *исходный файл*:

```
appletviewer NotHeloWorldApplet.java
```

Этот прием нельзя рекомендовать для постоянного применения, но он может оказаться удобным, если вы хотите минимизировать количество файлов, за которыми нужно следить во время отладки.



Совет. Аплеты можно также запускать на выполнение, находясь в текстовом редакторе или в интегрированной среде разработки. В редакторе Emacs выберите пункт меню JDE⇨Run Applet (JDE⇨Запустить аплет). В Eclipse выберите Run⇨Run as⇨Java Applet (Запустить⇨Запустить как⇨Java-аплет).

Программа `appletviewer` удобна на первых этапах тестирования, но в какой-то момент вам понадобится запустить аплет в браузере, чтобы почувствовать себя на месте пользователя. В частности, `appletviewer` показывает на экране только аплет без окружающего текста. Если файл в формате HTML содержит много аплетов, то `appletviewer` откроет на экране несколько отдельных окон.

Для того чтобы правильно просмотреть аплет, просто загрузите HTML-файл в браузер (рис. 10.13). Если аплет не отображается, возможно, браузер использует встроенную виртуальную машину. Чтобы решить эту проблему, вам придется настроить его на взаимодействие с Java Plug-In.



Совет. Если вы изменили аплет и скомпилировали его снова, вам нужно перезапустить браузер, чтобы он смог загрузить новые файлы классов. Простое обновление HTML-страницы не приводит к загрузке нового кода. Утомительной процедуры повторного запуска браузера можно избежать, открыв *консоль Java* и введя команду `x`, которая очищает кэш-память загрузчика классов. После этого вы можете повторно открывать HTML-файл, и на этот раз будет воспроизведен новый код аплета. Если вы работаете под управлением операционной системы Windows, щелкните на пиктограмме Java Plug-In, расположенной в панели управления. В среде Linux воспользуйтесь командой `jcontrol` и введите команду вывода на экран консоли Java. Консоль будет отображаться при каждой загрузке аплета.

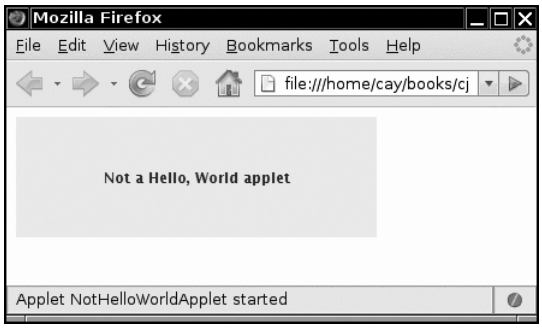


Рис. 10.13. Просмотр аплета в браузере

Преобразование приложения в аплет

Приложение с графическим интерфейсом, использующее библиотеку AWT (и допускающее запуск из командной строки) легко преобразовать в аплет и встроить в Web-страницу. Особенно важно, что при этом весь код пользовательского интерфейса можно оставить без изменений.

Для того решения данной задачи нужно выполнить следующие действия.

1. Создать HTML-страницу с соответствующим дескриптором для загрузки кода аплета.
2. Определить подкласс класса `JApplet`. Этот класс должен быть общедоступным (`public`), иначе аплет будет невозможно загрузить.
3. Удалить из приложения метод `main()`. Окно фрейма для приложения создавать не следует. Приложение будет отображаться в браузере.
4. Перенести все операторы инициализации из конструктора окна фрейма в метод аплета `init()`. Создавать объект аплета не обязательно — браузер сам создаст его и вызовет метод `init()`.
5. Удалить вызов метода `setSize()`. В аплетах размеры задаются в HTML-файле с помощью параметров `width` и `height`.
6. Удалить вызов метода `setDefaultCloseOperation()`. Аплет нельзя закрыть; он прекращает работу вместе с браузером.
7. Если в приложении есть вызов метода `setTitle()`, следует его удалить. Аплеты не имеют заголовков. (Разумеется, можно задать заголовок самой Web-страницы, используя HTML-дескриптор `<title>`.)
8. Не вызывать метод `setVisible(true)`. Аплет отображается на экране автоматически.



На заметку! Далее в этой главе вы узнаете, как можно создать программу, являющуюся одновременно и аплетом, и приложением.



java.applet.Applet 1.0

- `void init()`

  Вызывается при первой загрузке аплета. Переопределите этот метод и разместите в нем весь код, выполняющий инициализацию.

- `void start()`

  В этом методе размещается код, который подлежит выполнению *каждый раз*, когда пользователь посещает страницу браузера, содержащую данный аплет. Обычно этот метод повторно активизирует поток.

- `void stop()`

  В этом методе размещается код, который подлежит выполнению *каждый раз*, когда пользователь покидает страницу браузера, содержащую данный аплет. Обычно этот метод приостанавливает поток.

- `void destroy()`

  Переопределите этот метод, реализовав в нем код, который подлежит выполнению при выходе из браузера.

- `void resize(int width, int height)`

  Задает новые размеры аплета. Если бы этот метод влиял на аплеты, отображаемые на Web-страницах, он был бы очень удобен. К сожалению, с современными браузерами этот метод не работает, поскольку конфликтует с их механизмами компоновки.

HTML-дескрипторы и атрибуты для работы с аплетами

Дескриптор `<applet>` в его наиболее базовой форме выглядит приблизительно следующим образом:

```
<applet code>="NotHelloWorldApplet.class" width=100 height=100>
```

Как уже было сказано ранее, атрибут `code` задает имя класса; значение этого атрибута представляет собой имя файла, оканчивающееся расширением `.class`. Атрибуты `width` и `height` определяют размеры окна, в котором будет размещен аплет. Значения обоих параметров задаются в пикселях. Нужно также указать закрывающий дескриптор `</applet>`. Текст, размещенный между дескрипторами `<applet>` и `</applet>`, выводится на экран в том случае, если браузер не может отобразить аплет. Атрибуты `code` `width` и `height` необходимы. Если хотя бы один из них будет пропущен, браузер не сможет загрузить аплет.

Вся эта информация обычно встраивается в HTML-страницу и в самом минимальном виде может выглядеть так:

```
<head>
     <title>NotHelloWorldApplet</title>
  </head>
  <body>
    <p>Следующая строка текста выводится под управлением системы Java:</p>
    <applet code="NotHelloWorldApplet.class"
     width="100" height="100">
       Если ваш браузер поддерживает Java, на этом
       месте будет окно аплета.
    </applet>
  </body>
</html>
```

С дескриптором `applet` вы можете использовать описанные ниже атрибуты.

- `widtht`, `height`

  Эти атрибуты необходимы. Они задают ширину и высоту окна аплета в пикселях. В программе `appletviewer` эти параметры определяют первоначальный размер аплета. Программист может изменять размер любого окна, созданного программой `appletviewer`. В браузере изменить размеры аплета *невозможно*. Поэтому, включая аплет в состав документа, надо заранее продумать, сколько места понадобится аплету.

- `align`

  Этот атрибут задает вид выравнивания аплета. Значения атрибута те же, что и у атрибута `align` HTML-дескриптора `img`.

- `vspace`, `hspace`

  Эти необязательные атрибуты указывают количество пикселей над и под аплетом (`vspace`), а также по обеим его сторонам (`hspace`).

- `code`

  Данный атрибут задает имя файла, содержащего класс аплета. Имя берется относительно пути, заданного посредством атрибута `codebase` или относительно текущей страницы, если `codebase` не указан. Например, если класс аплета находится в пакете `com.mycompany`, то задается атрибут `code="com/mycompany/MyApplet.class"` или `code="com.mycompany.MyApplet.class"`. Атрибут `codebase` позволяет ссылаться на файл, расположенный в любой позиции файловой системы.

  Атрибут `code` задает только имя файла, содержащего класс аплета. Разумеется, аплет может содержать и другие файлы классов. Загружая класс, содержащий аплет, браузер распознает, какие файлы нужны еще, и загружает их дополнительно.

  Обязательным является либо атрибут `code`, либо атрибут `object` (см. ниже).

- `codebase`

  Этот необязательный атрибут информирует браузер о том, что файлы классов находятся в каталоге, указанном в качестве значения этого атрибута. По желанию вы можете указывать даже абсолютный URL каталога на другом сервере. Однако гораздо чаще задается относительный URL. Предположим, что структура каталогов выглядит так, как показано ниже.

  ```
  aDirectory/
        MyPage.html
        myApplets/
              CalculatorApplet.class
  ```

  В этом случае в документе `MyPage.html` может присутствовать следующий дескриптор:

  ```
  <applet code=CalculatorApplet.class" codebase="MyApplets" WIDTH=100 height=150>
  ```

- `archive`

  Этот необязательный атрибут перечисляет архивные файлы Java или файлы, содержащие классы и другие ресурсы для аплета. Такие файлы загружаются с сервера перед загрузкой аплета. Подобная технология значительно ускоряет работу, поскольку для загрузки JAR-файла, содержащего много мелких файлов, необходим лишь один HTML-запрос. В списке JAR-файлы разделяются запятыми.

  ```
  <applet code="CalculatorAPplet.class" archive=
    "CalculatorClasses.jar,corejava/CoreJavaClasses.jar"
    width=100 height=150>
  ```

- `object`

  Этот дескриптор позволяет задать имя файла, содержащего *сериализованный объект аплета*. (Объект называется *сериализованным*, если все его поля данных записаны в отдельном файле. Сериализация обсуждается во втором томе.) Перед отображением аплета объект десериализуется. Если в дескрипторе присутствует этот атрибут, метод `init()` не вызывается; выполнение аплета начинается с метода `start()`. Перед сериализацией объекта аплета необходимо вызвать его метод `stop`. Благодаря сериализации создается возможность реализовывать браузеры с сохранением состояния, в которых после повторной загрузки восстанавливается то состояние аплета, которое было в момент завершения предыдущего сеанса работы. К сожалению, эта возможность практически не используется разработчиками Web-страниц.

  В каждом дескрипторе `<applet>` должен содержаться атрибут `code` либо `object`.

  ```
  <applet object="CalculatorApplet.ser" width=100 height=150>
  ```

- `name`

  С помощью атрибута `name` аплету присваивается имя, которое затем можно использовать в сценарии. Браузеры Netscape и Internet Explorer позволяют вызывать методы аплета, размещенного на странице, средствами JavaScript. В этой книге вопросы, связанные с языком JavaScript, не рассматриваются, поэтому мы лишь вкратце опишем, как программа на языке Java вызывается из JavaScript-сценария.



На заметку! JavaScript представляет собой язык сценариев, которые можно использовать на Web-странице. Он разработан компанией Netscape и вначале назывался LiveScript. Этот язык никак не связан с языком Java. Название JavaScript было продиктовано маркетинговыми соображениями. Подмножество этого языка под названием ECMAScript было стандартизовано и получило название ECMA-262. Однако, что совершенно не удивительно, компании Netscape и Microsoft стали поддерживать в своих браузерах несовместимые между собой расширения этого стандарта. Дополнительную информацию по JavaScript можно найти в книге Девида Фланагана (David Flanagan) *JavaScript: The Definitive Guide* (O'Reilly Media, Inc., 2006).

Для того чтобы получить доступ к аплету из сценария на языке JavaScript, сначала нужно дать ему имя:

```
<applet code="CalculatorApplet.class" width=100 height=150 name="mine">
</applet>
```

Теперь этот объект доступен с помощью ссылки `document.applets.`*`имя_аплета`*:

```
var myApplet = document.applets.mine;
```

Используя интеграцию между языками Java и JavaScript, поддерживаемую браузерами Netscape и Internet Explorer, можно вызывать методы аплета:

```
myApplet.clear();
```

Атрибут `name` приобретает большое значение, если вы хотите поместить два взаимодействующих аплета на одну и ту же страницу. Сначала нужно присвоить имя каждому аплету. Затем следует передать строку с именем методу `getApplet()` класса `AppletComplex`. Механизм взаимодействия аплетов обсуждается далее в этой главе.



На заметку! В документе, расположенном по адресу `http://www.javaworld.com/javatips/jw-javatip80.html`, описывается, как можно использовать средства взаимодействия JavaScript и Java для решения классической проблемы: изменения размеров окна аплета, жестко заданных с помощью атрибутов `width` и `height`. Это хороший пример интеграции между Java и JavaScript.

- `alt`

  У некоторых пользователей Java может быть отключена в браузере — возможно, системным администратором-перестраховщиком. Тогда вы можете использовать атрибут `alt` для отображения сообщения таким пользователям:

```
<applet code="MyApplet.class" width="100" height="150"
alt="If you activated Java, you would see my applet here">
```

  Если браузер вообще не может обрабатывать аплеты, он игнорирует неизвестные ему дескрипторы `applet` и `param`. Весь текст между дескрипторами `<applet>` и `</applet>` отображается браузером. В противоположность этому, браузеры, оснащенные Java, не отображают текста между `<applet>` и `</applet>`. Вы можете отображать сообщения внутри этих дескрипторов для тех, кто до сих пор применяет доисторические браузеры.

```
<applet code="MyApplet.class" width="100" height="150">
Если бы ваш браузер понимал Java, здесь бы отображался аплет.
</applet>
```

Дескриптор <object>

Дескриптор `<object>` определен в спецификации HTML 4.0. Он предложен консорциумом W3C взамен дескриптора `<applet>`. У этого дескриптора есть 35 атрибутов, большинство из которых (например, `onkeydown`) предназначено для разработчиков, использующих Dynamic HTML. Атрибуты, определяющие размеры и местоположение аплета, например, `align` и `height`, ничем не отличаются от таких же атрибутов дескриптора `applet`. Ключевым атрибутом дескриптора `object`, имеющим отношение к Java-аплетам, является `classid`. Этот атрибут ссылается на объект аплета. Помимо аплетов, с помощью дескриптора `<object>` можно загружать и другие объекты, например, компоненты ActiveX. Тип загружаемого объекта указывается посредством атрибута `codetype`. Аплет, написанный на языке Java, имеет тип `application/java`. Для его загрузки нужно задать в дескрипторе `<object>` следующие атрибуты:

```
<object
   codetype="application/java"
   classid="java:CalculatorApplet.class"
   width="100" height="150">
```

Обратите внимание, что атрибут `classid` может сопровождаться атрибутом `codebase` точно так же, как это происходит в дескрипторе `<applet>`.

Передача информации аплетам через параметры

Подобно тому, как приложения могут воспринимать информацию, указанную в командной строке, аплеты могут использовать параметры, заданные в HTML-файле. Эта задача решается с помощью HTML-дескриптора `<param>`. Предположим, что вам нужно определить в составе Web-страницы стиль шрифта, использованного в аплете. Для этого понадобится задать следующие дескрипторы:

```
<applet code="FontParamApplet.class" width=200, height=200>
<param name="font" value="Helvetica" />
</applet>
```

Аплет получает значение параметра с помощью метода `getParameter()`, определенного в классе `Applet`.

```
public class FontParamApplet extends JApplet
{
   public void init()
   {
      String fontName = getParameter("font");
      ...
   }
   ...
}
```



На заметку! Вы можете вызывать метод `getParameter()` только в методе аплета `init()`, а не в конструкторе. Когда выполняется конструктор аплета, параметры еще не подготовлены. Поскольку компоновка большинства нетривиальных аплетов определяется параметрами, мы рекомендуем не применять конструкторов в аплетах. Просто поместите весь код инициализации в метод `init()`.

Параметры всегда возвращаются в виде строк. Вам нужно преобразовать строки в числовой тип, если они для этого вызваны. Это делается стандартным способом, с использованием соответствующего метода, такого как `parseInt()` из класса `Integer`.

Например, если бы требовалось добавить параметр размера шрифта, то код HTML мог бы выглядеть так:

```
<applet code="FontParamApplet.class" width="200" height="200">
  <param name="font" value="Helvetica"/>
  <param name="size" value="24"/>
</applet>
```

В следующем коде показано, как читать целочисленный параметр:

```
public class FontParamApplet extends JApplet
{
      public void init()
      {
               String fontName = getParameter("font");
               int fontSize = Integer.parseInt(getParameter("size"));
               . . .
      }
}
```



На заметку! Строки, используемые для определения параметров в дескрипторе `<param>`, и строки в методе `getParameter` должны совпадать абсолютно точно. В частности, в них должны различаться прописные и строчные буквы.

Кроме того, для гарантии, что параметры заданы правильно, нужно проверить, не пропущено ли значение параметра `size`. Для этого достаточно просто сравнить его со значением `null`.

```
int fontsize;
String sizeString = getParameter("size');
if (sizeString == null) fontSize = 12;
else fontSize = Integer.parsent(sizeString);
```

Ниже приведен пример аплета, широко использующего параметры. Этот аплет рисует на экране гистограмму, показанную на рис. 10.14.

Аплет получает в качестве параметров метки и числа, определяющие высоту столбиков гистограммы. Требуемые значения задаются в HTML-файле с помощью дескриптора <param>. Ниже показано, как выглядит HTML-файл, соответствующий рис. 10.14.

```
<applet code="Chart.class" width="400" height="300">
   <param name="title" value="Diameters of the Planets"/>
   <param name="values" value="9"/>
   <param name="name.1" value="Mercury"/>
   <param name="name.2" value="Venus"/>
   <param name="name.3" value="Earth"/>
   <param name="name.4" value="Mars"/>
   <param name="name.5" value="Jupiter"/>
   <param name="name.6" value="Saturn"/>
   <param name="name.7" value="Uranus"/>
   <param name="name.8" value="Neptune"/>
   <param name="name.9" value="Pluto"/>
   <param name="value.1" value="3100"/>
   <param name="value.2" value="7500"/>
   <param name="value.3" value="8000"/>
   <param name="value.4" value="4200"/>
   <param name="value.5" value="88000"/>
   <param name="value.6" value="71000"/>
   <param name="value.7" value="32000"/>
   <param name="value.8" value="30600"/>
   <param name="value.9" value="1430"/>
</applet>
```

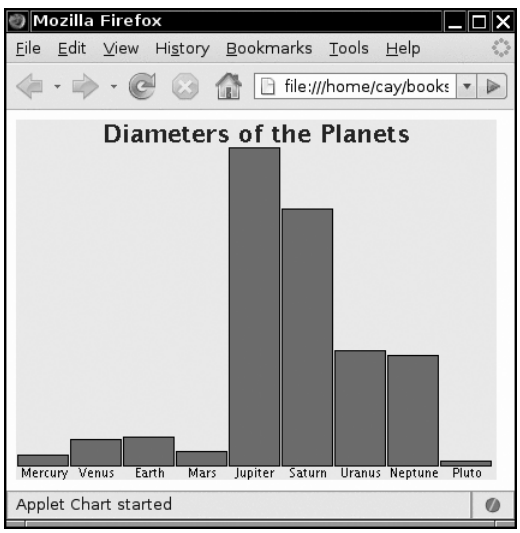


Рис. 10.14. Аплет, рисующий гистограмму

Эти значения можно было бы задать в виде массива строк или чисел, однако у дескрипторов `param` есть два преимущества. Во-первых, можно создать несколько копий одного и того же аплета на Web-странице, демонстрируя разные гистограммы: для этого достаточно поместить на страницу два дескриптора `applet` с разными наборами параметров. Во-вторых, данные, которые отображаются в виде гистограмм, можно изменять. Приведенный выше пример не очень хорошо демонстрирует сказанное, так как диаметры планет не изменяются. Поэтому предположим, что нам нужно поместить на Web-страницу другую гистограмму, отражающую объем еженедельных продаж. Web-страницу изменить легко, поскольку она содержит обычный текст. Редактировать и перекомпилировать файл на языке Java каждую неделю довольно утомительно.

Разумеется, существуют коммерческие компоненты JavaBeans, которые создают намного более сложные графические изображения, чем аплет, приведенный здесь в качестве примера. Купив один из таких компонентов, можно поместить его на свою Web-страницу и использовать, не задумываясь даже, как он работает.

В листинге 10.4 приведен исходный текст нашего аплета. Обратите внимание, что метод `init()` считывает параметры, а метод `paintComponent()` рисует гистограмму.

Листинг 10.4. Содержимое файла `Chart.java`

```
1. import java.awt.*;
2. import java.awt.font.*;
3. import java.awt.geom.*;
4. import javax.swing.*;
5.
6. /**
7.  * @version 1.33 2007-06-12
8.  * @author Cay Horstmann
9.  */
10. public class Chart extends JApplet
11. {
12.    public void init()
13.    {
14.       EventQueue.invokeLater(new Runnable()
15.          {
16.             public void run()
17.             {
18.                String v = getParameter("values");
19.                if (v == null) return;
20.                int n = Integer.parseInt(v);
21.                double[] values = new double[n];
22.                String[] names = new String[n];
23.                for (int i = 0; i < n; i++)
24.                {
25.                   values[i] = Double.parseDouble(getParameter("value." + (i + 1)));
26.                   names[i] = getParameter("name." + (i + 1));
27.                }
28.
29.                add(new ChartComponent(values, names, getParameter("title")));
30.             }
31.          });
32.    }
33. }
34.
```

```
35. /**
36.  * Панель с гистограммой.
37.  */
38. class ChartComponent extends JComponent
39. {
40.    /**
41.     * Конструирует ChartPanel.
42.     * @param v Массив значений гистограммы
43.     * @param n Массив меток гистограммы
44.     * @param t Заголовок гистограммы
45.     */
46.    public ChartComponent(double[] v, String[] n, String t)
47.    {
48.       values = v;
49.       names = n;
50.       title = t;
51.    }
52.
53.    public void paintComponent(Graphics g)
54.    {
55.       Graphics2D g2 = (Graphics2D) g;
56.
57.       // Вычисление минимального и максимального значений.
58.       if (values == null) return;
59.       double minValue = 0;
60.       double maxValue = 0;
61.       for (double v : values)
62.       {
63.          if (minValue > v) minValue = v;
64.          if (maxValue < v) maxValue = v;
65.       }
66.       if (maxValue == minValue) return;
67.
68.       int panelWidth = getWidth();
69.       int panelHeight = getHeight();
70.
71.       Font titleFont = new Font("SansSerif", Font.BOLD, 20);
72.       Font labelFont = new Font("SansSerif", Font.PLAIN, 10);
73.
74.       // Вычисление длины заголовка.
75.       FontRenderContext context = g2.getFontRenderContext();
76.       Rectangle2D titleBounds = titleFont.getStringBounds(title, context);
77.       double titleWidth = titleBounds.getWidth();
78.       double top = titleBounds.getHeight();
79.
80.       // Отображение заголовка.
81.       double y = -titleBounds.getY(); // ascent
82.       double x = (panelWidth - titleWidth) / 2;
83.       g2.setFont(titleFont);
84.       g2.drawString(title, (float) x, (float) y);
85.
86.       // Вычисление размеров меток гистограммы.
87.       LineMetrics labelMetrics = labelFont.getLineMetrics("", context);
88.       double bottom = labelMetrics.getHeight();
89.
90.       y = panelHeight - labelMetrics.getDescent();
91.       g2.setFont(labelFont);
92.
```

```
93.      // Вычисление масштаба и ширины столбиков.
94.      double scale = (panelHeight - top - bottom) / (maxValue - minValue);
95.      int barWidth = panelWidth / values.length;
96.
97.      // Отображение столбиков.
98.      for (int i = 0; i < values.length; i++)
99.      {
100.        // Получение координат прямоугольников.
101.        double x1 = i * barWidth + 1;
102.        double y1 = top;
103.        double height = values[i] * scale;
104.        if (values[i] >= 0) y1 += (maxValue - values[i]) * scale;
105.        else
106.        {
107.           y1 += maxValue * scale;
108.           height = -height;
109.        }
110.
111.        // Заполнение столбиков и рисование контуров.
112.        Rectangle2D rect = new Rectangle2D.Double(x1, y1, barWidth - 2, height);
113.        g2.setPaint(Color.RED);
114.        g2.fill(rect);
115.        g2.setPaint(Color.BLACK);
116.        g2.draw(rect);
117.
118.        // Рисование метки, выровненной по центру.
119.        Rectangle2D labelBounds = labelFont.getStringBounds(names[i], context);
120.
121.        double labelWidth = labelBounds.getWidth();
122.        x = x1 + (barWidth - labelWidth) / 2;
123.        g2.drawString(names[i], (float) x, (float) y);
124.     }
125.  }
126.
127.  private double[] values;
128.  private String[] names;
129.  private String title;
130.}
```

API java.applet.Applet 1.0

- `public String getParameter(String name)`

  Получает параметры, которые задаются посредством дескриптора `<param>`, расположенного на Web-странице. Прописные и строчные буквы различаются.

- `public String getAppletInfo()`

  Возвращает строку, содержащую информацию об авторе, версии программы и авторских правах на данный аплет. Требуемая информация задается при переопределении метода.

- `public String[][] getParameterInfo()`

  Данный метод переопределяется так, чтобы он возвращал массив параметров, поддерживаемых данным аплетом. В каждой строке содержится три элемента: имя, тип и описание параметра. Ниже приведен пример информации, возвращаемой в результате выполнения данного метода.

```
"fps", "1-10", "frames per second"
"repeat", "boolean", "repeat image loop?"
"images", "url", "directory containing images"
```

Обращение к файлам изображений и звуковым файлам

Аплеты могут отображать на экране графические изображения и воспроизводить аудиофайлы. Как известно, графические изображения представляются в форматах GIF или JPEG, а аудиофайлы — в форматах AU, AIFF, WAV или MIDI. Существует возможность создания анимационных GIF-изображений.

Вы указываете местоположение графических и звуковых файлов посредством относительных URL. Базовый URL обычно получается вызовом методов `getDocumentBase()` или `getCodeBase()`. Первый получает URL HTML-страницы, в которой содержится аплет, последний — URL каталога `codebase` аплета.



На заметку! В предыдущих версиях Java SE при описании обоих методов была допущена ошибка (зарегистрированная под номером #4456393; `http://bugs.sun.com/bugdatabase/index.jsp`). Эта ошибка была исправлена в Java SE 5.0.

Файлы с графическими изображениями и аудиоданные можно получать с помощью методов `getImage()` и `getAudioClip()`:

```
Image cat = getImage(getCodeBase(), "images/cat.gif");
AudioClip meow = getAudioClip(getCodeBase(), "audio/meow.au");
```

В главе 7 мы показали, как на экран выводится изображение. Для того чтобы воспроизвести аудиозапись, достаточно вызвать метод `play()`:

```
play(getCodeBase(), "audio/meow.au");
```

API java.applet.Applet 1.0

- `URL getDocumentBase()`

  Возвращает URL страницы, содержащей аплет.

- `URL getCodeBase()`

  Возвращает URL каталога, содержащего аплет. Это может быть либо абсолютный URL, либо ссылка на каталог, указываемый с помощью атрибута `codebase`, либо, если данный атрибут отсутствует, ссылка на каталог, в котором расположен HTML-файл.

- `void play(URL url)`
- `void play(URL url, String name)`

  Первый метод воспроизводит аудиофайлы, расположенные по указанному URL, а второй интерпретирует строку как адрес, определяемый относительно URL, заданного первым параметром. Если аудиофайл не найден, никакие действия не выполняются.

- `AudioClip getAudioClip(URL url)`
- `AudioClip getAudioClip(URL url, String name)`

  Первый метод получает аудиозапись, расположенную по заданному URL-адресу, а второй интерпретирует строку как адрес, определяемый относительно URL, заданного первым параметром. Если аудиофайл не найден, оба метода возвращают значение `null`.

- `Image getImage(URL url)`
- `Image getImage(URL url, String name)`

  Возвращает объект класса `Image`, инкапсулирующий графическое изображение, заданное своим URL. Если изображения по указанному адресу нет, метод немедленно возвращает `null`. В противном случае запускается отдельный поток для загрузки изображения.

Контекст аплета

Аплет запускается либо браузером, либо программой `appletviewer`. Аплет может поручить браузеру выполнить некоторые действия, например, извлечь аудиоданные, отобразить короткое сообщение в строке состояния или отобразить другую Web-страницу. Браузер, содержащий аплет, может удовлетворить его запросы либо проигнорировать их. Например, если аплет выполняется в среде `appletviewer`, его требование отобразить Web-страницу не будет удовлетворено.

Для взаимодействия с браузером аплет вызывает метод `getAppletContext()`. Этот метод возвращает объект, реализующий интерфейс `AppletContext`. Конкретную реализацию интерфейса `AppletContext` можно считать средством взаимодействия аплета и содержащего его браузера. В дополнение к методам `getAudioClip()` и `getImage()` в интерфейсе `AppletContext` объявлено несколько полезных методов, которые обсуждаются ниже.

Взаимодействие аплетов

Web-страница может содержать несколько аплетов. Если параметр `codebase` у этих аплетов один и тот же, они могут взаимодействовать друг с другом. Естественно, что взаимодействие аплетов — довольно сложный процесс, необходимость в котором возникает не часто.

Если для каждого аплета задать атрибут `name`, то с помощью метода `getApplet(String)`, объявленного в интерфейсе `AppletContext`, можно получить ссылку на любой аплет. Предположим, что в HTML-файле содержится дескриптор

```
<applet code="Chart.class" width=100 height=100 name="Chart1">
```

В этом случае в результате вызова приведенного ниже метода будет получена ссылка на аплет:

```
Applet chart1 = getAppletContext().getApplet("Chart1");
```

Что можно сделать с этой ссылкой? Если в классе `Chart` есть метод, предназначенный для получения новых данных и обновления гистограммы, его можно вызвать, выполнив соответствующее приведение типов:

```
((Chart)chart1).setData(3, "Earth", 9000);
```

Кроме того, можно перечислить все аплеты, размещенные на Web-странице, независимо от того, имеют они атрибут `name` или нет. Метод `getApplets()` возвращает объект `Enumeration` (он будет подробно рассмотрен во втором томе). Этот объект, в частности, предоставляет доступ ко всем аплетам. Ниже приведен фрагмент кода, выводящий имена аплетов, расположенных на текущей Web-странице.

```
Enumeration e = getAppletContext().getApplets();
while (e.hasMoreElements())
{
   Object a = e.nextElements();
   System.out.println(a.getClass().getName());
}
```

Аплет не может взаимодействовать с аплетами, находящимися на другой Web-странице.

Отображение элементов в браузере

У программиста есть доступ к двум областям браузера, содержащего аплет: строке состояния и области отображения Web-страницы. В обоих случаях используются методы класса `AppletContext`.

С помощью метода `showStatus()` можно вывести строку сообщения в строке состояния:

```
showStatus("Загрузка данных ... пожалуйста, подождите");
```



Внимание! Опыт показывает, что метод `showStatus()` имеет ограниченную область применения. Браузер также использует строку состояния и в большинстве случаев заменяет ранее выведенную строку своими сообщениями, например, `Applet running` (Аплет выполняется). Не используйте строку состояния для вывода важных сообщений, которые пользователь не должен пропустить.

С помощью метода `showDocument()` можно отобразить другую Web-страницу. Простейший способ — вызвать метод `showDocument()` с одним параметром, задающим URL нужного документа, например:

```
URL u = new URL("http://java.sun.com/index.html");
getAppletContext().showDocument(u);
```

При этом возникает одна проблема. Дело в том, что браузер открывает новую Web-страницу в том же самом окне, в котором отображался текущий документ, заменяя таким образом ваш аплет. Чтобы снова получить возможность работать с аплетом, пользователь должен щелкнуть на кнопке Back (Назад), расположенной в панели инструментов браузера.

Вызывая метод `showDocument()` с двумя параметрами, можно указать браузеру на то, что документ должен быть выведен в другом окне. Значения второго параметра описаны в табл. 10.2. Второй параметр является строкой. Если вы зададите специальную строку `"_blank"`, браузер откроет новое окно, не изменяя текущее. Что еще важнее, можно разделить окно браузера на несколько фреймов и присвоить каждому из них имя. Затем можно поместить аплет внутрь фрейма, а остальные документы отображать в других фреймах.

Таблица 10.2. Значения второго параметра метода `showDocument()`

Значение параметра	Место вывода документа
`"_self"` или отсутствует	Текущий фрейм.
`"_parent"`	Родительский фрейм.
`"_top"`	Фрейм самого высокого уровня.
`"_blank"`	Новое безымянное окно верхнего уровня.
Любая другая строка	Фрейм, имеющий заданное имя. Если фрейма с таким именем не существует, открывается новое окно, получающее заданное имя.



На заметку! Программа `appletviewer` от Sun не отображает Web-страницы, поэтому она игнорирует метод `showDocument()`.

Аплет или приложение? И то и другое!

Несколько лет назад юмористическая программа "Субботний вечер в прямом эфире" на американском коммерческом телевизионном канале показала спор супругов о белой желеобразной субстанции. Муж сказал: "Это — десертное блюдо". Жена сказала: "Это — мастика для паркета". А ведущий триумфально подвел итог: "Это — и то и другое!"

В этом разделе мы покажем, как писать программы, представляющие собой и аплет, и приложение одновременно. Такую программу можно выполнять с помощью `appletviewer` или в среде браузера либо запускать из командной строки под управлением интерпретатора `java`. Мы не уверены, что это так уж необходимо, — просто интересно, что подобная задача оказалась в принципе разрешимой.

На рис. 10.15 и 10.16 показана *одна и та же программа*, запущенная из командной строки как приложение и в среде `appletviewer` как аплет.

Каждый файл класса содержит только один общедоступный класс. Чтобы запустить аплет, этот класс должен быть подклассом класса `Applet`. Чтобы запустить приложение, нужен статический метод `main()`. Итак, нужно написать код, подобный приведенному ниже.

```
class MyAppletApplication extends JApplet
{
   public void init() { ... }
   ...
   public static void main(String[] args) { ... }
}
```

Как должен выглядеть метод `main()`? Обычно в нем создается экземпляр класса и вызывается метод `setVisible(true)`. Однако в нашем случае задача не так проста. Метод `setVisible()` для обычного аплета вызывать нельзя. Аплет должен находиться в составе фрейма.

Для того чтобы сформировать фрейм, создадим класс `AppletFrame`.

```
public class AppletFrame extends JFrame
{
   public AppletFrame(Applet anApplet)
   {
      applet = anApplet;
      add(applet);
      ...
   }
   ...
}
```



Рис. 10.15. Это аплет



Рис. 10.16. Это приложение

Конструктор фрейма помещает аплет (расширяющий класс `Component`) в панель содержимого. В методе `main()` нашего аплета/приложения создадим и отобразим `AppletFrame`.

```
class MyAppletApplication extends JApplet
{
   public void init() { . . . }
   . . .
   public static void main(String args[])
   {
      AppletFrame frame = new AppletFrame(new MyAppletApplication());
      frame.setTitle("MyAppletApplication");
      frame.setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

При запуске аплета должны вызываться методы `init()` и `start()`. Мы обеспечим их запуск, переопределив метод `setVisible()`.

```
public void setVisible(boolean b)
{
   if (b)
   {
      applet.init();
      super.setVisible(true);
      applet.start();
   }
   else
   {
      applet.stop();
      super.setVisible(false);
      applet.destroy();
   }
}
```

Однако здесь таится ловушка. Если программа запускается интерпретатором, а не программой `appletviewer`, и вызывает метод `getAppletContext()`, она получит в ответ ссылку `null`, поскольку запускается вне среды браузера. Во время работы программы попытка выполнить приведенный ниже оператор закончится неудачей.

```
getAppletContext().showStatus(message);
```

Мы не собираемся писать полнофункциональный браузер, достаточно правильно обработать указанный вызов. Он не выводит на экран никаких сообщений, но, по крайней мере, не приводит к аварийному завершению программы. Для того чтобы правильно его выполнить, достаточно реализовать два интерфейса: `AppletStub` и `AppletContext`.

Вы уже видели контексты аплетов в действии. Они выполняют загрузку графических и аудиофайлов, а также отображают на экране Web-страницы. Однако контексты могут вежливо отказаться выполнять команды, и именно это сделал контекст нашего аплета. Основная цель интерфейса `AppletStub` — обнаружить контекст аплета.

В нашем случае класс `AppletFrame` реализует интерфейсы `AppletStub` и `AppletContext` одновременно. Мы предусмотрели минимум функциональных возможностей, необходимых для реализации этих интерфейсов.

```
public class AppletFrame extends JFrame
   implements AppletStub, AppletContext
{
   . . .
   // Методы, объявленные в интерфейсе AppletStub
   public boolean isActive() { return true; }
   public URL getDocumentBase() { return null; }
   public URL getCodeBase() { return null; }
   public String getParameter(String name) { return ""; }
   public AppletContext getAppletContext() { return this; }
   public void appletResize(int width, int height) {}

   // Методы, объявленные в интерфейсе AppletContext
   public AudioClip getAudioClip(URL url) { return null; }
   public Image getImage(URL url) { return null; }
   public Applet getApplet(String name) { return null; }
   public Enumeration getApplets() { return null; }
   public void showDocument(URL url) {}
   public void showDocument(URL url, String target) {}
   public void showStatus(String status) {}
   public void setStream(String key, InputStream stream) {}
   public InputStream getStream(String key) { return null; }
   public Iterator getStreamKeys() { return null; }
}
```

Затем конструктор фрейма вызывает метод `setStub()` для аплета и создается его заглушка.

```
public AppletFrame(Applet anApplet)
{
   applet = anApplet;
   Container contentPane = getContentPane();
   contentPane.add(applet);
   applet.setStub(this);
}
```

Просто забавы ради мы оформили дескриптор `<applet>` в виде комментария в файле исходного кода. Теперь можно вызвать программу `appletviewer`, задав в командной строке имя исходного файла и не прибегая к помощи HTML-файлов.

Соответствующий код приведен в листингах 10.5 и 10.6. Запустить аплет/приложение можно с помощью следующих команд:

```
appletviewer NotHelloAppletApplication.java
java NotHelloAppletApplication
```

Листинг 10.5. Содержимое файла `AppletFrame.java`

```
1.  import java.awt.*;
2.  import java.applet.*;
3.  import java.io.*;
4.  import java.net.*;
5.  import java.util.*;
6.  import javax.swing.*;
7.
8.  /**
9.   * @version 1.32 2007-06-12
10.  * @author Cay Horstmann
11.  */
```

```
12. public class AppletFrame extends JFrame implements AppletStub, AppletContext
13. {
14.    public AppletFrame(Applet anApplet)
15.    {
16.       applet = anApplet;
17.       add(applet);
18.       applet.setStub(this);
19.    }
20.
21.    public void setVisible(boolean b)
22.    {
23.       if (b)
24.       {
25.          applet.init();
26.          super.setVisible(true);
27.          applet.start();
28.       }
29.       else
30.       {
31.          applet.stop();
32.          super.setVisible(false);
33.          applet.destroy();
34.       }
35.    }
36.
37.    // Методы AppletStub.
38.    public boolean isActive()
39.    {
40.       return true;
41.    }
42.
43.    public URL getDocumentBase()
44.    {
45.       return null;
46.    }
47.
48.    public URL getCodeBase()
49.    {
50.       return null;
51.    }
52.
53.    public String getParameter(String name)
54.    {
55.       return "";
56.    }
57.
58.    public AppletContext getAppletContext()
59.    {
60.       return this;
61.    }
62.
63.    public void appletResize(int width, int height)
64.    {
65.    }
66.
```

```
67.    // Методы AppletContext.
68.    public AudioClip getAudioClip(URL url)
69.    {
70.       return null;
71.    }
72.
73.    public Image getImage(URL url)
74.    {
75.       return Toolkit.getDefaultToolkit().getImage(url);
76.    }
77.
78.    public Applet getApplet(String name)
79.    {
80.       return null;
81.    }
82.
83.    public Enumeration<Applet> getApplets()
84.    {
85.       return null;
86.    }
87.
88.    public void showDocument(URL url)
89.    {
90.    }
91.
92.    public void showDocument(URL url, String target)
93.    {
94.    }
95.
96.    public void showStatus(String status)
97.    {
98.    }
99.
100.   public void setStream(String key, InputStream stream)
101.   {
102.   }
103.
104.   public InputStream getStream(String key)
105.   {
106.      return null;
107.   }
108.
109.   public Iterator<String> getStreamKeys()
110.   {
111.      return null;
112.   }
113.
114.   private Applet applet;
115. }
```

Листинг 10.6. Содержимое файла `AppletApplication.java`

```
1.  /*
2.   * Если для выполнения аплета используется appletviewer,
3.   * то она считывает указанные ниже дескрипторы.
4.   * В этом случае отдельный HTML-файл не нужен.
5.   * <applet code="AppletApplication.class"
6.   *  width="200" height="200">
7.   * </applet>
8.   */
9.
10. import java.awt.EventQueue;
11.
12. import javax.swing.*;
13.
14. /**
15.  * Это аплет. И это приложение. Два в одном!
16.  * @version 1.32 2007-04-28
17.  * @author Cay Horstmann
18.  */
19. public class AppletApplication extends NotHelloWorldApplet
20. {
21.    public static void main(String[] args)
22.    {
23.       EventQueue.invokeLater(new Runnable()
24.          {
25.             public void run()
26.             {
27.                AppletFrame frame = new AppletFrame(new NotHelloWorldApplet());
28.                frame.setTitle("NotHelloWorldApplet");
29.                frame.setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
30.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
31.                frame.setVisible(true);
32.             }
33.          });
34.    }
35.
36.    public static final int DEFAULT_WIDTH = 200;
37.    public static final int DEFAULT_HEIGHT = 200;
38. }
```

API `java.applet.Applet 1.2`

- `public AppletContext getAppletContext()`

  Предоставляет дескриптор среды браузера аплета. В большинстве браузеров вы можете использовать эту информацию для управления браузером, в котором запущен аплет.

- `void showStatus(String msg)`

  Отображает сообщение в строке состояния браузера.

- `Enumeration<Applet> getApplets()`

  Возвращает перечисление (см. главу 13) всех аплетов одного и того же контекста, т.е. находящихся на одной странице.

- `Applet getApplet(String name)`

  Возвращает аплет из текущего контекста по его имени; если аплет с таким именем не найден, возвращает `null`. Просматривается только текущая страница.

- `void showDocument(URL url)`
- `void showDocument(URL url, String target)`

  Отображает новую Web-страницу в том же фрейме браузера. В первой форме новая страница заменяет текущую. Вторая форма использует параметр `target` для идентификации целевого фрейма (см. табл. 10.2).

Хранение параметров настройки

Пользователи ваших приложений обычно ожидают, что их предпочтения и настройки будут сохранены, а при следующем запуске приложения восстановлены снова. Сначала мы рассмотрим простой подход к сохранению конфигурационной информации в файле свойств, который традиционно используется приложениями Java. Затем опишем более мощный механизм предпочтений, который появился в Java SE 1.4.

Карты свойств

Карта свойств (property map) — это структура данных, в которой хранятся пары "ключ–значение". Карты свойств часто используются для хранения информации о параметрах настройки и характеризуются тремя особенностями.

- Ключи и значения являются строками.
- Всю структуру легко записать в файл и прочитать оттуда.
- Существует вспомогательная таблица для значений, принимаемых по умолчанию.

Карты свойств используются при настройке программ. Рассмотрим пример.

```
Properties settings = new Properties();
settings.put("font", "Courier");
settings.put("size", "10");
settings.put("message", "Hello, World");
```

Для записи списка свойств в файл используется метод `store()`. В данном случае мы просто выведем карту свойств в файл `program.properties`. Второй параметр представляет собой комментарии, которые помещаются в файл.

```
FileOutputStream  out
   = new FileOutputStream("Myprog.properties");
settings.store(out, "Myprog Properties");
```

В результате выполнения приведенного фрагмента будут выведены следующие данные:

```
#Myprog Properties
#Sun Jul 14 07:22:52  2002
font=Courier
size=10
message=Hello,World
```

Чтобы загрузить эти свойства из файла, необходимо выполнить следующие операторы:

```
FileInputStream in
   = new FileInputStream("Myprog.properties");
settings.load(in);
```

Обычно свойства программы принято сохранять в подкаталоге домашнего каталога пользователя. Имя такого каталога часто начинается с точки — в системах UNIX это соглашение указывает на системный каталог, скрытый от пользователя. Наша программа следует этому соглашению.

Чтобы найти домашний каталог пользователя, вы можете вызвать метод `System.getProperties()`, который, так уж случилось, тоже использует объект `Properties` для описания системной информации. Домашний каталог имеет ключ `user.home`. Для чтения одиночного ключа предусмотрен удобный метод:

```
String userDir = System.getProperty("user.home");
```

Если в карте свойств есть свойство `title`, то `title` устанавливается в эту строку. В противном случае `title` устанавливается в `"Default title"`.

Если вы сочтете, что задавать значения по умолчанию при каждом вызове метода `getProperty()` слишком утомительно, он может упаковать все значения, принятые по умолчанию, во вспомогательную таблицу свойств и передать ее конструктору искомой таблицы.

```
Properties defaultSettings = new Properties();
defaultSettings.put("font", "Courier");
defaultSettings.put("size", "10");
defaultSettings.put("color.red", "0");
...
Properties settings = new Properties(defaultSettings);
FileStream in
   = new FileInputStream("CustomWorld.properties");
settings.load(in);
...
```

Вы даже можете специфицировать стандартные значения для значений по умолчанию, если передадите в качестве параметра другую карту свойств конструктору `defaultSettings`, но обычно так поступать не принято.

В листинге 10.7 показано, как можно использовать свойства для сохранения и загрузки состояния программы. Программа запоминает положение фрейма, размер и заголовок. Вы можете также вручную отредактировать файл `.corejava/program.properties` в вашем домашнем каталоге, чтобы придать программе желаемый внешний вид.



На заметку! Свойства хранятся в простой таблице, не имеющей иерархической структуры. Как правило, используется фиктивная иерархия с именами ключей `window.main.color`, `window.main.title` и т.п. Однако в классе `Properties` нет методов, позволяющих создать настоящую иерархию. Если в файле хранится сложная информация о конфигурации программы, следует использовать класс `Preferencs`, который будет описан в следующем разделе.

Листинг 10.7. Содержимое файла `PropertiesTest.java`

```
1.  import java.awt.EventQueue;
2.  import java.awt.event.*;
3.  import java.io.*;
4.  import java.util.Properties;
5.
6.  import javax.swing.*;
7.
```

```
8.  /**
9.  * Программа тестирования свойств. Запоминает размер,
10. * положение окна фрейма и его заголовок.
11. * @version 1.00 2007-04-29
12. * @author Cay Horstmann
13. */
14. public class PropertiesTest
15. {
16. public static void main(String[] args)
17. {
18.   EventQueue.invokeLater(new Runnable()
19.   {
20.     public void run()
21.     {
22.      PropertiesFrame frame = new PropertiesFrame();
23.      frame.setVisible(true);
24.     }
25.   });
26. }
27. }
28.
29. /**
30.  * Фрейм, восстанавливающий положение и размер
31.  * из файла свойств, и обновляющий его при
32.  * выходе.
33.  */
34. class PropertiesFrame extends JFrame
35. {
36. public PropertiesFrame()
37. {
38.   // Получить позицию, размер и заголовок из свойств.
39.
40.   String userDir = System.getProperty("user.home");
41.   File propertiesDir = new File(userDir,   ".corejava");
42.   if (!propertiesDir.exists()) propertiesDir.mkdir  ();
43.   propertiesFile = new File(propertiesDir,   "program.properties");
44.
45.   Properties defaultSettings = new Properties();
46.   defaultSettings.put("left", "0");
47.   defaultSettings.put("top", "0");
48.   defaultSettings.put("width", "" + DEFAULT_WIDTH);
49.   defaultSettings.put("height", "" + DEFAULT_HEIGHT);
50.   defaultSettings.put("title", "");
51.
52.   settings = new Properties(defaultSettings);
53.
54.   if (propertiesFile.exists()) try
55.   {
56.    FileInputStream in = new FileInputStream   (propertiesFile);
57.    settings.load(in);
58.   }
59.   catch (IOException ex)
60.   {
61.     ex.printStackTrace();
62.   }
63.
```

```
64.    int left = Integer.parseInt(settings.getProperty("left"));
65.    int top = Integer.parseInt(settings.getProperty("top"));
66.    int width = Integer.parseInt(settings.getProperty("width"));
67.    int height = Integer.parseInt(settings.getProperty("height"));
68.    setBounds(left, top, width, height);
69.
70.    // Если заголовка нет, запросить пользователя.
71.
72.    String title = settings.getProperty("title");
73.    if (title.equals(""))
74.      title = JOptionPane.showInputDialog("Please supply a frame title:");
75.    if (title == null) title = "";
76.    setTitle(title);
77.
78.    addWindowListener(new WindowAdapter()
79.    {
80.     public void windowClosing(WindowEvent event)
81.     {
82.      settings.put("left", "" + getX());
83.      settings.put("top", "" + getY());
84.      settings.put("width", "" + getWidth());
85.      settings.put("height", "" + getHeight());
86.      settings.put("title", getTitle());
87.      try
88.      {
89.        FileOutputStream out = new FileOutputStream(propertiesFile);
90.        settings.store(out, "Program Properties");
91.      }
92.      catch (IOException ex)
93.      {
94.        ex.printStackTrace();
95.      }
96.      System.exit(0);
97.     }
98.   });
99.  }
100.
101. private File propertiesFile;
102. private Properties settings;
103.
104. public static final int DEFAULT_WIDTH = 300;
105. public static final int DEFAULT_HEIGHT = 200;
106. }
```

API java.util.Properties 1.0

- `Properties()`

  Создает пустой список свойств.

- `Properties(Properties defaults)`

  Создает пустой список свойств со значениями, заданными по умолчанию.

  Параметр `defaults` Значения, заданные по умолчанию.

- `String getProperty(String key)`

  Возвращает строку, связанную с заданным ключом, или, если ключ в основной таблице отсутствует, строку, связанную с ключом из таблицы значений, принятых по умолчанию. Если указанного ключа в таблице значений, принятых по умолчанию, тоже нет, возвращается значение `null`.

  Параметр `key` Ключ, связанный с возвращаемой строкой.

- `String getProperty(String key, String defaultValue)`

  Возвращает строку, связанную с указанным ключом. Если ключа в таблице нет, возвращает свойство, заданное по умолчанию.

  Параметры `key` Ключ, связанный с возвращаемой строкой.

  `defaultValue` Строка, которая возвращается, если ключа в таблице нет.

- `void load(InputStream in) throws IOException`

  Загружает карту свойств из входного потока.

  Параметр `in` Входной поток

- `void store(OutputStream out, String header)` **1.2**

  Записывает карту свойств в выходной поток.

  Параметры `out` Выходной поток.

  `header` Заголовок, помещаемый в первую строку файла.

API `java.lang.System` 1.0

- `Properties getProperties()`

  Извлекает все системные свойства. Приложение должно иметь доступ ко всем свойствам, в противном случае генерируется исключение.

- `String getProperty(String key)`

  Извлекает системное свойство, связанное с заданным ключом. Если приложение не имеет доступа к данному свойству, генерируется исключение.

  Следующие свойства всегда доступны для извлечения:

```
java.version
java.vendor
java.vendor.url
java.class.version
os.name
os.version
os.arch
file.separator
path.separator
line.separator
java.specification.version
java.vm.specification.version
java.vm.specification.vendor
java.vm.specification.name
java.vm.version
java.vm.vendor
java.vm.name
```



На заметку! Все имена свободно доступных системных свойств можно найти в файле `security/java.policy` в каталоге исполняющей системы Java.

Интерфейс Preferences API

Как видим, класс `Properties` позволяет легко загружать и сохранять информацию о конфигурации системы. Однако файлы свойств обладают рядом недостатков.

- В ряде случаев невозможно хранить конфигурационные файлы в рабочем каталоге пользователя. Некоторые операционные системы (например, Windows 9x) вообще не поддерживают рабочие каталоги.
- Стандартные соглашения об именах конфигурационных файлов отсутствуют. Таким образом, если пользователь инсталлирует несколько приложений, возникает опасность конфликта имен.

Некоторые операционные системы хранят информацию о своих установках в центральном хранилище конфигурационных данных. Наиболее известным примером такого хранилища является системный реестр Windows. Класс `Preferences`, реализованный в Java SE 1.4, поддерживает подобное центральное хранилище, не зависящее от операционной системы. На платформе Windows класс `Preferences` использует системный реестр, на платформе Linux информация хранится в локальном системном файле. Разумеется, реализация хранилища абсолютно прозрачна для программиста, использующего класс `Preferences`.

Хранилище `Preferences` имеет древовидную структуру, в узлах которой содержатся имена путей, например `/com/mycompany/myapp`. Как и в случае пакетов, рекомендуется во избежание конфликта имен формировать имена на основе доменных имен, записывая их в обратном порядке. Разработчики интерфейса API предполагают, что пути к узлам совпадают с именами пакетов, используемых в программе.

Каждый узел в хранилище содержит отдельную таблицу пар "ключ–значение", которую можно использовать для записи чисел, строк или байтовых массивов. Возможность хранить сериализованные объекты не предусмотрена. Разработчики интерфейса API чувствовали, что формат сериализации слишком хрупок, и обеспечить его долговременное хранение непросто. Разумеется, если вы не согласны с этим, можно хранить сериализованные объекты в виде байтовых массивов.

Для обеспечения дополнительной степени гибкости предусмотрено несколько параллельных деревьев. Каждая программа использует одно дерево. Кроме того, существует дополнительное дерево, называемое системным и предназначенное для хранения настроек, общих для всех пользователей. Для доступа к соответствующему пользовательскому дереву класс `Preferences` использует понятие "текущий пользователь".

Поиск требуемого узла начинается с пользовательского или системного корня.

```
Preferences root = Preferences.userRoot();
```

или

```
Preferences root = Preferences.systemRoot();
```

Теперь для доступа к узлу достаточно указать соответствующий путь.

```
Preferences node = root.node("/com/mycompany/myapp");
```

Как правило, параметр `obj` является ссылкой `this`.

Получив доступ к узлу, можно обратиться к таблице "ключ–значение", используя указанные ниже методы.

```
String get(String key, String defval)
int getInt(String key, int defval)
long getLong(String key, long defval)
float getFloat(String key, float defval)
double getDouble(String key, double defval)
boolean getBoolean(String key, boolean defval)
byte[] getByteArray(String key, byte[] defval)
```

Обратите внимание, что если хранилище может оказаться недоступным, то при считывании этой информации необходимо указывать значения, предусмотренные по умолчанию. Эти значения необходимы по нескольким причинам. Во-первых, данные могут быть неполными, так как пользователи не всегда указывают все параметры. Во-вторых, на некоторых платформах при недостатке ресурсов хранилища не могут создаваться. В-третьих, переносные устройства могут быть временно отсоединены от хранилища.

С помощью методов `put` можно записать данные в хранилище:

```
put(String key, String value)
putInt(String key, int value)
```

и т.д.

Перечислить все ключи, сохраненные в узле, можно с помощью следующего метода:

```
String[] keys
```

Однако в настоящий момент нет способа определить тип значения, связанного с конкретным ключом.

Центральным хранилищам, например, системному реестру Windows, традиционно присущи два недостатка.

- В них постепенно накапливается устаревшая информация.
- Данные о конфигурации системы становятся все более запутанными, что затрудняет перенос установок на новую платформу.

Класс `Preferences` позволяет решить вторую проблему. Данные можно экспортировать в поддерево (или, в некоторых случаях, в отдельный узел), вызвав один из приведенных ниже методов.

```
void exportSubtree(outputStream out)
void exportNode(outputStream out)
```

Информация хранится в XML-формате. Ее можно импортировать в другое хранилище, вызвав следующий метод:

```
void importPreferences(inputStream in)
```

Рассмотрим пример XML-файла, содержащего конфигурационные данные.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE preferences SYSTEM
   "http://java.sun.com/dtd/preferences.dtd">
<preferences EXTERNAL_XML_VERSION="1.0">
  <root type="user">
    <map/>
    <node name="com">
      <map/>
      <node name="horstmann">
        <map/>
        <node name="corejava">
```

```
        <map>
          <entry key="left" value="11"/>
          <entry key="top" value="9"/>
          <entry key="width" value="453"/>
          <entry key="height" value="365"/>
          <entry key="title" value="Hello, World!"/>
        </map>
      </node>
    </node>
  </node>
 </root>
</preferences>
```

Если ваша программа использует параметры настройки, можно предоставить ее пользователям возможность экспортировать и импортировать их, что упрощает перенос программы с одного компьютера на другой. Код, приведенный в листинге 10.8, иллюстрирует данный подход. Программа сохраняет координаты и размер главного окна. Попробуйте изменить размеры окна, затем завершите приложение и запустите его снова. Размеры окна должны совпадать с размерами, указанными перед окончанием работы программы.

Листинг 10.8. Содержимое файла `PreferencesTest.java`

```
1. import java.awt.EventQueue;
2. import java.awt.event.*;
3. import java.io.*;
4. import java.util.prefs.*;
5. import javax.swing.*;
6.
7. /**
8.  * Программа тестирования средств настройки.
9.  * Запоминает координаты и размер фрейма.
10. * @version 1.02 2007-06-12
11. * @author Cay Horstmann
12. */
13.public class PropertiesTest
14.{
15.   public static void main(String[] args)
16.   {
17.      EventQueue.invokeLater(new Runnable()
18.         {
19.            public void run()
20.            {
21.               PropertiesFrame frame = new PropertiesFrame();
22.               frame.setVisible(true);
23.            }
24.         });
25.   }
26.}
27.
28./**
29. * Фрейм восстанавливает координаты и размер, заданные
30. * пользователем, и сохраняет установки перед выходом
31. * из программы.
32. */
```

```
33. class PropertiesFrame extends JFrame
34. {
35.    public PropertiesFrame()
36.    {
37.       // Получить позицию, размер и заголовок из свойств.
38.       String userDir = System.getProperty("user.home");
39.       File propertiesDir = new File(userDir, ".corejava");
40.       if (!propertiesDir.exists()) propertiesDir.mkdir();
41.       propertiesFile = new File(propertiesDir, "program.properties");
42.
43.       Properties defaultSettings = new Properties();
44.       defaultSettings.put("left", "0");
45.       defaultSettings.put("top", "0");
46.       defaultSettings.put("width", "" + DEFAULT_WIDTH);
47.       defaultSettings.put("height", "" + DEFAULT_HEIGHT);
48.       defaultSettings.put("title", "");
49.
50.       settings = new Properties(defaultSettings);
51.
52.       if (propertiesFile.exists()) try
53.       {
54.          FileInputStream in = new FileInputStream(propertiesFile);
55.          settings.load(in);
56.       }
57.       catch (IOException ex)
58.       {
59.          ex.printStackTrace();
60.       }
61.
62.       int left = Integer.parseInt(settings.getProperty("left"));
63.       int top = Integer.parseInt(settings.getProperty("top"));
64.       int width = Integer.parseInt(settings.getProperty("width"));
65.       int height = Integer.parseInt(settings.getProperty("height"));
66.       setBounds(left, top, width, height);
67.
68.       // Если нет заголовка, запросить пользователя.
69.
70.       String title = settings.getProperty("title");
71.       if (title.equals(""))
72.         title = JOptionPane.showInputDialog("Please supply a frame title:");
73.       if (title == null) title = "";
74.       setTitle(title);
75.
76.       addWindowListener(new WindowAdapter()
77.       {
78.          public void windowClosing(WindowEvent event)
79.          {
80.             settings.put("left", "" + getX());
81.             settings.put("top", "" + getY());
82.             settings.put("width", "" + getWidth());
83.             settings.put("height", "" + getHeight());
84.             settings.put("title", getTitle());
85.             try
86.             {
87.                FileOutputStream out = new FileOutputStream(propertiesFile);
88.                settings.store(out, "Program Properties");
89.             }
```

```
90.           catch (IOException ex)
91.           {
92.             ex.printStackTrace();
93.           }
94.           System.exit(0);
95.         }
96.       });
97.    }
98.
99.    private File propertiesFile;
100.   private Properties settings;
101.
102.   public static final int DEFAULT_WIDTH = 300;
103.   public static final int DEFAULT_HEIGHT = 200;
104. }
```

API java.util.prefs.Preferences 1.4

- `Preferences userRoot()`

  Возвращает из дерева настроек вызывающей программы корневой узел для пользователя.

- `Preference systemRoot()`

  Возвращает системный корневой узел из дерева настроек.

- `Preferences node(String path)`

  Возвращает узел, достижимый из текущего узла с помощью заданного пути. Если параметр `path` задает абсолютный путь (т.е. начинается с косой черты /), то узел достигается из корня дерева. Если узел отсутствует, он создается.

- `Preferences userNodeForPackage(Class c1)`
- `Preferences systemNodeForPackage(Class c1)`

  Возвращает узел из дерева текущего пользователя или системного дерева, абсолютный путь к которому соответствует имени пакета, содержащего класс `c1`.

- `String[] keys()`

  Возвращает все ключи, принадлежащие данному узлу.

- `String get(String key, String defval)`
- `int getInt(String key, int defval)`
- `long getLong(String key, long defval)`
- `float getFloat(String key, float defval)`
- `double getDouble(String key, double defval)`
- `boolean getBoolean(String key, boolean defval)`
- `byte[] getByteArray(String key, byte[] defval)`

  Эти методы возвращают значение, связанное с указанным ключом. Если значения в таблице нет, либо оно имеет неверный тип, либо сама таблица недоступна, возвращается значение, предусмотренное по умолчанию.

- `void put(String key, String value)`
- `void putInt(String key, int value)`
- `void putLong(String key, long value)`
- `void putFloat(String key, float value)`

- `void putDouble(String key, double value)`
- `void putBoolean(String key, boolean value)`
- `void putByteArray(String key, byte[] value)`

  Эти методы записывают пару "ключ–значение" в заданный узел.

- `void exportSubtree(OutputStream out)`

  Выводит в указанный поток параметры настройки, хранящиеся в заданном узле и узлах, дочерних по отношению к нему.

- `void exportNode(OutputStream out)`

  Выводит в указанный поток параметры настройки, хранящиеся в заданном узле (игнорируя дочерние узлы).

- `void importPreferences(InputStream in)`

  Импортирует параметры настройки, содержащиеся в указанном потоке.

На этом мы завершаем обсуждение способов доставки программного обеспечения, написанного на языке Java. В следующей главе описана обработка исключений, возникающих во время работы программы. Кроме того, мы дадим несколько советов, касающихся способов тестирования и отладки программ и позволяющих избежать многих проблем в ходе их выполнения.

Глава 11

ИСКЛЮЧЕНИЯ, ПРОТОКОЛИРОВАНИЕ, УТВЕРЖДЕНИЯ И ОТЛАДКА

В ЭТОЙ ГЛАВЕ...

- Обработка ошибок
- Перехват исключений
- Как использовать исключения
- Использование диагностических утверждений
- Протоколирование
- Приемы отладки
- Применение отладчика

В идеальном мире пользователи никогда не делают ошибок при вводе данных, файлы, которые они пытаются открыть, всегда существуют, а программы всегда работают правильно. До сих пор мы предполагали, что живем именно в таком мире. Теперь пришла пора спуститься на землю, населенную реальными людьми, делающими ошибки при программировании и вводе данных, и рассмотреть механизмы языка Java, которые позволяют в этом мире как-то существовать.

Сталкиваться с ошибками всегда неприятно. Если пользователь однажды потеряет все результаты своей работы из-за скрытой ошибки или непредвиденного стечения обстоятельств, он может навсегда отвернуться от такой программы. В подобной ситуации необходимо сделать, по крайней мере, следующее.

- Сообщить пользователю об обнаруженной ошибке.
- Сохранить все результаты его работы.
- Дать ему возможность аккуратно завершить программу.

В исключительных ситуациях, например, при некорректном вводе данных, которые могут привести к разрушению программы, язык Java использует механизм исправления ошибок, который называется *обработкой исключений*. Этот механизм аналогичен обработке исключений в C++ и Delphi. Он описывается в первой части этой главы.

Во время тестирования вы реализуете множество проверок, чтобы убедиться, что ваша программа работает правильно. Однако эти проверки зачастую оказываются слишком дорогостоящими по ресурсам и времени и потому по окончании тестирования — излишними. Вы можете просто удалить проверки и вернуть их обратно, когда потребуется дополнительное тестирование, хотя это утомительно. Вторая часть главы посвящена применению средств диагностических *утверждений* (assertions) для выборочной активизации проверок.

Когда ваша программа делает что-то не то, вы не всегда можете своевременно оказать помощь пользователю или прервать ее работу. Вместо этого вы хотите записать проблему для последующего анализа. Поэтому третья часть главы посвящена средствам протоколирования Java SE.

И, наконец, мы дадим вам несколько советов о том, как получить полезную информацию из работающего приложения Java, и как использовать отладчик в интегрированной среде обработки.

Обработка ошибок

Предположим, что в ходе выполнения программы, написанной на языке Java, обнаруживается ошибка. Она может быть вызвана неверной информацией в файле, неправильным сетевым соединением или (ненавидим упоминать об этом) выходом за пределы допустимого диапазона изменения индексов массива, а может быть попыткой использования ссылки, которая не указывает на какой-либо объект. Вполне естественно, пользователи надеются, что программа самостоятельно справится с возникшими неприятностями. Если из-за ошибки какая-либо операция не может быть завершена, то программа должна сделать одно из двух.

- Вернуться в безопасное состояние и позволить пользователю выполнить другие команды.
- Дать пользователю возможность сохранить результаты своей работы и аккуратно завершить работу.

Это может оказаться нелегко: фрагмент кода, в котором обнаруживается ошибка (или даже тот, выполнение которой становится причиной ошибки), обычно располагается довольно далеко от кода, который может восстановить данные и сохранить результаты, полученные пользователем. Механизм обработки исключений предназначен для передачи данных обработчику ошибок из того места, где возник сбой. Предусматривая обработку исключений в программе, нужно предвидеть возможные ошибки и проблемы, связанные с ними. Какие же ошибки следует рассмотреть в первую очередь?

- *Ошибки ввода.* В дополнение к неизбежным опечаткам пользователи часто предпочитают двигаться своим собственным маршрутом вместо задуманного разработчиком. Представим, что пользователь желает соединиться с Web-сайтом, но допустил синтаксическую ошибку, задавая URL. Ваша программа должна была бы проверить синтаксис, но, предположим, вы забыли это сделать. Тогда сетевое программное обеспечение сообщит об ошибке.
- *Сбои оборудования.* Аппаратное обеспечение не всегда работает так, как вы ожидаете. Принтер может оказаться выключенным, а Web-страница — временно недоступной. Оборудование часто отказывает в процессе работы, например, принтер может отказать на середине страницы.
- *Физические ограничения.* Диск может оказаться переполненным, а оперативная память — исчерпанной.
- *Ошибки программирования.* Какой-либо метод может работать неправильно. Например, он может возвращать неверный результат или некорректно вызывать другие методы. Выход за пределы массива, попытка найти несуществующий элемент хеш-таблицы, извлечение элемента из пустого стека — все это примеры ошибок программирования.

Обычно метод сообщает об ошибке, возвращая специальный код, который анализируется вызывающим методом. Например, методы, считывающие информацию из файлов, по достижении конца файла обычно возвращают значение –1. Такой способ обработки ошибок часто оказывается эффективным. В других случаях в качестве признака ошибки возвращается значение `null`. В главе 10 мы рассмотрели пример, в котором метод `getParameter()` класса `Applet` возвращал `null`, если требуемый параметр отсутствовал.

К сожалению, возвращать код ошибки не всегда возможно. Иногда не удается различить правильные данные и признак ошибки. Метод, возвращающий целое число, не может, обнаружив ошибку, вернуть значение –1, поскольку оно может быть результатом вычислений.

Как мы упоминали в главе 5, в языке Java есть возможность предусмотреть в каждом методе альтернативный выход, которым следует воспользоваться, если задачу невозможно выполнить до конца. При этом метод не станет возвращать значение, а *сгенерирует* (throw) объект, инкапсулирующий информацию об ошибке. Обратите внимание на то, что выход из метода происходит незамедлительно, и он не возвращает своего нормального значения. Более того, возобновить выполнение кода, вызвавшего данный метод, невозможно. Вместо этого начинается поиск *обработчика исключения*, который может справиться с возникшей ошибкой.

Исключения имеют свой собственный синтаксис и являются частью особой иерархии наследования. Сначала мы рассмотрим вопросы, связанные с синтаксисом, а затем дадим несколько советов по эффективному использованию обработчиков исключений.

Классификация исключений

В языке Java объект исключения всегда является экземпляром класса, производного от `Throwable`. Как мы вскоре убедимся, если стандартных классов недостаточно, можно создавать и свои собственные классы исключений.

На рис. 11.1 упрощенно показана иерархия исключений в языке Java.

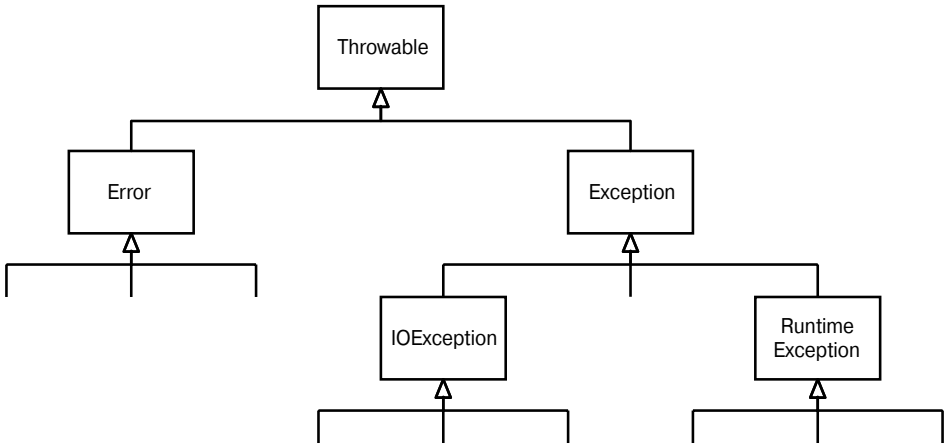


Рис. 11.1. Иерархия исключений в Java

Обратите внимание на то, что хотя все исключения имеют общего предка `Throwable`, иерархия сразу разделяется на две ветви: `Error` и `Exception`.

Иерархия `Error` описывает внутренние ошибки и ситуации, связанные с нехваткой ресурсов, в системе поддержки выполнения программ. Ни один объект этого типа самостоятельно сгенерировать невозможно. При возникновении внутренней ошибки возможности программиста крайне ограничены. Можно лишь уведомить пользователя и попытаться аккуратно прервать выполнение программы. Такие ситуации достаточно редки.

При программировании на Java основное внимание следует уделять иерархии `Exception`. Эта иерархия также разделяется на две ветви: исключения, производные от класса `RuntimeException`, и остальные. Исключения типа `RuntimeException` возникают вследствие ошибок программирования. Все другие исключения являются следствием непредвиденного стечения обстоятельств, например, ошибок ввода-вывода, возникающих при выполнении вполне корректных программ.

Исключения, производные от класса `RuntimeException`, связаны со следующими проблемами:

- неверное приведение типов;
- выход за пределы массива;
- попытка обратиться к объекту по ссылке `null`.

Остальные исключения возникают в следующих случаях:

- попытка чтения после конца файла;
- попытка установить соединение, используя неправильный URL;
- попытка получить объект `Class`, если в строке указан несуществующий класс.

Исключения RuntimeException практически всегда возникают по вине программиста. Исключения ArrayIndexOutBoundsException можно избежать, если всегда проверять индексы массива. Исключение NullPointerException никогда не возникнет, если перед использованием переменной вы проверите, не содержит ли она значение null.

А как насчет неверного URL-адреса? Всегда ли можно обнаружить ошибку заранее? Разные браузеры обрабатывают разные URL-адреса. Например, браузер Netscape может работать с URL-адресами вида mailto:, а программа appletviewer — нет. Следовательно, термин "неправильный" зависит от среды, а не от программы.

В спецификации языка Java любое исключение, производное от класса Error или RuntimeError, называется *неконтролируемым* (unchecked). Все остальные исключения называются *контролируемыми* (checked). Для всех контролируемых исключений компилятор проверяет наличие соответствующих обработчиков.



На заметку! Имя RuntimeException иногда вводит в заблуждение. Разумеется, все ошибки, которые мы обсуждаем, возникают в ходе выполнения программы.



На заметку! Если вы знакомы с иерархией исключений (намного более ограниченной) из стандартной библиотеки C++, то, возможно, заметили некоторое противоречие в именах. В C++ есть два основных класса, описывающих исключения: runtime_error и logic_error. Класс logic_error эквивалентен классу RuntimeException языка Java. Класс runtime_error является базовым для исключений, вызванных непредвиденным стечением обстоятельств. Он является эквивалентом всех исключений, предусмотренных в языке Java, не относящихся к типу RuntimeException.

Объявление контролируемых исключений

Метод может генерировать исключения, если возникают обстоятельства, с которыми он не справляется. Идея проста: метод не только сообщает компилятору, какие значения он может возвращать, но и предсказывает, *какие ошибки могут возникнуть*. Например, код, считывающий информацию из некоего файла, может предположить, что такого файла не существует или он пуст. Следовательно, код, предназначенный для чтения из файла, должен сообщить компилятору, что он может генерировать исключение IOException.

Объявление о том, что данный метод может генерировать исключение, помещается в его заголовок. Ниже в качестве примера приведено объявление одного из конструкторов класса FileInputStream из стандартной библиотеки. (Потоки рассматриваются в главе 12.)

```
public FileInputStream(String name) throws FileNotFoundException
```

Это объявление говорит о том, что данный конструктор создает объект FileInputStream на основе параметра String, но также может в определенных случаях, когда что-то идет не так, сгенерировать исключение FileNotFoundException. Если это произойдет, исполняющая система начнет поиск обработчика, предназначенного для работы с объектами класса FileNotFoundException.

Создавая свой собственный метод, не нужно объявлять все возможные исключения, который ваш метод фактически может генерировать. Чтобы понять, когда (и что) следует описывать с помощью выражения throws, имейте в виду, что исключение генерируется в следующих четырех случаях.

- Вызывается метод, генерирующий контролируемое исключение, например конструктор `FileInputStream`.
- Обнаружена ошибка, и с помощью оператора `throw` явным образом создается контролируемое исключение (оператор `throw` обсуждается в следующем разделе).
- Обнаружена ошибка программирования, например, в программе присутствует выражение `a[-1] = 0`, вследствие чего возникает неконтролируемое исключение, например, `ArrayIndexOutOfBoundsException`.
- Возникла внутренняя ошибка в виртуальной машине или библиотеке времени выполнения.

В первых двух случаях нужно сообщить программистам, которые будут использовать ваш метод, что возможно исключение. Зачем? Любой метод, генерирующий исключение, представляет собой смертельную угрозу. Если в программе не предусмотреть обработку исключения, то выполнение текущего потока прерывается.

Объявить, что метод может генерировать исключение, можно, включая в его заголовок выражение спецификацию исключения.

```
class MyAnimation
{
   ...
   public Image loadImage(String s) throws IOException
   {
      ...
   }
}
```

Если метод может генерировать несколько контролируемых исключений, всех их нужно перечислить в его заголовке, разделяя запятыми.

```
class MyAnimation
{
   ...
   public Image loadImage(String s)
      throws EOFException, MalformedURLException
   {
      ...
   }
}
```

Однако внутренние ошибки, т.е. исключения, производные от класса `Error`, объявлять не нужно. Эти ситуации могут генерироваться любыми методами, причем повлиять на них невозможно.

Точно так же не нужно объявлять неконтролируемые исключения, производные от класса `RuntimeException`.

```
class MyAnimation
{
   ...
   void drawImage(int i)
      throws ArrayIndexOutOfBoundsException // Плохой стиль!
   {
      ...
   }
}
```

Эти ошибки лежат полностью на вашей совести. Если вас так беспокоит возможность выхода индекса массива за пределы диапазона, лучше уделите больше внимания ее предотвращению, а не заявляйте, что она может возникнуть.

Отметим также, что метод должен объявлять все *контролируемые* исключения, которые он может генерировать. Неконтролируемые исключения либо находятся вне вашей компетенции (класс `Error`), либо являются следствием логических ошибок, которые не следовало допускать (класс `RuntimeException`). Если ваш метод не сообщает обо всех контролируемых исключениях, компилятор выдаст сообщение об ошибке.

Разумеется, как мы уже видели, вместо объявления исключений их можно перехватывать. В этом случае исключение не генерируется и спецификация `throws` не нужна. Позднее в этой главе мы рассмотрим вопрос, что лучше: перехватывать исключение самому или поручить это кому-нибудь другому.



Внимание! Метод подкласса не может генерировать более общие исключения, чем переопределяемый им метод суперкласса. (Исключения могут только конкретизироваться или не возникать вовсе.) В частности, если метод суперкласса вообще не генерирует контролируемые исключения, то и подкласс этого сделать не может. Например, если вы переопределяете метод `JComponent.paintComponent()`, ваш метод `paintComponent()` не должен генерировать ни одного контролируемого исключения, поскольку соответствующий метод суперкласса этого не делает.

Если метод объявляет, что может генерировать исключение определенного класса, то он может генерировать также исключения его подклассов. Например, конструктор `FileInputStream` может объявить о возможности генерации исключения `IOException`, причем неизвестно, какого именно. Это может быть ситуация как самого класса `IOException`, так и его потомков, например, класса `FileNotFoundException`.



На заметку! Спецификатор `throws` совпадает со спецификатором `throws` языка C++. Однако существует одно важное отличие. В C++ спецификатор `throws` действует только во время выполнения, а не на этапе компиляции. Это значит, что компилятор C++ игнорирует объявление исключений. Однако если исключение генерируется в функции, не являющейся частью списка `throws`, то вызывается функция `unexpected` и по умолчанию выполнение программы прекращается. Кроме того, в C++, если спецификатор `throws` не указан, функция может генерировать любое исключение.

Как сгенерировать исключение

Предположим, что с вашей программой случилось нечто ужасное. У вас есть метод `readData()`, считывающий данные из файла, в заголовке которого указано следующее:

```
Content-length: 1024
```

Однако после ввода 733 символов файл окончился. Вы считаете эту ситуацию настолько ненормальной, что решаете генерировать исключение.

Теперь необходимо решить, какого типа должно быть исключение. Нам подходит одна из разновидностей класса `IOException`. Просматривая документацию, вы обнаруживаете описание исключения `EOFException`: “Сигнализирует о том, что во время ввода данных неожиданно обнаружен признак конца файла `EOF`”. Отлично. Это исключение можно создать двумя способами:

```
throw new EOFException();
```

или, если вам так больше нравится:

```
EOFException e = new EOFException();
throw e;
```

Итак, для решения задачи нужно написать следующий фрагмент кода:

```
String readData(Scanner in) throws EOFException
{
   . . .
   while (. . .)
   {
      if (!in.hasNext()) // Обнаружен EOF.
      {
         if (n < len)
            throw new EOFException();
      }
      . . .
   }
   return s;
}
```

В классе `EOFException` есть второй конструктор, получающий в качестве параметра строку. Его можно использовать для более подробного описания исключения.

```
String gripe = "Content-length: " + len ", Received: + n;
throw new EOFException(gripe);
```

Задача облегчается, если один из существующих классов исключений подходит для вашей цели. В этом случае нужно сделать следующее.

1. Найти подходящий класс.
2. Создать экземпляр этого класса.
3. Генерировать исключение.

После генерации исключения управление уже не будет возвращено вызывающему методу. Это означает, что беспокоиться о возвращении значения, предусмотренного по умолчанию, или кода ошибки не нужно.



На заметку! Генерация исключений в языках C++ и Java происходит почти одинаково; есть лишь одно небольшое различие. В Java можно генерировать только объекты классов, производных от класса `Throwable`, а в C++ можно генерировать объекты любого типа.

Создание классов исключений

В программе может возникнуть ситуация, не предусмотренная ни одним из существующих классов. В этом случае можно достаточно просто создать свой собственный класс исключения. Очевидно, что он должен быть подклассом класса `Exception` или одного из его дочерних классов, например, `IOException`. Этот класс можно снабдить конструктором по умолчанию и конструктором, содержащим подробное сообщение. (Это сообщение может оказаться полезным при отладке. При его выводе используется метод `toString()` класса `Throwable`.)

```
class FileFormatException extends IOException
{
   public FileFormatException() {}
```

```
   public FileFormatException(String gripe) {}
   {
      super(gripe);
   }
}
```

Теперь мы готовы генерировать исключение своего собственного типа.

```
String readData(BufferReader in) throws FileFormatException
{
   ...
   while (...)
   {
      if (ch == -1) // Обнаружен маркер конца файла EOF.
      {
         if ( n < len)
            throw new FileFormatException();
      }
      ...
   }
   return s;
}
```

API **java.lang.Throwable 1.0**

- `Throwable()`

  Создает новый объект класса `Throwable`, не сопровождая его подробным сообщением.

- `Throwable(String message)`

  Создает новый объект класса `Throwable`, сопровождая его подробным сообщением. По соглашению все классы исключений должны содержать два конструктора — конструктор по умолчанию и конструктор с детальным сообщением.

- `String getMessage()`

  Получает подробное сообщение, предусмотренное в объекте `Throwable`.

Перехват исключений

Теперь мы знаем, как генерировать исключения. Ничего сложного в этом нет: сгенерировав исключение, о нем можно попросту забыть. Разумеется, должен существовать код, позволяющий обрабатывать его.

Если исключение возникает и нигде не перехватывается, то программа прекращает работу, выводя на консоль сообщение о типе исключения и содержимое стека. Графические программы (аплеты и приложения) выводят то же сообщение, возвращаясь затем в цикл обработки пользовательского интерфейса. (Отлаживая графические программы, лучше держать консольное окно на экране, не минимизируя его.)

Перехват исключения осуществляется с помощью блока `try/catch`. В простейшем случае этот блок имеет следующий вид:

```
try
{
   Код.
   Еще код.
   Еще код.
}
```

```
catch (тип_исключения e)
{
   Обработчик исключения данного типа.
}
```

Если фрагмент кода, содержащийся в блоке `try`, генерирует исключение типа, указанного в заголовке блока `catch`, то происходит следующее.

1. Программа пропускает оставшуюся часть кода в блоке `try`.
2. Программа выполняет код обработчика, указанный в блоке `catch`.

Если код в блоке `try` не генерирует исключение, то программа игнорирует блок `catch`.

Если какой-либо из операторов блока `try` генерирует исключение, отличное от типа, указанного в `catch`, выполнение метода немедленно прекращается. (Будем надеяться, что в вызывающем методе перехват исключения все же предусмотрен.)

Чтобы продемонстрировать, как это работает, рассмотрим совершенно типичный код чтения данных:

```
public void read(String filename)
{
   try
   {
      InputStream in = new FileInputStream(filename);
      int b;
      while ((b = in.read()) !=-1)
      {
         обработка ввода
      }
   }
   catch (IOException exception)
   {
      exception.printStackTrace();
   }
}
```

Обратите внимание на то, что бо́льшая часть кода в блоке `try` выполняет простейшие действия: считывает и обрабатывает строки, пока не обнаружится конец файла. Как указано в документации по API, существует возможность того, что метод `read()` сгенерирует исключение `IOException`. В данном случае мы пропускаем весь цикл `while`, переходим к разделу `catch` и генерируем трассировку стека. Для небольших программ это вполне разумно. А какие еще существуют возможности?

Зачастую лучше вовсе ничего не делать, а просто отправить исключение вызвавшему коду. Если в методе `read()` возникла ошибка, возложите ответственность за обработку этой ситуации на вызывающий метод! Придерживаясь этого подхода, достаточно указать, что метод `read()` может генерировать исключение `IOException`.

```
public void read(String filename) throws IOException
{
   InputStream in = new FileInputStream(filename);
   int b;
   while ((b = in.read()) !=-1)
   {
      обработка ввода
   }
}
```

Запомните, компилятор строго следит за спецификаторами throws. Вызывая метод, генерирующий контролируемое исключение, нужно либо самому обрабатывать его, либо делегировать обработку другому методу.

Что лучше? Как правило, следует перехватывать лишь те исключения, которые вы сами можете обработать, а остальные передавать дальше. Передавая исключение, нужно добавлять спецификатор throws, чтобы предупредить об этом вызывающий метод.

Просмотрите документацию по API и обратите внимание на методы, которые могут генерировать исключения. Затем примите решение, обрабатывать исключение или включить его в список throws. Во втором варианте нет ничего постыдного — лучше предоставить обработку исключения более компетентному обработчику.

Как мы упоминали ранее, у этого правила есть одно исключение. Если вы создаете метод, который переопределяет метод суперкласса, не генерирующий ни одного исключения (например, метод paintComponent класса JComponent), вы *обязаны* сами перехватывать каждое контролируемое исключение в теле метода. В этот метод подкласса нельзя добавить спецификатор throws, отсутствующий в методе суперкласса.



На заметку! Перехват исключений в Java и C++ осуществляется почти одинаково. Оператор

```
catch (Exception e) // Java
```

является аналогом оператора

```
catch (Exception& e) // C++
```

Перехват нескольких исключений

С помощью блока try можно перехватить несколько исключений, обработав их по отдельности. Для каждого типа исключения следует предусмотреть свой блок catch.

```
try
{
   Код, который может генерировать исключения.
}
catch (MalformedURLException e1)
{
   // Действия в случае некорректного формата URL.
}
catch (UnknownHostException e2)
{
   // Действия в случае, если URL не существует.
}

catch (IOException e3)
{
   // Действия при возникновении ошибок ввода-вывода.
}
```

Объект исключения (e1, e2 или e3) содержит информацию о нем. Для того чтобы узнать сведения о сообщении, нужно вызвать следующий метод:

```
e3.getMessage()
```

Фактический тип объекта исключения можно определить с помощью метода

```
e3.getClass().getName()
```

Повторная генерация исключения

Исключение можно генерировать в блоке `catch`, создавая тем самым цепочку исключений. Обычно разработчики поступают так в том случае, если им надо изменить тип исключения. Если вы создаете подсистему, используемую другими программистами, имеет смысл генерировать такие исключения, которые дали бы возможность сразу определить, что ошибка возникла именно в ней. В качестве примера подобного исключения можно привести `ServletException`. Возможно, что контейнеру сервлетов и не обязательно иметь информацию о том, какая именно возникла ошибка, а важно лишь знать, что некорректно работает сервлет.

Ниже показан пример перехвата исключения и повторной его генерации.

```
try
{
   Обращение к базе данных
}
catch (SQLException e)
{
   throw new ServletException("database error: " +
      e.getMessage());
}
```

В данном случае в конструкторе `ServletException` формируется сообщение. В Java SE 1.4 вы можете даже указать исходное исключение, сообщив, что оно стало причиной нового.

```
try
{
   access the database
}
catch (SQLException e)
{
   Throwable se = new ServletException("database error");
   se.setCause(e);
   throw se;
}
```

Теперь при перехвате исключения можно извлечь исходное исключение так, как показано ниже.

```
Throwable e = se.getCause();
```

Специалисты настоятельно рекомендуют использовать при разработке именно такой подход. Он позволяет генерировать исключения более высокого уровня, не теряя деталей исходного исключения.



Совет. Описанный подход удобен в том случае, если перехват исключения осуществляется в методе, который не имеет права генерировать контролируемые исключения. Вы можете перехватить контролируемое исключение и инкапсулировать его в составе исключения времени выполнения.



На заметку! В ряде классов, описывающих исключения, например, в `ClassNotFoundException`, `InvocationTargetException` и `RuntimeException`, поддерживается своя схема создания цепочки исключений. Начиная с версии Java SE 1.4, в них параллельно поддерживается стандартный механизм инкапсуляции исходного исключения. Таким образом, вы можете либо применять методы, типичные для этих классов, либо использовать метод `getCause()`.

Блок finally

Когда код генерирует исключение, операторы, оставшиеся до конца метода, не выполняются. Если метод задействовал какие-то локальные ресурсы, о которых известно лишь ему, то освободить их окажется невозможным. Можно, конечно, повторно генерировать все исключения, однако это решение не слишком удачно, поскольку ресурсы нужно освобождать в двух местах: в нормальном коде и в коде исключения.

В языке Java есть лучшее решение — воспользоваться блоком `finally`. Ниже приведен пример корректного вызова метода `dispose()` объекта `Graphics`. Если вы работаете с базами данных, то можете прмиенять тот же подход для того, чтобы правильно закрыть соединение с базой. Как вы увидите во втором томе, закрывать соединение с базой необходимо даже в том случае, если при выполнении программы возникло исключение.

Код в блоке `finally` выполняется независимо от того, возникло исключение или нет. В частности, в приведенном ниже примере метод `dispose()` выполняется *при любых условиях*.

```
Graphics g = image.getGraphics();
try
{
   // 1
   Код, который может генерировать исключения.
   // 2
}
catch (IOException e)
{
   // 3
   Выводим диалоговое окно.
   // 4
}
finally
{
   // 5
   g.dispose();
}
// 6
```

Рассмотрим три возможных ситуации, в которых программа выполняет раздел `finally`.

1. Код не генерирует никаких исключений. В этом случае программа сначала полностью выполняет блок `try`, а затем блок `finally`. Иными словами, при работе программы последовательно выполняются этапы 1, 2, 5 и 6.
2. Код генерирует исключение, которое перехватывается в разделе `catch` (в данном примере это исключение класса `IOException`). В этом случае программа сначала выполняет блок `try` до той точки, в которой возникает исключение. Остальная часть блока `try` пропускается. Затем программа выполняет код из соответствующего блока `catch` и в заключение — блок `finally`.

   Если в разделе `catch` исключения не генерируются, то программа выполняет первую строку вслед за блоком `try`. Таким образом, при описанной ситуации выполняются этапы 1, 3, 4, 5 и 6.

   Если исключение генерируется в блоке `catch`, то управление передается вызывающему методу, и выполняются лишь этапы 1, 3 и 5.

3. Код генерирует исключение, которое не обрабатывается `catch`. В этом случае программа выполняет блок `try` вплоть до той точки, в которой генерируется исключение. Оставшаяся часть блока `try` игнорируется. Затем программа выполняет код из блока `finally` и передает исключение обратно в вызывающий метод.

Блок `finally` можно использовать без блока `catch`. Рассмотрим следующий пример:

```
InputStream in = ...;
try
{
   Код, который может генерировать исключения.
}
finally
{
   in.close();
}
```

Оператор `in.close()` в блоке `finally` выполняется независимо от того, возникает ли какое-либо исключение в блоке `try`. Разумеется, если исключение возникает, она будет перехвачено другим разделом `catch`.

Фактически, как объясняется ниже, мы считаем очень хорошей идеей использовать конструкцию `finally` всякий раз, когда необходимо закрыть ресурс.



Совет. Мы настоятельно советуем разделять блоки `try/catch` и `try/finally`. При этом код программы становится более понятным. Рассмотрим следующий пример:

```
InputStream in = ...;
try
{
    try
    {
       Код, который может генерировать исключение.
    }
    finally
    {
        in.close();
    }
}
catch (IOException e)
{
  Отображение сообщения об ошибке.
}
```

Здесь внутренний блок `try` отвечает только за закрытие входного потока. Внешний блок `try` сообщает об ошибках. Такой код не только более понятен, но и более функционален, так как ошибки в блоке `finally` также распознаются.



Внимание! Если в блоке `finally` присутствует оператор `return`, результаты его выполнения могут быть неожиданными. Предположим, что в середине блока `try` вы вышли из метода с помощью оператора `return`. Перед тем как передать управление вызывающему методу, должен быть выполнен блок `finally`. Если он тоже содержит `return`, то возвращаемое значение будет замаскировано. Рассмотрим пример:

```
public static int f(int n)
{
    try
    {
        int r = n * n;
        return r;
    }
    finally
    {
        if (n == 2) return 0;
    }
}
```

Если выполнить вызов `f(2)`, то блок `try` вычислит значение `r = 4` и выполнит оператор `return`. Однако на самом деле до оператора `return` будет выполнен код раздела `finally`. Этот раздел заставит метод вернуть значение 0, игнорируя исходное значение, равное 4.

Иногда раздел `finally` повергает программистов в уныние, в частности, если методы очистки памяти могут генерировать исключения. Типичный случай — закрытие потока (подробнее об этом — во втором томе). Предположим, что при возникновении исключения мы хотим закрыть поток.

```
InputStream in;
try
{
  Код, который может генерировать исключения.
}
catch (IOException e)
{
  Отображение диалогового окна.
}
finally
{
  in.close();
}
```

Предположим также, что код в блоке `try` генерирует некоторое исключение, *отличающееся от класса* `IOException`. При этом выполняется блок `finally` и вызывается метод `close()`. Данный метод сам может генерировать исключение класса `IOException`! При этом исходное исключение будет потеряно и вместо нее будет возвращен объект класса `IOException`. Это совершенно противоречит духу обработки исключений.

Это всегда хорошая идея (к сожалению, ни один из проектировщиков класса `InputStream` ею не воспользовался) — генерировать исключение в таких операциях очистки, как `dispose()`, `close()` и т.п., вызова которых можно ожидать в блоках `finally`.



На заметку! Есть одно фундаментальное отличие между C++ и Java в отношении обработки исключений. В Java нет деструкторов; поэтому нет и раскручивания стека, как в C++. Это значит, что программист Java должен вручную поместить код, освобождающий ресурсы, в блок `finally`. Разумеется, поскольку Java выполняет сборку мусора, ресурсов, требующих ручного освобождения, намного меньше.

Анализ элементов трассировки стека

Трассировка стека — это список вызовов методов для данной точки программы. Вы наверняка неоднократно видели эту информацию, она отображается каждый раз, когда при выполнении Java-программы возникает неконтролируемое исключение.

В версиях, предшествующих Java SE 1.4, для получения текстового описания трассировки стека применялся метод `printStackTrace()` класса `Throwable`. Теперь разработчикам доступен метод `getStackTrace()`; он возвращает массив объектов `StackTraceElement`, которые можно анализировать в программе. Например:

```
Throwable t = new Throwable();
StackTraceElement[] frames = t.getStackTrace();
for (StackTraceElement frame : frames)
  // анализ frame
```

Класс `StackTraceElement` содержит методы, позволяющие получить имя файла, номер строки кода, а также имя класса и метода. Форматированную строку, содержащую эту информацию, предоставляет метод `toString()`.

В Java SE 5.0 был добавлен статический метод `Thread.getAllStackTraces()`, который позволяет получить трассировку стека для всех потоков. Следующий фрагмент кода демонстрирует его применение:

```
Map<Thread, StackTraceElement[]> map = Thread.getAllStackTraces();
for (Thread t : map.keySet())
{
   StackTraceElement[] frames = map.get(t);
   analyze frames
}
```

Более подробные сведения о потоках и интерфейсе `Map` приведены во втором томе.

В листинге 11.1 показан код программы, которая выводит трассировку стека для рекурсивной функции вычисления факториала. Например, при вычислении `factorial(3)` вы получите следующие данные.

```
factorial(3):
StackTraceTest.factorial(StackTraceTest.java:8)
StackTraceTest.main(StackTraceTest.java:23)
factorial(2):
StackTraceTest.factorial(StackTraceTest.java:8)
StackTraceTest.factorial(StackTraceTest.java:14)
StackTraceTest.main(StackTraceTest.java:23)
factorial(1):
StackTraceTest.factorial(StackTraceTest.java:8)
StackTraceTest.factorial(StackTraceTest.java:14)
StackTraceTest.factorial(StackTraceTest.java:14)
StackTraceTest.main(StackTraceTest.java:23)
return 1
return 2
return 6
```

Листинг 11.1. Содержимое файла `StackTraceTest.java`

```
1.  import java.util.*;
2.
3.  /**
4.   * Программа, отображающая результаты
5.   * трассировки стека для рекурсивного метода.
```

```
6.   * @version 1.01 2004-05-10
7.   * @author Cay Horstmann
8.   */
9.  public class StackTraceTest
10. {
11.    /**
12.     * Вычисление факториала.
13.     * @param n Неотрицательное число
14.     * @return n! = 1 * 2 * . . . * n
15.    */
16.    public static int factorial(int n)
17.    {
18.       System.out.println("factorial(" + n + "):");
19.       Throwable t = new Throwable();
20.       StackTraceElement[] frames = t.getStackTrace();
21.       for (StackTraceElement f : frames)
22.          System.out.println(f);
23.       int r;
24.       if (n <= 1) r = 1;
25.       else r = n * factorial(n - 1);
26.       System.out.println("return " + r);
27.       return r;
28.    }
29.
30.    public static void main(String[] args)
31.    {
32.       Scanner in = new Scanner(System.in);
33.       System.out.print("Enter n: ");
34.       int n = in.nextInt();
35.       factorial(n);
36.    }
37. }
```

API java.lang.Throwable 1.0

- `Throwable(Throwable cause)` **1.4**
- `Throwable(String message, Throwable cause)` **1.4**

  Создает объект класса `Throwable`, соответствующий объекту `cause`, который описывает причины возникновения исключения.

- `Throwable initCause(Throwable cause)` **1.4**

  Указывает причину создания объекта. Если причина создания объекта уже задана, генерирует исключение. Возвращает `this`.

- `Throwable getCause()` **1.4**

  Возвращает объект исключения, который был определен как причина создания данного объекта. Если причина создания объекта не была задана, возвращается значение `null`.

- `StackTraceElement[] getStackTrace()` **1.4**

  Возвращает трассировку стека в момент создания объекта.

API java.lang.Exception 1.0

- `Exception(Throwable cause)` **1.4**
- `Exception(String message, Throwable cause)`

  Создает объект класса `Exception`, соответствующий объекту `cause`.

API java.lang.RuntimeException 1.0

- `RuntimeException(Throwable cause)` **1.4**
- `RuntimeException(String message, Throwable cause)` **1.4**

  Создает объект класса `RuntimeException`, соответствующий объекту `cause`.

API java.lang.StackTraceElement 1.4

- `String getFileName()`

  Возвращает имя исходного файла, содержащего точку, в которой выполняется элемент. Если информация недоступна, возвращается значение `null`.

- `int getLineNumber()`

  Возвращает номер строки, содержащей точку, в которой выполняется элемент. Если информация недоступна, возвращается значение `-1`.

- `String getClassName()`

  Возвращает полное имя класса, содержащего точку, в которой выполняется элемент.

- `String getMethodName()`

  Возвращает имя метода, содержащего точку, в которой выполняется элемент. Имя конструктора — `<init>`. Имя статического инициализатора — `<clinit>`. Перегруженные методы, имеющие одинаковые имена, не различаются.

- `boolean isNativeMethod()`

  Если точка выполнения элемента находится внутри платформенно-зависимого метода, возвращает значение `true`.

- `String toString()`

  Возвращает форматированную строку, содержащую имя класса и метода, а также имя файла и номер строки (если эта информация доступна).

Как использовать исключения

Разные специалисты придерживаются различных мнений относительно использования исключений. Некоторые программисты считают, что контролируемые исключения — это не более чем досадная помеха, другие готовы затратить дополнительное время и усилия на их обработку. На наш взгляд, исключения (особенно контролируемые) — полезный механизм, но, применяя его, не стоит слишком увлекаться. В этом разделе мы дадим ряд рекомендаций относительно использования исключений в программах.

1. *Обработка исключений не может заменить собой простую проверку.*

   Для иллюстрации мы написали фрагмент программы, использующей встроенный класс `Stack`. Программа, приведенная в листинге 11.3, делает 1 000 000 попыток извлечь элемент из пустого стека. Сначала она определяет, пуст ли стек:

   ```
   if (!s.empty)) s.pop();
   ```

   Затем мы заставляем ее извлекать элемент, не обращая внимания на то, что стек пуст. После этого мы перехватываем исключение `EmptyStackException`, которое информирует нас о том, что мы не правы.

   ```
   try()
   {
     s.pop();
   }
   catch (EmptyStackException e)
   {
   }
   ```

   Время, которое мы затратили на эти операции, указано в табл. 11.1.

Таблица 11.1. Время, затраченное на выполнение операций над стеком

Проверка	Генерация/перехват исключения
646 мс	21 739 мс

   Как видим, перехват исключения занял намного больше времени, чем простая проверка. Вывод: используйте исключения только в тех случаях, когда это оправдано.

2. *Не увеличивайте без надобности количество блоков `try`.*

   Многие программисты каждый оператор помещают в отдельный блок `try`.

   ```
   OutputStream out;
   Stack s;

   for (i = 0; i < 100; i++)
   {
     try
     {
       n = s.pop();
     }
     catch (EmptyStackException s)
     {
       // Стек оказался пустым.
     }
     try
     {
       out.writeInt(n);
     }
     catch (IOException e)
     {
       // Проблема при записи в файл.
     }
   }
   ```

При таком подходе размер кода существенно увеличивается. Подумайте о задаче, которую должна решать ваша программа. В данном случае мы хотим извлечь из стека 100 чисел и записать их в файл. (Неважно, зачем нам это надо, — это всего лишь пример.) Сама по себе генерация исключения не решает проблем. Если стек пуст, он не заполнится по мановению волшебной палочки. Если при записи в файл возникает ошибка, она не исчезнет сама собой. Следовательно, имеет смысл поместить в блок `try` *весь фрагмент* программы. Если хотя бы одна из операций даст сбой, можно отказаться от решения всей задачи.

```
try
{
   for (i = 0; i < 100; i++)
   {
      n = s.pop();
      out.writeInt(n);
   }
}
catch (IOException e)
{
   // Проблема при записи в файл.
}
catch (EmptyStackException s)
{
   // Стек оказался пустым.
}
```

Этот фрагмент выполняет одно из своих основных заданий — *отделяет* нормальную работу программы от обработки исключений.

3. *Пользуйтесь возможностями, которые предоставляет иерархия исключений.*

   Не ограничивайтесь генерацией `RuntimeException`. Найдите подходящий подкласс или создайте собственный. Не перехватывайте `Throwable`. При таком подходе ваш код становится трудным для восприятия и поддержки.

   Различайте контролируемые и неконтролируемые исключения. Для обработки контролируемых исключений требуются дополнительные усилия, поэтому не применяйте их для оповещения об ошибках логики. (В библиотеке поддержки рефлексии это правило не выполняется. Пользователи часто вынуждены создавать код для перехвата тех исключений, которые гарантированно не возникнут.)

   При необходимости смело преобразуйте один тип исключения в другой, более подходящий в данной ситуации. Например, если вы выполняете разбор целого числа, извлекаемого из файла, перехватывайте `NumberFormatException` и генерируйте вместо него подкласс `IOException` или `MySubsystemException`.

4. *Не подавляйте исключения.*

   При написании программы на языке Java существует большой соблазн подавить исключения. Допустим, вы пишете метод, вызывающий другой метод, который может сгенерировать исключение один раз в сто лет. Компилятор пожалуется на это, поскольку вы не объявили это исключение в списке `throws` вашего метода. Предположим, вам не хочется указывать ее в списке `throws`, так как компилятор начнет выдавать сообщения об ошибке для всех методов, вызывающих ваш метод. В этом случае исключение можно просто подавить.

```
public Image loadImage(String s)
{
   try
   {
       Код, способный сгенерировать исключение.
   }
   catch (Exception e)
   {} // Действия не выполняются!
}
```

Теперь ваш код будет скомпилирован. Он будет прекрасно работать, но не в исключительной ситуации. Если она возникнет, то будет проигнорирована. Если вы считаете, что исключение достаточно важно, приложите усилия для его обработки.

5. *Генерируя исключения, не бойтесь показаться излишне жестким.*

   Некоторые программисты избегают генерировать исключения, информирующие об ошибках. Например, если методу переданы некорректные параметры, они предпочитают возвращать фиктивное значение. Например, в случае, когда стек пуст, программист может предусмотреть возврат методом `Stack.pop()` значения `null`, вместо того, чтобы сгенерировать исключение. На наш взгляд, лучше исключение `EmptyStackException` в той точке, где возникла ошибка, чем исключение `NullPointerException` впоследствии.

6. *Не бойтесь передавать исключения другим обработчикам.*

   Некоторые программисты считают себя обязанными перехватывать все исключения. Вызывая метод, генерирующий некоторое исключение, например конструктор класса `FileInputStream` или метод `readLine`, они инстинктивно перехватывают исключения, которые могут быть сгенерированы ими. Зачастую оказывается предпочтительнее передать исключение другому обработчику, а не обрабатывать его самому.

   ```
   public void readStuff(String filename)
     throws IOException // Вполне допустимое решение
   {
     InputStream in = new FileInputStream(filename);
     . . .
   }
   ```

   Методы более высокого уровня лучше информируют пользователя об ошибке или отказываются выполнять неправильные операторы.



На заметку! Правила 5 и 6 можно подытожить как “генерировать раньше, перехватывать позже”.

Использование диагностических утверждений

Диагностические утверждения (assertion) — это широко распространенные идиомы, применяемые для обеспечения надежности программирования. Допустим, вы убеждены в том, что конкретное свойство уже задано, и ссылаетесь на него в своей программе. Предположим, что вы вычисляете следующее выражение:

```
double y = Math.sqrt(x);
```

Вы уверены, что число x — неотрицательное. Возможно, оно получено в результате других вычислений, которые не могут порождать отрицательные числа, либо является параметром метода, которому можно передавать лишь положительные числа. Но вы все же хотите еще раз проверить свои предположения, чтобы в ходе выполнения программы не возникло ошибки. Разумеется, можно было бы просто сгенерировать исключение:

```
if (x < 0) throw new IllegalArgumentException("x < 0 ");
```

Однако у такого подхода есть досадный недостаток — код остается в программе, даже если тестирование давно закончено. Если в тексте программы расставлено много таких проверок, ее выполнение может замедлиться.

Механизм диагностических утверждений позволяет вставлять в текст программы проверки, предназначенные для ее тестирования, а затем удалять их из окончательного варианта.

Начиная с версии Java SE 1.4, в языке появилось новое ключевое слово — assert. Оно имеет две формы:

```
assert условие;
```

и

```
assert условие : выражение;
```

Оба оператора вычисляют некое условие и генерируют исключение AssertionError, если оно не выполняется. Во втором операторе выражение передается конструктору объекта класса AssertionError и превращается в строку сообщения.



На заметку! Единственная цель *выражения* при указании ключевого слова assert — создать строку сообщения. Объект класса AssertionError не хранит фактическое значение выражения, поэтому его невозможно использовать в дальнейшем. В документации, поставляемой совместно с JDK , непринужденно говорится, что это сделано для того, чтобы "утешить программистов, если диагностическое утверждение окажется неверным".

Чтобы проверить, является ли значение x неотрицательным числом, можно использовать такой оператор:

```
assert x >= 0;
```

Можно также передать фактическое значение переменной x объекту AssertionError, чтобы он впоследствии вывел его на экран:

```
assert x >= 0 : x;
```



На заметку! В языке C макрос assert превращает проверяемое условие в строку, которая выводится на экран, если оно не выполняется. Например, если выражение assert(x>=0) имеет значение false, на экран выводится строка, сообщающая, что условие x >= 0 не выполняется. В языке Java условие не является неотъемлемой частью сообщения об ошибке. Если вы хотите увидеть его, передайте его в виде строки объекту класса AssertionError: assert x >= 0 : "x >= 0".

Включение и отключение диагностических утверждений

По умолчанию диагностические утверждения считаются заблокированными. Разблокировать их в ходе выполнения программы можно с помощью опции -enableassertions или -ea:

```
java -enableassertions MyApp
```

Обратите внимание на то, что для подключения и отключения диагностических утверждений не нужно заново компилировать программу. Это задача *загрузчика классов*. Если диагностические утверждения заблокированы, загрузчик классов игнорирует их код, чтобы не замедлять выполнение программы.

Подключать диагностические утверждения можно в отдельных классах или в целых пакетах:

```
java -ea:MyClass -ea:com.mycompany.mylib ... MyApp
```

Эта команда подключает диагностические утверждения к классу `MyClass`, а также ко всем классам пакета `com.mycompany.mylib` и *его подпакетов*. Опция `-ea...` подключает диагностические утверждения ко всем классам пакета по умолчанию.

Кроме того, можно отключить диагностические утверждения в определенных классах и пакетах. Для этого достаточно задать опцию `-diableassertions` или `-da`:

```
java -ea:... -da:MyClass MyApp
```

Некоторые классы загружаются не загрузчиком классов, а непосредственно виртуальной машиной. Для выборочного включения или отключения диагностических утверждений, содержащихся в этих классах, также можно использовать опции `-ea` и `-da`. Однако эти опции, включающие и отключающие все диагностические утверждения одновременно, не могут применяться к "системным классам" без загрузчика. Для включения диагностических утверждений, находящихся в системных классах, следует использовать опцию `-enablesystemassertions/-esa`.

Можно даже управлять состоянием диагностических утверждений с помощью программных средств. Подробности вы найдете в документации по API.

Использование диагностических утверждений для проверки параметров

В языке Java есть три механизма обработки системных сбоев:

- генерация исключений;
- регистрация ошибок;
- использование диагностических утверждений.

Когда следует применять диагностические утверждения? Чтобы правильно ответить на этот вопрос, учтите следующее.

- Диагностические утверждения оказываются ложными, если произошла неустранимая ошибка.
- Диагностические утверждения включаются только на период отладки и тестирования программы. (Иногда это шутливо формулируют так: "одевайте спасательный жилет, плавая у берега, — в открытом море он вам не понадобится".)

Итак, диагностические утверждения не следует применять в качестве индикаторов несущественных и легко исправимых ошибок. Они предназначены для локализации серьезных неполадок во время тестирования.

Представим себе следующий сценарий: проверку параметров метода. Следует ли применять диагностические утверждения для проверки выхода индекса за пределы допустимого диапазона или для распознавания нулевой ссылки? Чтобы ответить на этот вопрос, необходимо обратиться к документации, описывающей метод. Например, рассмотрим метод `Array.sort` из стандартной библиотеки.

```
/**
   Упорядочивает указанный отрезок заданного массива
   в порядке возрастания. Упорядочиваемый отрезок начинается
   с индекса fromIndex и заканчивается индексом, предшествующим
   toIndex, т.е. элемент с индексом toIndex
   упорядочению не подлежит.
   @param a Упорядочиваемый массив
   @param fromIndex Индекс первого элемента (включительно)
   упорядочиваемого отрезка
   @param toIndex Индекс первого из элементов, лежащего
   за упорядочиваемым отрезком
   @throws IllegalArgumentException, если fromindex > toIndex
   @throws ArrayIndexOutOfBoundsException, если fromIndex < 0
     или toIndex > a.length
*/
   static void sort[] a, int fromIndex, int toIndex
```

В документации утверждается, что этот метод генерирует исключение, если значение индекса задано некорректно. Это поведение метода является частью контракта, который он заключает с вызывающим методом. Если вы реализуете этот метод, то должны придерживаться контракта и генерировать соответствующее исключение. В этом случае диагностические утверждения не подходят.

Следует ли применять их для распознавания нулевой ссылки? Как и прежде, ответ отрицательный. В документации о методе ничего не сказано о его реакции, если параметр `a` равен `null`. Вызывающий модуль имеет основания предполагать, что в этом случае метод будет выполнен и не сгенерирует исключение `AssertionError`.

Предположим, однако, что контракт метода слегка изменен:

```
@param a Упорядочиваемый массив. (Не должен быть null.)
```

Теперь вызывающий метод знает, что передавать нулевую ссылку данному методу запрещено. Следовательно, метод может начинаться с диагностического утверждения

```
assert(a != null);
```

Специалисты называют подобный контракт *предусловием* (precondition). В начале метод не накладывал на массив никаких предварительных условий, так как предполагалось, что он будет корректно работать во всех случаях. Модифицированный метод имеет одно предусловие — параметр `a` не должен принимать значение `null`. Если вызывающий модуль нарушает указанное предусловие, то контракт оказывается нарушенным и метод отказывается выполнять работу. Фактически имея диагностическое утверждение, при неправильном вызове метод может повести себя непредсказуемо. Иногда он генерирует исключение `assertionError`, а иногда — `NullPointerexception`, в зависимости от конфигурации загрузчика классов.

Использование утверждений для документирования предположений

Многие программисты оформляют свои предположения о работе программы в виде комментариев. Рассмотрим пример из `http://java.sun.com/javase/6/docs/technotes/guides/language/assert.html`:

```
if (i % 3 == 0)
   ...
else if (i % 3 == 1)
   ...
```

```
else // (i % 3 == 2)
   ...
```

В данном случае намного разумнее использовать диагностические утверждения.

```
if (i % 3 == 0)
   ...
else if (i % 3 == 1)
   ...
else
{
   assert(i % 3 == 2)
   ...
```

Естественно, было бы еще более разумно тщательнее сформулировать проверки. Какие значения может принимать выражение `i % 3`? Если переменная `i` положительна, остаток может равняться 0, 1 или 2. Если отрицательна — остаток принимает значения –1 или –2. Таким образом, намного логичнее предположить просто, что переменная `i` не является отрицательным числом. Логично использовать диагностическое утверждение следующего вида:

```
assert(i >= 0);
```

Его следует разместить перед оператором `if`.

Итак, диагностические утверждения — это тактическое средство, предназначенное для тестирования и отладки, а регистрация — стратегический механизм, сопровождающий программу на протяжении всего ее существования.

API java.lang.ClassLoader 1.0

- `void setDefaultAssertionStatus(boolean b)` **1.4**

  Включает и отключает диагностические утверждения во всех классах, загруженных указанным загрузчиком классов, в которых статус диагностических утверждений не задан явно.

- `void setClassAssertionStatus(String className, boolean b)` **1.4**

  Включает и отключает диагностические утверждения в заданном классе и его подклассах.

- `void setPackageAssertionStatus(String packageName, boolean b)` **1.4**

  Включает и отключает диагностические утверждения во всех классах заданного пакета и его подпакетов.

- `void clearAssertionStatus()` **1.4**

  Удаляет все диагностические утверждения, статус которых задан явно, и блокирует все диагностические утверждения в классах, загруженных данным загрузчиком.

Протоколирование

Программисты часто прибегают к вызовам метода `System.out.println` для отладки кода и для того, чтобы понять поведение программы. Разумеется, после устранения ошибки следует убрать промежуточный вывод и поместить его в другое место для выявления новой неисправности. Для упрощения отладки предусмотрен API протоколирования.

Преимущества его использования перечислены ниже.

- Все протокольные записи легко заблокировать и разблокировать.
- Заблокированные протокольные записи занимают немного места и не влияют на эффективность приложения.
- Протокольные записи можно направить разным обработчикам, вывести на консоль, записать в файл и т.п.
- Регистраторы и обработчики могут фильтровать записи. Фильтры отбрасывают ненужные записи, используя критерии, предложенные программистом.
- Протокольные записи допускают форматирование. Например, их можно представить в виде простого текста либо в формате XML.
- Приложения могут использовать несколько протоколов, имеющих иерархические имена, подобно именам пакета, например, `com.mycompany.myapp`.
- По умолчанию параметры настройки протоколирования задаются в конфигурационном файле. Приложение может изменить этот способ задания параметров.

Базовое протоколирование

Начнем с наиболее простого случая. Система протоколирования управляет регистратором `Logger.global`, который используется вместо `System.out`. Вызовем метод `info` для регистрации информационного сообщения.

```
Logger.global.info("Выбран пункт меню File->Open");
```

По умолчанию выводится следующая запись:

```
May 10, 2004 10:12:15 PM LoggingImageViewer fileOpen
INFO: Выбран пункт меню File->Open
```

(Обратите внимание на то, что время, а также имена класса и метода включаются автоматически.) Однако, если в соответствующее место (например, в начале метода `main`) поместить приведенный ниже оператор, запись будет заблокирована.

```
Logger.global.setLevel(Level.OFF);
```

Расширенное протоколирование

Теперь, ознакомившись с примитивным протоколированием, перейдем к средствам протоколирования, применяемым при создании программ производственного качества. В профессиональном приложении, как правило, все записи не концентрируются в одном глобальном протоколе. Вместо этого можно определить свои собственные средства протоколирования.

Регистратор протокола (logger) создается, когда он вызывается по имени в первый раз:

```
Logger myLogger = Logger.getLogger("com.mycompany.myapp");
```

При последующих вызовах регистратора с тем же именем будет возвращаться тот же объект.

Как и имена пакетов, имена регистраторов образуют иерархию. Фактически они являются еще *более* иерархическими, чем имена пакетов. Если между пакетом и его предком нет семантической связи, то регистратор и его дочерние классы обладают общими свойствами. Например, если в регистраторе `com.mycompany` задать уровень протоколирования, то дочерний регистратор унаследует этот уровень.

Существует семь уровней протоколирования:

- SEVERE
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST

По умолчанию используются первые три уровня. Остальные уровни нужно задавать с помощью метода setLevel:

```
logger.setLevel(level.FINE);
```

Теперь будут регистрироваться все сообщения, начиная с уровня FINE и выше.

Кроме того, можно использовать константу Level.ALL для включения регистрации всех уровней и константу Level.OFF для отключения регистрации.

Для всех уровней определены методы протоколирования, например:

```
logger.warning(message);
logger.fine(message);
```

Помимо этого, можно воспользоваться методом log и явно указывать уровень:

```
logger.log(Level.FINE, message);
```



Совет. По умолчанию регистрируются все записи, имеющие уровень INFO и выше. Следовательно, для отладочных сообщений, необходимых для диагностики программы, но совершенно не интересующих пользователя, следует использовать уровни CONFIG, FINE, FINER и FINEST.



Внимание! Если уровень регистрации превышает значение INFO, необходимо изменить конфигурацию обработчика регистрации. По умолчанию обработчик регистрации блокирует сообщения, имеющие уровень ниже, чем INFO.

Запись протокола, созданная по умолчанию, состоит из имени класса и метода, содержащего вызов регистратора. Однако если виртуальная машина оптимизирует процесс выполнений, точная информация о вызываемых методах может стать недоступной. Для уточнения вызывающих класса и метода следует применять метод logp():

```
void logp(Level l, String className, String methodName, String message)
```

Для трассировки выполнения программы существуют удобные методы:

```
void entering(String className, String methodName)
void entering(String className, String methodName, Object param)
void entering(String className, String methodName, Object[] params)
void exiting(String className, String methodName)
void exiting(String className, String methodName, Object result)
```

Ниже приведен пример использования этих методов.

```
int read(String file, String pattern)
{
  entering("com.mycompany.nylib.Reader", "read",
    new Object[] { file, pattern });
  ...
```

```
   exiting("com.mycompany.mylib.Reader", "read", count);
   return count;
}
```

Данные вызовы генерируют записи регистрации, имеющие уровень FINER и начинающиеся строками ENTRY и RETURN.



На заметку! В будущем методы протоколирования будут поддерживать переменное количество параметров. Тогда вы сможете использовать вызовы наподобие следующего:

```
logger.entering("com.mycompany.mylib.Reader", "read", file, pattern)
```

Обычно протоколирование используется для записи неожиданных исключений. Есть два удобных метода, позволяющих вставить имя исключения в протокольную запись:

```
void throwing(String className, String methodName, Throwable t)
void log(Level l, String message, Throwable t)
```

Эти методы, как правило, используются следующим образом:

```
if (. . .)
{
   IOException exception = new IOException(". . .");
   logger.throwing("com.mycompany.mylib.Reader",
      "read", exception);
   throw exception;
}
```

и

```
try
{
   . . .
}
catch (IOException e)
{
   Logger.getLogger("com.mycompany.myapp").
      log(Level.WARNING, "Reading image", e);
}
```

Вызов метода throwing регистрирует запись, имеющую уровень FINER, и сообщение, которое начинается с THROW.

Настройка диспетчера протоколирования

Свойства системы регистрации можно изменять, редактируя конфигурационный файл. По умолчанию этот файл находится по следующему адресу:

каталог_исполняющей_системы/lib/lggin.properties

Если вы хотите использовать другой файл, нужно при запуске приложения установить свойство java.util.logging.config.file:

```
java -Djava.util.logging.config.file=конфигурационный_файл класс
```



Внимание! Вызов метода System.setProperty("java.util.logging.config.file", file) из метода main() ничего не дает, поскольку диспетчер регистрации инициализируется при запуске виртуальной машины, еще до начала выполнения метода main().

Чтобы изменить уровень протоколирования, принятый по умолчанию, нужно отредактировать конфигурационный файл, изменив строку

```
.level=INFO
```

В собственных регистраторах уровни протоколирования задаются с помощью строк наподобие следующей:

```
com.mycompany.myapp.level=FINE
```

Иначе говоря, нужно добавить к имени регистратора суффикс `.level`.

Как мы увидим в дальнейшем, регистраторы на самом деле не посылают сообщения на консоль — это забота обработчиков. Для обработчиков также определены уровни. Чтобы увидеть на консоли сообщения, имеющие уровень `FINE`, необходимо выполнить следующую установку:

```
java.util.logging.ConsoleHandler.level=FINE
```



Внимание! Параметры настройки диспетчера протоколирования *не* являются системными свойствами. Запуск программы с опциями `-Dcom.mycompany.myapp.level=FINE` никак не отражается на регистраторе.



Внимание! По крайней мере, до версии Java SE 6, документация API для класса `LogManager` требовала установки свойств `java.util.logging.config.class` и `java.util.logging.config.file` через Preferences API. На самом деле это не так — см. `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4691587`.



На заметку! Файл, содержащий свойства системы протоколирования, обрабатывается классом `java.util.logging.LogManager`. Задав имя подкласса в качестве системного свойства `java.util.logging.manager`, можно указать другой диспетчер протоколирования. Кроме того, можно оставить стандартный диспетчер протоколирования, пропустив инициализационные установки в файле свойств. Детальное описание класса `LogManager` содержится в документации по API.

Можно изменять уровни протоколирования в работающей программе, используя консоль `jconsole`. За подробностями обращайтесь по адресу `http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html#LoggingControl`.

Локализация

Иногда возникает необходимость обеспечить отображение регистрационных сообщений на разных языках, чтобы они стали понятны пользователям из других стран. (Обеспечение поддержки разных языков приложением называют локализацией.) Интернационализация приложений обсуждается во втором томе. Ниже кратко рассматривается, что надо иметь в виду при локализации протокольных сообщений.

Информация, необходимая для локализации приложения, содержится в *пакетах ресурсов* (resource bundle). Пакет ресурсов представляет собой набор отображений, предназначенных для адаптации приложения для работы в разных странах (например, в США или Германии). Каждая таблица определяет соответствие между именами и объектами. Например, пакет ресурсов может отображать строку `"readingFile"` в строку `"Reading file"` или `"Achtung! Datei wird eingelesen"`.

Программа может содержать несколько пакетов ресурсов, например, один для меню, а другой для регистрационных сообщений. Каждый пакет ресурсов имеет имя (например, "com.mycompany.logmessages"). Для того чтобы добавить отображение к пакету ресурсов, надо предоставить файл для соответствующего языка. Отображение для английского языка находится в файле com/mycompany/logmessages_en.properties, а сообщение на немецком — в файле com/mycompany/logmessages_de.properties. (Здесь пары символов en и de представляют собой коды языков.) Объединение файлов осуществляется с помощью классов вашего приложения, в частности, класс ResourceBundle автоматически обнаруживает их.

Содержимое файлов представляется в текстовом формате и представляет собой записи, подобные приведенным ниже.

```
readingFile=Achtung! Datei wird eingelesen
renamingFile=Datei wird umbenannt
...
```

При вызове регистратора можно указать конкретный пакет ресурсов:

```
Logger logger = Logger.getlogger(loggerName,
   "com.mycompany.logmessages");
```

Затем для создания регистрационного сообщения нужно задать ключ пакета ресурсов (но не фактическую строку):

```
logger.info("readingFile");
```

Часто возникает необходимость включать в локализованные сообщения какие-либо параметры. В таком случае сообщение должно иметь заполнители {0}, {1} и т.д. Например, может потребоваться включить в регистрационное сообщение имя файла. Тогда заполнитель используется следующим образом:

```
Reading file {0};
Achtung! Datei {0} wird eingelesen.
```

Заполнители заменяются значениями при вызове одного из следующих методов:

```
logger.log(Level.INFO, "readingFile", fileName);
logger.log(Level.INFO, "renamingFile",
   new Object[]{ oldName, newName });
```

Обработчики

По умолчанию регистраторы посылают сообщения методу ConsoleHandler, который выводит их в поток System.err. Точнее говоря, регистраторы посылают запись предку обработчика, а его последний наследник (с именем "") содержит метод ConsoleHandler.

Как и регистраторы протоколирования, обработчики имеют уровни. Уровень протоколирования записи, подлежащей выводу, должен превышать пороговое значение *и регистратора, и обработчика*. Уровень консольного обработчика по умолчанию задается в файле, содержащем параметры настройки диспетчера регистрации:

```
java.util.logging.ConsoleHandler.level=INFO
```

Для регистрации записи, имеющей уровень FINE, необходимо изменить в конфигурационном файле как уровень регистрации, так и уровень обработчика, предусмотренные по умолчанию. Кроме того, можно совсем проигнорировать файл конфигурации и установить свой собственный обработчик.

```
Logger logger = Logger.getLogger("com.mycompany.myapp");
logger.setLevel(Level.FINE);
logger.setUseParentHandlers(false);
Handler handler = new ConsoleHandler();
handler.setLevel(Level.FINE);
logger.addHandler(handler);
```

По умолчанию регистратор посылает сообщения своему обработчику и его предку. Наш регистратор является наследником первоначального регистратора (имеющего имя ""), посылающего на консоль все записи уровня INFO и выше. Однако никому не нужно, чтобы все протокольные записи выводились на экран дважды, поэтому свойство useParentHandlers следует задать равным false.

Для того чтобы послать регистрационную запись еще куда-нибудь, нужно добавить другой обработчик. С этой целью API протоколирования предоставляет в наше распоряжение два класса: FileHandler и SocketHandler. Обработчик SocketHandler передает сообщения указанному хосту и порту. Наибольший интерес для нас представляет обработчик FileHandler, помещающий записи в файл.

Записи можно просто передать обработчику FileHandler по умолчанию.

```
FileHandler handler = new FileHandler();
logger.addHandler(handler);
```

Записи помещаются в файл javan.log, который находится в домашнем каталоге пользователя. Число *n* обеспечивает уникальность этого файла. Если на платформе понятие рабочего каталога пользователя не используется (например, в Windows 95/98/ME), файл записывается в каталог, предусмотренный по умолчанию, например, C:\Windows. Записи хранятся в формате XML. Обычная регистрационная запись имеет такой вид:

```
<record>
  <date>2002-02-04T07:45:15</date>
  <millis>1012837515710</millis>
  <sequence>1</sequence>
  <logger>com.mycompany.myapp</logger>
  <level>INFO</level>
  <class>com.mycompany.mylib.Reader</class>
  <method>read</method>
  <thread>10</thread>
  <message>Reading file corejava.gif</message>
</record>
```

Поведение обработчика FileHandler, определенное по умолчанию, можно изменить, задав другие параметры настройки диспетчера регистрации (табл. 11.2) или используя другой конструктор (см. заметки API в конце этого раздела).

Программисты, как правило, не используют имя файла протокола, предлагаемое по умолчанию. Следовательно, нужно задать другой шаблон, например, %h/myapp.log (переменные шаблона описаны в табл. 11.3).

Если несколько приложений (или несколько копий одного и того же приложения) используют один и тот же файл регистрации, следует установить флаг, управляющий добавлением данных в конец файла. Кроме того, можно использовать шаблон имени файла %u так, чтобы каждое приложение создавало уникальную копию файла регистрации.

Хорошей идеей является также “ротация” имени файла. В этом случае имена файлов регистрации формируются так: myapp.log.0, myapp.log.1, myapp.log.2 и т.д. Как только размер файла превысит допустимый предел, наиболее старый файл ре-

гистрации удаляется, остальные переименовываются, а новый файл получает имя с номером 0.

Таблица 11.2. Параметры настройки обработчика `FileHandler`

Свойство	Описание	Значение по умолчанию
`java.util.logging.FileHandler.level`	Уровень обработчика.	`Level.ALL`
`java.util.logging.FileHandler.append`	Определяет, должен обработчик добавлять записи в существующий файл или же открывать новый файл при каждом новом запуске программы.	`false`
`java.util.logging.FileHandler.limit`	Приблизительная оценка максимального объема файла. При превышении этого размера открывается новый файл (`0` — объем не ограничен).	В классе `FileHandler` — `0` (неограниченное количество), в конфигурации диспетчера регистрации по умолчанию — `50000`
`java.util.logging.FileHandler.pattern`	Шаблон имени файла регистрации (см. табл. 11.3).	`%h/java%u.log`
`java.util.logging.FileHandler.count`	Количество файлов регистрации, используемых для ротации.	`1` (ротация не выполняется)
`java.util.logging.FileHandler.filter`	Класс фильтра.	Фильтрация отсутствует
`java.util.logging.FileHandler.encoding`	Кодировка.	Кодировка, принятая в текущей операционной системе
`java.util.logging.FileHandler.formatter`	Средство форматирования записей.	`java.util.logging.XMLformatter`



Совет. Многие программисты используют средства регистрации для технической поддержки. Если в какой-то момент программа начинает работать некорректно, пользователь может обратиться к файлу регистрации. Чтобы это стало возможным, надо установить флаг, управляющий добавлением данных к имеющемуся файлу, использовать ротацию либо применить оба подхода одновременно.

Таблица 11.3. Переменные шаблона в имени файла регистрации

Переменная	Описание
`%h`	Значение системного свойства `user.home`
`%t`	Временный системный каталог
`%u`	Уникальный номер, позволяющий избежать конфликта имен
`%g`	Генерация номера для ротации
`%%`	Символ `%`

Можно также определять свои собственные обработчики, расширяя класс `Handler` или `StreamHandler`. Такой обработчик будет определен в программе, приведенной в качестве примера. Этот обработчик отображает записи на экране (рис. 11.2).

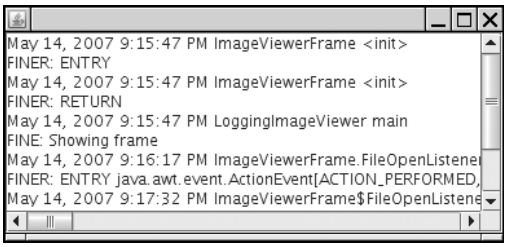


Рис. 11.2. Обработчик, отображающий записи на экране

Обработчик расширяет класс `StreanHandler` и устанавливает поток, содержащий метод `write()`, который выводит информацию в текстовую область.

```
class WindowHandler extends StreamHandler
{
   public WindowHandler()
   {
      . . .
      final JTextArea output = new JTextArea();
      setOutputStream(new
         OutputStream()
         {
            public void write(int b) {} // Не вызывается.
            public void write(byte[] b, int off, int len)
            {
               output.append(new String(b, off, len));
            }
         });
   }
   . . .
}
```

При таком подходе возникает лишь одна проблема — обработчик помещает записи в буфер и выводит их в поток только тогда, когда буфер переполняется. Следовательно, нужно переопределить метод `publish()`, чтобы выводить содержимое буфер после каждой записи.

```
class WindowHandler extends StreamHandler
{
   ...
   public void publish(LoigRecord record)
   {
      super.publish(record);
      flush();
   }
}
```

При желании можно написать более “экзотические” обработчики. Для этого достаточно расширить класс `Handler` и определить методы `publish()`, `flush()` и `close()`.

Фильтры

По умолчанию записи фильтруются в соответствии с уровнями протоколирования. Каждый регистратор и обработчик может иметь свой фильтр, выполняющий дополнительную фильтрацию. Для этого нужно реализовать интерфейс `Filter` и определить следующий метод:

```
boolean isloggable(LogRecord record)
```

Для анализа записей можно использовать любой критерий; метод должен возвращать значение `true` для тех записей, которые нужно поместить в файл протокола. Например, фильтр может пропускать только записи, созданные в начале выполнения метода и при возврате из него. Фильтр должен вызвать метод `record.getMessage()` и проверить, начинается запись с `ENTRY` или `RETURN`.

Для того чтобы задать фильтр в регистраторе или обработчике, вызовите метод `setFilter()`. Обратите внимание на то, что в каждый момент времени в вашем распоряжении может быть только один фильтр.

Средства форматирования

Классы `ConsoleHandler` и `FileHandler` порождают записи регистрации в текстовом и XML-форматах. Однако существует возможность определить свой собственный формат. Для этого необходимо расширить класс `Formatter` и переопределить следующий метод:

```
String format(logRecord record)
```

Затем следует форматировать запись любым выбранным способом и вернуть получившуюся строку. В методе `format()` может потребоваться вызов такого метода:

```
String formatMessage(LogRecord record)
```

Этот метод форматирует сообщение, содержащееся в регистрационной записи, заменяя параметры и выполняя локализацию.

Многие форматы файлов (например, XML) предполагают существование “головы” и “хвоста” файла, охватывающих форматированные записи. В этом случае следует переопределить следующие методы:

```
String getHead(Handler h)
String getTail(Handler h)
```

В заключение необходимо вызвать метод `setFormatter()`, инсталлирующий средства форматирования для обработчика.

Имея столько возможностей для регистрации, легко запутаться. Ниже приведены краткие рекомендации по выполнению основных операций.

“Рецепт” протоколирования

При таком обилии опций протоколирования очень легко выпустить из виду основы. Приведем “рецепт” установки системы протоколирования, подытожив наиболее часто используемые операции.

1. Для простых приложений выбирайте один регистратор. Желательно, чтобы имя регистратора совпадало с именем основного пакета приложения, например, `com.mycompany.myprog`. Создать регистратор можно с помощью такого вызова:

```
Logger.getLogger("com.mycompany.myprog");
```

Для удобства можно добавить в классы, интенсивно использующие регистрацию, статические поля:

```
private static final Logger logger = Logger.getLogger("com.mycompany.myprog");
```

2. По умолчанию все сообщения, имеющие уровень INFO и выше, выводятся на консоль. Пользователи могут изменить конфигурацию, предусмотренную по умолчанию, но, как мы уже видели, это довольно сложный процесс. Следовательно, лучше задать более оправданные установки приложения. Приведенный ниже код гарантирует, что все сообщения будут зарегистрированы в файле, связанном с конкретным приложением. Поместите его в метод `main()` вашего приложения.

```
if (System.getProperty("java.util.loging.config.class")== null   &&
System.getProperty("java.util.loging.config.file")== null)
{
   try
   {
      Logger.getLogger("").setLevel(Level.ALL);
      final int LOG_ROTATION_COUNT = 10;
      Handler handler = new FileHandler(%h/myapp.log,
         0, LOG_ROTATION_COUNT);
      Logger.getLogger("").addHandler(handler);
   }
   catch (IOException e)
   {
      logger.log (Level.SEVERE, "Can' create log file handler", e);
  }
}
```

3. Теперь все готово для регистрации сообщений. Помните, что все сообщения, имеющие уровень `INFO`, `WARNING` и `SEVERE`, выводятся на консоль. Следовательно, эти уровни нужно зарезервировать для сообщений, имеющих ценность для пользователей вашей программы. Уровень FINE лучше выделить для сообщений, предназначенных программистам. Там, где вы обычно использовали метод `System.out.println()`, регистрируйте сообщения:

```
logger.fine("File open dialog canceled");
```

Следует регистрировать также неожиданные исключения, например, так:

```
try
{
   ...
}
catch (SomeException exception)
{
   Logger.getlogger('com.mycompany.myprog".log(
      Level.FINE, "explanation", exception);
}
```

В листинге 11.2 приведен код программы, предназначенной для просмотра изображений, в которой предусмотрена регистрация событий.

Листинг 11.2. Содержимое файла `LoggingImageWindow.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.io.*;
4.  import java.util.logging.*;
5.  import javax.swing.*;
```

```
6.
7.  /**
8.   * Модификация программы для просмотра изображений.
9.   * В программе предусмотрена регистрация событий.
10.  * @version 1.02 2007-05-31
11.  * @author Cay Horstmann
12.  */
13. public class LoggingImageViewer
14. {
15.   public static void main(String[] args)
16.   {
17.     if (System.getProperty("java.util.logging.config.class") == null
18.     && System.getProperty("java.util.logging.config.file") == null)
19.     {
20.       try
21.       {
22.         Logger.getLogger("com.horstmann.corejava").setLevel(Level.ALL);
23.         final int LOG_ROTATION_COUNT = 10;
24.         Handler handler = new FileHandler("%h/LoggingImageViewer.log",
25.                           0, LOG_ROTATION_COUNT);
26.         Logger.getLogger("com.horstmann.corejava").addHandler(handler);
27.       }
28.       catch (IOException e)
29.       {
30.         Logger.getLogger("com.horstmann.corejava").log(Level.SEVERE,
31.           "Can't create log file handler", e);
32.       }
33.     }
34.     EventQueue.invokeLater(new Runnable()
35.     {
36.       public void run()
37.       {
38.         Handler windowHandler = new WindowHandler();
39.         windowHandler.setLevel(Level.ALL);
40.         Logger.getLogger("com.horstmann.corejava").addHandler(windowHandler);
41.         JFrame frame = new ImageViewerFrame();
42.         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
43.         Logger.getLogger("com.horstmann.corejava").fine("Showing frame");
44.         frame.setVisible(true);
45.       }
46.     });
47.   }
48. }
49.
50. /**
51.  * Фрейм для вывода изображений.
52.  */
53. class ImageViewerFrame extends JFrame
54. {
55.   public ImageViewerFrame()
56.   {
57.     logger.entering("ImageViewerFrame", "<init>");
58.     setTitle("LoggingImageViewer");
59.     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
60.
```

```
61.       // Создание строки меню.
62.       JMenuBar menuBar = new JMenuBar();
63.       setJMenuBar(menuBar);
64.       JMenu menu = new JMenu("File");
65.       menuBar.add(menu);
66.       JMenuItem openItem = new JMenuItem("Open");
67.       menu.add(openItem);
68.       openItem.addActionListener(new FileOpenListener());
69.       JMenuItem exitItem = new JMenuItem("Exit");
70.       menu.add(exitItem);
71.       exitItem.addActionListener(new ActionListener()
72.       {
73.          public void actionPerformed(ActionEvent event)
74.          {
75.             logger.fine("Exiting.");
76.             System.exit(0);
77.          }
78.       });
79.       // Для вывода изображений используется метка.
80.       label = new JLabel();
81.       add(label);
82.       logger.exiting("ImageViewerFrame", "<init>");
83.    }
84.    private class FileOpenListener implements ActionListener
85.    {
86.       public void actionPerformed(ActionEvent event)
87.       {
88.          logger.entering("ImageViewerFrame.FileOpenListener",
89.                          "actionPerformed", event);
90.          // Создание окна для выбора файлов.
91.          JFileChooser chooser = new JFileChooser();
92.          chooser.setCurrentDirectory(new File("."));
93.          // Принимать файлы с расширением .gif
94.          chooser.setFileFilter(new javax.swing.filechooser.FileFilter()
95.          {
96.             public boolean accept(File f)
97.             {
98.                return f.getName().toLowerCase().endsWith(".gif")
99.                   || f.isDirectory();
100.            }
101.            public String getDescription()
102.            {
103.               return "GIF Images";
104.            }
105.         });
106.         // Отображение диалогового окна для выбора файлов.
107.         int r = chooser.showOpenDialog(ImageViewerFrame.this);
108.         // Если файл выбран, устанавливается пиктограмма
109.         if (r == JFileChooser.APPROVE_OPTION)
110.         {
111.            String name = chooser.getSelectedFile().getPath();
112.            logger.log(Level.FINE, "Reading file {0}", name);
113.            label.setIcon(new ImageIcon(name));
114.         }
115.         else logger.fine("File open dialog canceled.");
116.         logger.exiting("ImageViewerFrame.FileOpenListener", "actionPerformed");
```

```
117.      }
118.   }
119.   private JLabel label;
120.   private static Logger logger = Logger.getLogger("com.horstmann.corejava");
121.   private static final int DEFAULT_WIDTH = 300;
122.   private static final int DEFAULT_HEIGHT = 400;
123. }
124.
125. /**
126.  * Обработчик для вывода в окне регистрационных записей.
127.  */
128. class WindowHandler extends StreamHandler
129. {
130.   public WindowHandler()
131.   {
132.      frame = new JFrame();
133.      final JTextArea output = new JTextArea();
134.      output.setEditable(false);
135.      frame.setSize(200, 200);
136.      frame.add(new JScrollPane(output));
137.      frame.setFocusableWindowState(false);
138.      frame.setVisible(true);
139.      setOutputStream(new OutputStream()
140.      {
141.         public void write(int b)
142.         {
143.         } // не вызывается
144.         public void write(byte[] b, int off, int len)
145.         {
146.            output.append(new String(b, off, len));
147.         }
148.      });
149.   }
150.   public void publish(LogRecord record)
151.   {
152.      if (!frame.isVisible()) return;
153.      super.publish(record);
154.      flush();
155.   }
156.   private JFrame frame;
157. }
```

API java.util.logging.Logger 1.4

- `Logger getLogger(String loggerName)`
- `Logger getLogger(String loggerName, String bundleName)`

Возвращает регистратор протоколирования с указанным именем. Если регистратор не существовал, он создается.

Параметры `loggerName` Иерархическое имя регистратора, например `com.mycompany.myapp`.

`bundleName` Имя пакета ресурсов для просмотра локализованных сообщений.

- `void severe(String message)`
- `void warning(String message)`
- `void info(String message)`
- `void config(String message)`
- `void fine(String message)`
- `void finer(String message)`
- `void finest(String message)`

Протоколируют запись, имеющую уровень, соответствующий имени метода, и содержащую заданное сообщение.

- `void entering(String className, String methodName)`
- `void entering(String className, String methodName, Object param)`
- `void entering(String className, String methodName, Object[] param)`
- `void exiting(String className, String methodName)`
- `void exiting(String className, String methodName, Object result)`

Протоколируют запись, описывающую вход и выход из метода с заданными параметрами или возвращаемым значением.

- `void throwing(String className, String methodName, Throwable t)`

Протоколирует запись, описывающую процесс генерации объекта исключения.

- `void log(Level level, String message)`
- `void log(Level level, String message, Object obj)`
- `void log(Level level, String message, Object[] obj)`
- `void log(Level level, String message, Throwable t)`

Протоколируют запись, имеющую заданный уровень и содержащую указанное сообщение. Запись может содержать обычные объекты или объект исключения. Для этой цели в сообщении предусмотрены заполнители `{0}`, `{1}` и т.д.

- `void logp(Level level, String className, String methodName, String message)`
- `void logp(Level level, String className, String methodName, String message, Object obj)`
- `void logp(Level level, String className, String methodName, String message, Object[] objs)`
- `void logp(Level level, String className, String methodName, String message, Throwable t)`

Протоколируют запись, имеющую заданный уровень и содержащую и информацию о вызове, и указанное сообщение. Запись может включать обычные объекты или объекты исключений.

- `void logrb(Level level, String className, String methodName, String bundleName, String message)`
- `void logrb(Level level, String className, String methodName, String bundleName, String message, Object obj)`
- `void logrb(Level level, String className, String methodName, String bundleName, String message, Object[] objs)`
- `void logrb(Level level, String className, String methodName, String bundleName, String message, Throwable t)`

Протоколируют запись, имеющую заданный уровень, содержащую информацию о вызове и указанное сообщение. Запись может включать обычные объекты или объекты исключений.

- `Level getLevel()`
- `void setLevel(Level l)`

  Возвращает и устанавливает уровень регистратора.

- `Logger getParent()`
- `void setParent(Logger l)`

  Возвращает и устанавливает родительский регистратор.

- `Handler[] getHandlers()`

  Возвращает все обработчики для регистратора.

- `void addHandler(Handler h)`
- `void removeHandler(Hadnler h)`

  Добавляет и удаляет обработчик для регистратора.

- `boolean getUseParentHandlers()`
- `void setUseParentHandlers(boolean b)`

  Возвращает и устанавливает свойство, управляющее использованием родительского обработчика. Если значение этого свойства равно `true`, регистратор направляет все зарегистрированные записи обработчикам своего родительского регистратора.

- `Filter getFilter()`
- `void setFilter(Filter f)`

  Возвращает и устанавливает фильтр для регистратора.

API java.util.logging.Handler 1.4

- `abstract void publish(LogRecord record)`

  Посылает сообщение по указанному адресу.

- `abstract void flush()`

  Выводит данные из буфера.

- `abstract void close()`

  Выводит данные из буфера и освобождает все ресурсы.

- `Filter getFilter()`
- `void setFilter(Filter f)`

  Возвращает и устанавливает фильтр для обработчика.

- `Formatter getFormatter()`
- `void setFormatter(Formatter f)`

  Возвращает и устанавливает средства форматирования для обработчика.

- `Level getlevel()`
- `void setLevel(Level l)`

  Возвращает и устанавливает уровень обработчика.

API java.util.logging.ConsoleHandler 1.4

- `ConsoleHandler()`

  Создает новый консольный обработчик.

API java.util.logging.FileHandler 1.4

- `FileHandler(String pattern)`
- `FileHandler(String pattern, boolean append)`
- `FileHandler(String pattern, int limit, int count)`
- `FileHandler(String pattern, int limit, int count, boolean append)`

Создают файловый обработчик.

Параметры

`pattern`	Шаблон для создания имени файла регистрации (см. табл. 11.3).
`limit`	Приблизительная оценка максимального объема файла регистрации. При превышении этого объема открывается новый файл.
`count`	Количество файлов при выполнении ротации.
`append`	Значение `true`, если вновь созданный объект файлового обработчика должен выводить данные в существующий файл регистрации.

API java.util.logging.LogRecord 1.4

- `Level getLevel()`

Возвращает уровень регистрации для записи.

- `String getLoggerName()`

Возвращает имя регистратора, который создал запись.

- `ResourceBundle getResourceBundle()`
- `String getResourceBundleName()`

Возвращают пакет ресурсов, предназначенный для локализации сообщения, или его имя. Если пакет не предусмотрен, возвращается значение `null`.

- `String getMessage()`

Возвращает "низкоуровневое" сообщение, не прошедшее локализацию и форматирование.

- `Object[] getParameters()`

Возвращает параметры или значение `null` при их отсутствии.

- `Throwable getThrown()`

Возвращает объект сгенерированного исключения или `null`, если объект не существует.

- `String getSourceClassName()`
- `String getSourceMethodName()`

Определяют местоположение кода, зарегистрировавшего данную запись. Эта информация может быть предоставлена кодом регистрации или автоматически выведена на основании анализа стека выполнения программы. Если код регистрации содержит неверное значение или программа была оптимизирована так, что определить местоположение кода невозможно, информация может оказаться неточной.

- `long getMillis()`

  Определяет время создания записи в миллисекундах, отсчитывая его от 1 января 1970 г.

- `long getSequenceNumber()`

  Возвращает уникальный номер записи.

- `int getThreadID()`

  Возвращает уникальный идентификационный номер потока, в котором была создана запись. Эти идентификационные номера присваиваются классом `LogRecord` и не имеют отношения к другим идентификаторам потоков.

API java.util.logging.Filter 1.4

- `boolean isLoggable(LogRecord record)`

  Если указанная запись должна быть зарегистрирована, возвращает значение `true`.

API java.util.logging.Formatter 1.4

- `abstract String format(LogRecord record)`

  Возвращает строку, полученную в результате форматирования регистрационной записи.

- `String getHead(Handler h)`
- `String getTail(Handler h)`

  Возвращают строки, которые должны появиться в начале и в конце документа, содержащего регистрационную запись. Эти методы определены в суперклассе `Formatter` и по умолчанию возвращают пустую строку. При необходимости их следует переопределить.

- `String format(LogRecord record)`

  Возвращает локализованное и форматированное сообщение, содержащееся в регистрационной записи.

Приемы отладки

Предположим, что вы написали программу и оснастили ее средствами для перехвата и соответствующей обработки исключений. Затем вы запускаете ее на выполнение, и она работает неправильно. Что теперь? (Если перед вами никогда не возникала такая проблема, можете не читать остальную часть главы.)

Разумеется, лучше всего было бы иметь под рукой удобный и мощный отладчик. Такие отладчики являются частью профессиональных сред программирования, например Eclipse, NetBeans. Позднее в этой главе мы еще вернемся к отладчикам, а пока в этом разделе покажем вам несколько приемов, которые стоит попробовать перед тем, как запускать отладчик.

1. Значение любой переменной можно вывести с помощью приведенного ниже выражения:

   ```
   System.out.println("x = " + x);
   ```

   Поместить это значение в регистрационный файл можно следующим образом:

   ```
   Logger.global.info("x=" + x);
   ```

Если в переменной x содержится числовое значение, оно преобразуется в строку. Если переменная x — объект, то вызывается метод toString. Состояние объекта, используемого в качестве неявного параметра, поможет определить следующий оператор:

```
Logger.global.info("this=" + this);
```

В большинстве классов языка Java метод toString переопределяется таким образом, чтобы предоставить пользователю как можно больше полезной информации. Это очень удобно для отладки. В своих классах следует самому позаботиться об этом.

2. Один малоизвестный, но очень полезный прием заключается в том, что в каждый класс можно включить отдельный метод main() и разместить в нем код, позволяющий протестировать этот класс отдельно от других.

```
public class MyClass
{
   методы и поля
   ...
   public static void main(String[] args)
   {
      код для проверки
   }
}
```

Создайте несколько объектов, вызовите все методы и проверьте, правильно ли они работают. Это делается с помощью методов main(); интерпретатор вызывается для каждого файла по отдельности. Если выполняется аплет, ни один из этих методов main() никогда не будет вызван, если выполняется приложение, то интерпретатор вызовет лишь метод main() из стартового класса.

3. Если предыдущая рекомендация пришлась вам по душе, попробуйте поработать с продуктом JUnit, который доступен по адресу http://junit.org. JUnit представляет собой популярную систему модульного тестирования, которая существенно упрощает работу по созданию наборов сценариев тестирования. Запускайте тесты после каждой модификации класса. При обнаружении неисправности добавляйте новый сценарий тестирования.

4. Протоколирующий прокси-объект представляет собой экземпляр подкласса, который перехватывает вызовы методов, протоколирует их, и обращается к суперклассу. Например, если у вас возникли проблемы с методом setBackground(), устанавливающим цвет фона панели, вы можете создать прокси-объект в виде безымянного класса.

```
JPanel panel = new
   JPanel()
   {
     public void setBackground(Color c)
     {
       Logger.global.info("setBackground: c=" + c);
       super.setBackground(c);
     }
   };
```

При каждом вызове метода setBackground() будет генерироваться регистрационное сообщение. Как выяснить вызывающий метод и сформировать трассировку стека, описывается ниже.

5. С помощью метода printStackTrace из класса Throwable можно получить трассировку стека для любого исключения. Приведенный ниже код перехватывает любое исключение, отображает объект исключения и трассировку стека, а затем повторно генерирует исключение, чтобы найти предназначенный для него обработчик.

```
try
{
   ...
}
catch (Throwable t)
{
   t.printStackTrace();
   throw t;
}
```

   Для трассировки стека не нужно даже перехватывать исключение. Просто включите следующее выражение в любое место кода:

```
Thread.dumpStack();
```

6. Обычно трассировка кода выводится в поток System.err. С помощью метода void PrintStackTrace(PrintWriter s) ее результаты можно записать в файл. При желании результаты трассировки стека можно вывести в окне, предварительно преобразовав в строку.

```
StringWriter out = nw StringWriter();
new Throwable().printStackTrace(new PrintWriter(out));
String trace = out.toString;
```

   (Классы PrinterWriter и StringWriter описываются во втором томе.)

7. Ошибки, возникающие при выполнении программы, удобно записывать в файл. Однако обычно ошибки посылаются в поток System.err, а не System.out. Следовательно, нельзя вывести ошибки в файл с помощью обычной команды:

```
java MyProgram > errors.txt
```

   Перехватите поток ошибок с помощью приведенной ниже команды.

```
java MyProgram 2> errors.txt
```

   Для того чтобы направить оба потока, Stream.err и Stream.out, в один и тот же файл, используйте такую команду:

```
java MyProgram 2>&1 errors.txt
```

   Это работает в bash и в командной оболочке Windows.

8. Трассировки стека не перехваченных исключений свидетельствует о том, что System.err не идеален. Его сообщения смущают конечных пользователей, если им случается их увидеть, и они не доступны для диагностических целей, когда в этом возникает потребность. Более правильный подход — протоколировать их в файле. Начиная с Java SE 5.0, можно заменять обработчик для не перехваченных исключений статическим методом Thread.setDefaultUncaughtExceptionHandler():

```
Thread.setDefaultUncaughtExceptionHandler(
new Thread.UncaughtExceptionHandler()
{
  public void uncaughtException(Thread t, Throwable e)
  {
    сохранить информацию в файле протокола
  };
});
```

9. Для того чтобы отследить загрузку класса, запустите виртуальную машину Java с опцией `-verbose`. На экране появятся такие строки:

```
[Opened /usr/local/jdk5.0/jre/lib/rt.jar]
[Opened /usr/local/jdk5.0/jre/lib/jsse.jar]
[Opened /usr/local/jdk5.0/jre/lib/jce.jar]
[Opened /usr/local/jdk5.0/jre/lib/charsets.jar]
[Loaded java.lang.Object from shared objects file]
[Loaded java.io.Serializable from shared objects file]
[Loaded java.lang.Comparable from shared objects file]
[Loaded java.lang.CharSequence from shared objects file]
[Loaded java.lang.String from shared objects file]
[Loaded java.lang.reflect.GenericDeclaration from shared objects file]
[Loaded java.lang.reflect.Type from shared objects file]
[Loaded java.lang.reflect.AnnotatedElement from shared objects file]
[Loaded java.lang.Class from shared objects file]
[Loaded java.lang.Cloneable from shared objects file]
```

   Этот способ оказывается полезным для диагностики ошибок, связанных с путями поиска классов.

10. Любуясь окнами из библиотеки Swing и удивляясь, как красиво и ровно расположены компоненты, естественно поинтересоваться, как именно они устроены. Для этого нужно нажать клавиши <Ctrl+Shift+F1>, и все компоненты будут выведены на экран по порядку.

```
FontDialog[frame0,0,0,300x200,layout=java.awt.BorderLayout,...
javax.swing.JRootPane[,4,23,292x173,layout=javax.swing.JRootPane$RootLayout,...
javax.swing.JPanel[null.glassPane,0,0,292x173,hidden,
layout=java.awt.FlowLayout,...
  javax.swing.JLayeredPane[null.layeredPane,0,0,292x173,...
javax.swing.JPanel[null.contentPane,0,0,292x173,
layout=java.awt.GridBagLayout,...
javax.swing.JList[,0,0,73x152,alignmentX=null,alignmentY=null,...
       javax.swing.CellRendererPane[,0,0,0x0,hidden]
        javax.swing.DefaultListCellRenderer$UIResource[,-73,-19,0x0,...
javax.swing.JCheckBox[,157,13,50x25,layout=javax.swing.OverlayLayout,...
javax.swing.JCheckBox[,156,65,52x25,layout=javax.swing.OverlayLayout,...
javax.swing.JLabel[,114,119,30x17,alignmentX=0.0,alignmentY=null,...
javax.swing.JTextField[,186,117,105x21,alignmentX=null,alignmentY=null,...
javax.swing.JTextField[,0,152,291x21,alignmentX=null,alignmentY=null,...
```

11. Если разработанный вами собственный компонент пользовательского интерфейса Swing на экране отображается некорректно, вы оцените удобство *графического отладчика Swing*. Даже если вы никогда не создавали свои собственные компоненты, очень поучительно проследить, как именно компоненты отображаются на экране. Чтобы переключиться на отладку компонента библиотеки Swing, нужно вызвать метод `setDebugGraphicsOptions()` класса `JComponent`. Этот метод имеет такие опции.

`DebugGraphics.FLASH_OPTION`	Перед тем, как отобразить линию, прямоугольник и текст на экране, показывает их в кадре.
`DebugGraphics.LOG_OPTION`	Выводит на экран сообщение о каждой графической операции.
`DebugGraphics.BUFFERED_OPTION`	Отображает операции, которые выполняются в видеобуфере.
`DebugGraphics.NONE_OPTION`	Отключает отладку графики.

Для того чтобы описанный выше механизм работал правильно, необходимо отключить опцию двойной буферизации, которая служит для подавления мерцания экрана при обновлении окна. Это можно сделать с помощью приведенных ниже операторов.

```
RepaintManager.currentManager(getRootPane()).setDoubleBufferingEnabled(false);
((JComponent)getContentPane()).setDebugGraphicsOptions(DebugGraphics.FLASH_OPTION);
```

Поместите эти строки в конец конструктора вашего фрейма. При выполнении программы вы увидите панель, которая медленно заполняется содержимым. Для локальной отладки каждого компонента воспользуйтесь методом `setDebugGraphicsOptions()`. Продолжительность, количество и цвет кадров можно регулировать (подробности — в описании класса `DebugGraphics` в документации).

12. В Java SE 5.0 была добавлена опция компилятора `-Xlint`, предназначенная для решения проблем, связанных с кодом программы. Например, если вы вызовете компилятор так, как указано ниже, то он проинформирует вас о пропущенных операторах `break` в блоках `switch`.

    ```
    javac -Xlint:fallthrough
    ```

    Термин `lint` ранее применялся для описания инструмента, предназначенного для обнаружения потенциальных проблем в программах, написанных на C. Теперь он обозначает все инструменты, отмечающие языковые конструкции, которые могут быть некорректными. Ниже описаны допустимые значения опции `-Xlint`.

`-Xlint` или `-Xlint:all`	Выполняет все проверки.
`-Xlint:deprecation`	То же, что и `-deprecation`, т.е. выполняется проверка на наличие методов, не рекомендованных к применению.
`-Xlint:fallthrough`	Проверяет наличие операторов `break` в блоках `switch`.
`-Xlint:finally`	Предупреждает о наличии выражений `finally`, которые не могут нормально завершиться.
`-Xlint:none`	Проверки не выполняются.
`-Xlint:path`	Проверяет, существуют ли каталоги, указанные в пути к классам и исходным файлам.
`-Xlint:serial`	Предупреждает о сериализуемых классах без `serialVersionUID` (подробно этот вопрос будет рассмотрен во втором томе).
`-Xlint:unchecked`	Предупреждает о сомнительных преобразованиях между универсальными и низкоуровневыми типами.

13. В Java SE 5.0 была реализована поддержка *мониторинга и управления* Java-приложениями. В виртуальной машине можно установить агентов, которые будут отслеживать расход памяти, использование потоков, загрузку классов и т.п. Эта возможность важна для работы с программами большого объема, работающими в течение длительного времени, например, с серверами приложений. Для демонстрации новых возможностей в комплект поставки JDK включен графический инструмент `jconsole`, который отображает статистические данные о производительности виртуальной машины (рис. 11.3). Для того чтобы включить мониторинг, надо указать при запуске виртуальной машины опцию `-Dcom.sun.management.jmxremote`. Затем следует найти идентификатор процесса операционной системы, поддерживающего работы виртуальной машины. В Unix/Linux

для этой цели надо использовать утилиту ps, а в системе Windows – диспетчер задач. Затем можно запускать программу jconsole.

```
jconsole processID
```

Консоль предоставит вам богатую информацию о вашей запущенной программе. Подробнее об этом – см. http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html.



На заметку! До Java SE 6 вы должны были запускать вашу программу с опцией -Dcom.sun.management.jmxremote:

```
java -Dcom.sun.management.jmxremote MyProgram
jconsole идентификатор_процесса
```

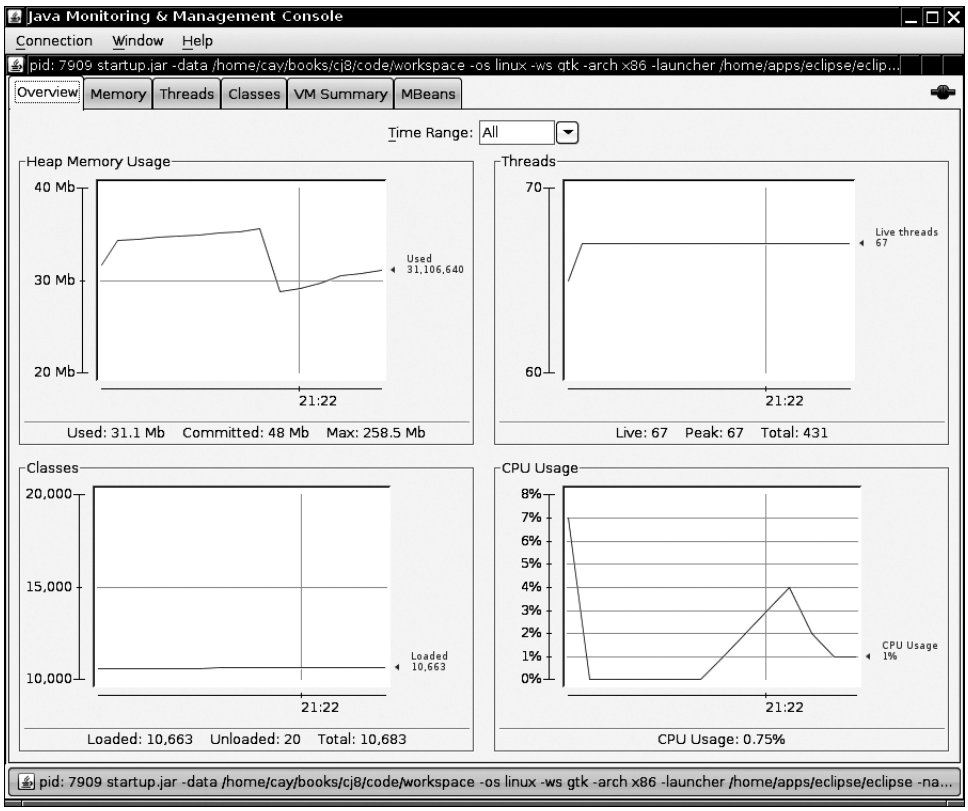


Рис. 11.3. Окно программы jconsole

14. Вы можете использовать утилиту jmap для получения дампа, показывающего каждый объект в куче. Введите следующие команды:

```
jmap -dump:format=b,file=имя_файла_дампа идентификатор_процесса
jhat имя_файла_дампа
```

Затем вы загружаете в своем браузере адрес localhost:7000. Там вы найдете Web-приложение, которое поможет разобраться в содержимом кучи на момент вывода дампа.

15. Если запустить виртуальную машину Java с опцией `-Xprof`, она запустит устаревший *профилировщик* (profiler), определяющий наиболее часто используемые методы. Профилировщик выводит информацию в поток `System.out`. В этом потоке также будет содержаться информация о том, какие методы были скомпилированы с помощью оперативной (JIT) компиляции.



Внимание! Опции компилятора, начинающиеся с `-X`, официально не поддерживаются и в некоторых версиях JDK могут отсутствовать. Получить список нестандартных опций можно с помощью команды `java -X`.

Использование консольного окна

Отлаживая аплет, вы можете просматривать сообщения об ошибках в окне: в панели конфигурации Java Plug-in установите флажок Show Java Console (Показать консоль Java), как описывалось в главе 10. Окно Java Console снабжено полосами прокрутки, так что в нем можно просматривать довольно пространные сообщения. Пользователи Windows имеют несомненное преимущество над пользователями системы DOS, в окне которой обычно появляются данные из потоков `System.err` и `System.out`.

Мы создадим класс окна, в котором можно будет просматривать сообщения отладчика. На рис. 11.4 показан наш класс `ConsoleWindow` в действии.

Использовать этот класс очень просто. Достаточно выполнить следующий вызов:

```
ConsoleWindow.init();
```

Затем можно выводить информацию в потоки `System.out` или `System.err`, как обычно.

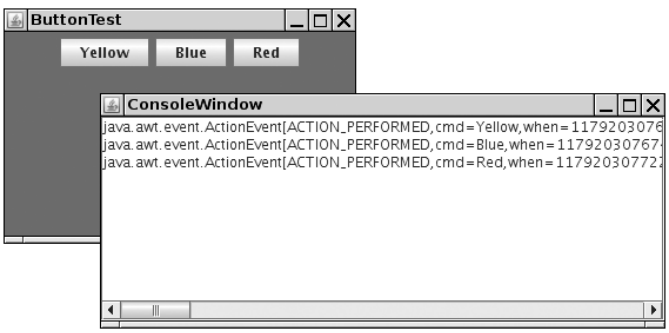


Рис. 11.4. Окно консоли

В листинге 11.3 показан код класса `ConsoleWindow`. Как видим, этот класс довольно прост. Сообщения отображаются в текстовой области `JTextArea` в панели `JScrollPane`. Для переключения потока вывода или потока ошибок на особый поток, выводящий сообщения в текстовую область, вызываются методы `System.setOut()` и `System.setErr()`.

Листинг 11.3. Содержимое файла `ConsoleWindow.java`

```
1.  import javax.swing.*;
2.  import java.io.*;
3.
```

```
4.  /**
5.   * Окно, предназначенное для отображения информации,
6.   * записанной в потоки System.out и System.err.
7.   * @version 1.01 2004-05-10
8.   * @author Cay Horstmann
9.  */
10. public class ConsoleWindow
11. {
12.   public static void init()
13.   {
14.      JFrame frame = new JFrame();
15.      frame.setTitle("ConsoleWindow");
16.      final JTextArea output = new JTextArea();
17.      output.setEditable(false);
18.      frame.add(new JScrollPane(output));
19.      frame.setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
20.      frame.setLocation(DEFAULT_LEFT, DEFAULT_TOP);
21.      frame.setFocusableWindowState(false);
22.      frame.setVisible(true);
23.
24.      // Определяется поток PrintStream, предназначенный
25.      // для вывода информации в текстовую область.
26.      PrintStream consoleStream = new PrintStream(new
27.        OutputStream()
28.        {
29.           public void write(int b) {} // Не вызывается.
30.           public void write(byte[] b, int off, int len)
31.           {
32.              output.append(new String(b, off, len));
33.           }
34.        });
35.
36.      // Данные System.out и System.err
37.      // будут передаваться в этот поток.
38.      System.setOut(consoleStream);
39.      System.setErr(consoleStream);
40.   }
41.
42.   public static final int DEFAULT_WIDTH = 300;
43.   public static final int DEFAULT_HEIGHT = 200;
44.   public static final int DEFAULT_LEFT = 200;
45.   public static final int DEFAULT_TOP = 200;
46. }
```

Трассировка событий AWT

Создавая пользовательский интерфейс на языке Java, мы должны знать, какие события происходят и каким компонентам они передаются. К сожалению, в документации AWT об этом говорится мало. Предположим, что мы хотим выводить подсказку в строке состояния, в зависимости от того, в какой части экрана перемещается курсор мыши. Библиотека AWT генерирует события, связанные с мышью и фокусом, которые можно перехватить.

Рассмотрим полезный класс EventTracer, предназначенный для слежения за этими событиями. Он выводит все методы, обрабатывающие события, а также их параметры. На рис. 11.5 показан результат трассировки событий.

Для того чтобы разобраться в этих сообщениях, добавьте компонент, события которого вы хотите отслеживать, в объект класса EventTracer:

```
EventTracer tracer = new EventTracer();
tracer.add(frame);
```

Он создает текстовое описание всех событий:

```
public abstract void java.awt.event.MouseListener.mouseExited(java.awt.
event.MouseEvent):
java.awt.event.MouseEvent[MOUSE_EXITED,(408,14),button=0,clickCount=0] on
javax.swing.JButton[,0,345,400x25,...]
public abstract void java.awt.event.FocusListener.focusLost(java.awt.event.
FocusEvent):
java.awt.event.FocusEvent[FOCUS_LOST,temporary,opposite=null] on javax.
swing.JButton[,0,345,400x25,...]
```

Эти сообщения можно отобразить в консольном окне или записать в файл, как объяснялось в предыдущем разделе.

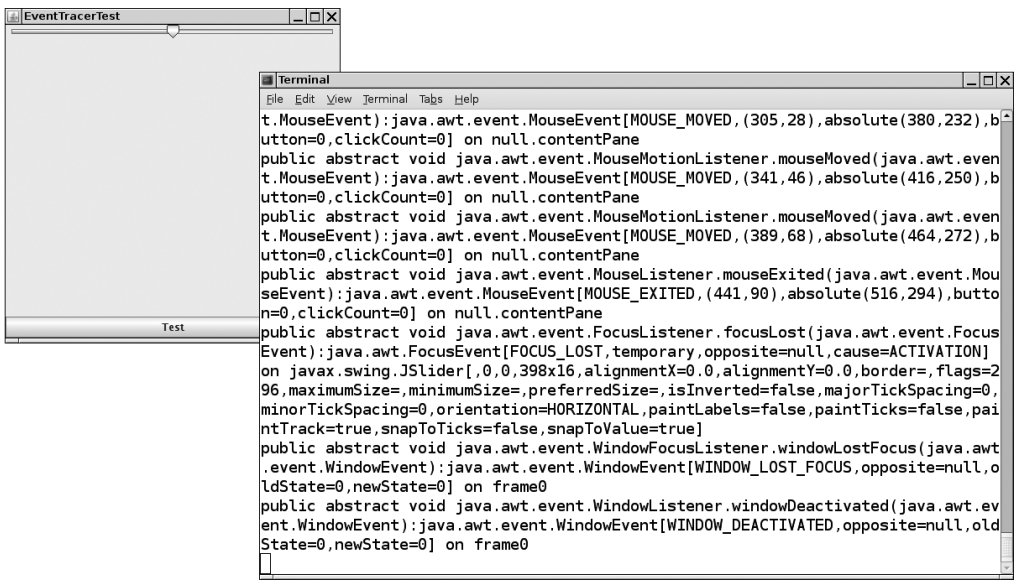


Рис. 11.5. Результат работы класса EventTracer

Код класса EventTracer приведен в листинге 11.4. Идея, лежащая в основе этого класса, довольно проста, хотя его реализация выглядит немного необычно. Вот что происходит "за кулисами".

1. Когда вы добавляете некий компонент в объект класса EventTracer с помощью метода add(), интроспективные средства JavaBeans ищут в нем методы, имеющие вид void add*Xxx*Listener(*Xxx*Event). (Информацию о JavaBeans можно найти во втором томе.) Для каждого найденного метода генерируется объект EventSetDescriptor. Каждый экземпляр этого класса передается методу addListener().

2. Если компонент является контейнером, следует перечислить вложенные в него компоненты и рекурсивно вызвать метод `add()` для каждого из них.
3. Методу `addListener()` передаются два параметра: компонент, за событиями которого мы следим, и объект `EventSetDescriptor`. Метод `getListenerType()` класса `EventSetDescriptor` возвращает объект `Class`, описывающий интерфейс обработчика события, например, `ActionListener` или `ChangeListener`. Для интерфейса создается прокси-объект. Он выводит имя и параметр вызванного метода события. Метод `getAddListenerMethod()` класса `EventSetDescriptor` возвращает объект `Method`. Этот объект используется для добавления прокси-объекта в компонент, в котором он выступает в качестве слушателя события.

   Эта программа представляет собой хорошую иллюстрацию возможностей механизма рефлексии. Мы можем не знать, что в классе `JButton` есть метод `addActionListener()`, а в классе `Jslider` — метод `addChangeListener()`. Механизм рефлексии выявит эти факты для нас.

В листинге 11.5 приведена программа, иллюстрирующая работу трассировщика событий. Эта программа выводит на экран фрейм с кнопкой и линейным регулятором и отслеживает события, генерируемые этими компонентами.

Листинг 11.4. Содержимое файла `EventTracer.java`

```
1.  import java.awt.*;
2.  import java.beans.*;
3.  import java.lang.reflect.*;
4.
5.  /**
6.   * @version 1.31 2004-05-10
7.   * @author Cay Horstmann
8.   */
9.  public class EventTracer
10. {
11.   public EventTracer()
12.   {
13.      // Обработчик всех прокси-объектов событий.
14.      handler = new InvocationHandler()
15.         {
16.            public Object invoke(Object proxy,
17.               Method method, Object[] args)
18.            {
19.               System.out.println(method + ":" + args[0]);
20.               return null;
21.            }
22.         };
23.   }
24.
25.   /**
26.    * Добавляет трассировщики событий для всех событий,
27.    * которые можно прослушивать у данного компонента
28.    * и его дочерних классов.
29.    * @param c Компонент
30.    */
31.   public void add(Component c)
32.   {
```

```
33.       try
34.       {
35.          // Получение всех событий, которые может
36.          // прослушивать этот компонент.
37.
38.          BeanInfo info = Introspector.getBeanInfo(c.getClass());
39.
40.          EventSetDescriptor[] eventSets =
41.             info.getEventSetDescriptors();
42.          for (EventSetDescriptor eventSet : eventSets)
43.             addListener(c, eventSet);
44.       }
45.       catch (IntrospectionException e)
46.       {
47.       }
48.       // Если генерируется исключение,
49.       // слушатели не добавляются.
50.
51.       if (c instanceof Container)
52.       {
53.          // Получение всех дочерних объектов
54.          // и рекурсивный вызов метода add.
55.          for (Component comp : ((Container) c).getComponents())
56.             add(comp);
57.       }
58.    }
59.
60.    /**
61.     * Добавление слушателя для заданного множества событий.
62.     * @param c Компонент
63.     * @param eventSet Описание интерфейса слушателя
64.     */
65.    public void addListener(Component c,
66.       EventSetDescriptor eventSet)
67.    {
68.       // Создание прокси-объекта для данного типа
69.       // слушателя и перенаправление вызовов обработчику.
70.       Object proxy = Proxy.newProxyInstance(null,
71.          new Class[] { eventSet.getListenerType() }, handler);
72.
73.       // Добавление прокси-объекта в качестве слушателя для компонента.
74.       Method addListenerMethod = eventSet.getAddListenerMethod();
75.       try
76.       {
77.          addListenerMethod.invoke(c, proxy);
78.       }
79.       catch(InvocationTargetException e) {}
80.       catch(IllegalAccessException e) {}
81.       // Если генерируется исключение,
82.       // слушатель не добавляется.
83.    }
84.
85.    private InvocationHandler handler;
86. }
```

Листинг 11.5. Содержимое файла `EventTracerTest.java`

```
1.  import java.awt.*;
2.  import javax.swing.*;
3.
4.  /**
5.   * @version 1.13 2007-06-12
6.   * @author Cay Horstmann
7.   */
8.  public class EventTracerTest
9.  {
10.   public static void main(String[] args)
11.   {
12.      EventQueue.invokeLater(new Runnable()
13.         {
14.            public void run()
15.            {
16.               JFrame frame = new EventTracerFrame();
17.               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18.               frame.setVisible(true);
19.            }
20.         });
21.   }
22. }
23.
24. class EventTracerFrame extends JFrame
25. {
26.   public EventTracerFrame()
27.   {
28.      setTitle("EventTracerTest");
29.      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
30.
31.      // Добавление линейного регулятора и кнопки.
32.      add(new JSlider(), BorderLayout.NORTH);
33.      add(new JButton("Test"), BorderLayout.SOUTH);
34.
35.      // Перехват всех событий, связанных с компонентами
36.      // в составе фрейма.
37.      EventTracer tracer = new EventTracer();
38.      tracer.add(this);
39.   }
40.
41.   public static final int DEFAULT_WIDTH = 400;
42.   public static final int DEFAULT_HEIGHT = 400;
43. }
```

Позволим роботу AWT выполнять его работу

В Java SE 1.3 появился класс `Robot`, который можно использовать для передачи нажатий клавиш и щелчков кнопками мыши в любую AWT-программу. Этот класс предназначен для автоматической проверки пользовательского интерфейса.

Для того чтобы создать объект класса `Robot`, нужно сначала получить объект класса `GraphicsDevice`. Устройство экрана, заданное по умолчанию, можно определить с помощью следующей последовательности вызовов:

```
GraphicsEnvitonment environment
   = GraphicsEnvironment.getLocalGraphicsEnvironment();
GraphicsDevice screen
   = environment.getDefaultScreenDevice();
```

Затем создается объект класса `Robot`:

```
Robot robot = new Robot(screen);
```

Чтобы передать нажатие клавиш в программу, объект `Robot` должен имитировать нажатие и отпускание клавиш:

```
robot.keyPress(KeyEvent.VK_TAB);
robot.keyRelease(KeyEvent.VK_TAB);
```

Для имитации щелчка мыши сначала нужно переместить ее курсор, а затем нажать и отпустить кнопку:

```
robot.mouseMove(x, y);  // x и y - абсолютные координаты экрана
robot.mousePress(InputEvent.BUTTON1_MASK);
robot.mouseRelease(InputEvent.BUTTON1_MASK);
```

Основная идея проверки состоит в том, что, имитируя нажатие клавиш и щелчки кнопками мыши, можно увидеть, правильно ли приложение реагирует на них. Затем нужно захватить содержимое экрана с помощью метода `createScreenCapture()`:

```
Rectangle rect = new Rectangle(x, y, width, height);
BufferedImage image = robot.createScreenCapture(rect);
```

Координаты, задаваемые в конструкторе `Rectangle`, также являются абсолютными координатами экрана и выражаются в пикселях.

Кроме того, чтобы приложение могло перехватить команды, выполняемые объектами класса `Robot`, между ними нужны небольшие задержки. Для этого предназначен метод `delay()`, при вызове которого указывается величина задержки в миллисекундах:

```
robot.delay(1000); // Задержка на 1000 миллисекунд.
```

Программа, приведенная в листинге 11.6, иллюстрирует применение объектов класса `Robot`. Эти объекты тестируют программу для работы с кнопками, рассмотренную нами в главе 8. Сначала в ответ на нажатие клавиши пробела активизируется левая кнопка. Затем объект `Robot` ожидает две секунды, чтобы дать пользователю возможность увидеть состояние экрана. По истечении этого срока объект класса `Robot` имитирует нажатие клавиши <Tab> и клавиши пробела для активизации следующей кнопки. После этого имитируется щелчок на третьей кнопке. (Для того чтобы действительно щелкнуть на этой кнопке, в программе нужно правильно задать параметры `x` и `y`.) В заключение содержимое экрана перехватывается и отображается в отдельном фрейме (рис. 11.6).

Этот пример показывает, что класс `Robot` сам по себе для тестирования пользовательского интерфейса не совсем удобен. Скорее, он представляет собой базовые "строительные блоки", из которых можно сконструировать средство для тестирования программ. Профессиональные средства тестирования могут фиксировать, хранить и воспроизводить сценарии поведения пользователя, а также определять расположение компонентов на экране, чтобы не щелкать кнопками мыши наугад. Надеемся, что с ростом популярности Java-приложений, появятся более совершенные средства тестирования.

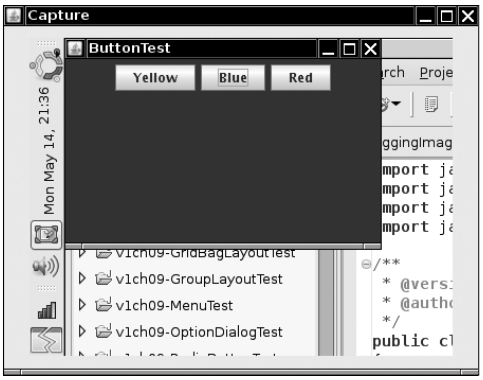


Рис. 11.6. Захват содержимого экрана с помощью объекта `Robot`

Листинг 11.6. Содержимое файла `RobotTest.java`

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.awt.image.*;
4. import javax.swing.*;
5.
6. /**
7.  * @version 1.03 2007-06-12
8.  * @author Cay Horstmann
9.  */
10. public class RobotTest
11. {
12.    public static void main(String[] args)
13.    {
14.       EventQueue.invokeLater(new Runnable()
15.          {
16.             public void run()
17.             {
18.                // Создание фрейма, содержащего панель с кнопками.
19.
20.                ButtonFrame frame = new ButtonFrame();
21.                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22.                frame.setVisible(true);
23.
24.                // Связывание объекта Robot с экранным устройством.
25.
26.                GraphicsEnvironment environment =
27.                      GraphicsEnvironment.getLocalGraphicsEnvironment();
28.                GraphicsDevice screen = environment.getDefaultScreenDevice();
29.
30.                try
31.                {
32.                   Robot robot = new Robot(screen);
33.                   runTest(robot);
34.                }
35.                catch (AWTException e)
36.                {
37.                   e.printStackTrace();
```

```
38.             }
39.          }
40.       });
41.   }
42.   /**
43.   * Запускает процедуру тестирования.
44.   * @param robot Объект Robot, связанный с экранным устройством
45.   */
46.   public static void runTest(Robot robot)
47.   {
48.      // Имитация нажатия пробела.
49.      robot.keyPress(' ');
50.      robot.keyRelease(' ');
51.      // Имитация нажатия клавиши <Tab> с последующим
52.      // нажатием клавиши пробела.
53.      robot.delay(2000);
54.      robot.keyPress(KeyEvent.VK_TAB);
55.      robot.keyRelease(KeyEvent.VK_TAB);
56.      robot.keyPress(' ');
57.      robot.keyRelease(' ');
58.      // Имитация щелчка правой кнопкой мыши.
59.      robot.delay(2000);
60.      robot.mouseMove(200, 50);
61.      robot.mousePress(InputEvent.BUTTON1_MASK);
62.      robot.mouseRelease(InputEvent.BUTTON1_MASK);
63.      // Захват содержимого экрана и вывод
64.      // полученного изображения.
65.      robot.delay(2000);
66.      BufferedImage image =
67.         robot.createScreenCapture(new Rectangle(0, 0, 400, 300));
68.      ImageFrame frame = new ImageFrame(image);
69.      frame.setVisible(true);
70.   }
71. }
72.
73. /**
74.  * Фрейм для вывода захваченного изображения.
75.  */
76. class ImageFrame extends JFrame
77. {
78.   /**
79.    * @param image Изображение для вывода
80.    */
81.   public ImageFrame(Image image)
82.   {
83.      setTitle("Capture");
84.      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
85.
86.      JLabel label = new JLabel(new ImageIcon(image));
87.      add(label);
88.   }
89.
90.   public static final int DEFAULT_WIDTH = 450;
91.   public static final int DEFAULT_HEIGHT = 350;
92. }
```

API java.awt.GraphicsEnvironment 1.2

- `static GraphicsEnvironment getLocalGraphicsEnvironment()`

  Возвращает локальное графическое окружение.

- `GraphicsDevice getDefaultScreenDevice()`

  Возвращает экранное устройство, заданное по умолчанию. Обратите внимание на то, что компьютеры с несколькими мониторами имеют по одному устройству на каждый экран, поэтому, чтобы получить массив, содержащий все устройства, используйте метод `getScreenDevices`.

API java.awt.Robot 1.3

- `Robot(GraphicsDevice device)`

  Создает объект класса `Robot`, который может взаимодействовать с данным устройством.

- `void keyPress(int key)`
- `void keyRelease(int key)`

  Имитирует нажатие и отпускание клавиши.

  Параметр `key` Код клавиши. Подробную информацию о кодах клавиш можно найти, рассмотрев класс `KeyStroke`.

- `void mouseMove(int x, int y)`

  Имитирует перемещение курсора мыши.

  Параметр `x, y` Абсолютные координаты курсора мыши, выраженные в пикселях.

- `void mousePress(int eventMask)`
- `void mouseRelease(int eventMask)`

  Имитирует нажатие и отпускание кнопки мыши.

  Параметр `eventMask` Маска события, описывающая кнопку мыши. Дополнительную информацию о масках событий можно получить, рассмотрев класс `InputEvent`.

- `void delay(int milliseconds)`

  Приостанавливает работу объекта `Robot` на заданное количество миллисекунд.

- `BufferedImage createScreenCapture(rectangle rect)`

  Захватывает часть экрана.

  Параметр `rect` Часть экрана, подлежащая захвату. Задается в абсолютных координатах.

Применение отладчика

Отладка программы с помощью вывода на экран промежуточных результатов — не самое приятное занятие. Вам приходится постоянно добавлять и удалять операторы, а затем компилировать программу заново. Отладчик намного облегчит вашу работу, поскольку в нем можно установить *точку прерывания* (breakpoint). Вплоть до этой точ-

ки программа выполняется без задержек, а после того, как выполнение программы приостанавливается, можно получить интересующие сведения о состоянии кода.

В листинге 11.7 показана программа `ButtonTest` из главы 8, в код которой внесены некоторые ошибки. (Мы разделили программу на части и поместили каждый класс в отдельный файл, так как в некоторых отладчиках при одновременной работе с несколькими классами возникают проблемы.)

После щелчка на любой кнопке ничего не произойдет. Посмотрите на код — предполагается, что после щелчка на кнопке фон должен окраситься в соответствующий цвет.

Листинг 11.7. Содержимое файла `BuggyButtonTest.java`

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import javax.swing.*;
4.
5.  /**
6.   * @version 1.22 2007-05-14
7.   * @author Cay Horstmann
8.   */
9.  public class BuggyButtonTest
10. {
11.   public static void main(String[] args)
12.   {
13.      EventQueue.invokeLater(new Runnable()
14.         {
15.            public void run()
16.            {
17.               BuggyButtonFrame frame = new BuggyButtonFrame();
18.               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.               frame.setVisible(true);
20.            }
21.         });
22.   }
23. }
24.
25. class BuggyButtonFrame extends JFrame
26. {
27.   public BuggyButtonFrame()
28.   {
29.      setTitle("BuggyButtonTest");
30.      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
31.
32.      // Добавление панели к фрейму.
33.
34.      BuggyButtonPanel panel = new BuggyButtonPanel();
35.      add(panel);
36.   }
37.
38.   public static final int DEFAULT_WIDTH = 300;
39.   public static final int DEFAULT_HEIGHT = 200;
40. }
41.
42. class BuggyButtonPanel extends JPanel
43. {
```

```
44.    public BuggyButtonPanel()
45.    {
46.       ActionListener listener = new ButtonListener();
47.
48.       JButton yellowButton = new JButton("Yellow");
49.       add(yellowButton);
50.       yellowButton.addActionListener(listener);
51.
52.       JButton blueButton = new JButton("Blue");
53.       add(blueButton);
54.       blueButton.addActionListener(listener);
55.
56.       JButton redButton = new JButton("Red");
57.       add(redButton);
58.       redButton.addActionListener(listener);
59.    }
60.
61.    private class ButtonListener implements ActionListener
62.    {
63.       public void actionPerformed(ActionEvent event)
64.       {
65.          String arg = event.getActionCommand();
66.          if (arg.equals("yellow")) setBackground(Color.yellow);
67.          else if (arg.equals("blue")) setBackground(Color.blue);
68.          else if (arg.equals("red")) setBackground(Color.red);
69.       }
70.    }
71. }
```

Ошибку в такой короткой программе можно обнаружить и без отладчика, для этого достаточно внимательно просмотреть код. Допустим, что наша программа очень сложна. Мы покажем вам, как использовать отладчик Eclipse для нахождения ошибки.



На заметку! Если вы используете автономный отладчик, такой как JSwat (`http://www.bluemarsh.com/java/jswat/`) или древний и тяжеловесный `jdb`, то должны сначала скомпилировать программу с опцией `-g`, например:

```
javac -g BuggyButtonTest.java
```

В интегрированной среде разработки это осуществляется автоматически.

Для запуска отладчика в Eclipse выберите пункт меню Run⇨Debug As⇨Java Application (Выполнить⇨Отладить как⇨Java-приложение). Ваша программа запустится.

Установите точку прерывания на первой строке метода `actionPerformed()`: щелкните правой кнопкой мыши на левом поле напротив строки кода и выберите Toggle Breakpoint (Включить точку прерывания) из контекстного меню.

Точка прерывания будет достигнута, как только Java начнет выполнять код метода `actionPerformed()`. Для этого щелкните в окне вашей программы на кнопке Yellow (Желтый). Отладчик остановится в начале метода `actionPerformed()`, как показано на рис. 11.7.

Для пошагового исполнения программы предусмотрено две базовых команды. Step Into входит в каждый вызов метода, а Step Over переходит к следующей строке, не входя внутрь вызовов методов. Eclipse следует использовать пункты меню Run⇨Step Into и Run⇨Step Over или клавиатурные сокращения <F5> и <F6>.

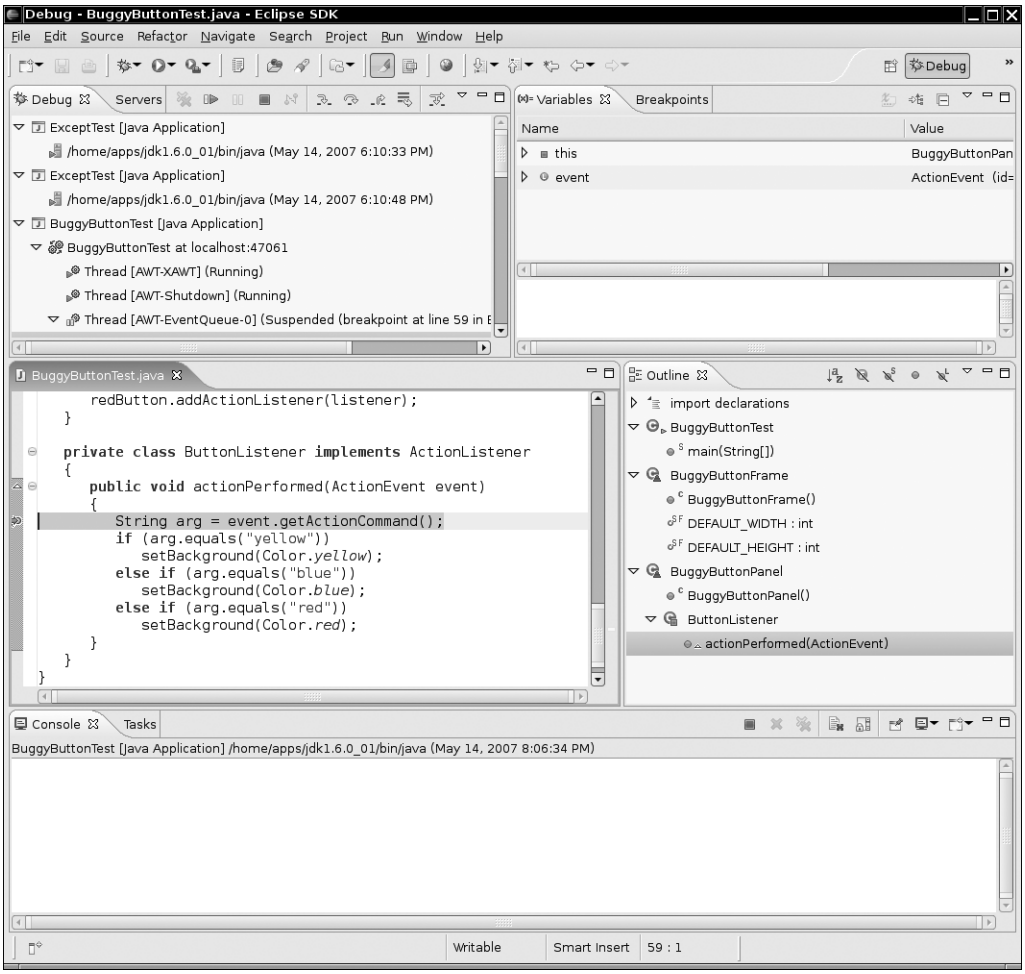
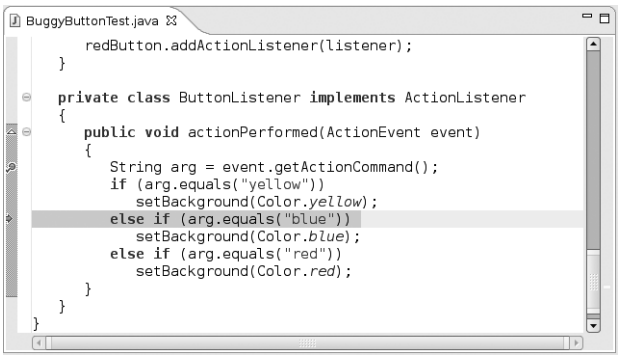


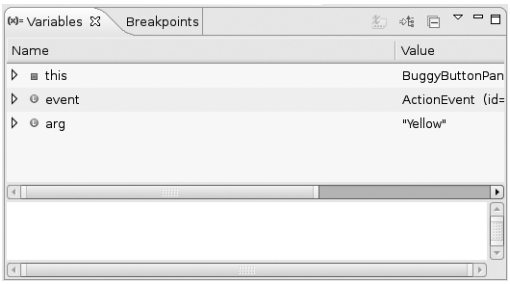
Рис. 11.7. Точка прерывания в отладчике

Два раза выполните команду Step Over и посмотрите, к чему это приведет.



Это явно не то, что должно было произойти. Предполагалось, что программа вызовет `setColor(Color.yellow)`, после чего выйдет из метода.

Просмотрим содержимое локальных переменных, чтобы узнать значение переменной `arg`:



Теперь вы можете видеть, что случилось. Значением `arg` было `"Yellow"`, т.е. с заглавной буквы “Y”, а сравнение осуществляется с `"yellow"`, т.е. с прописной:

```
if (arg.equals("yellow"))
```

Чтобы выйти из отладчика, выберите в меню пункт Run⇨Terminate (Выполнить⇨Завершить).

В отладчике Eclipse предусмотрено еще много расширенных команд, но, ограничиваясь даже самой простой техникой, которую мы продемонстрировали здесь, можно продвинуться достаточно далеко. Другие отладчики вроде NetBeans имеют очень похожий набор команд.

В этой главе вы ознакомились с обработкой исключений, а также получили несколько советов, касающихся тестирования и отладки. В следующей главе мы поговорим об обобщенном (generic) программировании и его наиболее важном применении: каркасе коллекций Java.

Глава 12

ОБОБЩЕННОЕ ПРОГРАММИРОВАНИЕ

В ЭТОЙ ГЛАВЕ...

- Назначение обобщенного программирования
- Определение простого обобщенного класса
- Обобщенные методы
- Ограничения переменных типов
- Обобщенный код и виртуальная машина
- Ограничения и лимиты
- Правила наследования обобщенных типов
- Подстановочные типы
- Рефлексия и обобщения

Обобщения (generics) представляют собой наиболее существенное изменение в языке программирования Java со времен версии 1.0. Появление обобщений в Java 5.0 стало результатом первых требований к спецификации Java (Java Specification Requests — JSR 14), которые были сформулированы в 1999 г. Группа экспертов потратила около пяти лет на разработку спецификаций и тестовые реализации.

Обобщения понадобились потому, что они позволяют писать более безопасный код, который легче читается, чем код, перегруженный переменными `Object` и приведениями типов. Обобщения особенно полезны для классов коллекций, таких как вездесущий `ArrayList`.

Обобщения похожи — по крайней мере, внешне — на шаблоны в C++. В C++, как и в Java, шаблоны впервые были добавлены для поддержки строго типизированных коллекций. Однако с годами были открыты и другие их применения. После прочтения этой главы, возможно, вы найдете новые, ранее неизвестные применения для обобщений в своих программах.

Назначение обобщенного программирования

Обобщенное программирование означает написание кода, который может быть многократно использован с объектами многих различных типов. Например, вы не хотите программировать отдельные классы для коллекционирования объектов `String` и `File`. И вам не нужно этого делать — единственный `ArrayList` может собирать объекты любого класса. Это лишь один пример обобщенного программирования.

До появления Java SE 5.0 обобщенное программирование в Java всегда реализовывалось посредством наследования. Класс `ArrayList` просто поддерживал массив ссылок на `Object`:

```
public class ArrayList // до Java SE 5.0
{
  public Object get(int i) { . . . }
  public void add(Object o) { . . . }
  . . .
  private Object[] elementData;
}
```

Такой подход порождает две проблемы. Всякий раз, когда извлекается значение, необходимо выполнить приведение типа:

```
ArrayList files = new ArrayList();
. . .
String filename = (String) names.get(0);
```

Более того, здесь нет проверки ошибок. Вы можете добавить значение любого класса:

```
files.add(new File(". . ."));
```

Этот вызов компилируется и запускается без ошибок. Но затем попытка выполнить приведение результата метода `get()` к типу `String` вызовет ошибку.

Обобщения предлагают лучшее решение: параметры типа. Класс `ArrayList` теперь принимает параметр типа, указывающий тип хранимых элементов:

```
ArrayList<String> files = new ArrayList<String>();
```

Это облегчает чтение вашего кода. Вы можете сразу понять, что этот конкретный массив-список содержит объекты `String`.

Компилятор также может успешно использовать эту информацию. Не требуется никакого приведения при вызове `get()`. Компилятор знает, что типом возврата этого метода является `String`, а не `Object`:

```
String filename = files.get(0);
```

Компилятор также знает, что метод `add()` в `ArrayList<String>` имеет параметр типа `String`. Это намного безопаснее, чем иметь дело с параметром типа `Object`. Теперь компилятор может проконтролировать, чтобы вы не вставили объект неверного типа. Например, следующий оператор не скомпилируется:

```
files.add(new File(". . ")); // в ArrayList<String> можно добавлять
                             // только объекты String
```

Ошибка компиляции — это намного лучше, чем исключение приведения типа во время выполнения. В этом привлекательность параметров типа: они делают вашу программу легче читаемой и более безопасной.

Кто хочет быть обобщенным программистом?

Такие обобщенные классы, как `ArrayList`, использовать легко. И большинство программистов Java просто применяют типы вроде `ArrayList<String>` — как если бы они были частью языка, подобно массивам `String[]`. (Разумеется, массивы-списки лучше простых массивов, поскольку могут автоматически расширяться.)

Однако реализовать обобщенный класс не так просто. Программисты, которые будут использовать ваш код, попытаются подставлять всевозможные классы вместо ваших параметров типа. Они ожидают, что все будет работать без досадных ограничений и запутанных сообщений об ошибках. Ваша задача как обобщенного программиста — предвидеть все возможные будущие применения вашего класса.

Насколько это трудно сделать? Приведем пример типичной проблемы, с которой сталкивались проектировщики стандартной библиотеки классов. Класс `ArrayList` содержит метод `addAll()`, предназначенный для добавления элементов другой коллекции. Программист может пожелать добавить все элементы из `ArrayList<Manager>` в `ArrayList<Employee>`. Но, конечно же, обратная задача не должна быть разрешена. Как разрешить один вызов и запретить другой? Проектировщики языка Java изобрели остроумную новую концепцию — *подстановочный тип* (wildcard type), чтобы решить эту проблему. Подстановочные типы довольно абстрактны, но они позволяют разработчику библиотеки сделать методы насколько возможно гибкими.

Обобщенное программирование разделяется на три уровня квалификации. На базовом уровне вы просто используете обобщенные классы — обычно это коллекции вроде `ArrayList` — причем делаете это, не задумываясь о том, как они работают. Большинство прикладных программистов предпочитают оставаться на этом уровне до тех пор, пока что-то не пойдет не так. Вы можете столкнуться с непонятным сообщением об ошибке, смешивая разные обобщенные классы, или же имея дело с унаследованным кодом, который ничего не знает о параметрах типа. В такой момент вам нужно знать достаточно об обобщениях Java, чтобы решить проблему системно, а не "методом тыка". И, наконец, конечно, вы можете решить реализовать свои собственные обобщенные классы и методы.

Прикладным программистам, вероятно, не придется писать много обобщенного кода. Разработчики из Sun уже выполнили самую тяжелую работу и предусмотрели параметры типа для всех классов коллекций. В качестве эмпирического правила можно сказать так: от применения параметров типа выигрывает только тот код, в котором традиционно присутствует много приведений от очень общих типов (таких как `Object` или интерфейс `Comparable`).

В этой главе мы расскажем вам все, что необходимо знать для реализации вашего собственного обобщенного кода. Однако вы надеемся, что наши читатели используют свои знания, прежде всего, для отыскания причин неполадок, а также для удовлетворения любопытства относительно внутреннего устройства параметризованных классов коллекций.

Определение простого обобщенного класса

Обобщенный класс — это класс с одной или более переменной типа. В этой главе мы используем в качестве примера простой класс `Pair`. Этот класс позволит сосредоточиться на механизме обобщений, не вдаваясь в подробности хранения данных. Код обобщенного класса Pair приведен ниже.

```
public class Pair<T>
{
    public Pair() { first = null; second = null; }
    public Pair(T first, T second) { this.first = first; this.second = second; }
    public T getFirst() { return first; }
    public T getSecond() { return second; }
    public void setFirst(T newValue) { first = newValue; }
    public void setSecond(T newValue) { second = newValue; }
    private T first;
    private T second;
}
```

Класс `Pair` вводит переменную типа `T`, заключенную в угловые скобки <>, следующие после имени класса. Обобщенный класс может иметь более одной переменной типа. Например, мы могли бы определить класс `Pair` с разными типами для первого и второго поля:

```
public class Pair<T, U> { . . . }
```

Переменные типа используются по всему определению класса для спецификации типа возврата методов и типов полей и локальных переменных. Например:

```
private T first; // используется переменная типа
```



На заметку! Использование заглавных букв для имен переменных типа — общепринятая практика, как и их краткость. Библиотека Java использует переменную `E` для обозначения типа элемента коллекции, `K` и `V` — для типов ключа и значения в таблице, а `T` (и соседние буквы `S` и `U` при необходимости) — для обозначения "любого типа вообще".

Вы *создаете экземпляр* обобщенного типа, подставляя имя типа вместо переменной типа:

```
Pair<String>
```

Вы можете воспринимать результат как обычный класс с конструкторами:

```
Pair<String>()
Pair<String>(String, String)
```

и методами:

```
String getFirst()
String getSecond()
void setFirst(String)
void setSecond(String)
```

Другими словами, обобщенный класс действует как фабрика обычных классов.

Программа в листинге 12.1 запускает класс `Pair` в работу. Статический метод `minmax()` проходит по элементам массива и одновременно вычисляет минимальное и максимальное значения. Он использует объект `Pair` для возврата результата. Напомним, что метод `compareTo()` сравнивает две строки, возвращая `0`, если строки идентичны, отрицательное число — если первая строка идет перед второй в алфавитном порядке, и положительное — в противоположном случае.



На заметку! На первый взгляд обобщенные классы Java похожи на шаблонные классы C++. Единственное очевидное отличие состоит в том, что в Java не предусмотрено специальное ключевое слово `template`. Однако, как вы увидите далее в этой главе, между этими двумя механизмами есть тонкое различие.

Листинг 12.1. Содержимое файла `PairTest.java`

```
1. /**
2.  * @version 1.00 2004-05-10
3.  * @author Cay Horstmann
4.  */
5. public class PairTest1
6. {
7.  public static void main(String[] args)
8.   {
9.      String[] words = { "Mary", "had", "a", "little", "lamb" };
10.     Pair<String> mm = ArrayAlg.minmax(words);
11.     System.out.println("min = " + mm.getFirst());
12.     System.out.println("max = " + mm.getSecond());
13.  }
14.}
15.class ArrayAlg
16.{
17./**
18. * Получает минимальное и максимальное значение массива строк.
19. * @param a Массив строк
20. * @return a Пара минимального и максимального значений или null,
21. * если a пуст
22. */
23.  public static Pair<String> minmax(String[] a)
24.  {
25.     if (a == null || a.length == 0) return null;
26.     String min = a[0];
27.     String max = a[0];
28.     for (int i = 1; i < a.length; i++)
29.     {
30.         if (min.compareTo(a[i]) > 0) min = a[i];
31.         if (max.compareTo(a[i]) < 0) max = a[i];
32.     }
33.     return new Pair<String>(min, max);
34.  }
35.}
```

Обобщенные методы

В предыдущем разделе вы видели, как определяется обобщенный класс. Вы можете также определить отдельный метод с параметрами типа.

```
class ArrayAlg
{
  public static <T> T getMiddle(T[] a)
  {
    return a[a.length / 2];
  }
}
```

Этот метод определен внутри обычного класса, а не внутри обобщенного. Однако этот метод — обобщенный, на что указывают угловые скобки и переменная типа. Обратите внимание, что переменная типа вставляется после модификаторов (`public static` в нашем случае) и перед типом возврата.

Вы можете определять обобщенные методы как внутри обычных классов, так и внутри обобщенных. Когда вызывается обобщенный метод, вы можете передать ему действительные типы, заключая их в угловые скобки перед именем метода:

```
String[] names = { "John", "Q.", "Public" };
String middle = ArrayAlg.<String>getMiddle(names);
```

В этом случае (как и в большинстве случаев) вы можете пропустить параметр типа `String` при вызове метода. У компилятора есть достаточно информации, чтобы предположить, какой метод вам нужен. Он сравнивает тип `names` (т.е. `String[]` с типом `T[]`) и соображает, что `T` должно быть `String`. То есть, вы можете просто вызвать так:

```
String middle = ArrayAlg.getMiddle(names);
```

Почти во всех случаях определение типа в обобщенных методах работает гладко. Иногда компилятор ошибается и вам приходится расшифровывать его сообщение об ошибке. Рассмотрим следующий пример:

```
double middle = ArrayAlg.getMiddle(3.14, 1729, 0);
```

Сообщение об ошибке будет таким: `found: java.lang.Number&java.lang.Comparable<? extends java.lang.Number&java.lang.Comparable<?>>, required: double`. Позднее в этой главе вы узнаете, как расшифровывается объявление типа "found". Вкратце скажем, что компилятор выполняет автоматическую упаковку (autoboxing) параметра в один объект `Double` и два объекта `Integer`, а затем пытается найти общий супертип для этих классов. Он находит их два: `Number` и интерфейс `Comparable`, который сам по себе является обобщенным типом. В этом случае решение заключается в передаче всех параметров как значений `double`.



Совет. Петер Ван Дер Ахе (Peter von der Ahe) рекомендует следующий трюк, если вы хотите увидеть, какой тип компилятор выводит при вызове обобщенного метода: намеренно допустите ошибку и изучите полученное в результате сообщение об ошибке. Например, рассмотрим вызов `ArrayAlg.getMiddle("Hello", 0, null)`. Присвоим результат `JButton`, что не может быть правильно. Мы получим такое сообщение об ошибке:

```
java.lang.Object&java.io.Serializable&java.lang.Comparable<? extends
java.lang.Object&java.io.Serializable&java.lang.Comparable<?>>."
```

Говоря просто, вы можете присвоить результат объектам типа `Object`, `Serializable` или `Comparable`.



На заметку! В C++ вы помещаете параметр типа после имени метода. Это может привести к неприятной неоднозначности при разборе кода. Например, `g(f<a,b>(c))` может означать "вызвать `g` с результатом вызова `f<a,b>(c)`", или же "вызвать `g` с двумя булевскими значениями `f<a` и `b>(c)`".

Ограничения переменных типов

Иногда класс или метод нуждается в наложении ограничений на переменные типов. Приведем типичный пример. Нужно вычислить минимальный элемент массива:

```
class ArrayAlg
{
  public static <T> T min(T[] a) // почти правильно
  {
    if (a == null || a.length == 0) return null;
    T smallest = a[0];
    for (int i = 1; i < a.length; i++)
       if (smallest.compareTo(a[i]) > 0) smallest = a[i];
    return smallest;
  }
}
```

Но здесь есть проблема. Взгляните на код метода `min()`. Переменная `smallest` имеет тип `T`, а это означает, что она может быть объектом произвольного класса. Откуда мы знаем, имеет ли класс `T` метод `compareTo()`?

Решение заключается в том, чтобы наложить ограничение на тип `T`, чтобы вместо него можно было подставлять только класс, реализующий `Comparable` — стандартный интерфейс с единственным методом `compareTo()`. Это делается добавлением ограничения (`bound`) переменной типа `T`:

```
public static <T extends Comparable> T min(T[] a) . . .
```

В действительности интерфейс `Comparable` сам является обобщенным типом. Пока мы проигнорируем эту сложность и предупреждения, которые генерирует компилятор. Далее в разделе "Подстановочные типы" мы обсудим, как правильно использовать параметры типа с интерфейсом `Comparable`.

Теперь обобщенный метод `min()` может вызываться только с массивами классов, реализующих интерфейс `Comparable`, таких как `String`, `Date` и т.п. Вызов `min()` с массивом `Rectangle` даст ошибку во время компиляции, поскольку класс `Rectangle` не реализует `Comparable`.



На заметку! В C++ вы не можете ограничить типы в параметрах шаблонов. Если программист создаст экземпляр шаблона с неправильным типом, сообщение об ошибке (часто невнятное) появится в шаблонном коде.

Вас может удивить, почему здесь используется ключевое слово `extends` вместо `implements`, ведь `Comparable` — это интерфейс? Обозначение

```
<T extends Ограничивающий_тип>
```

говорит о том, что T должен быть *подтипом* ограничивающего типа. Ключевое слово `extends` выбрано потому, что это резонное приближение концепции подтипа, и проектировщики Java не хотели добавлять в язык новое ключевое поле (такое как `sub`).

Переменная типа или тип подстановки могут иметь несколько ограничений. Например:

```
T extends Comparable & Serializable
```

Ограничивающие типы разделяются знаком &, потому что запятые используются для разделения переменных типа.

Как и в наследовании Java, вы можете иметь столько интерфейсных подтипов, сколько хотите, но только один из ограничивающих типов может быть классом. Если вы используете для ограничения класс, он должен быть первым в списке ограничений.

В следующем примере программы (листинг 12.2) мы перепишем метод minmax(), сделав его обобщенным. Метод вычисляет минимум и максимум в обобщенном массиве, возвращая Pair<T>.

Листинг 12.2. Содержимое файла PairTest2.java

```
1. import java.util.*;
2. /**
3.  * @version 1.00 2004-05-10
4.  * @author Cay Horstmann
5.  */
6. public class PairTest2
7. {
8.      public static void main(String[] args)
9.      {
10.        GregorianCalendar[] birthdays =
11.        {
12.          new GregorianCalendar(1906, Calendar.DECEMBER, 9), // G. Hopper
13.          new GregorianCalendar(1815, Calendar.DECEMBER, 10), // A. Lovelace
14.          new GregorianCalendar(1903, Calendar.DECEMBER, 3), // J. von Neumann
15.          new GregorianCalendar(1910, Calendar.JUNE, 22), // K. Zuse
16.        };
17.        Pair<GregorianCalendar> mm = ArrayAlg.minmax(birthdays);
18.        System.out.println("min = " + mm.getFirst().getTime());
19.        System.out.println("max = " + mm.getSecond().getTime());
20.     }
21. }
22. class ArrayAlg
23. {
24.   /**
25.    * Получает минимум и максимум из массива объектов типа T.
26.    * @param a Массив объектов типа T
27.    * @return Пара с минимальным и максимальным значениями
28.    *         или null, если a пуст или равен null
29.    */
30.   public static <T extends Comparable> Pair<T> minmax(T[] a)
31.   {
32.      if (a == null || a.length == 0) return null;
33.      T min = a[0];
34.      T max = a[0];
35.      for (int i = 1; i < a.length; i++)
36.      {
37.             if (min.compareTo(a[i]) > 0) min = a[i];
38.             if (max.compareTo(a[i]) < 0) max = a[i];
39.      }
40.      return new Pair<T>(min, max);
41.   }
42. }
```

Обобщенный код и виртуальная машина

Виртуальная машина не имеет дела с объектами обобщенных типов — все объекты принадлежат обычным классам. Ранние версии реализаций обобщения могли даже компилировать программу, использующую обобщения в файлы классов, которые могла исполнять виртуальная машина версии 1.0! Такая обратная совместимость позднее, в процессе работы над средствами обобщения Java, была отброшена. Если вы применяете компилятор Sun для компиляции кода, использующего средства обобщения Java, то результирующие файлы классов *не* будут выполняться на виртуальных машинах версий, предшествующих 5.0.



На заметку! Если вы хотите воспользоваться преимуществами обобщений, сохраняя совместимость байт-кода с более старыми виртуальными машинами, загляните на `http://sourceforge.net/projects/retroweaver`. Программа Retroweaver перепишет файлы классов, так что они будут совместимыми со старыми виртуальными машинами.

Всякий раз при определении обобщенного типа автоматически создается соответствующий ему "сырой" (raw) тип. Имя этого типа совпадает с именем обобщенного типа с удаленными параметрами типа. Переменные типа удаляются и заменяются ограничивающими типами (или типом `Object`, если переменная не имеет ограничений).

Например, "сырой" тип для `Pair<T>` выглядит, как показано ниже.

```
public class Pair
{
    public Pair(Object first, Object second)
    {
        this.first = first;
        this.second = second;
    }
    public Object getFirst() { return first; }
    public Object getSecond() { return second; }
    public void setFirst(Object newValue) { first = newValue; }
    public void setSecond(Object newValue) { second = newValue; }
    private Object first;
    private Object second;
}
```

Поскольку `T` — неограниченная переменная типа, она просто заменяется `Object`.

В результате получается обычный класс вроде тех, что вы могли реализовать раньше, до появления средств обобщений в язык программирования Java.

Ваши программы содержат разные варианты `Pair`, такие как `Pair<String>` или `Pair<GregorianCalendar>`, но такая "подчистка" превращает их в "сырой" тип `Pair`.



На заметку! В этом отношении обобщения Java очень отличаются от шаблонов C++. C++ производит разные типы для каждого экземпляра шаблона — этот феномен называется "разбуханием шаблонного кода". Java не страдает от этой проблемы.

"Сырой" тип заменяет переменные типов первым ограничением или же `Object`, если никаких ограничений не предусмотрено. Например, переменная типа в классе `Pair<T>` не имеет явных ограничений, а потому сырой тип заменяет `T` на `Object`.

Предположим, что объявлен немного отличающийся тип:

```
public class Interval<T extends Comparable & Serializable> implements Serializable
{
    public Interval(T first, T second)
    {
        if (first.compareTo(second) <= 0) { lower = first; upper = second; }
        else { lower = second; upper = first; }
    }
    . . .
    private T lower;
    private T upper;
}
```

Сырой тип `Interval` выглядит следующим образом:

```
public class Interval implements Serializable
{
    public Interval(Comparable first, Comparable second) { . . . }
    . . .
    private Comparable lower;
    private Comparable upper;
}
```



На заметку! Вы можете спросить, что случится, если поменять местами ограничения: `class Interval<Serializable & Comparable>`. В этом случае сырой тип заменит `T` на `Serializable`, и компилятор вставит, где нужно, приведения к `Comparable`. Поэтому для эффективности вы должны помещать пустые интерфейсы (т.е. интерфейсы без методов) в конец списка ограничений.

Трансляция обобщенных выражений

Когда ваша программа вызывает обобщенный метод, компилятор вставляет приведения, когда удаляется возвращаемый тип. Например, рассмотрим следующую последовательность операторов:

```
Pair<Employee> buddies = . . .;
Employee buddy = buddies.getFirst();
```

Подчистка `getFirst()` приведет к возврату типа `Object`. Компилятор автоматически вставит приведение к `Employee`. То есть компилятор транслирует вызов метода в две инструкции виртуальной машины.

- Вызов сырого метода `Pair.getFirst()`.
- Приведение возвращенного объекта `Object` к типу `Employee`.

Приведения также вставляются при обращении к обобщенному полю. Предположим, что поля `first` и `second` были бы общедоступными, т.е. `public`. (Возможно, это нехороший стиль программирования, но вполне допустимый в Java.) Тогда в оператор

```
Employee buddy = buddies.first;
```

в результирующем байт-коде также будут вставлены приведения.

Трансляция обобщенных методов

Подчистка типов также происходит и с обобщенными методами. Программисты обычно воспринимают обобщенные методы вроде этого:

```
public static <T extends Comparable> T min(T[] a)
```

как целое семейство методов. Но после подчистки остается только один метод:

```
public static Comparable min(Comparable[] a)
```

Обратите внимание, что параметр типа `T` удален, а остается только ограничивающий тип `Comparable`.

Подчистка методов вносит пару сложностей. Рассмотрим пример:

```
class DateInterval extends Pair<Date>
{
    public void setSecond(Date second)
    {
        if (second.compareTo(getFirst()) >= 0)
            super.setSecond(second);
    }
    . . .
}
```

Интервал дат задается парой объектов `Date`, и мы хотим переопределить эти методы, чтобы гарантировать, что второе значение никогда не будет меньше первого. Этот класс очищается до следующего:

```
class DateInterval extends Pair // после чистки
{
    public void setSecond(Date second) { . . . }
    . . .
}
```

Возможно, это неожиданно, но существует и другой метод `setSecond()`, унаследованный от `Pair`, а именно:

```
public void setSecond(Object second)
```

Это явно другой метод, поскольку он имеет параметр другого типа — `Object` вместо `Date`. Но он *не должен* быть другим. Рассмотрим последовательность операторов:

```
DateInterval interval = new DateInterval(. . .);
Pair<Date> pair = interval; // OK—присвоение суперклассу
pair.setSecond(aDate);
```

Мы ожидаем, что вызов `setSecond()` является полиморфным и потому будет вызван соответствующий метод. Поскольку `pair` ссылается на объект `DateInterval`, это должен быть `DateInterval.setSecond()`. Но проблема в том, что подчистка типов пересекается с полиморфизмом. Чтобы решить эту проблему, компилятор генерирует *метод-мост* (bridge method) в классе `DateInterval`:

```
public void setSecond(Object second) { setSecond((Date) second); }
```

Чтобы понять, почему это работает, тщательно проследим за выполнением приведенного ниже оператора:

```
pair.setSecond(aDate)
```

Переменная `pair` объявлена с типом `Pair<Date>`, и этот тип имеет только один метод по имени `setSecond`, а именно — `setSecond(Object)`. Виртуальная маши-

на вызывает этот метод на объекте, на который ссылается pair. Этот объект имеет тип DateInterval, потому вызывается метод DateInterval.setSecond(Object). Этот метод и есть синтезированный метод-мост. Он вызывает DateInterval.setSecond(Date), что нам и нужно.

Метод-мост может быть еще более странным. Предположим, что метод DateInterval также переопределяет getSecond:

```
class DateInterval extends Pair<Date>
{
    public Date getSecond() { return (Date) super.getSecond().clone(); }
    . . .
}
```

В очищенном типе есть два метода getSecond:

```
Date getSecond()     // определен в DateInterval
Object getSecond()   // определен в Pair
```

Вы не сможете написать такой Java-код без параметров — было бы некорректно иметь два метода с одинаковыми типами параметров. Однако в виртуальной машине типы параметров *и тип возврата* специфицируют метод. Поэтому компилятор может произвести байт-код двух методов, отличающихся только типом возврата, и виртуальная машина корректно справляется с такой ситуацией.



На заметку! Методы-мосты не ограничиваются обобщенными типами. Мы уже упоминали в главе 5 о том, что, начиная с версии Java SE 5.0, допускается, чтобы метод специфицировал более ограниченный тип возврата, когда он переопределяет другой метод. Например:

```
public class Employee implements Cloneable
{
  public Employee clone() throws CloneNotSupportedException { ... }
}
```

О методах Object.clone и Employee.clone говорят, что это *ковариантные типы*. В действительности класс Employee имеет два таких метода:

```
Employee clone() // определен выше
Object clone() // синтезированный метод-мост, переопределяет Object.clone
```

Синтезированный метод-мост вызывает вновь определенный метод.

В итоге вам нужно помнить следующие факты о трансляции обобщений Java.

- Для виртуальной машины обобщений не существует, есть только обычные классы и методы.
- Все параметры типа заменяются типами-ограничителями.
- Методы-мосты синтезируются для предохранения полиморфизма.
- Приведения вставляются по необходимости для обеспечения безопасности типов.

Вызов унаследованного кода

До появления Java SE 5.0 был написан огромный объем кода Java. Если бы обобщенные классы не могли взаимодействовать с этим кодом, они не получили бы широкого распространения. К счастью, достаточно просто использовать обобщенные классы вместе с их "сырыми" эквивалентами из унаследованного API.

Рассмотрим конкретный пример. Чтобы установить метки JSlider, вы используете метод

```
void setLabelTable(Dictionary table)
```

В главе 9 мы применяли следующий код для наполнения таблицы меток:

```
Dictionary<Integer, Component> labelTable = new Hashtable<Integer, Component>();
labelTable.put(0, new JLabel(new ImageIcon("nine.gif")));
labelTable.put(20, new JLabel(new ImageIcon("ten.gif")));
. . .
slider.setLabelTable(labelTable); // WARNING
```

В Java SE 5.0 классы `Dictionary` и `Hashtable` превратились в обобщенные классы. Поэтому можно сформировать `Dictionary<Integer, Component>` вместо использования "сырого" `Dictionary`. Однако когда вы передаете объект `Dictionary<Integer, Component>` методу `setLabelTable`, компилятор выдает предупреждение:

```
Dictionary<Integer, Component> labelTable = . . .;
slider.setLabelTable(labelTable); // предупреждение
```

В конце концов, компилятор не знает о том, что `setLabelTable()` может делать с объектом `Dictionary`. Этот метод может заменить все ключи строками. Это нарушит гарантию того, что ключи имеют тип `Integer`, и тогда будущие операции могут вызвать исключения приведения.

С этим предупреждением мало что можно сделать, кроме как обдумать его и спросить себя, что, скорее всего, `JSlider` собирается делать с объектом `Dictionary`? В нашем случае вполне ясно, что `JSlider` только читает информацию, так что мы можем проигнорировать предупреждение.

Теперь рассмотрим противоположный случай, когда вы получаете объект "сырого" типа от унаследованного класса. Вы можете присвоить его переменной параметризованного типа, но конечно, при этом получите предупреждение. Например:

```
Dictionary<Integer, Components> labelTable =
    slider.getLabelTable(); // предупреждение
```

Все хорошо — просмотрите предупреждение и убедитесь, что таблица меток действительно содержит объекты `Integer` и `Component`. Конечно, абсолютной гарантии нет. Злоумышленник может инсталлировать другой `Dictionary` в `JSlider`. Но, опять-таки, эта ситуация на хуже той, что была до версии Java SE 5.0. В худшем случае ваша программа сгенерирует исключение.

После того, как вы обратили внимание на предупреждение, вы можете использовать *аннотацию* для того, чтобы оно исчезло. Аннотацию следует поместить перед методом, чей код генерирует предупреждение, как показано ниже:

```
@SuppressWarnings("unchecked")
public void configureSlider() { . . . }
```

К сожалению, эта аннотация отключает проверку всего кода внутри метода. Будет хорошей идеей изолировать потенциально небезопасный код в отдельных методах, чтобы его можно было легко просмотреть.



На заметку! `Hashtable` — конкретный подкласс абстрактного класса `Dictionary`. И `Dictionary`, и `Hashtable` объявлены устаревшими, с тех пор как в Java SE 1.2 для них появилась замена в виде интерфейса `Map` и класса `HashMap`. Несмотря на это, упомянутые устаревшие классы до сих пор живут и здравствуют. В конце концов, класс `JSlider` был добавлен только в Java 1.3. Знали ли его разработчики о классе `Map` в то время? Можно ли рассчитывать, что они адаптируют его к обобщениям в будущем? В принципе, подобное всегда происходит с унаследованным кодом.

Ограничения и лимиты

В следующих разделах мы обсудим ряд ограничений, которые следует учитывать при работе с обобщениями Java. Большинство этих ограничений являются следствием "подчистки" типов.

Параметры типа не могут принимать примитивные типы

Нельзя подставить примитивные тип вместо параметра типа. То есть не бывает `Pair<double>`, а только `Pair<Double>`. Причина, конечно же, в "подчистке" типов. После подчистки класс `Pair` имеет поля типа `Object`, и вы не можете использовать их для хранения значений `double`.

Это досадно, но такое ограничение согласуется с особым статусом примитивных типов в языке Java. Этот недостаток не носит фатального характера — существует всего восемь примитивных типов, и вы всегда можете обработать их с помощью отдельных классов и методов, когда нельзя подставить вместо них типы-оболочки.

Исследование типов времени выполнения работает только с "сырыми" типами

В виртуальной машине объекты всегда имеют определенный не обобщенный тип. Поэтому все динамические опросы типов порождают только "сырой" тип. Например, следующий оператор в действительности только проверяет, является ли `a` экземпляром `Pair` любого типа:

```
if (a instanceof Pair<String>) // то же, что и instanceof Pair
```

То же справедливо и в отношении оператора

```
if (a instanceof Pair<T>) // T игнорируется
```

и в отношении приведения

```
Pair<String> p = (Pair<String>) a;
// предупреждение — проверяет только, что a является Pair
```

Чтобы напомнить вам об опасности, компилятор выдает предупреждения всякий раз, когда вы используете операцию `instanceof` или выражения приведения с обобщенными типами.

В том же духе метод `getClass()` всегда возвращает сырой тип. Например:

```
Pair<String> stringPair = . . .;
Pair<Employee> employeePair = . . .;
if (stringPair.getClass() == employeePair.getClass()) // эквивалентны
```

Сравнение дает `true`, потому что оба вызова `getClass` вернут `Pair.class`.

Генерировать и перехватывать исключения-экземпляры обобщенного класса нельзя

Вы не можете использовать переменную типа в конструкции `catch`. Например, следующий метод компилироваться не будет:

```
public static <T extends Throwable> void doWork(Class<T> t)
{
    try
    {
```

```
            выполнять работу
        }
        catch (T e) // Ошибка — нельзя перехватить переменную типа
        {
            Logger.global.info(...)
        }
    }
```

Однако можно использовать переменные типа в спецификациях исключений. Приведенный ниже метод вполне допустим:

```
public static <T extends Throwable> void doWork(T t) throws T // OK
{
    try
    {
        выполнить какие-то действия
    }
    catch (Throwable realCause)
    {
        t.initCause(realCause);
        throw t;
    }
}
```

Массивы параметризованных типов не разрешены

Вы не можете объявить массив параметризованных типов наподобие

```
Pair<String>[] table = new Pair<String>[10]; // Ошибка
```

Что же здесь не так? После подчистки типом таблицы становится `Pair[]`. Вы можете преобразовать его в `Object[]`:

```
Object[] objarray = table;
```

Массив запоминает тип своих компонентов и генерирует исключение `ArrayStoreException`, если вы пытаетесь сохранить элемент неверного типа:

```
objarray[0] = "Hello"; // Ошибка — тип компонента - Pair
```

Но подчистка изменяет этот механизм для обобщенных типов неэффективно. Следующее присваивание пройдет проверку сохранения в массиве, но даст ошибку типа:

```
objarray[0] = new Pair<Employee>();
```

Поэтому массивы параметризованных типов не допускаются.



Совет. Если вам нужно хранить объекты параметризованных типов, просто используйте `ArrayList`: `ArrayList<Pair<String>>` — это безопасно и эффективно.

Нельзя создавать экземпляры переменных типов

Переменные типа нельзя использовать в выражениях вроде `new T(...)`, `new T[...]` или `T.class`. Например, следующий конструктор `Pair<T>` недопустим:

```
public Pair() { first = new T(); second = new T(); } // ОШИБКА
```

Подчистка типа может изменить `T` на `Object`, а вы, конечно, не хотите вызывать `new Object()`.

В качестве обходного пути можно конструировать обобщенные объекты через рефлексию, вызывая метод Class.newInstance().

К сожалению, детали достаточно сложны. Вы не можете поступить так:

```
first = T.class.newInstance(); // ОШИБКА
```

Выражение T.class недопустимо. Вместо этого вы должны разработать API, чтобы иметь дело с объектом Class, как показано ниже:

```
public static <T> Pair<T> makePair(Class<T> cl)
{
    try { return new Pair<T>(cl.newInstance(), cl.newInstance()) }
    catch (Exception ex) { return null; }
}
```

Этот метод может быть вызван следующим образом:

```
Pair<String> p = Pair.makePair(String.class);
```

Обратите внимание, что класс Class сам является обобщенным. Например, String.class — это экземпляр (на самом деле единственный) Class<String>. Поэтому метод makePair может выводить тип создаваемой пары.

Вы не можете конструировать обобщенный массив:

```
public static <T extends Comparable> T[] minmax(T[] a) { T[] mm =
     new T[2]; . . . } // ОШИБКА
```

Подчистка типа может заставить этот метод всегда конструировать массив Object[2].

Если массив используется только как приватное поле экземпляра класса, вы можете объявлять массив как Object[] и использовать приведения при извлечении элементов. Например, класс ArrayList может быть реализован так:

```
public class ArrayList<E>
{
    private Object[] elements;
    @SuppressWarnings("unchecked") public E get(int n) { return (E)
    elements[n]; }
    public void set(int n, E e) { elements[n] = e; } // приведение не нужно
    . . .
}
```

Действительная реализация не так ясна:

```
public class ArrayList<E>
{
    private E[] elements;
    public ArrayList() { elements = (E[]) new Object[10]; }
    . . .
}
```

Здесь приведение к E[] — полная ложь, но подчистка типа делает ее незаметной. Эта техника не работает с нашим методом minmax, поскольку мы возвращаем массив T[], и ошибка времени выполнения возникнет, если мы будет хитрить с типом. Предположим, что мы реализуем

```
public static <T extends Comparable> T[] minmax(T[] a)
{
    Object[] mm = new Object[2];
    . . .;
    return (T[]) mm; // компилируется с предупреждением
}
```

Тогда вызов

```
String[] ss = minmax("Tom", "Dick", "Harry");
```

компилируется без каких-либо предупреждений. `ClassCastException` случается, когда ссылка `Object[]` присваивается переменной `String[]`.

В этой ситуации вы можете использовать рефлексию и вызвать `Array.newInstance`:

```
public static <T extends Comparable> T[] minmax(T[] a)
{
    T[] mm = (T[]) Array.newInstance(a.getClass().getComponentType(), 2);
    . . .
}
```

Методу `toArray()` класса `ArrayList` не так повезло. Ему нужно произвести массив `T[]`, но у него нет типа компонентов. Поэтому возможны два варианта:

```
Object[] toArray()
T[] toArray(T[] result)
```

Второй метод принимает параметр-массив. Если этот массив достаточно велик, он используется. В противном случае создается новый массив подходящего размера, использующий тип компонента `result`.

Переменные типа в статическом контексте обобщенных классов не разрешены

Вы не можете ссылаться на переменные типа в статических полях или метода. Например, следующая замечательная идея работать не будет:

```
public class Singleton<T>
{
    public static T getSingleInstance() // ОШИБКА
    {
           if (singleInstance == null) конструирует новый экземпляр T
                  return singleInstance;
    }
    private static T singleInstance; // ОШИБКА
}
```

Если бы такое было возможно, то программа должна была бы сконструировать `Singleton<Random>` для разделения общего генератора случайных чисел, и `Singleton <JFileChooser>` — для разделения диалога выбора файлов. Но это не может работать. После подчистки типов остается только один класс `Singleton` и только одно поле `singleInstance`. По этой причине статические поля и методы с переменными типа просто объявлены “вне закона”.

Остерегайтесь конфликтов после подчистки

Не допускается создавать условия, которые вызовут конфликты после подчистки обобщенных типов. Вот пример. Предположим, что мы добавляем метод `equals` в класс `Pair`:

```
public class Pair<T>
{
    public boolean equals(T value)
      { return first.equals(value) && second.equals(value); }
    . . .
}
```

Рассмотрим Pair<String>. Концептуально он имеет два метода equals:

```
boolean equals(String) // определен в Pair<T>
boolean equals(Object) // унаследован от Object
```

Но интуиция приводит к заблуждению. Подчистка метода

```
boolean equals(T)
```

даст

```
boolean equals(Object)
```

что конфликтует с методом Object.equals.

Решение, конечно же, заключается в переименовании метода-виновника.

Спецификация обобщений объявляет другое правило: для поддержки трансляции подчисткой налагается ограничение, что класс или переменная типа не могут одновременно быть подтипами двух интерфейсных типов, которые представляют собой разные параметризации одного и того же интерфейса. Например, следующий код неверен:

```
class Calendar implements Comparable<Calendar> { . . . }
class GregorianCalendar extends Calendar implements Comparable<GregorianCalendar>
    { . . . } // ОШИБКА
```

GregorianCalendar затем должен реализовывать одновременно Comparable <Calendar> и Comparable<GregorianCalendar>, представляющие разные параметризации одного и того же интерфейса.

Непонятно, что это ограничение должно делать с подчисткой типов. В конце концов, не обобщенная версия совершенно законна:

```
class Calendar implements Comparable { . . . }
class GregorianCalendar extends Calendar implements Comparable { . . . }
```

Причина намного тоньше. Может случиться конфликт с синтезированными методами-мостами. Класс, реализующий Comparable<X>, получает метод-мост

```
public int compareTo(Object other) { return compareTo((X) other); }
```

Вы не можете иметь два таких метода с разным типом X.

Правила наследования обобщенных типов

При работе с обобщенными классами вам нужно выучить несколько правил, касающихся наследования и подтипов. Начнем с ситуации, которую многие программисты считают интуитивно понятной. Рассмотрим класс и подкласс вроде Employee и Manager. Является ли Pair<Manager> подклассом Pair<Employee>? Может, это будет сюрпризом, но нет. Например, следующий код не скомпилируется:

```
Manager[] topHonchos = . . .;
Pair<Employee> result = ArrayAlg.minmax(topHonchos); // ОШИБКА
```

Метод minmax возвращает Pair<Manager>, а не Pair<S> и Pair<T> — независимо от того, как S и T соотносятся между собой (рис. 12.1).

Это может показаться суровым ограничением, но оно необходимо для безопасности типов. Представим, что мы могли бы преобразовать Pair<Manager> в Pair<Employee>. Рассмотрим следующий код:

```
Pair<Manager> managerBuddies = new Pair<Manager>(ceo, cfo);
Pair<Employee> employeeBuddies =
   managerBuddies; // недопустимо, но предположим, что было бы можно
employeeBuddies.setFirst(lowlyEmployee);
```

Ясно, что последний оператор недопустим. Но `employeeBuddies` и `managerBuddies` ссылаются на один и тот же объект. Теперь мы трактуем высшее руководство как рядовых сотрудников, что невозможно для `Pair<Manager>`.

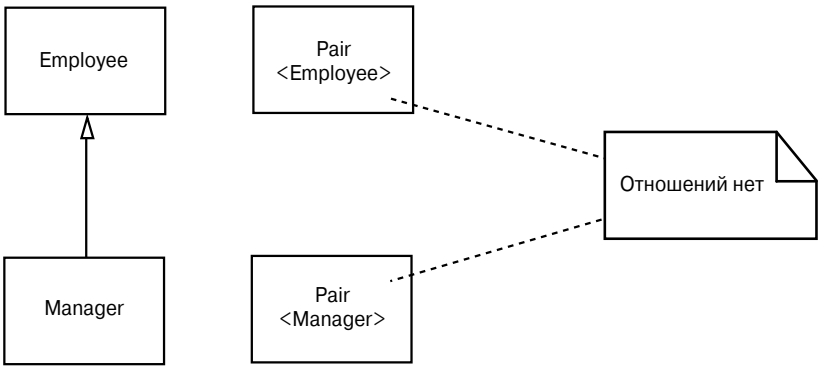


Рис. 12.1. Между классами `Pair` отношение наследования отсутствует



На заметку! Вы только что увидели важное отличие между обобщенными типами и массивами Java. Массив `Manager[]` можно присвоить переменной типа `Employee[]`:

```
Manager[] managerBuddies = { ceo, cfo };
Employee[] employeeBuddies = managerBuddies; // OK
```

Однако массивы снабжены дополнительной защитой. Если вы попытаетесь сохранить рядового сотрудника в `employeeBuddies[0]`, то виртуальная машина сгенерирует исключение `ArrayStoreException`.

Вы всегда можете преобразовать параметризованный тип в "сырой" тип. Например, `Pair<Employee>` — это подтип "сырого" типа `Pair`. Такое преобразование необходимо для взаимодействия с унаследованным кодом.

Можно ли конвертировать сырой тип и затем вызвать ошибку типа? К сожалению, да. Рассмотрим пример:

```
Pair<Manager> managerBuddies = new Pair<Manager>(ceo, cfo);
Pair rawBuddies = managerBuddies; // OK
rawBuddies.setFirst(new File(". . ")); // только предупреждение при компиляции
```

Эту звучит пугающе. Однако следует иметь в виду, что в данном отношении ситуация не хуже, чем она была в старых версиях Java. Безопасность виртуальной машины не безгранична. Когда внешний объект извлекается методом `getFirst()` и присваивается переменной `Manager`, генерируется исключение `ClassCastException` — как и в старые добрые времена. Вы просто лишаетесь дополнительной защиты, которую обычно предоставляет обобщенное программирование.

И, наконец, обобщенные классы могут расширять или реализовывать другие обобщенные классы. В этом отношении они не отличаются от обычных классов. Например, класс `ArrayList<T>` реализует интерфейс `List<T>`. Это значит, что `ArrayList<Manager>` может быть преобразован в `List<Manager>`. Однако, как вы только что видели, `ArrayList<Manager>` — это *не* `ArrayList<Employee>` или `List<Employee>`. На рис. 12.2 показаны все эти отношения.

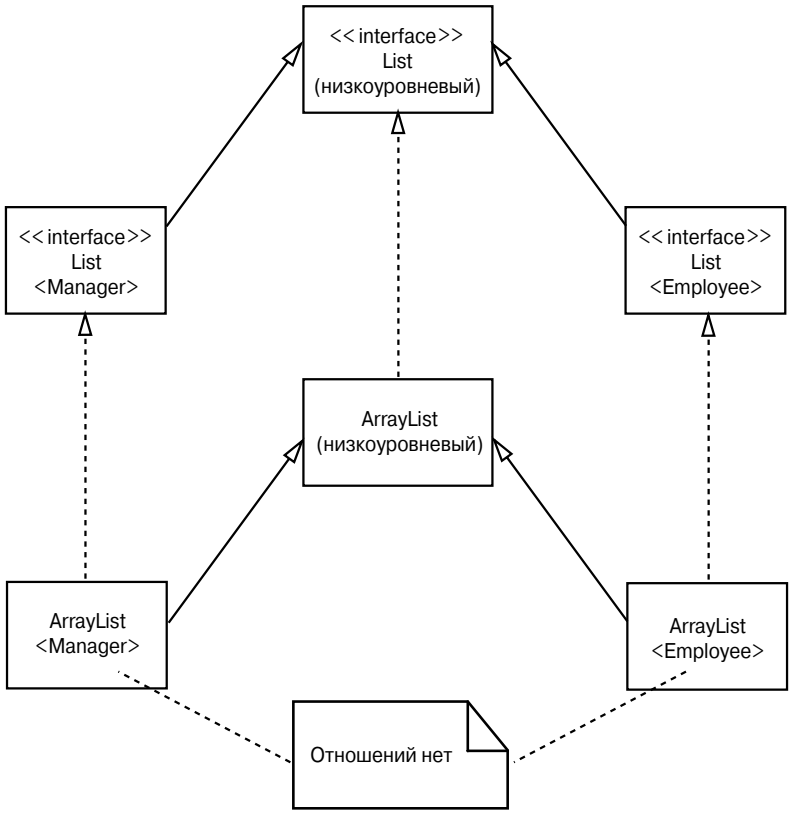


Рис. 12.2. Отношения подтипов между обобщенными списочными типами

Подстановочные типы

Исследователям систем типов уже давно известно, что жесткие системы обобщенных типов использовать довольно неприятно. Проектировщики Java придумали оригинальный (но безопасный) "запасной выход": подстановочные (wildcard) типы. Например, следующий подстановочный тип

```
Pair<? extends Employee>
```

обозначает любой обобщенный тип `Pair`, чей параметр типа является подклассом `Employee`, такой как `Pair<Manager>`, но не `Pair<String>`.

Скажем, вы хотите написать метод, который печатает пары сотрудников, вроде такого:

```
public static void printBuddies(Pair<Employee> p)
{
   Employee first = p.getFirst();
   Employee second = p.getSecond();
   System.out.println(first.getName() + " and " + second.getName() + " are buddies.");
}
```

Как вы видели в предыдущем разделе, передать Pair<Manager> методу невозможно, что довольно стеснительно. Однако есть простое решение — использовать подстановочный тип:

```
public static void printBuddies(Pair<? extends Employee> p)
```

Тип Pair<Manager> является подтипом Pair<? extends Employee> (рис. 12.3).

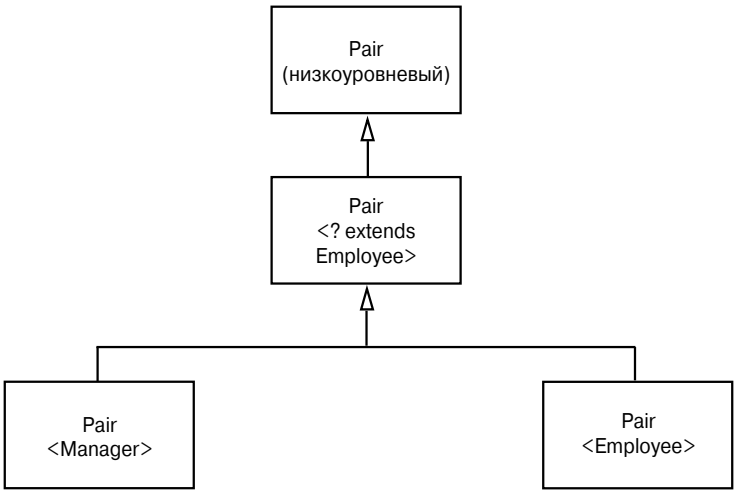


Рис. 12.3. Отношения подтипов с подстановками

Можно ли использовать подстановки, чтобы разрушить Pair<Manager> через ссылку Pair<? extends Employee>?

```
Pair<Manager> managerBuddies = new Pair<Manager>(ceo, cfo);
Pair<? extends Employee> wildcardBuddies = managerBuddies; // OK
wildcardBuddies.setFirst(lowlyEmployee); // ошибка времени компиляции
```

Какое-либо повреждение невозможно. Вызов setFirst() даст ошибку, связанную с типом. Чтобы понять, почему, внимательно рассмотрим тип Pair<? extends Employee>. Его методы выглядят следующим образом:

```
? extends Employee getFirst()
void setFirst(? extends Employee)
```

Это делает невозможным вызов метода setFirst(). Компилятор знает только, что ему нужен какой-нибудь подтип Employee, но не знает какой именно. Он отвергает любой специфический тип — в конце концов, ? может и не соответствовать ему.

С методом getFirst() такой проблемы нет. Вполне законно присвоить возвращаемый тип getFirst() ссылке на Employee.

В этом заключается ключевая идея, лежащая в основе ограниченных подстановок. У нас теперь есть способ различать безопасные методы доступа (accessor) от небезопасных изменяющих методов.

Ограничения супертипа для подстановок

Ограничения подстановок подобны ограничениям переменных типов, однако у них есть дополнительная возможность — вы можете специфицировать *ограничение супертипа*:

```
? super Manager
```

Эта подстановка ограничена всеми супертипами Manager. (Большая удача, что существующее ключевое слово super так точно описывает отношения.)

Зачем это может понадобиться? Подстановка с ограничением супертипа дает вам поведение, противоположное тому, что описано выше в разделе "Подстановочные типы". Вы можете передавать параметры методам, но не можете использовать возвращаемые значения. Например, Pair<? super Manager> имеет методы:

```
void setFirst(? super Manager)
? super Manager getFirst()
```

Компилятор не знает точного типа метода setFirst, но может вызвать его с любым объектом типа Manager, Employee или Object, но не с подтипами, такими как Executive. Однако если вы вызываете getFirst, то нет никаких гарантий относительно типа возвращаемого объекта. Вы можете лишь присвоить его Object.

Приведем типичный пример. У нас есть массив менеджеров, и мы хотим вставить менеджера с минимальным и максимальным размером премии в объект Pair. Какого рода Pair? Будет справедливо, если это окажется Pair<Employee> или же Pair<Object> (рис. 12.4). Следующий метод примет любой подходящий Pair:

```
public static void minmaxBonus(Manager[] a, Pair<? super Manager> result)
{
    if (a == null || a.length == 0) return;
    Manager min = a[0];
    Manager max = a[0];
    for (int i = 1; i < a.length; i++)
    {
        if (min.getBonus() > a[i].getBonus()) min = a[i];
        if (max.getBonus() < a[i].getBonus()) max = a[i];
    }
    result.setFirst(min);
    result.setSecond(max);
}
```

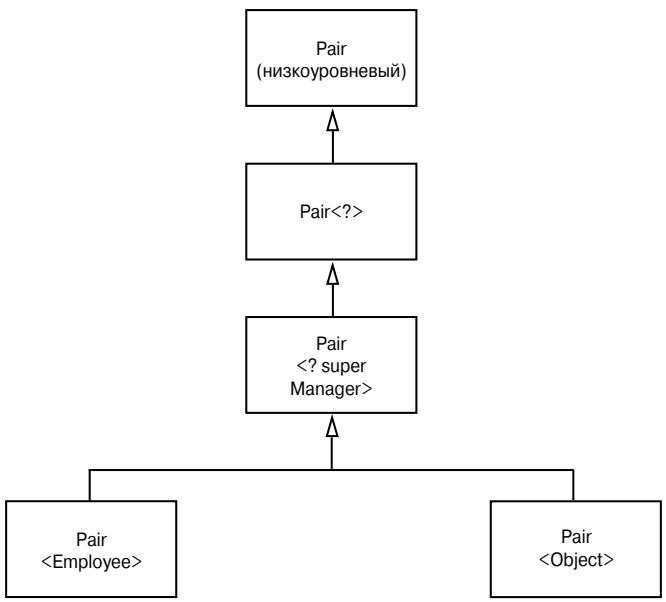


Рис. 12.4. Подстановка с ограничением супертипа

Интуитивно рассуждая, подстановки с ограничениями супертипа позволяют писать обобщенный объект, а подстановки с ограничениями подтипа — читать из обобщенного объекта.

А вот другое использование ограничений супертипа. Интерфейс `Comparable` сам является обобщенным типом. Он объявлен следующим образом:

```
public interface Comparable<T>
{
    public int compareTo(T other);
}
```

Здесь переменная типа указывает тип параметра `other`. Например, класс `String` реализует `Comparable<String>`, и его метод `compareTo()` объявлен следующим образом:

```
public int compareTo(String other)
```

Это прекрасно — явный параметр имеет корректный тип. До версии Java SE 5.0 `other` был `Object`, и в реализации метода требовалось приведение.

Поскольку `Comparable` — обобщенный тип, возможно, мы сможем лучше выполнить работу с методом `min` в классе `ArrayList`? Мы могли бы объявить его как

```
public static <T extends Comparable<T>> T min(T[] a)
```

Это выглядит более цельно, чем просто использование `T extends Comparable`, и должно хорошо работать со многими классами. Например, если вы вычисляете минимум в массиве `String`, то `T` будет типом `String`, а `String` — подтип `Comparable<String>`. Но мы сталкиваемся с проблемой, обрабатывая массив объектов `GregorianCalendar`. Так случилось, что `GregorianCalendar` — это подкласс `Calendar`, а `Calendar` реализует `Comparable<Calendar>`. Поэтому `GregorianCalendar` реализует `Comparable<Calendar>`, а не `Comparable<GregorianCalendar>`.

В ситуации, подобной этой, на помощь приходят супертипы:

```
public static <T extends Comparable<? super T>> T min(T[] a) . . .
```

Теперь метод `compareTo` имеет следующую форму:

```
int compareTo(? super T)
```

Может быть, он объявлен для приема объекта типа `T` или — например, когда `T` представляет собой `GregorianCalendar` — супертипа `T`. Как бы то ни было, можно безопасно передать объект типа `T` методу `compareTo`.

На непосвященных объявление вроде `<T extends Comparable<? super T>>` может наводить ужас. Это прискорбно, потому что назначение этого объявления — помочь прикладным программистам, исключая ненужные ограничение на параметры вызова. Прикладные программисты, не интересующиеся обобщениями, вероятно, предпочтут поскорее пропустить такие объявления и принять на веру, что разработчики библиотеки знали, что делали. Если же вы — программист, разрабатывающий библиотеку, вам придется использовать подстановки, или же ваши пользователи проклянут вас и станут пробовать в коде разнообразные приведения, пока он не скомпилируется.

Неограниченные подстановки

Вы можете даже использовать подстановки вообще без ограничений, например, `Pair<?>`. На первый взгляд это выглядит идентичным сырому типу `Pair`. На самом деле разница между ними есть. Тип `Pair<?>` имеет методы наподобие приведенных ниже.

```
? getFirst()
void setFirst(?)
```

Значение, возвращаемое `getFirst()`, может быть присвоено только `Object`. Метод `setFirst()` никогда не может быть вызван, даже с параметром `Object`. В этом существенная разница между `Pair<?>` и `Pair`: вы можете вызвать метод `setObject()` "сырого" класса `Pair` с *любым* `Object`.

Но зачем вообще может понадобиться такой непонятный тип? Оказывается, он удобен для очень простых операций. Например, следующий метод проверяет, содержит ли пара заданный объект. Ему никогда не понадобится действительный тип.

```
public static boolean hasNulls(Pair<?> p)
{
    return p.getFirst() == null || p.getSecond() == null;
}
```

Вы могли бы избежать применения подстановочного типа, превратив `contains()` в обобщенный метод:

```
public static <T> boolean hasNulls(Pair<T> p)
```

Однако версия с подстановочным типом выглядит легче читаемой.

Захват подстановок

Давайте напишем метод, обменивающий элементы в паре:

```
public static void swap(Pair<?> p)
```

Подстановка не является переменной типа, так что мы не можем использовать `?` как тип. Другими словами, следующие операторы неверны:

```
? t = p.getFirst(); // ОШИБКА
p.setFirst(p.getSecond());
p.setSecond(t);
```

Это проблема, потому что нам нужно временно запоминать первый элемент, когда мы выполняем обмен. К счастью, у этой проблемы есть интересное решение. Можно написать вспомогательный метод `swapHelper()` — примерно так:

```
public static <T> void swapHelper(Pair<T> p)
{
    T t = p.getFirst();
    p.setFirst(p.getSecond());
    p.setSecond(t);
}
```

Обратите внимание, что `swapHelper()` — обобщенный метод, в то время как `swap()` — нет. Он имеет фиксированный параметр типа `Pair<?>`.

Теперь мы можем вызвать `swapHelper()` из `swap()`:

```
public static void swap(Pair<?> p) { swapHelper(p); }
```

В этом случае параметр `T` метода `swapHelper()` *захватывает подстановку*. Известно, какой тип означает подстановка, и это совершенно определенный тип, а потому определение `<T>swapHelper()` имеет четкий смысл, когда `T` означает тип.

Конечно, в этом случае мы не обязаны использовать подстановку. Мы могли бы напрямую реализовать `<T> void swap(Pair<T> p)` как обобщенный метод без подстановок. Однако рассмотрим следующий пример, в котором подстановочный тип появляется естественным образом в середине вычислений:

```
public static void maxminBonus(Manager[] a, Pair<? super Manager> result)
{
    minmaxBonus(a, result);
    PairAlg.swapHelper(result);
       //OK -- swapHelper захватывает подстановочный тип
}
```

Здесь без механизма захвата подстановки невозможно обойтись.

Захват подстановки допускается в очень ограниченных ситуациях. Компилятор должен иметь возможность гарантировать, что подстановка представляет один, определенный тип. Например, `T` в `ArrayList<Pair<T>>` никогда не сможет захватить подстановку в `ArrayList<Pair<?>>`. Список-массив может содержать два `Pair<?>`, в каждом из которых вместо `?` будет подставлен свой тип.

Тестовая программа в листинге 12.3 подытоживает разнообразные методы, которые мы обсудили в предыдущих разделах, чтобы вы могли увидеть их в контексте.

Листинг 12.3. Содержимое файла `PairTest3.java`

```
1.  import java.util.*;
2.  /**
3.   * @version 1.00 2004-05-10
4.   * @author Cay Horstmann
5.   */
6.  public class PairTest3
7.  {
8.  public static void main(String[] args)
9.  {
10.     Manager ceo = new Manager("Gus Greedy", 800000, 2003, 12, 15);
11.     Manager cfo = new Manager("Sid Sneaky", 600000, 2003, 12, 15);
12.     Pair<Manager> buddies = new Pair<Manager>(ceo, cfo);
13.     printBuddies(buddies);
14.     ceo.setBonus(1000000);
15.     cfo.setBonus(500000);
16.     Manager[] managers = { ceo, cfo };
17.     Pair<Employee> result = new Pair<Employee>();
18.     minmaxBonus(managers, result);
19.     System.out.println("first: " + result.getFirst().getName()
20.            + ", second: " + result.getSecond().getName());
21.     maxminBonus(managers, result);
22.     System.out.println("first: " + result.getFirst().getName()
23.            + ", second: " + result.getSecond().getName());
24. }
25. public static void printBuddies(Pair<? extends Employee> p)
26. {
27.     Employee first = p.getFirst();
28.     Employee second = p.getSecond();
29.     System.out.println(first.getName() + " and " + second.getName()
30.        + " are buddies.");
31. }
32. public static void minmaxBonus(Manager[] a, Pair<? super Manager> result)
33. {
34.     if (a == null || a.length == 0) return;
35.     Manager min = a[0];
36.     Manager max = a[0];
37.     for (int i = 1; i < a.length; i++)
38.     {
```

```
39.          if (min.getBonus() > a[i].getBonus()) min = a[i];
40.          if (max.getBonus() < a[i].getBonus()) max = a[i];
41.      }
42.      result.setFirst(min);
43.      result.setSecond(max);
44. }
45. public static void maxminBonus(Manager[] a, Pair<? super Manager> result)
46. {
47.      minmaxBonus(a, result);
48.      PairAlg.swapHelper(result); // OK--swapHelper captures wildcard type
49. }
50. }
51. class PairAlg
52. {
53. public static boolean hasNulls(Pair<?> p)
54. {
55.      return p.getFirst() == null || p.getSecond() == null;
56. }
57. public static void swap(Pair<?> p) { swapHelper(p); }
58. public static <T> void swapHelper(Pair<T> p)
59. {
60.      T t = p.getFirst();
61.      p.setFirst(p.getSecond());
62.      p.setSecond(t);
63. }
64. }
65. class Employee
66. {
67. public Employee(String n, double s, int year, int month, int day)
68. {
69.      name = n;
70.      salary = s;
71.      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
72.      hireDay = calendar.getTime();
73. }
74. public String getName()
75. {
76.      return name;
77. }
78. public double getSalary()
79. {
80.      return salary;
81. }
82. public Date getHireDay()
83. {
84.      return hireDay;
85. }
86. public void raiseSalary(double byPercent)
87. {
88.      double raise = salary * byPercent / 100;
89.      salary += raise;
90. }
91. private String name;
92. private double salary;
93. private Date hireDay;
94. }
```

```
95. class Manager extends Employee
96. {
97.  /**
98.   * @param n Имя сотрудника
99.   * @param s Зарплата
100.  * @param year Год найма
101.  * @param month Месяц найма
102.  * @param day День найма
103.  */
104. public Manager(String n, double s, int year, int month, int day)
105. {
106.     super(n, s, year, month, day);
107.     bonus = 0;
108. }
109. public double getSalary()
110. {
111.     double baseSalary = super.getSalary();
112.     return baseSalary + bonus;
113. }
114. public void setBonus(double b)
115. {
116.     bonus = b;
117. }
118. public double getBonus()
119. {
120.     return bonus;
121. }
122. private double bonus;
123. }
```

Рефлексия и обобщения

Класс `Class` теперь обобщенный. Например, `String.class` — на самом деле объект (фактически, единственный объект) класса `Class<String>`.

Параметр типа удобен, потому что он позволяет методам `Class<T>` быть более точным в отношении возвращаемых типов. Следующие методы `Class<T>` использует преимущество параметра типа:

```
T newInstance()
T cast(Object obj)
T[] getEnumConstants()
Class<? super T> getSuperclass()
Constructor<T> getConstructor(Class... parameterTypes)
Constructor<T> getDeclaredConstructor(Class... parameterTypes)
```

Метод `newInstance` возвращает экземпляр класса, полученный от конструктора по умолчанию. Его тип возврата теперь может быть объявлен как `T` — тот же тип, что и класс, описанный `Class<T>`. Это экономит на приведении.

Метод `cast` возвращает данный объект, теперь объявленный, как имеющий тип `T`, если его тип действительно является подтипом `T`. В противном случае он генерирует исключение `BadCastException`.

Метод `getEnumConstants` возвращает `null`, если этот класс не является классом перечисления (`enum`) или массивом перечислимых значений, о которых известно, что они принадлежат к типу `T`.

И, наконец, методы getConstructor() и getDeclaredConstructor() возвращают объект Constructor<T>. Класс Constructor также стал обобщенным, так что метод newInstance теперь имеет корректный тип возврата.

API java.lang.Class<T> 1.0

- T newInstance() **5.0**

  Возвращает новый экземпляр, созданный конструктором по умолчанию.

- T cast(Object obj) **5.0**

  Возвращает obj, если он null либо может быть преобразован в тип T, иначе генерирует исключение BadCastException.

- T[] getEnumConstants() **5.0**

  Возвращает массив всех значений, если T — перечислимый тип, или null в противном случае.

- Class<? super T> getSuperclass() **5.0**

  Возвращает суперкласс данного класса или null, если T не является классом либо является классом Object.

- Constructor<T> getConstructor(Class... parameterTypes) **5.0**
- Constructor<T> getDeclaredConstructor(Class... parameterTypes) **5.0**

  Получает общедоступный конструктор, либо конструктор с заданными типами параметров.

API java.lang.reflect.Constructor<T> 1.1

- T newInstance(Object... parameters) **5.0**

  возвращает новый экземпляр, созданный конструктором с заданными параметрами

Использование параметров Class<T> для сравнения типов

Иногда бывает полезно сравнить переменные типа параметра Class<T> в обобщенном методе. Ниже приведен канонический пример:

```
public static <T> Pair<T> makePair(Class<T> c) throws InstantiationException,
    IllegalAccessException
{
    return new Pair<T>(c.newInstance(), c.newInstance());
}
```

Если вы вызовете

```
makePair(Employee.class)
```

то Employee.class — это объект типа Class<Employee>. Параметр типа T метода makePair соответствует Employee, и компилятор может отсюда заключить, что метод возвращает Pair<Employee>.

Информация об обобщенных типах в виртуальной машине

Одним из замечательных средств обобщений Java является подчистка обобщенных типов в виртуальной машине. Это покажется неожиданным, но подчищенные классы все еще сохраняют некоторую память о своих обобщенных истоках. Например, "сырой" класс `Pair` знает, что он происходит от обобщенного класса `Pair<T>`, даже несмотря на то, что объект типа `Pair` не может отличить, был он сконструирован как `Pair<String>` или же как `Pair<Employee>`.

Аналогично, рассмотрим метод

```
public static Comparable min(Comparable[] a)
```

который является результатом подчистки

```
public static <T extends Comparable<? super T>> T min(T[] a)
```

Вы можете использовать расширения API рефлексии из Java SE 5.0, чтобы определить следующее.

1. Обобщенный метод имеет параметр по имени `T`.
2. Параметр типа имеет ограниченный подтип, который сам является обобщенным.
3. Ограниченный тип имеет подставляемый параметр.
4. Подставляемый параметр имеет ограниченный супертип.
5. Обобщенный метод имеет параметр — обобщенный массив.

Другими словами, вы можете реконструировать все, что было объявлено в реализации обобщенных классов и методов. Однако вы не знаете, как разрешались параметры типа для конкретных объектов и вызовов методов.



На заметку! Информация о типе, содержащаяся в файле класса для обеспечения рефлексии обобщений, не совместима с более старыми виртуальными машинами.

Для того чтобы выразить объявления обобщенных типов, в Java SE 5.0 в пакете `java.lang.reflect` появился новый интерфейс `Type`. У этого интерфейса есть следующие подтипы.

- Класс `Class`, описывающий конкретные типы.
- Интерфейс `TypeVariable`, описывающий переменные типа (такие как `T extends Comparable<? super T>`).
- Интерфейс `WildcardType`, описывающий подстановки (такие как `? super T`).
- Интерфейс `ParameterizedType`, описывающий обобщенный класс или типы интерфейсов (такие как `Comparable<? super T>`).
- Интерфейс `GenericArrayType`, описывающий обобщенные массивы (такие как `T[]`).

На рис. 12.5. показана иерархия наследования. Обратите внимание, что последние четыре подтипа являются интерфейсами — виртуальная машина создает экземпляры подходящих классов, реализующих эти интерфейсы.

Код в листинге 12.4 использует API рефлексии обобщений для распечатки того, что он может рассказать о данном классе. Если вы запустите его с классом `Pair`, то получите такой отчет:

```
class Pair<T> extends java.lang.Object
public T getFirst()
public T getSecond()
public void setFirst(T)
public void setSecond(T)
```

Если вы запустите его с `ArrayAlg` в каталоге `PairTest2`, то отчет покажет следующий метод:

```
public static <T extends java.lang.Comparable> Pair<T> minmax(T[])
```

Врезки, касающиеся API, в конце этого раздела описывают методы, использованные в примере программы.

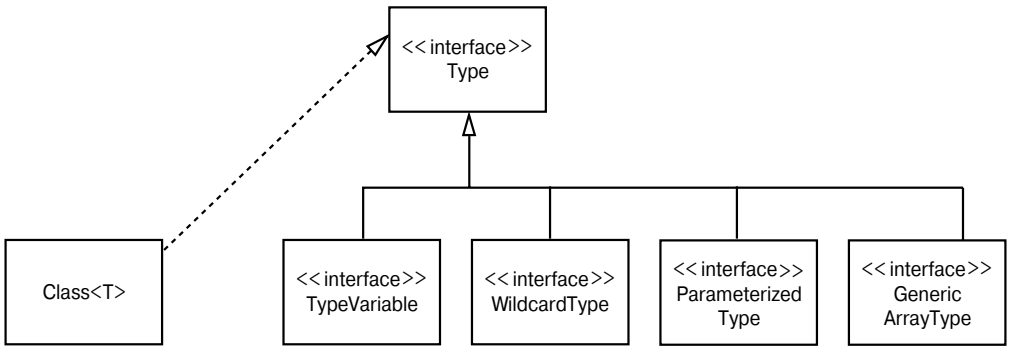


Рис. 12.5. Класс `Type` и его наследники

Листинг 12.4. Содержимое файла `GenericReflectionTest.java`

```
1. import java.lang.reflect.*;
2. import java.util.*;
3. /**
4.  * @version 1.10 2004-05-15
5.  * @author Cay Horstmann
6.  */
7. public class GenericReflectionTest
8. {
9.    public static void main(String[] args)
10.   {
11.      //Читает имя класса из аргументов командной строки или ввода пользователя.
12.      String name;
13.      if (args.length > 0) name = args[0];
14.      else
15.      {
16.           Scanner in = new Scanner(System.in);
17.           System.out.println("Enter class name (e.g. java.util.Collections): ");
18.           name = in.next();
19.      }
20.      try
21.      {
22.           //Печатает информацию об обобщениях для класса и общедоступных методов.
23.           Class cl = Class.forName(name);
24.           printClass(cl);
25.           for (Method m : cl.getDeclaredMethods())
26.              printMethod(m);
27.      }
```

```
28.     catch (ClassNotFoundException e)
29.     {
30.          e.printStackTrace();
31.     }
32. }
33. public static void printClass(Class cl)
34. {
35.     System.out.print(cl);
36.     printTypes(cl.getTypeParameters(), "<", ", ", ">", true);
37.     Type sc = cl.getGenericSuperclass();
38.     if (sc != null)
39.     {
40.         System.out.print(" extends ");
41.         printType(sc, false);
42.     }
43.     printTypes(cl.getGenericInterfaces(), " implements ", ", ", "", false);
44.     System.out.println();
45. }
46. public static void printMethod(Method m)
47. {
48.     String name = m.getName();
49.     System.out.print(Modifier.toString(m.getModifiers()));
50.     System.out.print(" ");
51.     printTypes(m.getTypeParameters(), "<", ", ", "> ", true);
52.     printType(m.getGenericReturnType(), false);
53.     System.out.print(" ");
54.     System.out.print(name);
55.     System.out.print("(");
56.     printTypes(m.getGenericParameterTypes(), "", ", ", "", false);
57.     System.out.println(")");
58. }
59. public static void printTypes(Type[] types, String pre, String sep,
60.    String suf, boolean isDefinition)
61. {
62.     if (pre.equals(" extends ") && Arrays.equals(types, new Type[]
63.        { Object.class })) return;
64.     if (types.length > 0) System.out.print(pre);
65.     for (int i = 0; i < types.length; i++)
66.     {
67.         if (i > 0) System.out.print(sep);
68.         printType(types[i], isDefinition);
69.     }
70.     if (types.length > 0) System.out.print(suf);
71. }
72. public static void printType(Type type, boolean isDefinition)
73. {
74.     if (type instanceof Class)
75.     {
76.         Class t = (Class) type;
77.         System.out.print(t.getName());
78.     }
79.     else if (type instanceof TypeVariable)
80.     {
81.         TypeVariable t = (TypeVariable) type;
82.         System.out.print(t.getName());
```

```
83.          if (isDefinition)
84.          printTypes(t.getBounds(), " extends ", " & ", "", false);
85.      }
86.      else if (type instanceof WildcardType)
87.      {
88.          WildcardType t = (WildcardType) type;
89.          System.out.print("?");
90.          printTypes(t.getUpperBounds(), " extends ", " & ", "", false);
91.          printTypes(t.getLowerBounds(), " super ", " & ", "", false);
92.      }
93.      else if (type instanceof ParameterizedType)
94.      {
95.          ParameterizedType t = (ParameterizedType) type;
96.          Type owner = t.getOwnerType();
97.          if (owner != null)
98.          {
99.             printType(owner, false);
100.            System.out.print(".");
101.         }
102.         printType(t.getRawType(), false);
103.         printTypes(t.getActualTypeArguments(), "<", ", ", ">", false);
104.     }
105.     else if (type instanceof GenericArrayType)
106.     {
107.         GenericArrayType t = (GenericArrayType) type;
108.         System.out.print("");
109.         printType(t.getGenericComponentType(), isDefinition);
110.         System.out.print("[]");
111.     }
112.  }
113. }
```

API java.lang.Class<T> 1.0

- `TypeVariable[] getTypeParameters()` **5.0**

  Получает переменные обобщенного типа, если этот тип был объявлен как обобщенный, или же массив длиной 0 в противном случае.

- `Type getGenericSuperclass()` **5.0**

  Получает обобщенный тип суперкласса, который был объявлен для этого типа, или `null`, если данный тип является `Object` или же не является типом класса.

- `Type[] getGenericInterfaces()` **5.0**

  Получает обобщенные типы интерфейсов, которые были объявлены для этого типа, в порядке объявления, либо массив длиной 0, если данный тип не реализует интерфейсов.

API java.lang.reflect.Method 1.1

- `TypeVariable[] getTypeParameters()` **5.0**

  Получает переменные обобщенного типа, если этот тип был объявлен как обобщенный, или же массив длиной 0 в противном случае.

- `Type getGenericReturnType()` **5.0**

  Получает обобщенный тип возврата, с которым был объявлен данный метод.

- `Type[] getGenericParameterTypes()` **5.0**

  Получает обобщенные параметры типов, с которыми был объявлен данный метод. Если метод не имеет параметров, возвращается массив длиной 0.

API java.lang.reflect.TypeVariable 5.0

- `String getName()`

  Получает имя переменной типа.

- `Type[] getBounds()`

  Получает ограничения подкласса этой переменной типа или массив длиной 0, если переменная не ограничена.

API java.lang.reflect.WildcardType 5.0

- `Type[] getLowerBounds()`

  Получает ограничение подкласса (`extends`) данной переменной типа либо массив длиной 0, если ограничений подкласса нет.

- `Type[] getUpperBounds()`

  Получает ограничения суперкласса (`super`) этой переменной типа или массив длиной 0, если ограничений суперкласса нет.

API java.lang.reflect.ParameterizedType 5.0

- `Type getRawType()`

  Получает "сырой" тип данного параметризованного типа.

- `Type[] getActualTypeArguments()`

  Получает параметр типа, с которым был объявлен данный параметризованный тип.

- `Type getOwnerType()`

  Получает тип внешнего класса, если данный тип — внутренний, или `null`, если это тип верхнего уровня.

API java.lang.reflect.GenericArrayType 5.0

- `Type getGenericComponentType()`

  Получает обобщенный тип компонента, с которым ассоциирован тип данного массива.

Теперь вы знаете, как использовать обобщенные классы и как программировать собственные обобщенные классы и методы, если возникнет такая необходимость. Не менее важно и то, что вы знаете, как расшифровать объявления обобщенных типов, которые вы можете встретить в документации по API и в сообщениях об ошибках. Чтобы ознакомиться с совершенно исчерпывающей дискуссией обо всем, что нужно знать о механизме обобщений в Java, обратитесь к блестящему списку часто (и не очень часто) задаваемых вопросов по адресу `http://angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html`.

В следующей главе вы увидите, как каркас коллекций Java заставляет работать механизм обобщений.

Глава 13

КОЛЛЕКЦИИ

В ЭТОЙ ГЛАВЕ...

- Интерфейсы коллекций
- Конкретные коллекции
- Каркас коллекций
- Алгоритмы
- Унаследованные коллекции

Выбираемые вами структуры данных могут весьма отличаться друг от друга как в отношении реализации методов в "естественном стиле", так и в отношении производительности. Нужен ли быстрый поиск среди тысяч (или даже миллионов) сортированных элементов? Нужна ли быстрая вставка и удаление элементов в середине упорядоченной последовательности? Требуется ли устанавливать ассоциации между ключами и значениями?

В этой главе мы покажем, как с помощью библиотеки Java можно реализовать традиционные структуры данных, необходимые при серьезном программировании. На специальностях, связанных с вычислительной техникой, как правило, предусмотрен курс по структурам данных, поэтому есть очень много книг, посвященных этой важной теме. Наше изложение отличается от упомянутого курса; мы пропустим теорию и просто расскажем о том, как используются классы коллекций из стандартной библиотеки.

Интерфейсы коллекций

Начальная версия Java предлагала лишь небольшой набор классов наиболее часто употребляемых структур данных: `Vector`, `Stack`, `Hashtable`, `BitSet` и интерфейс `Enumeration`, предоставляющий абстрактный механизм для посещения элементов в произвольном контейнере. Безусловно, это было мудрое решение — на реализацию всеобъемлющей библиотеки классов коллекций потребовалось время и опыт.

К появлению Java SE 1.2 проектировщики поняли, что настало время создать полноценный набор структур данных. При этом они столкнулись с множеством противоречивых требований. Они хотели, чтобы библиотека была маленькой и легкой в изучении. Нужно было избежать сложности стандартной библиотеки шаблонов (STL) языка C++, но при этом позаимствовать "обобщенные алгоритмы", впервые появившиеся в STL. Необходимо было обеспечить совместимость унаследованных классов коллекций с новым каркасом. Как это случается со всеми проектировщиками библиотек коллекций, перед ними не раз вставал сложный выбор, и по пути было найдено множество оригинальных решений. В этом разделе мы раскроем основы дизайна каркаса коллекций Java, продемонстрируем, как заставить их работать, и объясним смысл наиболее спорных средств.

Отделение интерфейса от реализации коллекций

Как это принято в современных библиотеках структур данных, библиотека коллекций Java отделяет *интерфейсы* от *реализаций*. Давайте рассмотрим, как это разделение реализовано в знакомой всем структуре данных — очереди.

Интерфейс очереди специфицирует, что вы можете добавлять элементы в хвост очереди, удалять из ее головы, а также узнавать, сколько элементов находится в очереди в данный момент. Очереди применяются тогда, когда нужно накапливать объекты и извлекать их в режиме "первый вошел — первый вышел" (рис. 13.1).

Минимальная форма интерфейса очереди может выглядеть примерно так:

```
interface Queue<E> // упрощенная форма интерфейса из стандартной библиотеки
{
    void add(E element);
    E remove();
    int size();
}
```

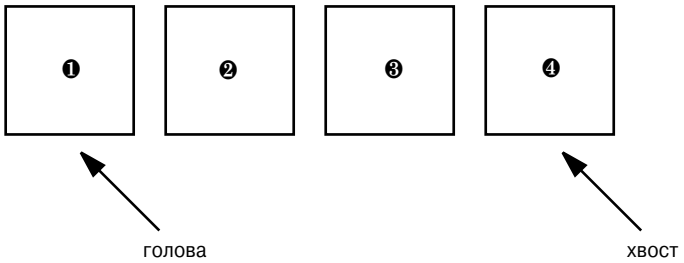


Рис. 13.1. Очередь

Интерфейс ничего не говорит вам о том, как реализована очередь. Одна из распространенных реализаций очереди использует "циклический массив", а другая — связный список (рис. 13.2).

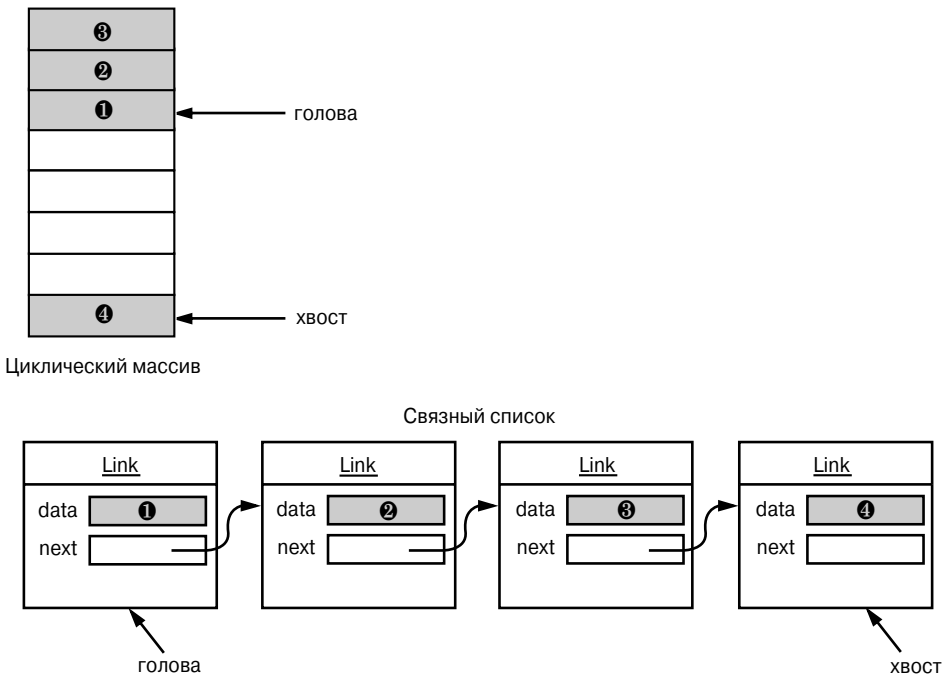


Рис. 13.2. Реализация очереди



На заметку! Начиная с Java SE 5.0, классы коллекций стали обобщенными классами с параметрами типа. Подробнее об обобщенных классах читайте в главе 12.

Каждая реализация может быть выражена классом, реализующим интерфейс `Queue`.

```
class CircularArrayQueue<E> implements Queue<E> // не настоящий
                                                // библиотечный класс
{
 CircularArrayQueue(int capacity) { . . . }
 public void add(E element) { . . . }
```

```
 public E remove() { . . . }
 public int size() { . . . }

 private E[] elements;
 private int head;
 private int tail;
}

class LinkedListQueue<E> implements Queue<E> // не настоящий библиотечный класс
{
 LinkedListQueue() { . . . }
 public void add(E element) { . . . }
 public E remove() { . . . }
 public int size() { . . . }

 private Link head;
 private Link tail;
}
```



На заметку! Библиотека Java на самом деле не содержит классов `CircularArrayQueue` и `LinkedListQueue`. Мы используем эти классы в качестве примеров, чтобы пояснить концептуальное отличие между интерфейсом и реализацией. Если вам нужна очередь на основе циклического массива, используйте класс `ArrayDeque`, который появился в Java SE 6. Если же необходима очередь — связный список, просто воспользуйтесь классом `LinkedList` — он реализует интерфейс `Queue`.

Когда вы применяете в своей программе очередь, вам не нужно знать, какая именно реализация используется для конструирования коллекции. Таким образом, имеет смысл использовать конкретный класс только тогда, когда вы конструируете объект коллекции. Применяйте *интерфейсный тип* для ссылки на коллекцию.

```
Queue<Customer> expressLane = new CircularArrayQueue<Customer>(100);
expressLane.add(new Customer("Harry"));
```

При таком подходе, если вдруг вы передумаете, то легко сможете использовать другую реализацию. Нужно будет внести изменение только в одном место программы — вызове конструктора. Если вы решите, что `LinkedListQueue` — лучший выбор из всех, код станет таким:

```
Queue<Customer> expressLane = new LinkedListQueue<Customer>();
expressLane.add(new Customer("Harry"));
```

Почему вы предпочитаете одну реализацию другой? Интерфейс ничего не говорит об эффективности реализации. Циклический массив в некотором отношении более эффективен, чем связный список, так что обычно он предпочтительнее. Однако, как всегда, за все нужно платить. Циклический массив — *ограниченная* коллекция, и он имеет определенную окончательную емкость. Если вам не известен верхний предел количества объектов, которые должна накапливать ваша программа, то вероятно, вам стоит выбрать реализацию на основе связного списка.

Изучая документацию API, вы найдете другой набор классов, чьи имена начинаются с `Abstract`, такие как `AbstractQueue`. Эти классы предназначены для разработчиков библиотек. В случае (наверно, маловероятном), когда вы захотите реализовать свой собственный класс очереди, вы обнаружите, что это легче сделать, расширив `AbstractQueue`, чем реализовывать все методы интерфейса `Queue`.

Интерфейсы Collection и Iterator из библиотеки Java

Фундаментальным интерфейсом классов коллекций в библиотеке Java является интерфейс Collection. Он включает два основных метода:

```
public interface Collection<E>
{
  boolean add(E element);
  Iterator<E> iterator();
  . . .
}
```

В дополнение к этим есть еще несколько методов; мы поговорим о них позже.

Метод add() добавляет элемент в коллекцию. Он возвращает true, если добавление элемента в действительности изменило коллекцию, и false — если коллекция осталась неизменной. Например, если вы попытаетесь добавить объект к набору (set), а такой объект уже там есть, то запрос add() не будет иметь эффекта, поскольку набор не допускает дублирования.

Метод iterator() возвращает объект, реализующий интерфейс Iterator. Вы можете выбрать объект итератора для перебора всех элементов коллекции один за другим.

Итераторы

Интерфейс Iterator включает следующие три метода:

```
public interface Iterator<E>
{
  E next();
  boolean hasNext();
  void remove();
}
```

Многократно вызывая метод next(), вы можете "посетить" каждый элемент коллекции — один за другим. Однако если вы достигнете конца коллекции, то метод next() сгенерирует исключение NoSuchElementException. Поэтому перед вызовом next() следует вызывать hasNext(). Этот метод возвращает true, если у объекта итератора все еще есть объекты, которые можно "посетить". Если вы хотите перебрать все элементы коллекции, то вы запрашиваете итератор, и продолжаете вызывать метод next(), пока hasNext() возвращает true. Например:

```
Collection<String> c = . . .;
Iterator<String> iter = c.iterator();
while (iter.hasNext())
{
   String element = iter.next();
   делать что-то с element
}
```

Начиная с Java SE 5.0, для этого цикла имеется элегантное сокращение. Его можно записать более изящно с использованием цикла в стиле "for each":

```
for (String element : c)
{
  делать что-то с element
}
```

Компилятор просто транслирует цикл "for each" в цикл с итератором.

Цикл "for each" работает с любым объектом, реализующим интерфейс `Iterable`, который имеет единственный метод:

```
public interface Iterable<E>
{
  Iterator<E> iterator();
}
```

Интерфейс `Collection` расширяет интерфейс `Iterable`. Поэтому вы можете использовать цикл "for each" с любой коллекцией из стандартной библиотеки.

Порядок, в котором перебираются элементы, зависит от типа коллекции. Если вы проходите по `ArrayList`, то итератор начинает с индекса 0 и увеличивает его значение на каждом шаге. Однако если вы проходите по элементам `HashSet`, то получите их в совершенно случайном порядке. Можно быть лишь уверенным, что за время итерации вы переберете все элементы коллекции, хотя нельзя сделать никаких предположений относительно их порядка. Обычно это не является проблемой, потому что порядок не имеет значения при таких вычислениях, как, например, подсчет суммы или количества совпадений.



На заметку! Ветераны Java заметят, что методы `next` и `hasNext` интерфейса `Iterator` служат той же цели, что и методы `nextElement` и `hasMoreElements` из `Enumeration`. Проектировщики библиотеки коллекцией Java могли бы отдать предпочтение использованию интерфейса `Enumeration`. Но им не понравились громоздкие имена методов, и потому они разработали новый интерфейс с более короткими именами методов.

Существует важная концептуальная разница между итераторами в библиотеке коллекцией Java и итераторами из других библиотек. В традиционных библиотеках коллекций, таких как Standard Template Library в C++, итераторы моделируются на основе индексов массива. Имея такой итератор, вы можете найти элемент массива, находящийся в этой позиции, подобно тому, как вы находите элемент `a[i]`, если у вас есть индекс массива `i`. Независимо от просмотра элементов, вы можете передвинуть итератор на следующую позицию. Это — та же операция, что и перемещение индекса вызовом `i++`, без выполнения просмотра. Однако итераторы Java работают иначе. Просмотр (поиск) и изменение позиции тесно взаимосвязаны. Единственный способ найти элемент — вызвать `next()`, и эта операция одновременно перемещает позицию.

Вместо этого итераторы Java нужно воспринимать так, как будто их позиции находятся *между элементами*. Когда вы вызываете `next()`, итератор перепрыгивает следующий элемент и возвращает ссылку на элемент, который он только что прошел (рис. 13.3).



На заметку! Есть еще одна удобная аналогия. Можно воспринимать `Iterator.next` как эквивалент `InputStream.read`. Прочтение байта из потока автоматически "потребляет" этот байт. Следующий вызов `read` потребляет и возвращает следующий байт из входа. Аналогично, повторяющиеся вызовы next позволяют вам читать элементы коллекции.

Удаление элемента

Метод `remove()` интерфейса `Iterator` удаляет элемент, который был возвращен последним вызовом `next()`. Во многих ситуациях это имеет смысл — вам нужно увидеть элемент, прежде чем вы решите, что его надо удалить. Но если вы хотите удалить элемент, находящийся в определенной позиции, вам придется пройти через него.

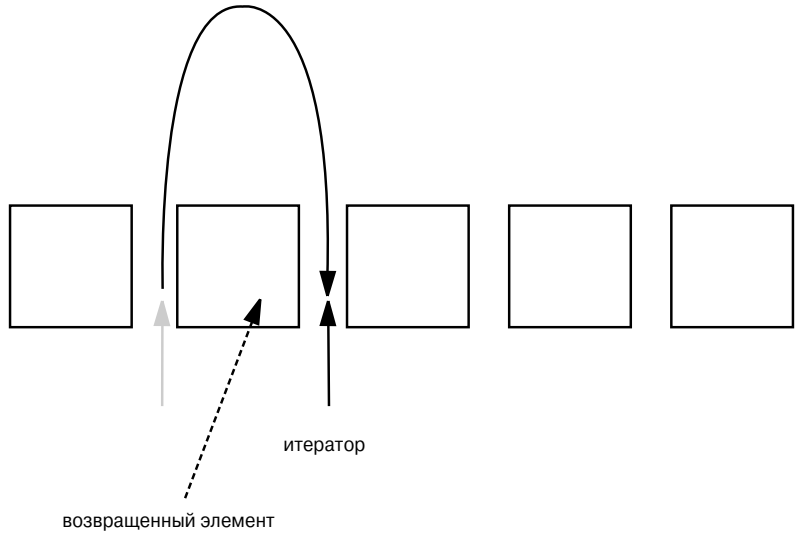


Рис. 13.3. Перемещение итератора

Например, вот как удалить первый элемент в коллекции строк:

```
Iterator<String> it = c.iterator();
it.next();          // пропустить первый элемент
it.remove();        // теперь удалить его
```

Что более важно, существует связь между вызовами методов `next()` и `remove()`. Не допускается вызывать `remove()`, если перед ним не был вызван `next()`. Если вы попытаетесь это сделать, сгенерируется исключение `IllegalStateException`.

Если вы хотите удалить два соседних элемента, вы не можете просто вызвать `remove()` два раза:

```
it.remove();
it.remove(); // Ошибка!
```

Вместо этого вы должны сначала вызвать `next()`, чтобы перепрыгнуть через элемент, подлежащий удалению:

```
it.remove();
it.next();
it.remove(); // OK
```

Обобщенные вспомогательные методы

Поскольку интерфейсы `Collection` и `Iterator` обобщенные, можно написать вспомогательные методы, оперирующие коллекциями любого вида. Например, ниже приведен обобщенный метод, проверяющий, содержит ли произвольная коллекция заданный элемент.

```
public static <E> boolean contains(Collection<E> c, Object obj)
{
  for (E element : c)
   if (element.equals(obj))
     return true;
  return false;
}
```

Проектировщики библиотеки Java решили, что некоторые из этих вспомогательных методов настолько полезны, что библиотека должна сделать их доступными. Таким образом, пользователи библиотеки будут избавлены от необходимости заново изобретать колесо. Один из таких методов — `contains()`.

Фактически интерфейс `Collection` объявляет немало методов, которые должны использовать все реализующие его классы.

```
int size()
boolean isEmpty()
boolean contains(Object obj)
boolean containsAll(Collection<?> c)
boolean equals(Object other)
boolean addAll(Collection<? extends E> from)
boolean remove(Object obj)
boolean removeAll(Collection<?> c)
void clear()
boolean retainAll(Collection<?> c)
Object[] toArray()
<T> T[] toArray(T[] arrayToFill)
```

Многие из этих методов самоочевидны; полную документацию вы найдете в заметках по API в конце этого раздела.

Конечно, было бы утомительно, если бы каждый класс, реализующий интерфейс `Collection`, должен был бы реализовывать настолько много рутинных методов. Чтобы облегчить жизнь разработчикам, библиотека предлагает класс `AbstractCollection`, который оставляет фундаментальные методы `size()` и `iterator()` абстрактными, но реализует рутинные методы. Например:

```
public abstract class AbstractCollection<E>
implements Collection<E>
{
  . . .
  public abstract Iterator<E> iterator();
  public boolean contains(Object obj)
  {
   for (E element : c) // вызывает iterator()
   if (element.equals(obj))
      return = true;
   return false;
  }
  . . .
}
```

Конкретный класс коллекции теперь может расширить класс `AbstractCollection`. Теперь на совести конкретного класса лежит реализация метода `iterator()`, но о методе `contains()` уже позаботился класс `AbstractCollection`. Однако если подкласс может предложить более эффективный способ реализации `contains()`, он вполне может это сделать.

Это хороший дизайн каркаса коллекций. В распоряжении пользователей классов коллекций есть богатый набор доступных методов в обобщенном интерфейсе, но разработчики реальных структур данных могут не утруждать себя реализацией рутинных методов.

API java.util.Collection<E> 1.2

- `Iterator<E>()`

  Возвращает итератор, который можно использовать для перебора элементов коллекции.

- `int size()`

  Возвращает количество элементов, хранящихся в коллекции в данный момент.

- `boolean isEmpty()`

  Возвращает `true`, если коллекция не содержит элементов.

- `boolean contains(Object obj)`

  Возвращает `true`, если коллекция содержит объект, эквивалентный `obj`.

- `boolean containsAll(Collection<?> other)`

  Возвращает `true`, если коллекция содержит все элементы другой коллекции.

- `boolean add(Object element)`

  Добавляет элемент в коллекцию. Возвращает `true`, если в результате этого вызова коллекция изменилась.

- `boolean addAll(Collection<? Extends E> other)`

  Добавляет все элементы из другой коллекции в данную. Возвращает `true`, если в результате этого вызова коллекция изменилась.

- `boolean remove(Object obj)`

  Удаляет объект, эквивалентный `obj` из коллекции. Возвращает `true`, если в результате этого вызова коллекция изменилась.

- `boolean removeAll(Collection<?> other)`

  Удаляет из данной коллекции все элементы другой коллекции. Возвращает `true`, если в результате этого вызова коллекция изменилась.

- `void clear()`

  Удаляет все элементы из данной коллекции.

- `boolean retainAll(Collection<?> other)`

  Удаляет все элементы из данной коллекции, которые не эквивалентны ни одному из элементов другой коллекции. Возвращает `true`, если в результате этого вызова коллекция изменилась.

- `Object[] toArray()`

  Возвращает массив объектов коллекции.

- `<T> T[] toArray(T[] arrayToFill)`

  Возвращает массив объектов в коллекции. Если `arrayToFill` имеет достаточную длину, он заполняется элементами данной коллекции. Если остается место, добавляются элементы `null`. В противном случае выделяется новый массив с тем же типом компонентов, что и `arrayToFill`, и той же длины, и этот массив заполняется значениями `arrayToFill`.

API java.util.Collection<E> 1.2

- `boolean hasNext()`

  Возвращает `true`, если в коллекции есть еще элементы для "посещения".

- `E next()`

  Возвращает следующий объект для "посещения". Генерирует исключение `NoSuchElementException`, если уже достигнут конец коллекции.

- `void remove()`

  Удаляет последний посещенный объект. Этот метод должен следовать немедленно после посещения элемента. Если коллекция была модифицирована после посещения последнего элемента, этот метод генерирует исключение `IllegalStateException`.

Конкретные коллекции

Вместо того чтобы углубляться дальше в детали этих интерфейсов, мы полагаем, что было бы полезно сначала обсудить конкретные структуры данных, предлагаемые библиотекой Java. Основательно описав классы, которые вам могут понадобиться, мы вернемся к абстрактным рассуждениям и увидим, как каркас коллекций организует все эти классы. В табл. 13.1 показаны коллекции библиотеки Java вместе с кратким описанием назначения каждого из классов коллекций. (Для простоты мы пропускаем здесь безопасные к потокам коллекции, о которых речь пойдет к главе 14.) Все классы из табл. 13.1 реализуют интерфейс `Collection`, за исключением классов, чьи имена оканчиваются на `Map`. Эти классы реализуют вместо него интерфейс `Map`. Об интерфейсе `Map` мы поговорим ниже, в разделе "Карты".

Таблица 13.1. Конкретные коллекции из библиотеки Java

Тип коллекции	Описание
`ArrayList`	Индексированная последовательность, динамически растягивающаяся и сжимающаяся.
`LinkedList`	Упорядоченная последовательность, позволяющая эффективную вставку и удаление в любой позиции.
`ArrayDeque`	Упорядоченная последовательность, реализованная в виде циклического массива.
`HashSet`	Неупорядоченная коллекция, исключающая дубликаты.
`TreeSet`	Сортированный набор.
`EnumSet`	Множество значений перечислимого типа.
`LinkedHashSet`	Набор, запоминающий порядок вставки элементов.
`PriorityQueue`	Коллекция, позволяющая эффективно удалять наименьший элемент .
`HashMap`	Структура данных, сохраняющая ассоциации "ключ/значение".
`TreeMap`	Карта с сортированными ключами.
`EnumMap`	Карта, в которой ключи относятся к перечислимому типу.
`LinkedHashMap`	Карта, запоминающая порядок, в котором добавлялись элементы.
`WeakHashMap`	Карта со значениями, которые могут удаляться сборщиком мусора, если таковые нигде не используются.
`IdentityHashMap`	Карта с ключами, сравниваемыми с помощью `==`, а не `equals()`.

Связные списки

Во многих примерах нашей книги мы уже использовали массивы и их динамического родственника — класс `ArrayList`. Однако массивы и массивы-списки страдают существенным недостатком. Удаление элемента из середины массива обходится дорого, потому что все элементы, находящиеся за удаляемым, приходится перемещать в сторону начала массива (рис. 13.4). То же самое верно в отношении вставки элементов в середину.

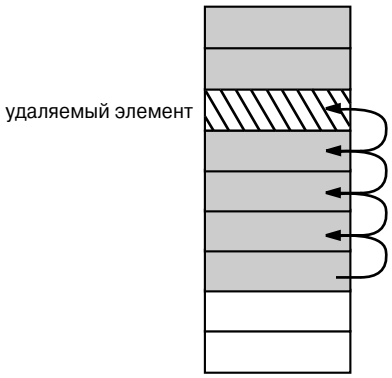


Рис. 13.4. Удаление элемента из массива

Другая широко известная структура данных — *связный список*, решает эту проблему. В то время как массив хранит ссылки на объекты в последовательных участках памяти, связный список каждый объект в отдельном *звене* (link). В языке программирования Java все связные списки в действительности являются *двунаправленными*; т.е. каждое звено хранит ссылку на предшественника (рис. 13.5).

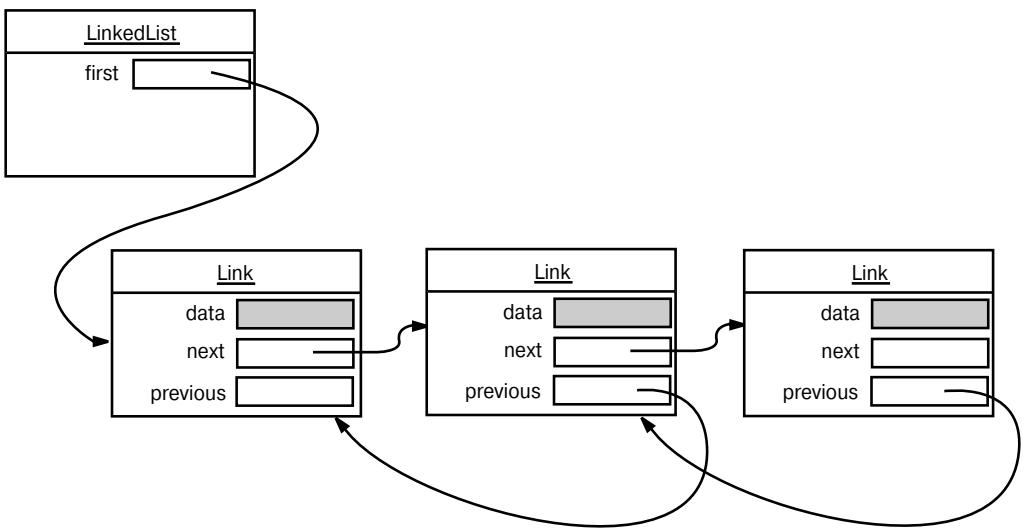


Рис. 13.5. Двунаправленный связный список

Удаление элемента из средины связного списка — недорогая операция, так как при этом нужно обновить лишь соседние с удаляемым элементом звенья (рис. 13.6).

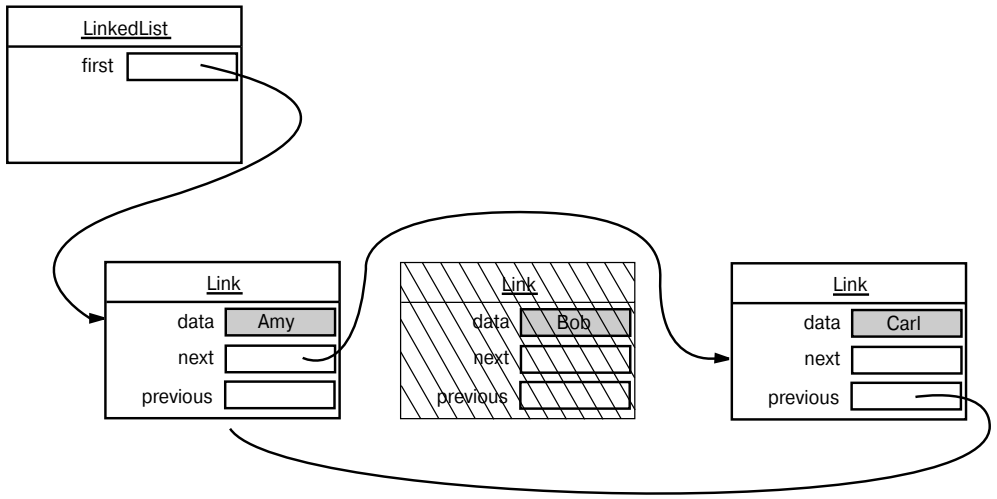


Рис. 13.6. Удаление элемента из связного списка

Возможно, вы уже прошли курс по структурам данных, в котором изучали реализацию связных списков. Возможно, вы уже подзабыли, как нужно привязывать звенья при удалении или добавлении элементов связного списка. Если так, то вы будете приятно удивлены, узнав, что в библиотеке коллекций Java предусмотрен готовый к использованию класс `LinkedList`.

Следующий пример кода добавляет три элемента и затем удаляет второй из них.

```
List<String> staff = new LinkedList<String>(); // LinkedList реализует List
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");
Iterator iter = staff.iterator();
String first = iter.next();   // посетить первый элемент
String second = iter.next();  // посетить второй элемент
iter.remove(); // удалить последний посещенный элемент
```

Однако есть существенная разница между связными списками и обобщенными коллекциями. Связный список — это *упорядоченная коллекция*, в которой положение объектов имеет значение. Метод `LinkedList.add()` добавляет объект в конец списка. Но часто вам нужно добавлять объекты куда-то в середину списка.

Зависящий от позиции метод `add()` находится в ответственности итератора, поскольку итератор описывает позиции в коллекции. Использование итераторов для добавления элементов имеет смысл только для коллекций, имеющих естественный порядок. Например, тип данных `set`, о котором пойдет речь в следующем разделе, не подразумевает никакого порядка элементов. Поэтому у интерфейса `Iterator` нет метода `add()`. Вместо него библиотека коллекций предусматривает подинтерфейс `ListIterator`, включающий метод `add()`:

```
interface ListIterator<E> extends Iterator<E>
{
  void add(E element);
  . . .
}
```

В отличие от `Collection.add()`, этот метод не возвращает `boolean`. Предполагается, что операция `add` всегда модифицирует список. В дополнение итератор `ListIterator` имеет два метода, которые вы можете использовать для обратного прохода по списку:

```
E previous()
boolean hasPrevious()
```

Как и `next()`, метод `previous()` возвращает объект, который он прошел.

Метод `listIterator()` класса `LinkedList` возвращает объект итератора, реализующего интерфейс `ListIterator`:

```
ListIterator<String> iter = staff.listIterator();
```

Метод `add()` добавляет новый элемент *перед* текущей позицией итератора. Например, следующий код пропускает первый элемент связного списка и добавляет "Juliet" перед вторым его элементом (рис. 13.7).

```
List<String> staff = new LinkedList<String>();
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");
ListIterator<String> iter = staff.listIterator();
iter.next(); // пропускает первый элемент
iter.add("Juliet");
```

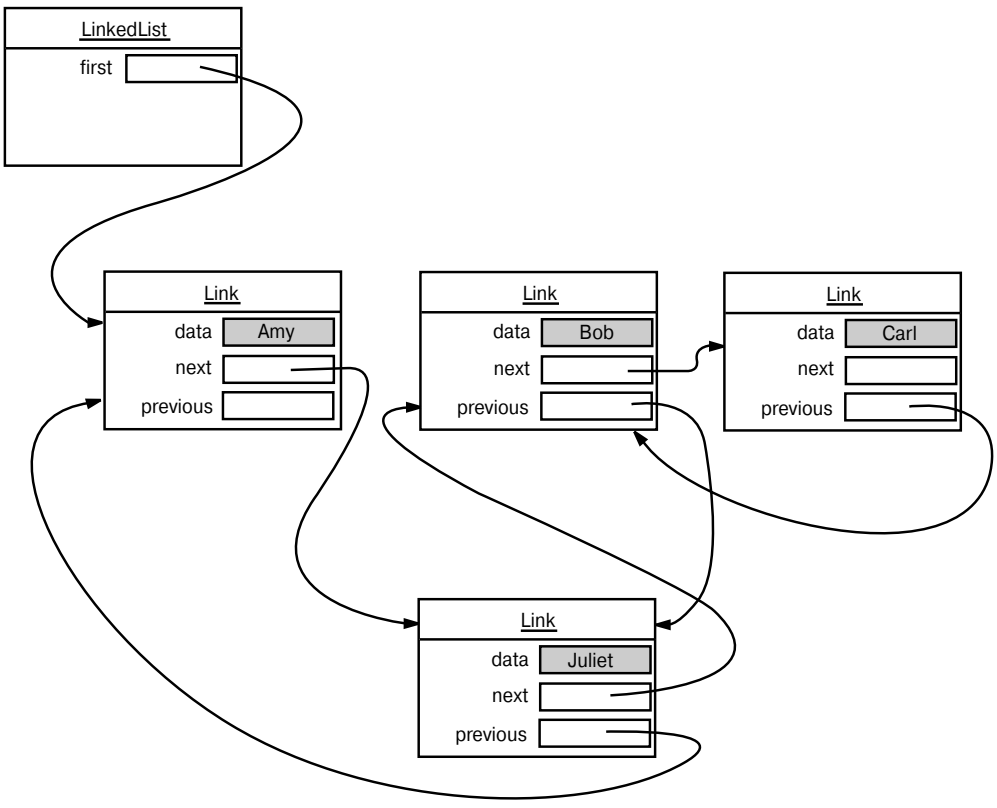


Рис. 13.7. Добавление элемента в связный список

Если вы вызываете метод `add()` несколько раз, элементы просто добавляются в порядке их передачи. Все они добавляются по очереди перед текущей позицией итератора.

Когда вы используете операцию `add()` с итератором, только что возвращенным методом `listIterator()`, и указывающим на начало связного списка, то вновь добавляемые элементы становятся в его начало. Когда же итератор проходит за последний элемент списка (т.е. `hasNext()` возвращает `false`), добавляемые элементы становятся в конец списка. Если связный список содержит n элементов, т.е. $n + 1$ мест для добавления нового элемента. Эти места соответствуют $n + 1$ возможным позициям итератора. Например, если связный список содержит три элемента — `A`, `B` и `C`, — то имеется четыре возможных позиции (они помечены `|`) для вставки нового элемента:

```
|ABC
A|BC
AB|C
ABC|
```



На заметку! Нужно быть осторожным с аналогией с "курсором". Операция удаления работает не совсем так, как клавиша <Backspace>. Сразу после вызова `next()` метод `remove()` действительно удаляет элемент слева от итератора — так же, как это делает нажатие клавиши <Backspace>. Однако если вы только что вызвали `previous()`, удаляется элемент, стоящий справа. К тому же вы не можете вызвать `remove()` дважды на строке.

В отличие от метода `add()`, который зависит только от позиции итератора, метод `remove()` зависит от состояния итератора.

И, наконец, метод `set()` заменяет последний элемент, возвращенный вызовом `next()` или `previous()`, новым. Например, следующий код заменяет первый элемент списка новым значением:

```
ListIterator<String> iter = list.listIterator();
String oldValue = iter.next(); // возвращает первый элемент
iter.set(newValue); // устанавливает первый элемент в newValue
```

Как вы можете представить, если итератор проходит коллекцию в то время, когда другой итератор модифицирует ее, могут возникнуть конфликтные ситуации. Например, предположим, что итератор установлен перед элементом, который только что был удален другим итератором. Теперь этот итератор недействителен и не должен более использоваться. Итераторы связного списка способны обнаруживать такие модификации. Если итератор обнаруживает, что коллекция была модифицирована другим итератором или же методом самой коллекции, то он генерирует исключение `ConcurrentModificationException`. Для примера рассмотрим следующий код.

```
List<String> list = . . .;
ListIterator<String> iter1 = list.listIterator();
ListIterator<String> iter2 = list.listIterator();
iter1.next();
iter1.remove();
iter2.next(); // генерирует ConcurrentModificationException
```

Вызов `iter2.next` генерирует `ConcurrentModificationException`, поскольку `iter2` обнаруживает, что список был модифицирован извне.

Чтобы избежать исключений, вызванных конкурирующими модификациями, вы можете присоединить единственный итератор, который может и читать, и писать.

Обнаружение конкурирующей модификации достигается простым способом. Коллекция отслеживает количество изменяющих ее операций (таких как добавление и удаление элементов). Каждый итератор хранит отдельный счетчик операций изменения, вызванных им самим. В начале каждого метода итератора, итератор просто сравнивает значения своего счетчика изменений со значением счетчика изменений коллекции. Если они не равны, генерируется исключение `ConcurrentModificationException`.



На заметку! Однако есть одно любопытное исключение из обнаружения конкурирующих модификаций. Связный список отслеживает лишь *структурные* модификации наподобие добавления и удаления звеньев. Метод `set()` не считается структурной модификацией. Вы можете присоединить к списку множество итераторов, и все они могут вызывать `set()` для изменения содержимого списка. Эта возможность востребована множеством алгоритмов классов `Collection`, о которых мы поговорим далее в этой главе.

Теперь вы знакомы с основными методами класса `LinkedList`. Вы используете `ListIterator` для прохода по элементам связного списка в любом направлении и для добавления и удаления элементов.

Как вы видели в предыдущем разделе, в интерфейсе `Collection` объявлено много других полезных методов для операций со связными списками. Большей частью они реализованы в суперклассе `AbstractCollection` класса `LinkedList`. Например, метод `toString()` вызывает `toString()` на всех элементах и генерирует одну длинную строку в формате `[A, B, C]`. Это удобно для отладки. Используйте метод `contains()` для проверки наличия элемента в связном списке.

Например, вызов `staff.contains("Harry")` возвращает `true`, если связный список уже содержит строку, эквивалентную строке `"Harry"`.

Библиотека также предусматривает ряд методов, которые с теоретической точки зрения несколько избыточны. Связные списки не поддерживают произвольного доступа. Если вы хотите видеть n-й элемент связного списка, то должны начать с начала и пропустить первые $n - 1$ элементов от начала. Это невозможно сократить. По этой причине обычно программисты не используют связные списки в тех случаях, когда нужно обратиться к элементу по индексу.

Тем не менее, в классе `LinkedList` предусмотрен метод `get()`, который позволяет обратиться к определенному элементу:

```
LinkedList<String> list = . . .;
String obj = list.get(n);
```

Конечно, этот метод не слишком эффективен. Всякий раз, когда с его помощью вы обращаетесь к другому элементу, поиск начинается с начала списка. Объект `LinkedList` не предпринимает никаких попыток кэширования информации о позициях элементов.



На заметку! В методе `get()` предусмотрена одна небольшая оптимизация: если указанный индекс больше `size()/2`, то поиск элемента начинается с конца списка.

Интерфейс итератора списка также имеет метод, который сообщит вам индекс текущей позиции. Фактически, поскольку концептуально итераторы Java указывают на позицию между элементами, таких методов два. Метод `nextIndex()` возвращает целочисленный индекс элемента, который должен быть возвращен следующим вызовом `next()`; метод `previousIndex()` возвращает индекс элемента, который был бы возвращен следующим вызовом `previous()`. Конечно, он будет на единицу меньше, чем

nextIndex. Эти методы эффективны — итератор помнит текущую позицию. И, наконец, если у вас есть целочисленный индекс n, то list.listIterator(n) вернет итератор, установленный прямо перед элементом с индексом n. То есть вызов next() порождает тот же элемент, что и list.get(n); получение этого итератора не эффективно.

Если ваш связный список содержит лишь горстку элементов, то вам не стоит особо беспокоиться о стоимости методов set() и get(). Но тогда зачем вы вообще используете связный список? Единственная причина его применения — минимизация расходов на вставку и удаление в средине списка. Если элементов всего несколько, используйте ArrayList.

Мы рекомендуем держаться подальше от всех методов, которые используют целочисленный индекс для обозначения позиции в связном списке. Если вам нужен произвольный доступ к коллекции, используйте массив или ArrayList, а не связный список.

Программа в листинге 13.1 представляет пример работы связного списка. Она просто создает два списка, объединяет их, затем удаляет каждый второй элемент из второго списка, и в конце тестирует метод removeAll(). Мы рекомендуем вам тщательно исследовать порядок выполнения этой программы, уделив особое внимание итераторам. Возможно, полезно будет нарисовать диаграмму позиций итератора — примерно так:

```
|ACE |BDFG
A|CE |BDFG
AB|CE B|DFG
. . .
```

Обратите внимание, что вызов

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

печатает все элементы связного списка, вызывая метод toString() из AbstractCollection.

Листинг 13.1. Содержимое файла LinkedListTest.java

```
1.  import java.util.*;
2.
3.  /**
4.   * Программа демонстрирует операции со связными списками.
5.   * @version 1.10 2004-08-02
6.   * @author Cay Horstmann
7.   */
8.  public class LinkedListTest
9.  {
10.   public static void main(String[] args)
11.   {
12.      List<String> a = new LinkedList<String>();
13.      a.add("Amy");
14.      a.add("Carl");
15.      a.add("Erica");
16.
17.      List<String> b = new LinkedList<String>();
18.      b.add("Bob");
19.      b.add("Doug");
20.      b.add("Frances");
21.      b.add("Gloria");
22.
23.      // Объединить слова из b с a.
24.      ListIterator<String> aIter = a.listIterator();
```

```
25.       Iterator<String> bIter = b.iterator();
26.       while (bIter.hasNext())
27.       {
28.          if (aIter.hasNext()) aIter.next();
29.          aIter.add(bIter.next());
30.       }
31.       System.out.println(a);
32.
33.       // Удалить каждое второе слово из b.
34.
35.       bIter = b.iterator();
36.       while (bIter.hasNext())
37.       {
38.          bIter.next(); // пропустить один элемент
39.          if (bIter.hasNext())
40.          {
41.             bIter.next();   // пропустить следующий элемент
42.             bIter.remove(); // удалить этот элемент
43.          }
44.       }
45.       System.out.println(b);
46.       // Групповая операция: удаление всех слов b из a.
47.       a.removeAll(b);
48.       System.out.println(a);
49.    }
50. }
```

API java.util.List<E> 1.2

- `ListIerator<E> listIterator()`

  Возвращает итератор списка, который можно использовать для перебора элементов списка.

- `ListIerator<E> listIterator(int index)`

  Возвращает итератор списка для посещения элементов списка, чей первый вызов `next()` вернет элемент с заданным индексом.

- `void add(int i, E element)`

  Добавляет элемент в указанную позицию.

- `void addAll(int i, Collection<? Extends E> elements)`

  Добавляет все элементы из коллекции в указанную позицию.

- `E remove(int i)`

  Удаляет и возвращает элемент из указанной позиции.

- `E get(int i)`

  Получает элемент из указанной позиции.

- `E set(int i, E element)`

  Заменяет элемент в указанной позиции новым элементом и возвращает старый.

- `int indexOf(Object element)`

  Возвращает позицию первого вхождения элемента, эквивалентного указанному, или `-1`, если соответствующий элемент не найден.

- `int lastIndexOf(Object element)`

  Возвращает позицию последнего вхождения элемента, эквивалентного указанному, или `-1`, если соответствующий элемент не найден.

API java.util.ListIterator<E> 1.2

- `void add(E newElement()`

  Добавляет элемент перед текущей позицией.

- `void set(E newElement)`

  Заменяет последний элемент, посещенный вызовом `next()` или `previous()`, новым элементом. Генерирует исключение `IllegalStateException`, если структура списка была модифицирована после последнего вызова `next()` или `previous()`.

- `boolean hasPrevious()`

  Возвращает `true`, если есть еще элемент для посещения при обратной итерации по списку.

- `E previous()`

  Возвращает предыдущий объект. Генерирует исключение `NoSuchElementException`, если достигнуто начало списка.

- `int nextIndex()`

  Возвращает индекс элемента, который должен быть возвращен следующим вызовом `next()`.

- `int previousIndex()`

  Возвращает индекс элемента, который должен быть возвращен следующим вызовом `previous()`.

API java.util.LinkedList<E> 1.2

- `LinkedList()`

  Конструирует пустой связный список.

- `LinkedList(Collection<? Extends E> elements)`

  Конструирует связный список и добавляет элементы из коллекции.

- `void addFirst(E element)`
- `void addLast(E element)`

  Добавляют элемент в начало или конец списка.

- `E getFirst()`
- `E getLast()`

  Возвращают элемент из начала или конца списка.

- `void removeFirst()`
- `void removeLast()`

  Удаляют и возвращают элемент из начала или конца списка.

Списки-массивы

В предыдущем разделе вы видели интерфейс `List` и класс `LinkedList`, реализующий его. Интерфейс `List` описывает упорядоченную коллекцию, в которой положение элемента имеет значение. Существует два протокола для обхода элементов: посредством итератора и произвольным доступом с помощью методов `get()` и `set()`. Второй не слишком подходит для связных списков, но, конечно, `get()` и `set()` совершенно оправданы для массивов. Библиотека коллекций предлагает уже знакомый вам класс `ArrayList`, также реализующий интерфейс `List`. `ArrayList` инкапсулирует динамически выделенный массив объектов.



На заметку! Если вы — ветеран в программировании на Java, то наверно использовали класс `Vector` всякий раз, когда возникала потребность в динамическом массиве. Почему стоит вместо `ArrayList` применять `Vector`? По одной простой причине: все методы `Vector` *синхронизированы.* Можно безопасно обращаться к объекту `Vector` одновременно из двух потоков. Но если вы обращаетесь к вектору только из одного потока — что случается намного чаще — ваш код впустую тратит время на синхронизацию. В отличие от этого, методы `ArrayList` не синхронизированы. Мы рекомендуем всегда, когда нет необходимости в синхронизации, использовать `ArrayList` вместо `Vector`.

Хеш-наборы

Связные списки и массивы позволяют вам указать порядок, в котором должны следовать элементы. Однако если вам нужен определенный элемент и вы не помните его позиции, то вам придется перебрать все элементы, пока вы его не найдете соответствия. Это может потребовать времени, если коллекция содержит достаточно много элементов. Если вас не заботит порядок элементов, то на этот случай предусмотрена структура данных, которая позволит находить элементы намного быстрее. Ее недостаток в том, что эта структура не обеспечивает никакого контроля за порядком появления элементов. Эта структура данных организует элементы в порядке, удобном для ее собственных целей.

Эта широко известная структура данных для быстрого нахождения объектов называется *хеш-таблицей.* Хеш-таблица вычисляет целое число, называемое *хеш-кодом*, для каждого объекта. Хеш-код — это целое число, которое некоторым образом производится на основе полей экземпляра объекта, причем предполагается, что объекты с разными данными порождают разные коды. В табл. 13.2 перечислено несколько примеров хеш-кодов, являющихся результатом вызова метода `hashCode()` класса `String`.

Таблица 13.2. Хеш-коды, возвращенные функцией `hashCode()`

Строка	Хеш-код
`"Lee"`	76268
`"lee"`	107020
`"eel"`	100300

Если вы определяете свои собственные классы, то на вас ложится ответственность за реализацию собственного метода `hashCode()` — подробнее об этом читайте в главе 5. Ваша реализация должна быть совместима с методом `equals()`: если `a.equals(b)` равно `true`, то `a` и `b` должны иметь одинаковые хеш-коды.

Что важно — так это то, чтобы хеш-коды вычислялись быстро, и вычисление зависело только от состояния объекта, подлежащего хешированию, а не от других объектов в хеш-таблице.

В Java хеш-таблицы реализованы в виде массивов или связных списков. Каждый список называется *группой* (рис. 13.8). Чтобы найти место объекта в таблице, вычислите его хеш-код и уменьшите его модуль до общего количества групп. Полученное в результате число и будет индексом группы, содержащей элемент. Например, если хеш-код объекта равен 76268, и всего есть 128 групп, значит, объект должен быть помещен в группу номер 108 (поскольку остаток от целочисленного деления 76268 на 128 равен 108). Возможно, вам повезет, и в этой группе не окажется других элементов. Тогда вы просто вставляете элемент в группу. Конечно, в конце концов, вы наткнетесь на непустую группу. Это называется *коллизией хеш-таблицы*. Тогда вы сравниваете новый объект со всеми объектами группы, чтобы проверить, нет ли его там уже. Предполагая сравнительно равномерное распределение хеш-кодов и при достаточно большом количестве групп, понадобится лишь несколько сравнений.

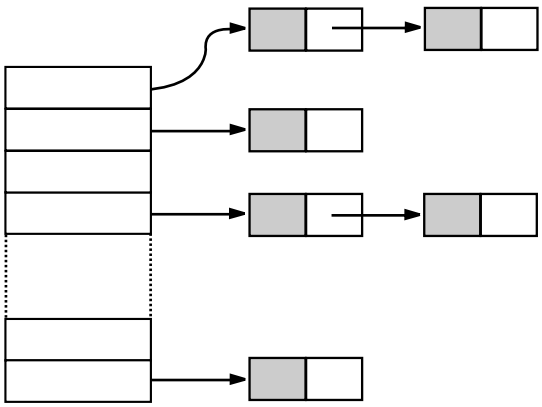


Рис. 13.8. Хеш-таблица

Если вы хотите более тщательно управлять производительностью хеш-таблицы, то можно специфицировать начальное количество групп. Это количество даст вам число групп, используемых для сбора объектов с идентичными значениями хеша. Если в хеш-таблицу вставляется слишком много элементов, количество коллизий растет, и производительность извлечения элементов страдает.

Если вы примерно знаете, сколько элементов в конечном итоге будут в вашей таблице, можете установить количество групп. Обычно вы устанавливаете его где-то между 75% и 150% от ожидаемого числа элементов. Некоторые исследователи полагают, что количество групп должно быть выражено простым числом, чтобы предотвратить кластеризацию ключей. Однако на этот счет нет общего мнения. Стандартная библиотека использует количества групп, выраженные степенью двойки, по умолчанию — 16. (Любое указанное вами значение размера таблицы автоматически округляется до следующей степени 2.)

Конечно, вы не всегда знаете, сколько элементов придется хранить, или же ваше начальное предположение может оказаться слишком заниженным. Если хеш-таблица становится слишком полной, для нее должно быть выполнено *повторное хеширование*. Для повторного хеширования создается новая таблица с большим количеством групп, все элементы старой вставляются в новую, а старая таблица отбрасывается.

Коэффициент загрузки (load factor) определяет момент рехеширования таблицы. Например, если коэффициент загрузки равен 0,75 (по умолчанию так и есть), а таблица наполнена более чем на 75%, она автоматически повторно хешируется с вдвое увеличенным количеством групп. Для большинства приложений разумно оставить коэффициент загрузки равным 0,75.

Хеш-таблицы можно использовать для реализации нескольких важных структур данных. Простейшая среди них — тип *набор* (set). Набор — это коллекция элементов, не содержащая дубликатов. Метод `add()` набора сначала пытается найти добавляемый объект, и добавляет только в случае, когда его в наборе еще нет.

Библиотека коллекций Java предлагает класс `HashSet`, реализующий набор на основе хеш-таблицы. Вы добавляете элементы в него методом `add()`. Метод `contains()` переопределен, чтобы осуществлять быстрый поиск элемента в наборе. Он проверяет элементы только одной группы, а не все элементы коллекции.

Итератор хеш-набора перебирает все группы по очереди. Поскольку хеширование разбрасывает элементы по таблице, они перебираются внешне в случайном порядке. Вы должны использовать `HashSet` только тогда, когда порядок элементов в коллекции для вас не имеет значения.

Пример программы, представленный в конце этого раздела (листинг 13.2) читает слова из `System.in`, добавляет их в набор, а затем печатает все слова в наборе. Например, вы можете направить программе англоязычный текст “Алисы в стране чудес” (который доступен по адресу `http://www.gutenberg.net`), запустив ее из командной оболочки следующим образом:

```
java SetTest < alice30.txt
```

Программа читает все слова из входного потока и добавляет их в хеш-набор. Затем выполняется итерация по уникальным словам набора и печатается их количество. (“Алиса в стране чудес” состоит из 5909 уникальных слов, включая пометки об авторских правах в начале.) Слова извлекаются в случайном порядке.



Внимание! Будьте осторожны с изменением элементов набора. Если хеш-код элемента изменится, то этот элемент уже не будет находиться в правильной позиции структуры данных.

Листинг 13.2. Содержимое файла `SetTest.java`

```
1.  import java.util.*;
2.
3.  /**
4.   * Эта программа использует набор для распечатки всех уникальных
5.   * слов из System.in.
6.   * @version 1.10 2003-08-02
7.   * @author Cay Horstmann
8.   */
9.  public class SetTest
10. {
11.   public static void main(String[] args)
12.   {
13.      Set<String> words = new HashSet<String>(); // HashSet реализует Set
14.      long totalTime = 0;
15.      Scanner in = new Scanner(System.in);
16.      while (in.hasNext())
17.      {
18.         String word = in.next();
```

```
19.          long callTime = System.currentTimeMillis();
20.          words.add(word);
21.          callTime = System.currentTimeMillis() - callTime;
22.          totalTime += callTime;
23.       }
24.       Iterator<String> iter = words.iterator();
25.       for (int i = 1; i <= 20; i++)
26.          System.out.println(iter.next());
27.       System.out.println(". . .");
28.       System.out.println(words.size() + " уникальных слов. " + totalTime
29.          + " миллисекунд.");
30.    }
31. }
```

API java.util.HashSet<E> 1.2

- `HashSet()`

  Конструирует пустой хеш-набор.

- `HashSet(Collection<? Extends E> elements)`

  Конструирует хеш-набор и добавляет в него все элементы коллекции.

- `HashSet(int initialCapacity)`

  Конструирует пустой хеш-набор заданной емкости (количество групп).

- `HashSet(int initialCapacity, float loadFactor)`

  Конструирует пустой хеш-набор заданной емкости и с заданным коэффициентом загрузки (число между `0.0` и `1.0`, определяющее предельный процент заполнения хеш-таблицы, по достижении которого происходит повторное хеширование).

API java.lang.Object 1.0

- `int hashCode()`

  Возвращает хеш-код данного объекта. Хеш-код может быть любым целым числом — положительным или отрицательным. Определения методов `equals()` и `hashCode()` должны быть согласованы: если `x.equals(y)` равно `true`, то `x.hashCode()` должно возвращать то же значение, что и `y.hashCode()`.

Древовидные наборы

Класс `TreeSet` (древовидный набор) подобен хеш-набору, но с одним дополнительным усовершенствованием. Он представляет собой *сортированную коллекцию*. Вы вставляете элементы в коллекцию в любом порядке. Когда же выполняете итерацию по ее элементам, значения оказываются автоматически представленными в отсортированном порядке. Например, предположим, что вы вставили три строки и затем прошлись по всем добавленным элементам:

```
SortedSet<String> sorter = new TreeSet<String>(); // TreeSet реализует SortedSet
sorter.add("Bob");
sorter.add("Amy");
sorter.add("Carl");
for (String s : sorter) System.println(s);
```

Затем значения печатаются в отсортированном порядке: `Amy, Bob, Carl`. Как следует из имени класса, сортировка обеспечивается древовидной структурой данных. (Современная реализация использует "красно-белое дерево" (red-black tree). Детальное описание этих структур приведено в издании *Алгоритмы: построение и анализ. 2-е издание. Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн* (ИД "Вильямс", 2006 г.).) Всякий раз при добавлении элемента в дерево, он помещается в правильную отсортированную позицию. Таким образом, итератор всегда перебирает элементы в сортированном порядке.

Добавление элемента в дерево происходит медленнее, чем добавление в хеш-таблицу, но все же намного быстрее, чем в правильное место массива или связного списка. Если дерево содержит n элементов, то в среднем требуется $\log_2 n$ сравнений, чтобы найти правильное положение для нового элемента. Например, если дерево уже содержит 1000 элементов, добавление нового требует около 10 сравнений.

Таким образом, добавление элементов в `TreeSet` несколько медленнее, чем добавление в `HashSet` — см. табл. 13.3. для сравнения, — но зато `TreeSet` автоматически сортирует элементы.

Таблица 13.3. Хеш-коды, возвращенные функцией `hashCode()`

Документ	Всего слов	Количество уникальных слов	HashSet	TreeSet
"Алиса в стране чудес"	28195	5909	5 с	7 с
"Граф Монте-Кристо"	466300	37545	75 с	98 с

API `java.util.TreeSet<E> 1.2`

- `TreeSet()`

  Конструирует пустой древовидный набор.

- `TreeSet(Collection<? Extends E> elements)`

  Конструирует древовидный набор и добавляет все элементы коллекции.

Сравнение объектов

Откуда `TreeSet` узнает о том, как ему следует сортировать элементы? По умолчанию он предполагает, что вы вставляете элементы, реализующие интерфейс `Comparable`. Этот интерфейс определяет единственный метод:

```
public interface Comparable<T>
{
   int compareTo(T other);
}
```

Вызов `a.compareTo(b)` должен вернуть `0`, если `a` и `b` эквивалентны, отрицательное число, если `a` следует перед `b` в порядке сортировки, или же положительное число, если `a` следует после `b`. Точное значение не играет роли; важен только знак (>0, 0, или <0). Несколько стандартных классов платформы Java реализуют интерфейс `Comparable`. Примером может служить класс `String`. Его метод `compareTo()` сравнивает строки в алфавитном порядке (иногда называемом *лексикографическим порядком*).

Если вы вставляете свои собственные объекты, то должны определить порядок их сортировки самостоятельно, реализуя интерфейс `Comparable`. В классе `Object` не предусмотрена реализация `Comparable` по умолчанию.

Например, вот как можно сортировать объекты Item по номеру детали (partNumber):

```
class Item implements Comparable<Item>
{
public int compareTo(Item other)
{
    return partNumber - other.partNumber;
}
. . .
}
```

Если вы сравниваете два *положительных* целых, таких как номер детали в нашем примере, то вы можете просто вернуть разницу — она будет отрицательной, если первый элемент должен следовать перед вторым, будет равна нулю, если номера деталей равны, и в противном случае окажется положительной.



Внимание! Этот трюк работает, только если целые числа принадлежат к относительно небольшому диапазону. Если x окажется очень большим положительным числом, а y — большим отрицательным, то их вычитание может привести к переполнению.

Однако использование интерфейса Comparable для определения порядка сортировки имеет очевидные ограничения. Каждый данный класс может реализовать интерфейс лишь однажды. Но что вы будете делать, если нужно сортировать группу элементов по номеру детали в одной коллекции, и по описанию — в другой? Более того, что делать, если нужно сортировать объекты класса, чей разработчик не позаботился о реализации интерфейса Comparable?

В таких ситуациях вы указываете древовидному набору, чтобы он использовал другой метод сравнения, передав ему объект Comparator в конструкторе TreeSet. Интерфейс Comparator объявляет метод compare() с двумя явными параметрами:

```
public interface Comparator<T>
{
  int compare(T a, T b);
}
```

Точно так же, как в случае метода compareTo(), метод compare() возвращает отрицательное целое, если a идет перед b, ноль — если они идентичны, и положительное целое в противном случае.

Для сортировки элементов по их описанию просто определите класс, реализующий интерфейс Comparator:

```
class ItemComparator implements Comparator<Item>
{
  public int compare(Item a, Item b)
  {
     String descrA = a.getDescription();
     String descrB = b.getDescription();
     return descrA.compareTo(descrB);
  }
}
```

Затем вы передаете объект этого класса конструктору древовидного набора:

```
ItemComparator comp = new ItemComparator();
SortedSet<Item> sortByDescription = new TreeSet<Item>(comp);
```

Если вы конструируете дерево с компаратором, оно использует этот объект всякий раз, когда нужно сравнить два элемента.

Отметим, что этот компаратор элементов не имеет собственных данных. Он служит просто "держателем" метода сравнения. Такие объекты иногда называют *функциональными объектами*.

Функциональные объекты часто создаются "на лету" — как экземпляры анонимных внутренних классов:

```
SortedSet<Item> sortByDescription = new TreeSet<Item>(
   new Comparator<Item>()
{
    public int compare(Item a, Item b)
    {
        String descrA = a.getDescription();
        String descrB = b.getDescription();
        return descrA.compareTo(descrB);
    }
});
```



На заметку! В действительности интерфейс `Comparator<T>` объявлен с двумя методами: `compare()` и `equals()`. Разумеется, *каждый* класс имеет метод `equals()`, поэтому кажется не слишком целесообразным добавление этого метода в декларацию интерфейса. Документация по API объясняет, что вы не обязаны переопределять метод `equals()`, но если вы это сделаете, то в некоторых случаях можно повысить производительность. Например, метод `addAll()` класса `TreeSet` работает наиболее эффективно, если вы добавляете элементы из другой коллекции, использующей такой же компаратор.

Если вы вернетесь к табл. 13.3, у вас может возникнуть вопрос: почему бы всегда не использовать древовидный набор вместо хеш-набора? В конце концов, добавление элементов происходит не намного медленнее, а элементы при этом автоматически сортируются. Ответ зависит от данных, с которыми приходится иметь дело. Если вам не нужны сортированные данные, нет смысла платить за сортировку. Что более важно, для некоторых данных намного сложнее определить порядок сортировки, чем хеш-функцию. Хеш-функция должна лишь достаточно хорошо (равномерно) распределять объекты, в то время как функция сравнения должна различать объекты с абсолютной точностью.

Чтобы нагляднее проявить это различие, рассмотрим задачу с коллекционированием набора прямоугольников. Если вы используете `TreeSet`, то придется применять `Comparator<Rectangle>`. Как следует сравнивать два прямоугольника? По их площади? Но это не работает. Можно создать два прямоугольника с разными координатами, но одинаковой площадью. Порядок сортировки для дерева должен быть *полностью упорядоченным*. Любые два элемента должны быть сравнимы, и сравнение должно вернуть 0, если элементы эквивалентны. Можно представить такой порядок для прямоугольников (лексикографическое упорядочивание координат), но он представляется не естественным и сложным в вычислении. По контрасту хеш-функция всегда определена для класса `Rectangle`. Она просто хеширует координаты.



На заметку! Начиная с версии Java SE 6, класс `TreeSet` реализует интерфейс `NavigableSet`. Этот интерфейс добавляет несколько удобных методов для нахождения элементов и обратного прохода. Подробности читайте во врезках, посвященных API.

Листинг 13.3. Содержимое файла `TreeSetTest.java`

```
1. /**
2.  * @version 1.10 2004-08-02
3.  * @author Cay Horstmann
4.  */
5.
6. import java.util.*;
7.
8. /**
9.  * Программа сортирует набор элементов,
10. * сравнивая их описания.
11. */
12.public class TreeSetTest
13.{
14.  public static void main(String[] args)
15.  {
16.     SortedSet<Item> parts = new TreeSet<Item>();
17.     parts.add(new Item("Toaster", 1234));
18.     parts.add(new Item("Widget", 4562));
19.     parts.add(new Item("Modem", 9912));
20.     System.out.println(parts);
21.
22.     SortedSet<Item> sortByDescription = new TreeSet<Item>(new
23.       Comparator<Item>()
24.             {
25.             public int compare(Item a, Item b)
26.             {
27.                    String descrA = a.getDescription();
28.                    String descrB = b.getDescription();
29.                    return descrA.compareTo(descrB);
30.             }
31.             });
32.
33.     sortByDescription.addAll(parts);
34.     System.out.println(sortByDescription);
35.  }
36.}
37.
38./**
39. * Элемент с описанием и номером партии.
40. */
41.class Item implements Comparable<Item>
42.{
43.  /**
44.   * Конструирует элмент.
45.   * @param aDescription Описание элемента
46.   * @param aPartNumber Номер детали элемента
47.   */
48.  public Item(String aDescription, int aPartNumber)
49.  {
50.     description = aDescription;
51.     partNumber = aPartNumber;
52.  }
```

```
53.  /**
54.   * Извлекает описание элемента.
55.   * @return Описание
56.   */
57.  public String getDescription()
58.  {
59.     return description;
60.  }
61.
62.  public String toString()
63.  {
64.     return "[descripion=" + description
65.           + ", partNumber=" + partNumber + "]";
66.  }
67.
68.  public boolean equals(Object otherObject)
69.  {
70.     if (this == otherObject) return true;
71.     if (otherObject == null) return false;
72.     if (getClass() != otherObject.getClass()) return false;
73.     Item other = (Item) otherObject;
74.     return description.equals(other.description)
75.           && partNumber == other.partNumber;
76.  }
77.
78.  public int hashCode()
79.  {
80.     return 13 * description.hashCode() + 17 * partNumber;
81.  }
82.
83.  public int compareTo(Item other)
84.  {
85.     return partNumber - other.partNumber;
86.  }
87.
88.  private String description;
89.  private int partNumber;
90. }
```

API java.lang.Comparable<T> 1.2

- `int compareTo(T other)`

  Сравнивает этот объект с другим и возвращает отрицательное значение, если `this` следует перед `other`, ноль, если они рассматриваются как идентичные в данном порядке сортировки, и положительное значение — если `this` следует после `other`.

API java.lang.Comparator<T> 1.2

- `int compare(T a, T b)`

  Сравнивает два объекта и возвращает отрицательное значение, если `a` следует перед `b`, ноль, если они рассматриваются как идентичные при данном порядке сортировки, и положительное значение, если `a` следует после `b`.

API java.lang.SortedSet<E> 1.2

- `Comparator<? super E> comparator()`

  Возвращает компаратор для сортировки элементов или `null`, если элементы сравниваются методом `comapreTo()` интерфейса `Comparable`.

- `E first()`
- `E last()`

  Возвращают минимальный и максимальный элементы сортированного набора.

API java.lang.NavigableSet<E> 6

- `E higher(E value)`
- `E lower(E value)`

  Возвращают минимальный элемент, больший `value`, либо максимальный, меньший `value`, или же `null`, если такого элемента не найдено.

- `E ceiling(E value)`
- `E floor(E value)`

  Возвращают минимальный элемент, больший или равный `value`, либо максимальный, меньший или равный `value`, или же `null`, если такого элемента не найдено.

- `E pollFirst()`
- `E pollLast()`

  Удаляют и возвращают минимальный или максимальный элемент в наборе, либо же `null`, если набор пуст.

- `Iterator<E> descendingIterator()`

  Возвращает итератор, проходящий набор в обратном порядке.

API java.lang.TreeSet<E> 6

- `TreeSet()`

  Конструирует древовидный набор для сортировки объектов `Comparable`.

- `TreeSet(Comparator<? Super E> c)`

  Конструирует древовидный набор и использует указанный компаратор для сортировки элементов.

- `TreeSet(SortedSet<? extends E> elements)`

  Конструирует древовидный набор, добавляет в него элементы сортированного набора, и использует тот же компаратор элементов, что и данный сортированный набор.

Однонаправленные и двунаправленные очереди

Как мы уже говорили, очередь позволяет эффективно добавлять элементы в хвост и удалять элементы из головы. Двусторонняя очередь (deque) позволяет эффективно добавлять и удалять элементы на обоих концах. Добавление элементов в середину очереди не поддерживается. В Java SE 6 появился интерфейс `Deque`. Он реализован клас-

сами ArrayDeque и LinkedList — оба представляют двустороннюю очередь, которая может расти по мере необходимости. В главе 14 вы увидите примеры ограниченных однонаправленных и двунаправленных очередей.

API java.util.Queue<E> 5.0

- boolean add(E element)
- boolean offer(E element)

  Добавляют заданный элемент в конец очереди и возвращают true, если очередь не полна. Если очередь полна, первый метод генерирует исключение IllegalStateException, в то время как второй возвращает false.

- E remove()
- E poll()

  Удаляют и возвращают элемент из головы очереди, если очередь не пуста. Если же очередь пуста, то первый метод генерирует исключение NoSuchElementException, в то время как второй возвращает null.

- E element()
- E peek()

  Возвращают элемент из головы очереди, не удаляя его, если очередь не пуста. Если же очередь пуста, то первый метод генерирует исключение NoSuchElementException, в то время как второй возвращает null.

- void addFirst(E element)
- void addLast(E element)
- boolean offerFirst(E element)
- boolean offerLast(E element)

  Добавляют заданный элемент в голову или хвост двунаправленной очереди. Если очередь полна, первые два метода генерируют исключение IllegalStateException, в то время как последние два возвращают false.

- E removeFirst()
- E removeLast()
- E pollFirst()
- E pollLast()

  Удаляют и возвращают элемент из головы очереди, если очередь не пуста. Если же она пуста, первые два метода генерируют исключение NoSuchElementException, в то время как последние два возвращают null.

- E getFirst()
- E getLast()
- E peekFirst()
- E peekLast()

  Возвращают элемент из головы очереди, не удаляя ее, если очередь не пуста. Если же она пуста, первые два метода генерируют исключение NoSuchElementException, в то время как последние два возвращают null.

API java.util.ArrayDeque<E> 6

- ArrayDeque()

- `ArrayDeque(int initialCapacity)`

  Конструируют неограниченные двунаправленные очереди с начальной емкостью 16 либо заданной начальной емкостью.

Очереди с приоритетами

Очереди с приоритетами извлекают элементы в отсортированном порядке после того, как они были вставлены в произвольном порядке. То есть при всяком вызове метода `remove()` вы получаете самый маленький элемент, находящийся в очереди. Однако очередь с приоритетами не сортирует все свои элементы. Если выполняется итерация по элементам, они не обязательно оказываются отсортированными. Очередь с приоритетами использует элегантную и эффективную структуру данных под названием *куча* (heap). Куча — это самоорганизующееся бинарное дерево, в котором операции `add()` и `remove()` вызывают перемещение наименьшего элемента в корень, не тратя времени на сортировку всех элементов.

Подобно `TreeSet`, очередь с приоритетами может как содержать элементы класса, реализующего интерфейс `Comparable`, либо принимать объект `Comparator` в конструкторе.

Типичное использование очереди с приоритетами — планирование задач. Каждая задача имеет приоритет. Задачи добавляются в случайном порядке. Когда новая задача может быть запущена, то наиболее высокоприоритетная очередь удаляется из очереди. (Поскольку традиционно приоритет 1 считается наивысшим, операция `remove()` выдает минимальный элемент.)

Код в листинге 13.4 демонстрирует очередь с приоритетами в действии. В отличие от итерации в `TreeSet`, здесь итерация не проходит по элементам в отсортированном порядке. Однако удаление всегда касается наименьшего элемента.

Листинг 13.4. Содержимое файла `PriorityQueueTest.java`

```
1.  import java.util.*;
2.  /**
3.   * Эта программа демонстрирует применение очереди с приоритетами.
4.   * @version 1.00 2004-08-03
5.   * @author Cay Horstmann
6.   */
7.  public class PriorityQueueTest
8.  {
9.   public static void main(String[] args)
10.  {
11.    PriorityQueue<GregorianCalendar> pq = new PriorityQueue<GregorianCalendar>();
12.    pq.add(new GregorianCalendar(1906, Calendar.DECEMBER, 9)); // G. Hopper
13.    pq.add(new GregorianCalendar(1815, Calendar.DECEMBER, 10)); // A. Lovelace
14.    pq.add(new GregorianCalendar(1903, Calendar.DECEMBER, 3)); // J. von Neumann
15.    pq.add(new GregorianCalendar(1910, Calendar.JUNE, 22)); // K. Zuse
16.    System.out.println("Итерация по элементам...");
17.    for (GregorianCalendar date : pq)
18.      System.out.println(date.get(Calendar.YEAR));
19.    System.out.println("Удаление элементов...");
20.    while (!pq.isEmpty())
21.      System.out.println(pq.remove().get(Calendar.YEAR));
22.  }
23. }
```

API java.util.PriorityQueue<E> 5.0

- `PriorityQueue()`
- `PriorityQueue(int initialCapacity)`

  Конструируют очередь с приоритетами для хранения объектов `Comparable`.

- `PriorityQueue(int initialCapacity, Comparator<? super E> c)`

  Конструирует очередь с приоритетами и использует указанный компаратор для сортировки элементов.

Карты

Набор (set) — это коллекция, которая позволяет быстро находить существующий элемент. Однако для того, чтобы искать этот элемент, нужно обладать его точной копией. Это не слишком распространенная операция поиска — обычно у вас есть некоторая ключевая информация, и вы хотите найти ассоциированный с ней элемент. Для этой цели предназначена структура данных типа карты (map). Карта хранит пары "ключ/значение". Вы можете найти значение, если предоставите ключ. Например, вы можете сохранить таблицу записей о сотрудниках, где ключами служат идентификаторы сотрудника, а значениями — объекты `Employee`.

Библиотека Java предлагает две реализации карт общего назначения: `HashMap` и `TreeMap`. Оба класса реализуют интерфейс `Map`.

Хеш-карта хеширует ключи, а карта-дерево использует общий порядок ключей для организации поискового дерева. Функции хеширования или сравнения применяются *только к ключам*. Значения, ассоциированные с ключами, не хешируются и не сравниваются.

Когда использовать хеш-карту, а когда карту-дерево? Как и в случае с наборами, хеширование работает немного быстрее, и является предпочтительным выбором, если вам не нужно перебирать ключи в отсортированном порядке.

Вот как можно создать хеш-карту для хранения записей о сотрудниках:

```
Map<String, Employee> staff = new HashMap<String, Employee>();
   // HashMap реализует Map
Employee harry = new Employee("Harry Hacker");
staff.put("987-98-9996", harry);
. . .
```

Всякий раз, когда вы добавляете объект к карте, вы должны указать также его ключ. В нашем случае ключом является строка, а соответствующим значением — объект `Employee`.

Чтобы извлечь объект, вы должны использовать (и, следовательно, помнить) ключ:

```
String s = "987-98-9996";
e = staff.get(s); // получает harry
```

Если в карте нет информации с указанным ключом, `get()` возвращает `null`.

Ключи должны быть уникальными. Вы не можете сохранить два значения с одинаковыми ключами. Если дважды вызвать метод `put()` с одним и тем же ключом, то второе значение заменит первое. Фактически `put()` возвращает предыдущее значение, сохраненное с параметром ключа.

Метод `remove()` удаляет элемент с заданным ключом из карты. Метод `size()` возвращает количество элементов в карте.

Каркас коллекций не рассматривает саму карту как коллекцию. (Другие каркасы структур данных рассматривают карту как коллекцию *пар*, или коллекцию значений, индексированных ключами.) Однако вы можете получить представления (views) карты — объекты, реализующие интерфейс `Collection`, либо один из его подинтерфейсов.

Существует три таких представления: набор ключей, коллекция значений (не набор) и набор пар "ключ/значение". Ключи и пары "ключ/значение" формируют наборы, потому что в карте может быть только по одной копии каждого ключа. Методы

```
Set<K> keySet()
Collection<K> values()
Set<Map.Entry<K, V>> entrySet()
```

возвращают эти три представления. (Элементы последнего набора являются объектами статического внутреннего класса `Map.Entry`.)

Обратите внимание, что `keySet` — это не `HashSet` или `TreeSet`, а объект некоторого другого класса, реализующего интерфейс `Set`. Интерфейс `Set` расширяет интерфейс `Collection`. Поэтому вы можете использовать `keySet` подобно любой другой коллекции.

Например, можно перечислить все ключи в карте:

```
Set<String> keys = map.keySet();
for (String key : keys)
{
    делать что-нибудь с key
}
```



Совет. Если вы хотите просмотреть и ключи и значения, то можно избежать поиска значений перечислением *вхождений* (entries). Используйте следующий скелет кода:

```
for (Map.Entry<String, Employee> entry : staff.entrySet())
{
     String key = entry.getKey();
     Employee value = entry.getValue();
     делать что-то с key, value
}
```

Если вы вызываете метод `remove()` итератора, то в действительности удаляете из карты ключ *и ассоциированное значение*. Однако вы не можете *добавить* элемент к представлению — набору ключей. Нет смысла в добавлении ключа без добавления ассоциированного с ним значения. Если вы попытаетесь вызвать метод `add()`, он сгенерирует исключение `UnsupportedOperationException`. Представление набора вхождений подчиняется тому же ограничению, даже несмотря на то, что операция добавления новой пары "ключ/значение" имеет концептуальный смысл.

Код в листинге 13.5 иллюстрирует работу карты. Сначала мы добавляем в карту пары "ключ/значение". Затем удаляем один ключ из карты, который также удаляет ассоциированное с ним значение. Затем изменяем значение, ассоциированное с ключом, и вызываем метод `get()` для нахождения значения. И, наконец, выполняем итерацию по набору вхождений.

Листинг 13.5. Содержимое файла `MapTest.java`

```
1. import java.util.*;
2. /**
3.  * Эта программа демонстрирует использование карты с ключом типа String
4.  * и значением Employee.
5.  * @version 1.10 2004-08-02
6.  * @author Cay Horstmann
7.  */
8. public class MapTest
9. {
10.  public static void main(String[] args)
11.  {
12.     Map<String, Employee> staff = new HashMap<String, Employee>();
13.     staff.put("144-25-5464", new Employee("Amy Lee"));
14.     staff.put("567-24-2546", new Employee("Harry Hacker"));
15.     staff.put("157-62-7935", new Employee("Gary Cooper"));
16.     staff.put("456-62-5527", new Employee("Francesca Cruz"));
17.     // Печать всех вхождений.
18.     System.out.println(staff);
19.     // Удаление вхождения.
20.     staff.remove("567-24-2546");
21.     // Замена вхождения.
22.     staff.put("456-62-5527", new Employee("Francesca Miller"));
23.     // Поиск значения.
24.     System.out.println(staff.get("157-62-7935"));
25.     // Итерация по вхождениям.
26.     for (Map.Entry<String, Employee> entry : staff.entrySet())
27.     {
28.        String key = entry.getKey();
29.        Employee value = entry.getValue();
30.        System.out.println("key=" + key + ", value=" + value);
31.     }
32.  }
33. }
34. /**
35.  * Минималистский класс, описывающий сотрудника для тестирования.
36.  */
37. class Employee
38. {
39.  /**
40.   * Конструирует объект сотрудника с зарплатой, равной $0.
41.   * @param n the employee name
42.   */
43.  public Employee(String n)
44.  {
45.     name = n;
46.     salary = 0;
47.  }
48.  public String toString()
49.  {
50.     return "[name=" + name + ", salary=" + salary + "]";
51.  }
52.  private String name;
53.  private double salary;
54. }
```

API java.util.Map<K, V> 1.2

- `V get(K key)`

  Получает значение, ассоциированное с ключом; возвращает объект, ассоциированный с ключом, или `null`, если ключ не найден в карте. Ключ может быть `null`.

- `V put(K key, V value)`

  Помещает в карту ассоциацию ключа со значением. Если ключ уже присутствует, новый объект заменяет старый, ранее ассоциированный с тем же ключом. Этот метод возвращает старое значение ключа или `null`, если ключ еще не присутствовал. Ключ может быть `null`, но значение не должно быть равно `null`.

- `void putAll(Map<? extends K, ? extends V> entries)`

  Добавляет вхождения из указанной карты в данную.

- `boolean containsKey(Object key)`

  Возвращает `true`, если ключ присутствует в карте.

- `boolean containsValue(Object value)`

  Возвращает `true`, если значение присутствует в карте.

- `Set<Map.Entry<K, V>> entrySet()`

  Возвращает представление-набор объектов `Map.Entry`, т.е. пар "ключ/значение", хранящихся в карте. Вы можете удалять элементы из этого набора, но не можете добавлять новые.

- `Set<K> keySet()`

  Возвращает представление — набор всех ключей карты. Вы можете удалять элементы из этого набора, при этом ассоциированные значения также будут удалены, но вы не можете добавлять элементы.

- `Collection<V> values()`

  Возвращает представление — набор всех значений карты. Можно удалять элементы этого набора, при этом из карты будут удалены и соответствующие ключи, но добавлять элементы невозможно.

API java.util.HashMap<K, V> 1.2

- `HashMap()`
- `HashMap(int initialCapacity)`
- `HashMap(int initialCapacity, float loadFactor)`

  Конструируют пустую карту указанной емкости и с указанным коэффициентом загрузки (число между `0.0` и `1.0`, определяющее процент наполнения хеш-таблицы, после которого она должна быть хеширована повторно). Коэффициент наполнения по умолчанию равен `0.75`.

API java.util.HashMap<K, V> 1.2

- `TreeMap(Comparator<? super K> c)`

  Конструирует карту-дерево, используя указанный компаратор для сортировки ключей.

- `TreeMap(Map<? extends K, ? extends V> entries)`

  Конструирует карту-дерево и добавляет все вхождения из переданной карты.

- `TreeMap(SortedMap<? extends K, ? extends V> entries)`

  Конструирует карту-дерево, добавляет все вхождения из переданной сортированной карты, используя тот же компаратор, что и переданная сортированная карта.

API **java.util.SortedMap<K, V> 1.2**

- `Comparator<? super K> comparator()`

  Возвращает компаратор, используемый для сортировки ключей, либо `null`, если ключи сравниваются методом `compareTo()` интерфейса `Comparable`.

- `K firstKey()`
- `K lastKey()`

  Возвращает минимальный или максимальный ключи в карте.

Специализированные классы `Set` и `Map`

Библиотека классов коллекций включает несколько классов карт для специализированных нужд, и мы опишем их кратко в настоящем разделе.

"Слабые" хеш-карты

Класс `WeakHashMap` был спроектирован для решения одной интересной проблемы. Что случится со значением, чей ключ более не используется нигде в вашей программе? Предположим, что последняя ссылка на ключ исчезла. Значит, не остается никакого способа сослаться на объект-значение. Но поскольку ни одна часть программы не содержит обращений к данному ключу, то и соответствующая пара "ключ/значение" не может быть удалена из карты. Почему бы сборщику мусора не удалить ее? Разве это не его работа — удалять неиспользуемые объекты?

К сожалению, все не так просто. Сборщик мусора отслеживает *живые* объекты. До тех пор, пока объект карты жив, все группы в нем живы и не могут быть освобождены. Поэтому ваша программа должна позаботиться об удалении неиспользуемых значений из долгоживущих карт. Или же вы должны вместо этого использовать `WeakHashMap`. Эта структура данных сотрудничает со сборщиком мусора для удаления пар "ключ/значение", когда единственной ссылкой на ключ остается ссылка из вхождения данной карты.

Поясним, как работает этот механизм. `WeakHashMap` использует *слабые ссылки* для хранения ключей. Объект `WeakHashMap` содержит ссылку на другой объект, в нашем случае — ключ хеш-таблицы. Объекты этого типа трактуются сборщиком мусора специальным образом. Обычно если сборщик мусора обнаруживает, что определенный объект не имеет ссылок на него, он просто освобождает память, занятую им. Однако если объект доступен только из `WeakHashMap`, сборщик мусора также освобождает его, но помещает слабую ссылку на него в очередь. Операции `WeakHashMap` периодически проверяют эту очередь на предмет появления новых слабых ссылок. Появление такой ссылки в очереди говорит о том, что ключ более не используется никем, и что он должен быть удален. Тогда `WeakHashMap` удаляет ассоциированное вхождение.

Связные хеш-наборы и карты

В Java SE 1.4 появились классы `LinkedHashSet` и `LinkedHashMap`, запоминающие порядок вставки в них элементов. Таким образом, вы избегаете кажущегося случайным порядка элементов в хеш-таблице. По мере вставки вхождений в таблицу, они присоединяются к двунаправленному связному списку (рис. 13.9).

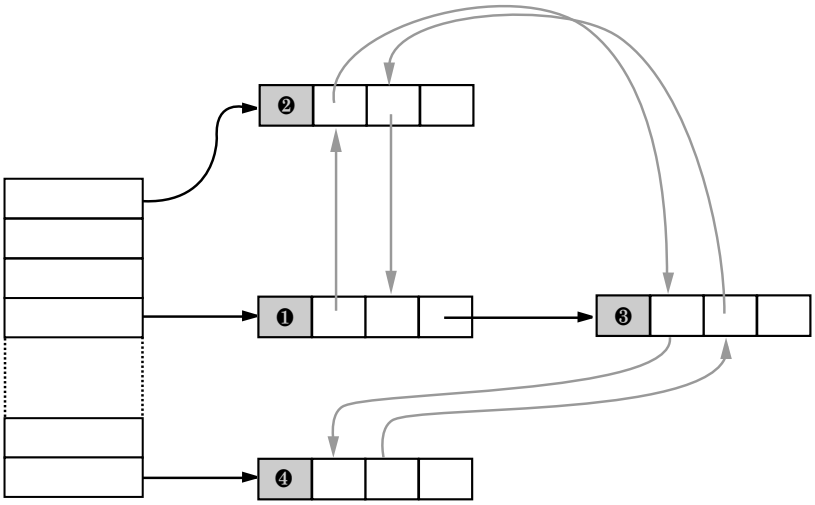


Рис. 13.9. Связная хеш-таблица

Например, рассмотрим следующие вставки в карту из листинга 13.5:

```
Map staff = new LinkedHashMap();
staff.put("144-25-5464", new Employee("Amy Lee"));
staff.put("567-24-2546", new Employee("Harry Hacker"));
staff.put("157-62-7935", new Employee("Gary Cooper"));
staff.put("456-62-5527", new Employee("Francesca Cruz"));
```

Затем `staff.keySet().iterator()` перечисляет ключи в следующем порядке:

```
144-25-5464
567-24-2546
157-62-7935
456-62-5527
```

А `staff.values().iterator()` перечисляет значения в таком порядке:

```
Amy Lee
Harry Hacker
Gary Cooper
Francesca Cruz
```

Связная хеш-карта для итерации по вхождениям может альтернативно использовать *порядок доступа*, а не порядок вставки. При каждом вызове `get()` или `put()` затронутое вхождение удаляется из его текущей позиции и вставляется в *конец* связного списка вхождений. (Затрагивается только позиция в связном списке, а не в группах хеш-таблицы. Вхождение всегда остается в том месте, которое соответствует хеш-коду его ключа.) Чтобы сконструировать такую хеш-карту, воспользуйтесь следующим вызовом:

```
LinkedHashMap<K, V>(initialCapacity, loadFactor, true)
```

Порядок доступа удобен для реализации дисциплины кэширования под названием "с наиболее давним использованием". Например, вы можете пожелать сохранять часто используемые вхождения в памяти и читать менее часто используемые из базы данных. Когда вы не находите вхождения в таблице, а таблица уже почти полна, тогда вы получаете итератор таблицы и удаляете несколько первых элементов, которые он перечисляет. Эти элементы использовались наиболее давно.

Этот процесс можно даже автоматизировать. Сформируйте подкласс `LinkedHashMap` и переопределите метод

```
protected boolean removeEldestEntry(Map.Entry<K, V> eldest)
```

Добавление нового вхождения затем вызовет удаление вхождения `eldest` в каждом случае, когда ваш метод вернет `true`. Например, следующий кэш удерживается в максимальном размере в 100 элементов:

```
Map<K, V> cache = new
LinkedHashMap<K, V>(128, 0.75F, true)
{
   protected boolean removeEldestEntry(Map.Entry<K, V> eldest)
   {
     return size() > 100;
   }
};
```

Альтернативно можно просмотреть вхождение `eldest`, чтобы решить, стоит ли его удалять. Например, вы можете проверить временную метку, хранящуюся в этом вхождении.

Перечислимые наборы и карты

`EnumSet` — эффективная реализация набора с элементами, относящимися к перечислимому типу. Поскольку перечислимый тип имеет ограниченное количество экземпляров, `EnumSet` внутренне реализован в виде битовой последовательности. Бит установлен в 1, если соответствующее значение перечисления присутствует в наборе.

Класс `EnumSet` не имеет общедоступного конструктора. Для конструирования такого набора используется статический фабричный метод:

```
enum Weekday { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY };
EnumSet<Weekday> always = EnumSet.allOf(Weekday.class);
EnumSet<Weekday> never = EnumSet.noneOf(Weekday.class);
EnumSet<Weekday> workday = EnumSet.range(Weekday.MONDAY, Weekday.FRIDAY);
EnumSet<Weekday> mwf = EnumSet.of(Weekday.MONDAY, Weekday.WEDNESDAY, Weekday.FRIDAY);
```

Вы можете использовать обычные методы интерфейса `Set` для модификации `EnumSet`.

`EnumMap` — это карта с ключами, относящимися к перечислимому типу. Она реализована просто и эффективно в виде массива значений. Вам нужно специфицировать тип ключа в конструкторе:

```
EnumMap<Weekday, Employee> personInCharge =
   new EnumMap<Weekday, Employee>(Weekday.class);
```



На заметку! В документации по API `EnumSet` вы увидите странно выглядящие параметры типа вроде `E extends Enum<E>`. Это означает просто "`E` является перечислимым типом". Перечислимые типы расширяют обобщенный класс `Enum`. Например, `Weekday` расширяет `Enum<Weekday>`.

Хеш-карты идентичности

В Java SE 1.4 появился еще один класс — `IdentityHashMap`, предназначенный для довольно специальных целей, где хеш-значения ключей должны вычисляться не методом `hashCode()`, а `System.identityHashCode`. Это метод, который использует `Object.hashCode()` для вычисления хеш-кода на основании адреса объекта в памяти. К тому же для сравнения объектов `IdentityHashMap` применяет операцию ==, а не `equals()`.

Другими словами, разные объекты-ключи рассматриваются как различающиеся, даже если они имеют одинаковое содержимое. Этот класс удобен для реализации алгоритмов обхода объектов (таких как сериализация объектов), когда вам нужно отслеживать, какие объекты уже были пройдены.

API java.util.WeakHashMap<K, V> 1.2

- `WeakHashMap()`
- `WeakHashMap(int initialCapacity)`
- `WeakHashMap(int initialCapacity, float loadFactor)`

  Конструируют пустую хеш-карту с указанной емкостью и коэффициентом загрузки.

API java.util.LinkedHashSet<K, V> 1.4

- `LinkedHashSet()`
- `LinkedHashSet(int initialCapacity)`
- `LinkedHashSet(int initialCapacity, float loadFactor)`

  Конструируют пустой хеш-набор с указанной емкостью и коэффициентом загрузки.

API java.util.LinkedHashMap<K, V> 1.4

- `LinkedHashMap()`
- `LinkedHashMap(int initialCapacity)`
- `LinkedHashMap(int initialCapacity, float loadFactor)`
- `LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)`

  Конструируют пустую связную хеш-карту с заданной емкостью, коэффициентом загрузки и порядком. Параметр `accessOrder` в случае равенства `true` задает прямой порядок, в случае `false` — обратный.

- `protected boolean removeEldestEntry(Map.Entry<K, V> eldest)`

  Метод должен быть переопределен, чтобы возвращать `true`, если вы хотите удалить вхождение `eldest`. Параметр `eldest` — это вхождение, чье удаление рассматривается. Метод вызывается после того, как вхождение добавлено в карту. Реализация по умолчанию возвращает `false` — т.е. по умолчанию старые элементы не удаляются. Однако вы можете переопределить этот метод для выборочного возврата `true`; например, если наиболее старое вхождение отвечает определенным условиям или размер карты достигает определенной величины.

API java.util.EnumSet<E extends Enum<E>> 5.0

- `static <E extends Enum<E>> EnumSet<E> allOf(Class<E> enumType)`

  Возвращает набор, содержащий все значения заданного типа перечисления.

- `static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> enumType)`

  Возвращает пустой набор, способный хранить значений заданного перечислимого типа.

- `static <E extends Enum<E>> EnumSet<E> range(E from, E to)`

  Возвращает набор, содержащий все значения от `from` до `to` включительно.

- `static <E extends Enum<E>> EnumSet<E> of(E value)`
- `static <E extends Enum<E>> EnumSet<E> of(E value, E... values)`

  Возвращают набор, содержащий заданные значения.

API java.util.EnumMap<K, extends Enum<V>, K> 5.0

- `EnumMap(Class<K> keyType)`

  Конструирует пустую карту с ключами заданного типа.

API java.util.IdentityHashMap<K, V> 1.4

- `IdentityHashMap()`
- `IdentityHashMap(int expectedMaxSize)`

  Конструируют пустую хеш-карту идентичности, чья емкость равна минимальной степени 2, превышающей `1.5*expectedMaxSize` (по умолчанию `expectedMaxSize` принимается равным 21).

API java.lang.System 1.0

- `static int identityHashCode(Object obj)` **1.1**

  Возвращает хеш-код (полученный на основе адреса памяти объекта), вычисленный методом `Object.hashCode()`, даже если класс, к которому относится `obj`, переопределяет метод `hashCode()`.

Каркас коллекций

Каркас (framework) — это набор классов, формирующих базу для построения расширенной функциональности. Каркас содержит суперклассы, обладающие полезной функциональностью, политики и механизмы. Пользователь каркаса формирует подклассы для расширения функциональности без изменения базовых механизмов. Например, Swing — каркас для построения пользовательских интерфейсов.

Библиотека коллекций Java формирует каркас классов коллекций. Она определяет множество интерфейсов и абстрактных классов, необходимых для реализации коллекций (рис. 13.10), а также предписывает определенные механизмы, такие как протокол итераций. Вы можете использовать эти классы коллекций, ничего не зная о каркасе — именно так мы поступали в предыдущих разделах. Однако если вы хотите реализовать обобщенные алгоритмы, работающие с множеством типов коллекций, или хотите добавить новый тип коллекции, полезно будет разобраться в каркасе в целом.

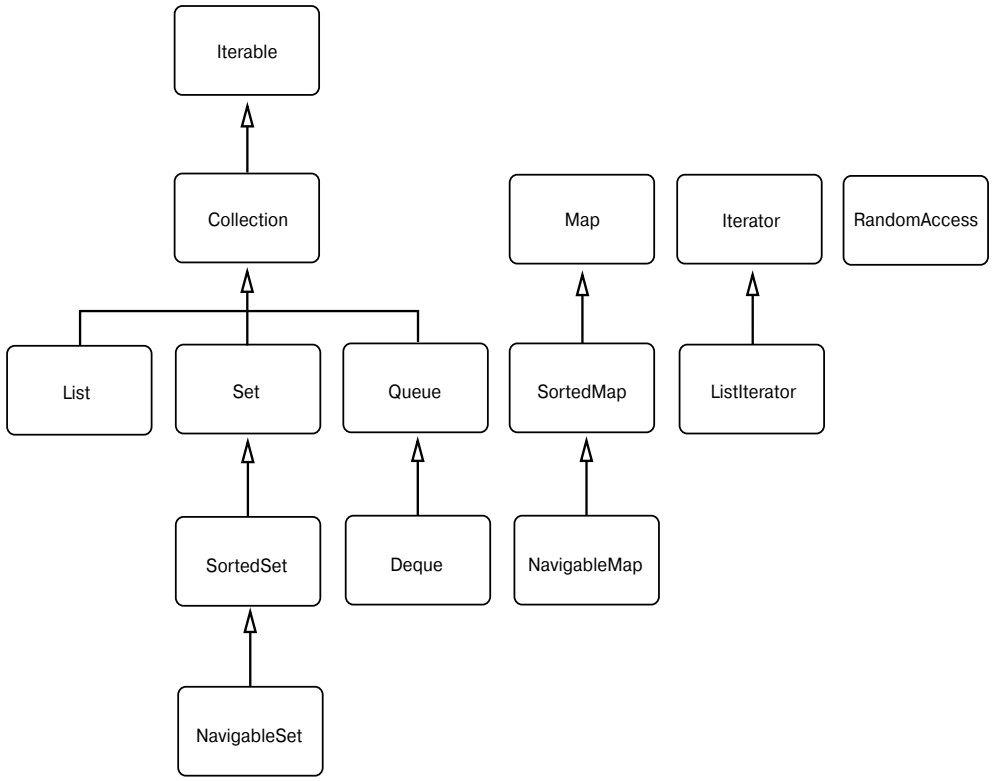


Рис. 13.10. Интерфейсы каркаса коллекций

Существует два фундаментальных интерфейса коллекций: `Collection` и `Map`. Вы вставляете элементы в коллекцию следующим методом:

```
boolean add(E element)
```

Однако карты содержат пары "ключ/значение", и вы используете метод `put()` для вставки этих пар:

```
V put(K key, V value)
```

Чтобы прочитать элементы из коллекции, вы посещаете их итератором. Однако вы можете читать значения из карты методом `get()`:

```
V get(K key)
```

Список представляет собой *упорядоченную коллекцию*. Элементы добавляются в определенную позицию контейнера. Объект может быть помещен в позицию двумя способами: по целочисленному индексу и через итератор списка. Интерфейс `List` определяет методы для произвольного доступа:

```
void add(int index, E element)
E get(int index)
void remove(int index)
```

Как уже говорилось, интерфейс `List` предоставляет методы произвольного доступа, независимо от того, эффективны ли они в данной реализации. Чтобы избежать

дорогостоящих операций произвольного доступа, в Java SE 1.4 появился новый интерфейс RandomAccess. Этот интерфейс не имеет методов, но вы можете использовать его для проверки, поддерживает ли конкретная коллекция эффективный произвольный доступ:

```
if (c instanceof RandomAccess)
{
   использовать алгоритм произвольного доступа
}
else
{
   использовать алгоритм последовательного доступа
}
```

Классы ArrayList и Vector реализуют интерфейс RandomAccess.



На заметку! С теоретической точки зрения имело бы смысл выделить интерфейс Array, расширяющий интерфейс List и объявляющий методы произвольного доступа. Если бы существовал отдельный интерфейс Array, то те алгоритмы, которые требуют произвольного доступа, могли бы использовать параметры Array, и вы никогда не смогли бы непреднамеренно применить их к коллекциям с медленным произвольным доступом. Однако проектировщики каркаса коллекций решили не определять отдельного интерфейса, поскольку хотели сохранить количество интерфейсов в библиотеке небольшим. К тому же они не хотели занимать патерналистскую позицию по отношению к программистам. Вы вольны применять к связным спискам алгоритмы прямого доступа — просто при этом следует отдавать себе отчет о том, что за это придется платить снижением производительности.

Интерфейс ListIterator определяет метод для добавления элемента перед позицией итератора:

```
void add(E element)
```

Чтобы получать и удалять элементы в определенной позиции, вы просто применяете методы next() и remove() интерфейса Iterator.

Интерфейс Set идентичен интерфейсу Collection, но поведение его методов определено более сжато. Метод add() набора должен отвергать дубликаты. Метод equals() набора должен быть определен так, что два набора считаются идентичными, если содержат одинаковые элементы, но не обязательно в одинаковом порядке. Метод hashCode() должен быть определен так, чтобы два набора с одинаковыми элементами порождали одинаковый хеш-код.

Зачем объявлять отдельные интерфейсы, если сигнатуры методов совпадают? Концептуально не все коллекции являются наборами. Наличие интерфейса Set позволяет программистам писать методы, принимающие только наборы.

Интерфейсы SortedSet и SortedMap применяют объект-компаратор, применяемый для сортировки, а также определяют методы для получения представлений подмножеств коллекций. Об этих представлениях мы поговорим в следующем разделе.

И, наконец, в Java SE 6 представлены интерфейсы NavigableSet и NavigableMap, содержащие дополнительные методы для поиска и обхода сортированных наборов и карт. (В идеале эти методы должны быть просто включены в интерфейсы SortedSet и SortedMap.) Классы TreeSet и TreeMap реализуют эти интерфейсы.

Теперь перейдем от интерфейсов к реализующим их классам. Мы уже говорили, что интерфейсы коллекций предусматривают относительно немного методов, которые могут быть реализованы тривиально на основе более фундаментальных методов.

Абстрактные классы применяют многие из таких рутинных реализаций:

```
AbstractCollection
AbstractList
AbstractSequentialList
AbstractSet
AbstractQueue
AbstractMap
```

Если вы реализуете свой собственные класс коллекции, то вероятно, захотите расширить один из этих классов так, чтобы можно было предложить свои реализации рутинных операций.

Библиотека Java предлагает следующие конкретные классы коллекций:

```
LinkedList
ArrayList
ArrayDeque
HashSet
TreeSet
PriorityQueue
HashMap
TreeMap
```

На рис. 13.11 показаны отношения между этими классами.

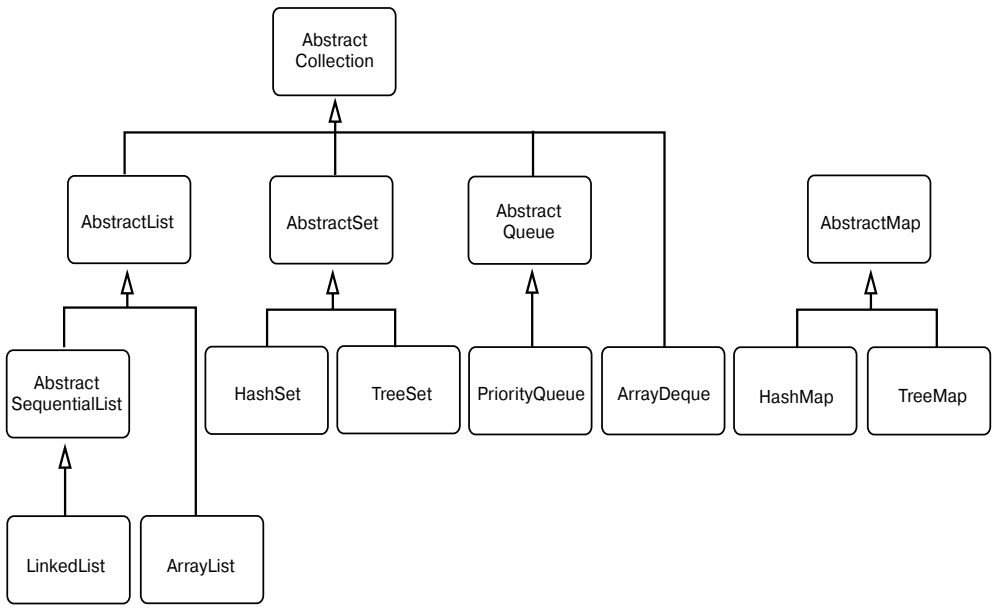


Рис. 13.11. Классы из каркаса коллекций

И, наконец, по-прежнему поддерживаются несколько “унаследованных” контейнерных классов, впервые появившихся в первой версии Java, когда еще не было нынешнего каркаса коллекций:

```
Vector
Stack
Hashtable
Properties
```

Все они были интегрированы в каркас коллекций, показанный на рис. 13.12. Мы обсудим их позже в этой главе.

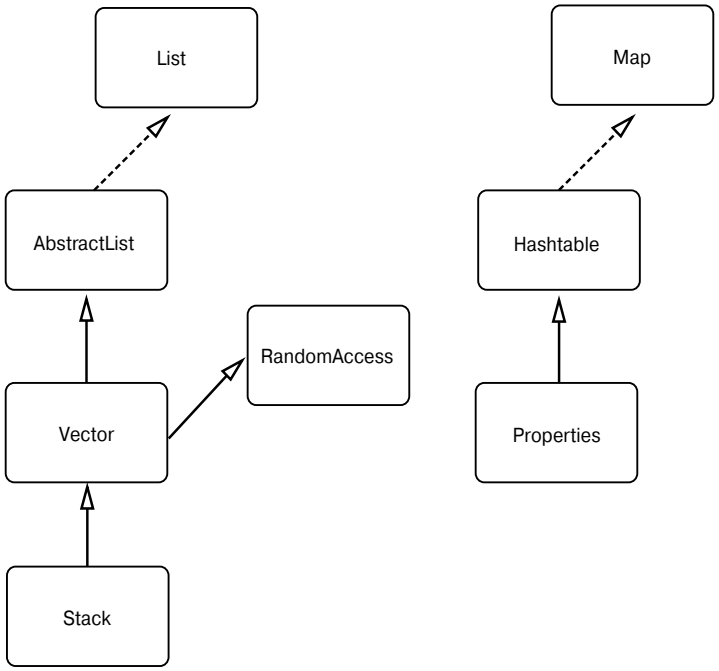


Рис. 13.12. Унаследованные классы в каркасе коллекций

Представления и "оболочки"

Если посмотреть на рис. 13.10 и 13.11, то можно подумать, что это избыточно — иметь так много интерфейсов и абстрактных классов для реализации столь скромного количества конкретных классов коллекций. Однако эти рисунки не отражают всей картины. Используя *представления* (views) вы можете получить другие объекты, реализующие интерфейсы `Collection` или `Map`. Один пример этого вы уже видели — метод `keySet()` в классах карт. На первый взгляд это выглядит так, будто данный метод создает новый набор, заполняет его всеми ключами карты и возвращает его. Однако это не так. Вместо этого метод `keySet` возвращает объект класса, реализующего интерфейс `Set`, чьи методы манипулируют исходной картой. Такая коллекция называется *представлением*.

Техника представлений имеет ряд полезных применений в каркасе коллекций. Мы обсудим эти применения в последующих разделах главы.

Легковесные "оболочки" коллекций

Статический метод `asList()` класса `Arrays` возвращает `List` — оболочку вокруг простого массива Java. Этот метод позволяет вам передать массив методу, принимающему аргумент — список или коллекцию. Например:

```
Card[] cardDeck = new Card[52];
. . .
List<Card> cardList = Arrays.asList(cardDeck);
```

Возвращенный объект — *не* `ArrayList`. Это объект-представление с методами `get()` и `set()`, имеющими доступ к лежащему в основе массиву. Все методы, которые могли бы изменить размер массива (такие как `add()` и `remove()` ассоциированного итератора) генерируют исключение `UnsupportedOperationException`.

Начиная с Java SE 5.0, метод `asList()` объявлен с переменным количеством аргументов. Вместо передачи массива можно передавать индивидуальные элементы. Например:

```
List<String> names = Arrays.asList("Amy", "Bob", "Carl");
```

Вызов метода

```
Collections.nCopies(n, anObject)
```

возвратит неизменяемый объект, реализующий интерфейс `List` и создаст иллюзию наличия `n` элементов, каждый из которых выглядит как `anObject`.

Например, следующий вызов создает `List`, содержащий 100 строк с одинаковым значением `"DEFAULT"`:

```
List<String> settings = Collections.nCopies(100, "DEFAULT");
```

Хранение такого списка обходится очень дешево — объект сохраняется лишь однажды. Такое вот разумное приложение техники представлений.



На заметку! Класс `Collections` содержит множество вспомогательных методов с параметрами или возвращаемыми значениями — коллекциями. Не путайте его с интерфейсом `Collection`.

Вызов метода

```
Collections.singleton(anObject)
```

возвращает объект представления, реализующий интерфейс `Set` (в отличие от `ncopies`, который возвращает `List`). Возвращенный объект реализует неизменяемый одноэлементный набор без накладных расходов структуры данных. Методы `singletonList()` и `singletonMap()` ведут себя аналогично.

Поддиапазоны

Для многих коллекций можно формировать представления — поддиапазоны. Например, предположим, что у вас есть список `staff`, и вы хотите извлечь элементы с 10 по 19. Для этого вы должны применить метод `subList()` для получения представления поддиапазона списка.

```
List group2 = staff.subList(10, 20);
```

Первый индекс включается в результат, второй исключается — точно так же, как действуют параметры операции `substring()` класса `String`.

Вы можете применять любые операции к полученному поддиапазону, и они будут автоматически отражаться на весь список. Например, вы можете удалить весь поддиапазон:

```
group2.clear(); // сокращение штатов
```

Выбранные элементы автоматически удаляются из списка `staff`, а группа `group2` становится пустой.

Чтобы сформировать поддиапазоны сортированных наборов и карт, вы используете порядок сортировки, а не позиции элементов. Интерфейс `SortedSet` объявляет три метода:

```
SortedSet<E> subSet(E from, E to)
SortedSet<E> headSet(E to)
SortedSet<E> tailSet(E from)
```

Все они возвращают подмножества всех элементов, которые больше или равны `from` и меньше, чем `to`. Для сортированных карт аналогичные методы

```
SortedMap<K, V> subMap(K from, K to)
SortedMap<K, V> headMap(K to)
SortedMap<K, V> tailMap(K from)
```

возвращают представления карт, состоящих из всех вхождений, в которых ключи попадают в заданный диапазон.

Интерфейс `NavigableSet`, который появился в Java SE 6, предоставляет больше контроля над этими операциями с поддиапазонами. Вы можете специфицировать — включать ли указанные границы:

```
NavigableSet<E> subSet(E from, boolean fromInclusive, E to, boolean toInclusive)
NavigableSet<E> headSet(E to, boolean toInclusive)
NavigableSet<E> tailSet(E from, boolean fromInclusive)
```

Немодифицируемые представления

Класс `Collections` включает методы, производящие *немодифицируемые* представления коллекций. Эти представления добавляют проверку времени выполнения к существующим коллекциям. Если обнаружена попытка модификации коллекции, генерируется исключение и коллекция остается нетронутой.

Получить немодифицируемые представления можно следующими шестью методами:

```
Collections.unmodifiableCollection
Collections.unmodifiableList
Collections.unmodifiableSet
Collections.unmodifiableSortedSet
Collections.unmodifiableMap
Collections.unmodifiableSortedMap
```

Каждый метод определен для работы с интерфейсом. Например, `Collections.unmodifiableList()` работает с `ArrayList`, `LinkedList` или любым другим классом, реализующим интерфейс `List`.

Например, предположим, что вы хотите предоставить некоторой части вашего кода возможность просматривать, но не изменять содержимое коллекции. Вот как это можно сделать:

```
List<String> staff = new LinkedList<String>();
. . .
lookAt(new Collections.unmodifiableList(staff));
```

Метод `Collections.unmodifiableList()` возвращает объект класса, реализующего интерфейс `List`. Его методы доступа извлекают значения из коллекции `staff`. Конечно, метод `lookAt()` может вызывать все методы интерфейса `List`, а не только методы доступа (accessors).

Но все методы, изменяющие коллекцию (такие как `add()`) переопределены так, что немедленно генерируют исключение `UnsupportedOperationException` вместо передачи вызова лежащей в основе коллекции.

Немодифицируемое представление не делает саму коллекцию немодифицируемой. Вы можете модифицировать коллекцию через ее исходную ссылку (`stuff` в данном случае). И вы можете вызывать модифицирующие методы на элементах коллекции.

Поскольку представления помещают в оболочку *интерфейс*, а не реальный объект коллекции, вам нужен лишь доступ к тем методам, которые определены в интерфейсе. Например, класс `LinkedList` включает удобные методы `addFirst()` и `addLast()`, не являющиеся частью интерфейса `List`. Эти методы не доступны через немодифицируемое представление.



Внимание! Метод `unmodifiableCollection()` (как и методы `synchronizedCollection()` и `checkedCollection()`, о которых мы поговорим ниже) возвращают коллекцию, чей метод `equals()` *не* вызывает метода `equals()` лежащей в основе коллекции. Вместо этого он наследует метод `equals()` класса `Object`, который просто проверяет идентичность объектов. Если вы превратите набор или список просто в коллекцию, то не сможете проверить эквивалентность содержимого. Представление действует подобным образом, потому что проверка эквивалентности не определена четко на этом уровне иерархии. Аналогичным образом представления трактуют метод `hashCode()`.

Однако классы объектов, возвращенных `unmodifiableSet` и `unmodifiableList`, используют методы `equals()` и `hashCode()` лежащей в основе коллекции.

Синхронизированные представления

Если вы обращаетесь к коллекции из нескольких потоков, нужно исключить ее непреднамеренное повреждение. Например, было бы губительно, если один поток попытается добавить элемент в хеш-таблицу, в то время как другой выполняет повторное хеширование ее элементов.

Вместо реализации безопасных к потокам классов коллекций разработчики библиотеки использовали механизм представлений, чтобы сделать безопасными к потокам обычные коллекции. Например, статический метод `synchronizedMap()` в классе `Collections` может превратить любую карту в `Map` с синхронизированными методами доступа:

```
Map<String, Employee> map =
   Collections.synchronizedMap(new HashMap<String, Employee>());
```

После этого вы можете обращаться к объекту `map` из нескольких потоков. Такие методы, как `get()` и `put()`, сериализованы — каждый вызов метода должен полностью завершаться до того, как другой поток сможет его вызвать. Проблемы синхронизированного доступа к структурам данных мы обсудим в главе 14.

Контролируемые представления

В Java SE 5.0 был добавлен набор "контролируемых" (checked) представлений, задуманных как средство поддержки отладки проблем, сопровождающий обобщенные типы. Как объяснялось в главе 12, существует возможность "протащить" в обобщенную коллекцию элементы неверного типа. Например:

```
ArrayList<String> strings = new ArrayList<String>();
ArrayList rawList = strings;  // выдается предупреждение, а не ошибка,
                              // для совместимости с унаследованным кодом
rawList.add(new Date());      // теперь строка содержит объект Date!
```

Ошибочная команда `add()` не будет обнаружена во время выполнения. Вместо этого произойдет исключение приведения класса, когда другая часть кода вызовет `get()` и попытается привести результат к `String`. Контролируемое представление может обнаружить эту проблему. Определим безопасный список:

```
List<String> safeStrings = Collections.checkedList(strings, String.class);
```

Метод представления `add()` проверит принадлежность вставляемого объекта к заданному классу и немедленно сгенерирует исключение `ClassCastException`, если обнаружит несоответствие. Преимущество этого в том, что ошибка случится в правильном месте:

```
ArrayList rawList = safeStrings;
rawList.add(new Date()); // Контролируемый список сгенерирует
ClassCastException
```



На заметку! Контролируемые представления ограничены проверками времени выполнения, которые может выполнить виртуальная машина. Например, если у вас есть `ArrayList<Pair<String>>`, вы не сможете защитить его от вставки `Pair<Date>`, поскольку виртуальная машина знает о единственном "сыром" классе `Pair`.

Примечание об необязательных операциях

Обычно представление подчиняется некоторому ограничению: оно может быть доступно только для чтения, может не допускать изменений размера или поддерживать удаления, но запрещать вставку — как в случае представления ключей карты. Ограниченное представление генерирует исключение `UnsupportedOperationException`, если вы пытаетесь выполнить неразрешенную операцию.

В документации API по интерфейсам коллекций и итераторов многие методы описаны, как "необязательные операции". Это кажется противоречащим самому понятию интерфейса. В конце концов, разве не в том назначение интерфейсов, чтобы объявлять методы, которые класс *обязан* реализовать? На самом деле такая организация неудовлетворительна с теоретической точки зрения. Может быть, лучше было бы разработать отдельные интерфейсы для представлений "только для чтения" и представлений, которые не могут изменять размер коллекции. Однако это привело бы к значительному размножению интерфейсов, что проектировщики библиотеки сочли неприемлемым.

Должны ли вы расширять технику "необязательных" методов в своем собственном дизайне? Мы полагаем, что нет. Даже несмотря на то, что коллекции используются часто, стиль кодирования для их реализации не типичен для других предметных областей. Проектировщики библиотеки классов коллекций должны были удовлетворить жесткий набор противоречивых требований. Пользователи хотят, чтобы библиотеку было легко изучить, удобно использовать, чтобы она была полностью обобщенной, защищенной от неумелого обращения, и в то же время столь же эффективной, как и разработанные вручную алгоритмы. Почти невозможно достичь всех этих целей одновременно, или даже приблизиться к этому. Но в вашей собственной практике программирования вы редко столкнетесь с таким суровым набором ограничений. Вы должны быть готовы находить решения, которые не полагаются на крайнюю меру — применение "необязательных" интерфейсных операций.

API java.util.Collection 1.2

- `static <E> Collection unmodifiableCollection(Collection<E> c)`
- `static <E> List unmodifiableList(List<E> c)`
- `static <E> Set unmodifiableSet(Set<E> c)`
- `static <E> SortedSet unmodifiableSortedSet(SortedSet<E> c)`
- `static <K, V> Map unmodifiableMap(Map<K, V> c)`
- `static <K, V> SortedMap unmodifiableSortedMap(SortedMap<K, V> c)`

Конструируют представления коллекций, чьи изменяющие методы генерируют исключение `UnsupportedOperationException`.

- `static <E> Collection<E> synchronizedCollection(Collection<E> c)`
- `static <E> List synchronizedList(List<E> c)`
- `static <E> Set synchronizedSet(Set<E> c)`
- `static <E> SortedSet synchronizedSortedSet(SortedSet<E> c)`
- `static <K, V> Map<K, V> synchronizedMap(Map<K, V> c)`
- `static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> c)`

Конструируют представления коллекций с синхронизированными методами.

- `static <E> Collection checkedCollection(Collection<E> c, Class<E> elementType)`
- `static <E> List checkedList(List<E> c, Class<E> elementType)`
- `static <E> Set checkedSet(Set<E> c, Class<E> elementType)`
- `static <E> SortedSet checkedSortedSet(SortedSet<E> c, Class<E> elementType)`
- `static <K, V> Map checkedMap(Map<K, V> c, Class<K> keyType, Class<V> valueType)`
- `static <K, V> SortedMap checkedSortedMap(SortedMap<K, V> c, Class<K> keyType, Class<V> valueType)`

Конструируют представления, чьи методы генерируют исключение `ClassCastException` при попытке вставить элемент неверного типа.

- `static <E> List<E> nCopies(int n, E value)`
- `static <E> Set<E> singleton(E value)`

Конструируют представление объекта, как либо немодифицируемый список с `n` идентичных элементов, либо набор из одного элемента.

API java.util.Arrays 1.2

- `static <E> List<E> asList(E... array)`

Возвращает представление-список элементов массива, который является модифицируемым, но не изменяемым в размерах.

API java.util.List<E> 1.2

- `List<E> subList(int firstIncluded, int firstExcluded)`

Возвращает представление-список элементов внутри диапазона позиций.

API java.util.SortedSet<E> 1.2

- `SortedSet<E> subSet(E firstIncluded, E firstExcluded)`
- `SortedSet<E> headSet(E firstExcluded)`
- `SortedSet<E> tailSet(E firstIncluded)`

Возвращает представление элементов из диапазона.

API java.util.NavigableSet<E> 6

- `NavigableSet<E> subSet(E from, boolean fromIncluded, E to, boolean toIncluded)`
- `NavigableSet<E> headSet(E to, boolean toIncluded)`
- `NavigableSet<E> tailSet(E from, boolean fromIncluded)`

Возвращает представление элементов внутри диапазона. Флаги `boolean` определяют включение граничных значений в представление.

API java.util.SortedMap<K, V> 1.2

- `SortedMap<K, V> subMap(K firstIncluded, K firstExcluded)`
- `SortedMap<K, V> headMap(K firstExcluded)`
- `SortedMap<K, V> tailMap(K firstIncluded)`

Возвращает представление элементов, чьи ключи находятся внутри диапазона.

API java.util.NavigableMap<K, V> 6

- `NavigableMap<K, V> subMap(K from, boolean fromIncluded, K to, boolean toIncluded)`
- `NavigableMap<K, V> headMap(K from, boolean fromIncluded)`
- `NavigableMap<K, V> tailMap(K to, boolean toIncluded)`

Возвращает представление вхождений, чьи ключи находятся внутри диапазона. Флаги `boolean` определяют включение граничных значений в представление.

Пакетные операции

До сих пор большинство наших примеров использовали итератор для обхода элементов коллекции, один за другим. Однако часто вы можете избежать итераций, применяя *пакетные операции* (bulk operations).

Представим, что вы хотите найти *пересечение* двух наборов, т.е. элементы, общие для обоих наборов. Для начала создадим новый набор для помещения в него результата:

```
Set<String> result = new HashSet<String>(a);
```

Здесь используется тот факт, что каждая коллекция имеет конструктор, чьим параметром является другая коллекция, содержащая неинициализированные значения.

Теперь применим метод `retainAll()`:

```
result.retainAll(b);
```

Он оставляет в коллекции только те элементы, которые есть в `b`. Вы получаете пересечение двух множеств без всякого программирования.

Эту идею можно развить и дальше, применив пакетную операцию к *представлению*. Например, предположим, что у вас есть карта, связывающая идентификаторы сотрудников с объектами, описывающими сотрудников, и у вас есть набор идентификаторов сотрудников, подлежащих увольнению.

```
Map<String, Employee> staffMap = . . .;
Set<String> terminatedIDs = . . .;
```

Поскольку набор ключей — это представление карты, ключи и имена ассоциированных сотрудников автоматически удаляются из карты.

Используя представление поддиапазона, вы можете ограничить пакетные операции подсписками и поднаборами. Например, предположим, что вы хотите добавить первые 10 элементов списка в другой контейнер.

Сформируйте подсписок из первых 10 элементов:

```
relocated.addAll(staff.subList(0, 10));
```

Поддиапазон также может быть предметом изменяющей операции:

```
staff.subList(0, 10).clear();
```

Преобразования между коллекциями и массивами

Поскольку значительная часть API платформы Java была спроектирована до появления каркаса коллекций, рано или поздно вам понадобится транслировать традиционные массивы в более современные коллекции.

Если у вас есть массив, вам нужно превратить его в коллекцию. Оболочка `Arrays.asList` служит этой цели. Например:

```
String[] values = . . .;
HashSet<String> staff = new HashSet<String>(Arrays.asList(values));
```

Получение массива из коллекции немного сложнее. Конечно, вы можете применить метод `toArray()`:

```
Object[] values = staff.toArray();
```

Но результатом будет массив *объектов*. Даже если вы знаете, что ваша коллекция содержит объекты определенного типа, вы не можете применить приведение:

```
String[] values = (String[]) staff.toArray(); // Ошибка!
```

Массив, возвращенный методом `toArray()`, был создан как массив `Object[]`, и вы не можете изменить этот тип. Вместо этого вы используете вариант метода `toArray()`. Передайте ему массив длиной 0 требуемого типа. Возвращенный в результате массив будет *того же типа*:

```
String[] values = staff.toArray(new String[0]);
```

Если хотите, вы можете сразу сконструировать массив нужного размера:

```
staff.toArray(new String[staff.size()]);
```

В этом случае новый массив не создается.



На заметку! Вы можете спросить, почему бы просто не передать объект `Class` (такой как `String.class`) методу `toArray`? Однако этот метод несет на себе “двойную нагрузку”, заполняя существующий массив (если он достаточной длины) и создавая новый.

Алгоритмы

Обобщенные интерфейсы коллекций имеют огромное преимущество – вам нужно реализовать ваш алгоритм лишь однажды. Например, рассмотрим простой алгоритм вычисления максимального элемента в коллекции. Традиционно программисты реализуют его в цикле. Вот как можно найти самый большой элемент массива:

```
if (a.length == 0) throw new NoSuchElementException();
T largest = a[0];
for (int i = 1; i < a.length; i++)
  if (largest.compareTo(a[i]) < 0)
    largest = a[i];
```

Конечно, чтобы найти максимум в массиве-списке, придется написать код несколько по-другому.

```
if (v.size() == 0) throw new NoSuchElementException();
T largest = v.get(0);
for (int i = 1; i < v.size(); i++)
  if (largest.compareTo(v.get(i)) < 0)
    largest = v.get(i);
```

А как насчет связного списка? Для связного списка не существует эффективного произвольного доступа, но можно использовать итератор.

```
if (l.isEmpty()) throw new NoSuchElementException();
Iterator<T> iter = l.iterator();
T largest = iter.next();
while (iter.hasNext())
{
  T next = iter.next();
  if (largest.compareTo(next) < 0)
     largest = next;
}
```

Все эти циклы писать достаточно утомительно, и они чреваты ошибками. Нет ли ошибки выхода за диапазон? Правильно ли будут работать эти циклы с пустыми коллекциями? Как насчет контейнеров с одним элементом? Вы не хотите тестировать и отлаживать этот код каждый раз, но также вы не хотите реализовывать уйму методов вроде таких:

```
static <T extends Comparable> T max(T[] a)
static <T extends Comparable> T max(ArrayList<T> v)
static <T extends Comparable> T max(LinkedList<T> l)
```

И здесь на помощь приходят интерфейсы коллекций. Подумайте о *минимальном* интерфейсе коллекции, который понадобится для эффективной реализации алгоритма. Произвольный доступ с помощью `get()` и `set()` стоит выше в "цепочке потребления", чем простая итерация. Как вы видели в примере вычисления максимального элемента в связном списке, произвольный доступ не обязателен для этой задачи. Вычисление максимума может быть выполнено с помощью итерации по элементам. Поэтому вы можете реализовать метод `max()` так, чтобы он принимал *любой* объект, реализующий интерфейс `Collection`.

```
public static <T extends Comparable> T max(Collection<T> c)
{
  if (c.isEmpty()) throw new NoSuchElementException();
  Iterator<T> iter = c.iterator();
```

```
    T largest = iter.next();
    while (iter.hasNext())
    {
      T next = iter.next();
      if (largest.compareTo(next) < 0)
        largest = next;
    }
    return largest;
  }
```

Теперь с единственным методом вы можете вычислить максимум в связном списке, в списке-массиве или простом массиве.

Это весьма мощная концепция. Фактически стандартная библиотека C++ имеет десятки полезных алгоритмов, каждый из которых оперирует обобщенной коллекцией. Библиотека Java не настолько богата, но в ней есть основы: сортировка, двоичный поиск, а также некоторые служебные алгоритмы.

Сортировка и перетасовка

Ветераны программирования иногда вспоминают о том, как им приходилось использовать перфокарты и как вручную программировать алгоритмы сортировки. В наши дни, конечно, алгоритмы сортировки являются частью стандартной библиотеки в большинстве языков программирования, и язык программирования Java — не исключение.

Метод `sort()` класса `Collections` сортирует коллекцию, реализующую интерфейс `List`.

```
List<String> staff = new LinkedList<String>();
// наполнение коллекции . . .;
Collections.sort(staff);
```

Этот метод предполагает, что элементы списка реализуют интерфейс `Comparable`. Если вы хотите сортировать список каким-то другим способом, вы можете передать объект `Comparator` в качестве второго параметра конструктора. (Мы обсуждали компараторы выше в разделе “Сравнение объектов”.) Вот как можно отсортировать список элементов:

```
Comparator<Item> itemComparator = new
Comparator<Item>()
{
   public int compare(Item a, Item b)
   {
     return a.partNumber - b.partNumber;
   }
});
Collections.sort(items, itemComparator);
```

Если вы захотите отсортировать список в *порядке убывания*, используйте удобный статический метод `Collections.reverseSort()`. Он вернет компаратор, возвращающий `b.compareTo(a)`. Например, показанный ниже вызов сортирует элементы списка `staff` в обратном порядке — согласно упорядочиванию, заданному методом `compareTo()` типа элементов.

```
Collections.sort(staff, Collections.reverseOrder())
```

Аналогично, следующий вызов обращает порядок, задаваемый `itemComparator`.

```
Collections.sort(items, Collections.reverseOrder(itemComparator))
```

Вас может заинтересовать, каким образом метод `sort()` сортирует список. Обычно, если посмотреть на алгоритм сортировки в книге по алгоритмам, он представляется для массивов с использованием произвольного доступа к элементам. Однако произвольный доступ к элементам списка неэффективен. Вы можете эффективно сортировать списки, используя форму сортировки слиянием (см., например, книгу *Algorithms in C++* Роберта Седжвика (Robert Sedgewick) (Addison-Wesley, 1998)). Однако реализация на языке Java не делает этого. Она просто сбрасывает все элементы в массив, сортирует этот массив, применяя другой вариант сортировки слиянием, и затем копирует отсортированную последовательность обратно в список.

Алгоритм сортировки слиянием, используемый в библиотеке коллекций, немного медленнее *быстрой сортировки* (quick sort) — традиционного выбора для алгоритмов сортировки общего назначения. Однако он имеет одно важное преимущество: он *стабилен*, т.е. не меняет местами эквивалентные элементы. Зачем беспокоиться о порядке эквивалентных элементов? Рассмотрим распространенный сценарий. Предположим, что у вас есть список сотрудников, который вы уже отсортировали по именам. Теперь вы сортируете по зарплате. Что случится с сотрудниками с одинаковой зарплатой? При стабильной сортировке порядок по именам сохраняется. Другими словами, в результате получим список, отсортированный сначала по зарплате, потом по имени.

Поскольку коллекциям не обязательно реализовывать все "необязательные" методы, все методы, принимающие параметры-коллекции, должны указывать, когда передавать коллекцию алгоритму безопасно. Например, очевидно, что вы не захотите передать список `unmodifiableList` алгоритму сортировки. Какого рода списки вы *можете* передавать? Согласно документации, список должен быть модифицируемым, но не должен быть изменяемым в размере.

Термины определяются следующим образом.

- Список является *модифицируемым*, если он поддерживает метод `set()`.
- Список является *изменяемым в размере*, если он поддерживает методы `add()` и `remove()`.

Класс `Collections` имеет алгоритм `shuffle`, который выполняет противоположную сортировке задачу: случайным образом изменяет порядок элементов в списке. Например:

```
ArrayList<Card> cards = . . .;
Collections.shuffle(cards);
```

Если вы примените список, который не реализует интерфейс `RandomAccess`, то метод `shuffle` скопирует все элементы в массив, перетасует этот массив, после чего скопирует перетасованные элементы обратно в список.

Программа в листинге 13.6 заполняет массив-список 49 объектами `Integer`, содержащими числа от 1 до 49. Затем она случайно тасует их в списке и выбирает первые 6 значений из перетасованного списка. И, наконец, она сортирует выбранные значения и печатает их.

Листинг 13.6. Содержимое файла `ShuffleTest.java`

```
1.  import java.util.*;
2.
3.  /**
4.   * Эта программа демонстрирует алгоритмы случайного тасования и сортировки.
5.   * @version 1.10 2004-08-02
6.   * @author Cay Horstmann
7.   */
```

```
8.  public class ShuffleTest
9.  {
10.   public static void main(String[] args)
11.   {
12.      List<Integer> numbers = new ArrayList<Integer>();
13.      for (int i = 1; i <= 49; i++)
14.         numbers.add(i);
15.      Collections.shuffle(numbers);
16.      List<Integer> winningCombination = numbers.subList(0, 6);
17.      Collections.sort(winningCombination);
18.      System.out.println(winningCombination);
19.   }
20. }
```

API java.util.Collections 1.2

- `static <T extends Comparable<? super T>> void sort(List<T> elements)`
- `static <T> void sort(List<T> elements, Comparator<? super T> c)`

  Сортируют элементы в списке, используя алгоритм стабильной сортировки. Гарантируется исполнение алгоритма за время $O(n \log n)$, где n — длина списка.

- `static void shuffle(List<?> elements)`
- `static void shuffle(List<?> elements, Random r)`

  Случайно тасует элементы в списке. Этот алгоритм исполняется за время $O(n\, a(n))$, где n — длина списка, а $a(n)$ — среднее время доступа к элементу.

- `static <T> Comparator<T> reverseOrder()`

  Возвращает компаратор, сортирующий элементы в порядке, обратном задаваемому методом `compareTo()` интерфейса `Comparable`.

- `static <T> Comparator<T> reverseOrder(Comparator<T> comp)`

  Возвращает компаратор, сортирующий элементы в порядке, обратном заданному `comp`.

Бинарный поиск

Чтобы найти объект в массиве, обычно вы перебираете все элементы, пока не найдете соответствующий критерию поиска. Однако если массив отсортирован, то вы можете сразу перейти к среднему элементу и сравнить, больше ли он искомого элемента. Если да, значит нужно продолжить поиск в первой половине массива; в противном случае — во второй. Это сразу "урезает" проблему вдвое. Дальше вы продолжаете в том же духе. Например, если массив содержит 1024 элемента, то вы найдете соответствие (или убедитесь, что его нет) после 10 шагов, в то время как линейный поиск в среднем потребует 512 шагов, если элемент присутствует, и 1024 — чтобы подтвердить, что его нет.

Метод `binarySearch()` класса `Collections` реализует этот алгоритм. Обратите внимание, что коллекция уже должна быть отсортирована, иначе алгоритм вернет неверный ответ. Чтобы найти элемент, передайте коллекцию (которая должна реализовывать интерфейс `List`) и элемент, который нужно найти. Если коллекция не сортирована методом `compareTo` интерфейса `Comparable`, то вы должны применить объект-компаратор.

```
i = Collections.binarySearch(c, element);
i = Collections.binarySearch(c, element, comparator);
```

Возвращенное методом `binarySearch()` значение, большее и равное 0, означает индекс найденного объекта. То есть `c.get(i)` эквивалентно `element` по порядку сравнения. Если возвращено отрицательное значение, значит, элемент не найден. Однако вы можете использовать возвращенное значение для вычисления местоположения, куда вы *должны* вставить `element`, чтобы сохранить порядок сортировки. Место вставки вычисляется следующим образом:

```
insertionPoint = -i - 1;
```

Здесь не берется просто `-i`, потому что значение 0 было бы неоднозначным. Другими словами, приведенная ниже операция добавляет элемент в правильное место.

```
if (i < 0)
 c.add(-i - 1, element);
```

Для эффективной работы бинарный поиск нуждается в произвольном доступе. Если вы должны выполнять итерацию по одному элементу на половине связного списка, чтобы найти средний элемент, то теряются все преимущество бинарного поиска. Поэтому алгоритм `binarySearch()` превращается в линейный поиск, если вы передаете ему связный список.



На заметку! В Java SE 1.3 не было отдельного интерфейса для упорядоченной коллекции с эффективным произвольным доступом, и метод `binarySearch()` использовал очень грубую проверку — он проверял, расширяет ли параметр `list` класс `AbstractSequentialList`. Это было исправлено в Java SE 1.4. Теперь метод `binarySearch()` проверяет, реализует ли параметр `list` интерфейс `RandomAccess`. Если это так, то метод выполняет бинарный поиск. В противном случае применяется линейный поиск.



java.util.Collections 1.2

- `static <T extends Comparable<? super T>> int binarySearch(List<T> elements, T key)`
- `static <T> int binarySearch(List<T> elements, T key, Comparator<? super T> c)`

  Ищут `key` в сортированном списке, используя линейный поиск, если `element` расширяет класс `AbstractSequentialList`, и бинарный поиск во всех других случаях. Гарантируется выполнение метода за время $O(a(n) \log n)$, где n — длина списка, а $a(n)$ — среднее время доступа к элементу. Метод возвращает либо индекс `key` в списке, либо отрицательное значение `i`, если `key` не найден в списке. В этом случае `key` должен быть вставлен в позицию с индексом `-i-1`, чтобы список остался отсортированным.

Простые алгоритмы

Класс `Collections` включает несколько простых, но полезных алгоритмов. Среди них — пример из начала этого раздела, находящий максимальное значение в коллекции. Другие включают копирование элементов из одного списка в другой, заполнение контейнера константным значением и обращение списка. Зачем включать такие простые алгоритмы в стандартную библиотеку? Несомненно, большинство программистов могут легко реализовать их с помощью простых циклов. Нам нравятся алгорит-

мы, поскольку они облегчают жизнь программистам, *читающим* код. Когда вы читаете цикл, реализованный кем-то другим, вам приходится расшифровывать намерения этого программиста. Когда вы видите вызов такого метода, как `Collections.max()`, вы знаете точно, что этот код делает. Следующие врезки, посвященные API, описывают простые алгоритмы из класса `Collections`.

API java.util.Collections 1.2

- `static <T extends Comparable<? super T>> T min(Collection<T> elements)`
- `static <T extends Comparable<? super T>> T max(Collection<T> elements)`
- `static <T> min(Collection<T> elements, Comparator<? super T> c)`
- `static <T> max(Collection<T> elements, Comparator<? super T> c)`

  Возвращают минимальный или максимальный элемент коллекции (границы параметров упрощены для ясности).

- `static <T> void copy(List<? super T> to, List<T> from)`

  Копирует все элементы из исходного списка в те же позиции целевого списка. Целевой список должен быть не короче исходного.

- `static <T> void fill(List<? super T> l, T value)`

  Устанавливает во все позиции списка одно и то же значение.

- `static <T> boolean addAll(Collection<? super T> c, T... values)` **5.0**

  Добавляет все значения в заданную коллекцию и возвращает `true`, если в результате этого коллекция изменилась.

- `static <T> boolean replaceAll(List<T> l, T oldValue, T newValue)` **1.4**

  Заменяет все элементы, эквивалентные `oldValue`, на `newValue`.

- `static int indexOfSubList(List<?> l, List<?> s)` **1.4**
- `static int lastIndexOfSubList(List<?> l, List<?> s)` **1.4**

  Возвращает индекс первого и последнего подсписков `l`, эквивалентных `s`, или `-1`, если никакой подсписок `l` не эквивалентен `s`. Например, если `l` содержит `[s, t, a, r]`, а `s` содержит `[t, a, r]`, значит, оба метода вернут `1`.

- `static void swap(List<?> l, int i, int j)` **1.4**

  Меняет местами элементы в указанных позициях.

- `static void reverse(List<?> l)`

  Меняет порядок элементов в списке. Например, обращение списка `[t, a, r]` порождает список `[r, a, t]`. Этот метод выполняется за время $O(n)$, где n — длина списка.

- `static void rotate(List<?> l, int d)` **1.4**

  Меняет порядок элементов в списке, перемещая вхождение с индексом `i` в позицию `(i+d)%l.size()`. Например, вращение списка `[t, a, r]` на 2 порождает список `[a, r, t]`. Этот метод выполняется за время $O(n)$, где n — длина списка.

- `static int frequency(Collection<?> c, Object o)` **5.0**

  Возвращает количество элементов в `c`, эквивалентных объекту `o`.

- `boolean disjoint(Collection<?> c1, Collection<?> c2)` **5.0**

  Возвращает `true`, если коллекции не имеют общих элементов.

Написание собственных алгоритмов

Если вы пишете свой собственный алгоритм (фактически — любой метод, принимающий в параметре коллекцию), то где только возможно должны работать с *интерфейсами*, а не конкретными реализациями. Например, представим, что вы хотите заполнить `JMenu` набором пунктов меню. Традиционно такой метод может быть реализован примерно так:

```
void fillMenu(JMenu menu, ArrayList<JMenuItem> items)
{
  for (JMenuItem item : items)
    menu.addItem(item);
}
```

Однако так вы ограничиваете код, вызывающий ваш метод — он должен передавать пункты в `ArrayList`. Если это окажется другой контейнер, его придется сначала перепаковать. Гораздо лучше принимать более обобщенную коллекцию.

Вы можете спросить себя: какой интерфейс коллекции является наиболее общим, который мог бы выполнить эту работу? В данном случае нужно просто перебрать все элементы, что умеет базовый интерфейс `Collection`. И вот как можно переписать этот метод `fillMenu()`, чтобы он принимал коллекцию любого вида:

```
void fillMenu(JMenu menu, Collection<JMenuItem> items)
{
  for (JMenuItem item : items)
    menu.addItem(item);
}
```

Теперь любой может вызвать этот метод, передав ему `ArrayList`, `LinkedList` или даже массив, помещенный в оболочку `Arrays.asList`.



На заметку! Если это такая хорошая идея — использовать интерфейс коллекции в качестве параметра метода, почему же библиотека Java не так часто следует этому правилу? Например, класс `JComboBox` имеет два конструктора:

```
JComboBox(Object[] items)
JComboBox(Vector<?> items)
```

Причина проста: библиотека Swing была создана до библиотеки коллекций.

Если вы пишете метод, *возвращающий* коллекцию, то можете также вернуть интерфейс вместо класса, поскольку позднее можете передумать и реализовать его с другой коллекцией.

Например, давайте напишем метод `getAllItems()`, возвращающий все пункты меню.

```
List<MenuItem> getAllItems(JMenu menu)
{
  ArrayList<MenuItem> items = new ArrayList<MenuItem>()
  for (int i = 0; i < menu.getItemCount(); i++)
    items.add(menu.getItem(i));
  return items;
}
```

Позднее, если вы решите, что не хотите *копировать* элементы, то можно будет вернуть их представление. Это делается возвратом анонимного подкласса `AbstractList`.

```
List<MenuItem> getAllItems(final JMenu menu)
{
  return new
   AbstractList<MenuItem>()
   {
     public MenuItem get(int i)
     {
          return item.getItem(i);
     }
     public int size()
     {
          return item.getItemCount();
     }
   };
}
```

Конечно, это — совершенная техника. Если вы используете ее, не забудьте точно документировать, какие "необязательные" операции поддерживаются. В данном случае следует предупредить пользователя этого метода, что возвращенная коллекция — немодифицируемый список.

Унаследованные коллекции

В этом разделе мы поговорим о классах коллекций, которые существуют в языке программирования Java с самого начала: о классе `Hashtable` и его удобном подклассе `Properties`, о подклассе `Stack` класса `Vector`, а также о классе `BitSet`.

Класс `Hashtable`

Класс `Hashtable` служит той же цели, что и `HashMap` — по сути, с тем же интерфейсом. Как и методы класса `Vector`, методы `Hashtable` синхронизированы. Если вам не нужна синхронизация или совместимость с унаследованным кодом, вы должны использовать вместо него `HashMap`.



На заметку! Имя класса записывается как `Hashtable`, с прописной "t". В среде Windows вы получите странное сообщение об ошибке, если используете `HashTable`, потому что файловая система Windows не является чувствительной к регистру символов, в то время как компилятор Java является.

Интерфейс `Enumeration`

Унаследованные коллекции используют интерфейс `Enumeration` для обхода последовательностей элементов. Интерфейс `Enumeration` имеет два метода — `hasMoreElements()` и `nextElement()`. Это полные аналоги методов `hasNext()` и `next()` интерфейса `Iterator`.

Например, метод `elements()` класса `Hashtable` порождает объект для перечисления значений из таблицы:

```
Enumeration<Employee> e = staff.elements();
while (e.hasMoreElements())
{
  Employee e = e.nextElement();
  . . .
}
```

Иногда вам придется столкнуться с унаследованным методом, который ожидает параметра-перечисления. Статический метод `Collections.enumeration` порождает объект перечисления всех элементов коллекции. Например:

```
List<InputStream> streams = . . .;
SequenceInputStream in = new SequenceInputStream(Collections.enumeration(streams));
// конструктор SequenceInputStream ожидает перечисление
```



На заметку! В C++ довольно часто используются итераторы в качестве параметров. К счастью, при программировании на платформе Java лишь очень немногие программисты используют эту идиому. Разумнее передавать коллекцию, чем итератор. Объект коллекции удобнее. Код-приемник может всегда получить итератор из коллекции, когда ему это нужно, к тому же он получает в свое распоряжение все методы коллекции. Однако в унаследованном коде вы встретите перечисления, поскольку раньше они были единственным доступным механизмом обобщенных коллекций до появления каркаса коллекций в Java SE 1.2.



java.util.Enumeration<E> 1.0

- `boolean hasMoreElements()`

  Возвращает `true`, если еще остались элементы для просмотра.

- `E nextElement()`

  Возвращает следующий элемент для просмотра. Не вызывайте этот метод, если `hasMoreElements()` вернул `false`.



java.util.Hashtable<K, V> 1.0

- `Enumeration<K> keys()`

  Возвращает объект перечисления, который проходит по всем ключам хеш-таблицы.

- `Enumeration<V> elements()`

  Возвращает объект перечисления, проходящий по всем значениям хеш-таблицы.



java.util.Vector<E> 1.0

- `Enumeration<E> elements()`

  Возвращает объект перечисления, который проходит по всем элементам вектора.

Карты свойств

Карта свойств (property map) — это структура карты очень специального типа. Она обладает следующими характеристиками:

1. Ключи и значения являются строками.
2. Таблица может быть сохранена в файл и загружена из файла.
3. Используется вторичная таблица для установок по умолчанию.

Класс платформы Java, реализующий карту свойств, называется `Properties`.

Карты свойств широко используются для спецификации конфигурационных настроек для программ — см. главу 10.

API java.util.Properties 1.0

- `Properties()`

  Создает пустую карту свойств.

- `Properties(Properties defaults)`

  Создает пустую карту свойств с набором установок по умолчанию.

- `String getProperty(String key)`

  Получает ассоциацию свойства; возвращает строку, ассоциированную с ключом, или же строку, ассоциированную с ключом, но из таблицы установок по умолчанию, если она не представлена в карте.

- `String getProperty(String key, String defaultValue)`

  Получает свойство со значением по умолчанию, если ключ не найден; возвращает строку, ассоциированную с ключом, либо строку по умолчанию, если ключ не представлен в карте.

- `void load(InputStream in)`

  Загружает карту свойств из `InputStream`.

- `void store(OutputStream out, String commentString)`

  Сохраняет карту свойств в `OutputStream`.

Стеки

Начиная с версии 1.0, стандартная библиотека включает класс `Stack` со знакомыми методами `push()` и `pop()`. Однако класс `Stack` расширяет класс `Vector`, что неудовлетворительно с теоретической точки зрения — вы можете применить к нему операции, не характерные для стека, такие как `insert()` и `remove()`, чтобы вставлять или удалять значения откуда угодно, а не только с вершины стека.

API java.util.Stack<E> 1.0

- `E push(E item)`

  Заталкивает элемент в стек и возвращает `item`.

- `E pop()`

  Выталкивает и возвращает элемент из вершины стека. Не вызывайте этот метод на пустом стеке.

- `E peek()`

  Возвращает элемент, находящийся на вершине стека, не выталкивая его. Не вызывайте этот метод на пустом стеке.

Битовые наборы

Класс `BitSet` платформы Java хранит последовательности бит. (Это не *множество* в математическом смысле — битовый *вектор*, или битовый *массив* — возможно, более подходящие термины.) Используйте битовый набор, когда вам нужно эффективно хранить последовательность бит (например, флагов). Поскольку битовый набор помещает биты в байты, использование их намного более эффективно, чем массив-список `ArrayList` объектов `Boolean`.

Класс BitSet предоставляет удобный интерфейс для чтения, установки или переустановки индивидуальных бит. Применение этого интерфейса позволяет избежать маскирования или других операций с битами, которые были бы необходимы в случае хранения бит в переменных типа long.

Например, для BitSet по имени bucketOfBits следующий вызов:

```
bucketOfBits.get(i)
```

возвращает true, если i-й бит установлен, и false — в противном случае. Аналогично, вызов

```
bucketOfBits.set(i)
```

включает i-й бит. И, наконец, вызов

```
bucketOfBits.clear(i)
```

выключает i-й бит.



На заметку! Шаблон bitset в C++ имеет ту же функциональность, что и BitSet платформы Java.



java.util.BitSet 1.0

- BitSet(int initialCapacity)

  Конструирует битовый набор.

- int length()

  Возвращает "логическую длину" битового набора: 1 + индекс самого старшего установленного бита.

- boolean get(int bit)

  Получает бит.

- void set(int bit)

  Устанавливает бит.

- void clear(int bit)

  Очищает бит.

- void and(BitSet set)

  Выполняет операцию логического "И" над данным битовым набором и другим.

- void or(BitSet set)

  Выполняет операцию логического "ИЛИ" над данным битовым набором и другим.

- void xor(BitSet set)

  Выполняет операцию логического исключающего "ИЛИ" над данным битовым набором и другим.

- void andNot(BitSet set)

  Очищает все биты данного битового набора, которые установлены в другом битовом наборе.

Реализация “решета Эратосфена”

В качестве примера использования битовых наборов мы продемонстрируем реализацию алгоритма “решето Эратосфена”, служащего для нахождения простых чисел. (Простое число — это число вроде 2, 3 или 5, которое делится только на себя и на 1, а решето Эратосфена было одним из первых методов, изобретенных для перечисления этих фундаментальных строительных блоков.) Это не самый лучший алгоритм для нахождения простых чисел, но по некоторым причинам он получил популярность в качестве теста производительности компилятора. (Хотя это и не лучший тест, поскольку он тестирует в основном битовые операции.)

Отдадим дань традиции, и включим его реализацию. Программа подсчитывает все простые числа от 2 до 2 000 000. (В этом диапазоне 148 933 простых числа, так что вы вряд ли захотите распечатать их все.)

Не слишком вдаваясь в детали, поясним: ключ к достижению цели лежит в проходе по битовому набору из 2 миллионов бит. Сначала мы “включаем” все эти биты — выставляем их в 1. После этого начинаем выключать те биты, которые кратны известным простым числам. Позиции битов, оставшихся после этого процесса сами представляют простые числа. В листинге 13.7 показана эта программа на языке Java, а в листинге 13.8 представлен код на C++.



На заметку! Несмотря на то что “решето Эратосфена” — не лучший тест, мы не преминули сравнить время работы этих двух реализаций алгоритма. Вот что получилось на двухядерном процессоре 1,66 ГГц ThinkPad с 2 Гбайт ОЗУ, функционирующего под управлением ОС Ubuntu 7.04:

- C++ (g++ 4.1.2): 360 мс
- Java (Java SE 6): 105 мс

Мы запускали этот тест для семи изданий настоящей книги, и в трех последних редакциях язык Java легко “победил” C++. Справедливости ради надо сказать, что при повышении уровня оптимизации компилятора C++ он обгонял Java на 60 мс. Java мог сравниться с этим показателем только при достаточно долгой работе программы, чтобы вступил в работу JIT-компилятор Hotspot.

Листинг 13.7. Содержимое файла `Sieve.java`

```
1.  import java.util.*;
2.
3.  /**
4.   * Эта программа выполняет тест "решето Эратосфена", подсчитывая
5.   * простые числа до 2 000 000.
6.   * @version 1.21 2004-08-03
7.   * @author Cay Horstmann
8.   */
9.  public class Sieve
10. {
11.   public static void main(String[] s)
12.   {
13.      int n = 2000000;
14.      long start = System.currentTimeMillis();
15.      BitSet b = new BitSet(n + 1);
16.      int count = 0;
17.      int i;
18.      for (i = 2; i <= n; i++)
19.         b.set(i);
```

```
      i = 2;
      while (i * i <= n)
      {
         if (b.get(i))
         {
            count++;
            int k = 2 * i;
            while (k <= n)
            {
               b.clear(k);
               k += i;
            }
         }
         i++;
      }

      while (i <= n)
      {
         if (b.get(i)) count++;
         i++;
      }

      long end = System.currentTimeMillis();
      System.out.println(count + " простых чисел");
      System.out.println((end - start) + " мс");
   }
}
```

Листинг 13.8. Содержимое файла `Sieve.cpp`

```
/**
 * @version 1.21 2004-08-03
 * @author Cay Horstmann
 */

#include <bitset>
#include <iostream>
#include <ctime>

using namespace std;

int main()
{
  const int N = 2000000;
  clock_t cstart = clock();

  bitset<N + 1> b;
  int count = 0;
  int i;
  for (i = 2; i <= N; i++)
      b.set(i);
  i = 2;
  while (i * i <= N)
  {
```

```
25.       if (b.test(i))
26.       {
27.          count++;
28.          int k = 2 * i;
29.          while (k <= N)
30.          {
31.             b.reset(k);
32.             k += i;
33.          }
34.       }
35.       i++;
36.    }
37.    while (i <= N)
38.    {
39.       if (b.test(i))
40.          count++;
41.       i++;
42.    }
43.
44.    clock_t cend = clock();
45.    double millis = 1000.0
46.       * (cend - cstart) / CLOCKS_PER_SEC;
47.
48.    cout << count << " простых чисел\n"
49.         << millis << " мс\n";
50.
51.    return 0;
52. }
```

На этом мы завершаем нашу экскурсию по каркасу коллекций Java. Как вы убедились, библиотека Java предлагает широкое разнообразие классов коллекций для ваших программистских потребностей. В завершающей главе книги мы обсудим важнейшую тему программирования с параллелизмом.

Глава 14

МНОГОПОТОЧНОСТЬ

В ЭТОЙ ГЛАВЕ...

- Что такое потоки?
- Прерывание потоков
- Состояния потока
- Свойства потока
- Синхронизация
- Блокирующие очереди
- Коллекции, безопасные к потокам
- Интерфейсы **`Callable`** и **`Future`**
- Класс **`Executors`**
- Синхронизаторы
- Потоки и Swing

Вероятно, вы знакомы с *многопоточностью* операционной системы — возможностью одновременного выполнения более чем одной программы. Например, вы можете печатать во время редактирования или приема электронной почты. В наше время, вполне возможно, что ваш компьютер оснащен более чем одним центральным процессором, однако число одновременно выполняющихся процессов не ограничено количеством процессоров. Операционная система выделяет время процессора "кусочками" для каждого процесса, создавая впечатление параллельной работы.

Многопоточные программы расширяют идею многозадачности, добавляя один нижний уровень: индивидуальные приложения могут выполнять множество задач в одно и то же время. Каждая задача обычно называется потоком (thread) — имеется в виду "поток управления". Программы, которые могут выполнять более одного потока одновременно, называются *многопоточными*.

Так в чем же отличие между многими *процессами* и многими *потоками*? Существенная разница состоит в том, что в то время как каждый процесс имеет собственный набор переменных, потоки могут разделять одни и те же данные. Это кажется несколько рискованным, и в самом деле так оно и есть, как вы убедитесь далее в этой главе. Однако разделенные переменные обеспечивают более высокую эффективность взаимодействия между потоками и облегчают коммуникации между потоками. Вдобавок в некоторых операционных системах потоки являются более "легковесными", чем процессы — они требуют меньше накладных расходов на создание и уничтожение индивидуальных потоков по сравнению с созданием и уничтожением процессов.

Многопоточность имеет исключительную практическую ценность. Например, браузер должен уметь одновременно загружать множество изображений. Web-серверу нужно обслуживать параллельные запросы. Программы с графическим пользовательским интерфейсом имеют отдельные потоки для сбора интерфейсных событий от окружающей операционной среды. В этой главе мы покажем, как добавлять многопоточные возможности к Java-приложениям.

Многопоточность существенно изменилась в Java SE 5.0 — добавилось огромное количество классов и интерфейсов, представляющих высококачественные реализации механизмов, которые необходимы большинству программистов. В этой главе мы расскажем о средствах, которые появились в Java SE 5.0, наряду с классическими механизмами синхронизации, и поможем вам осуществлять выбор между ними.

Внимание! Многопоточность может оказаться очень сложной. В этой главе мы опишем все инструменты, которые, вероятно, понадобятся прикладным программистам. Однако для более сложного программирования системного уровня мы советуем обратиться к более серьезным источникам, таким как книга Брайана Гоетца (Brian Goetz) *Java Concurrency in Practice* (Addison-Wesley Professional, 2006).

Что такое потоки?

Начнем с рассмотрения программы, которая не использует средств многопоточности, и, как следствие, затрудняет пользователю выполнение нескольких задач сразу. После того, как мы проанализируем ее, мы покажем, как легко заставить эту программу запускать отдельные потоки. Эта программа будет анимировать прыгающий мяч, постоянно перемещая его по экрану и меняя направление движения, когда он встретит преграду в виде стены (рис. 14.1).

Как только вы щелкнете на кнопке Start (Пуск), программа выбросит мяч из правого левого угла и он начнет прыгать. Обработчик события кнопки Start вызывает метод `addBall`. Этот метод содержит цикл из 1000 движений. Каждый вызов `move()`

перемещает мяч на небольшое расстояние, изменяет направление движения, если он отскакивает от стены, и затем перерисовывает панель.

```
Ball ball = new Ball();
panel.add(ball);
for (int i = 1; i <= STEPS; i++)
{
   ball.move(panel.getBounds());
   panel.paint(panel.getGraphics());
   Thread.sleep(DELAY);
}
```

Статический метод `sleep()` класса `Thread` реализует задержку на заданное количество миллисекунд.

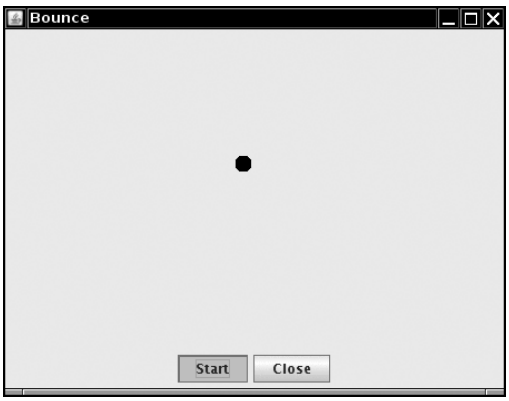


Рис. 14.1. Использование потока для анимации прыгающего мяча

Вызов `Thread.sleep()` не создает нового потока, этот статический метод класса `Thread` временно приостанавливает действие текущего потока.

Метод `sleep()` может сгенерировать исключение `InterruptedException`. Позже мы поговорим об этом исключении и его правильной обработке. А пока мы просто прекратим прыжки мяча, если случится такое исключение.

Если вы запустите эту программу, то мяч будет прыгать почти замечательно, однако при этом он полностью захватит управление программой. Если вам надоест наблюдать движение мяча до того, как он выполнит 1000 движений, и вы щелкнете на кнопке Close (Закрыть), мяч все равно продолжит прыгать. Вы не можете взаимодействовать с программой до тех пор, пока мяч не завершит свои прыжки.



На заметку! Если вы внимательно присмотритесь к коду, приведенному в конце этого раздела, то заметите вызов `comp.paint(comp.getGraphics())` внутри метода `addBall` класса `BounceFrame`. Это довольно странно — ведь обычно вы вызываете `repaint()` и позволяете AWT позаботиться о получении графического контекста и выполнении рисования. Но если вы попытаетесь вызвать `comp.repaint()` в этой программе, панель никогда не будет перерисована, потому что метод `addBall()` полностью захватил управление. Также обратите внимание на то, что компонент расширяет класс `JPanel`; это облегчает очистку фона. В следующей программе, где мы используем отдельный поток для вычисления положения мяча, можно будет вернуться к привычному применению `repaint()` и `JComponent`.

Очевидно, что поведение этой программы никуда не годится. Оно совершенно не подходит для программ, выполняющих более или менее длительную работу. В конце концов, когда вы читаете данные по сетевому соединению, часто случается иметь дело с задачами, которые вы *действительно* хотели бы прервать. Например, предположим, что вы загружаете огромное графическое изображение и, увидев его часть, решаете, что вам не нужно видеть остальное; конечно же, неплохо иметь возможность щелчком на кнопке Stop (Останов) или Back (Назад) прервать процесс загрузки. В следующем разделе мы покажем, как предоставить пользователю контроль над таким процессом, запуская критичные части кода в отдельном *потоке*.

Листинг 14.1. Содержимое файла `Bounce.java`

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * Демонстрирует анимированный прыгающий мяч.
7.  * @version 1.33 2007-05-17
8.  * @author Cay Horstmann
9.  */
10. public class Bounce
11. {
12.  public static void main(String[] args)
13.  {
14.   EventQueue.invokeLater(new Runnable()
15.      {
16.        public void run()
17.        {
18.             JFrame frame = new BounceFrame();
19.             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20.             frame.setVisible(true);
21.        }
22.      });
23.  }
24. }
25.
26. /**
27.  * Фрейм с компонентом-мячом и кнопками.
28.  */
29. class BounceFrame extends JFrame
30. {
31. /**
32.  * Конструирует фрейм с компонентом для отображения
33.  * прыгающего мяча и кнопками Start и Close.
34.  */
35.  public BounceFrame()
36.  {
37.   setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
38.   setTitle("Bounce");
39.
40.   comp = new BallComponent();
41.   add(comp, BorderLayout.CENTER);
42.   JPanel buttonPanel = new JPanel();
```

```
43.   addButton(buttonPanel, "Start", new ActionListener()
44.     {
45.        public void actionPerformed(ActionEvent event)
46.        {
47.            addBall();
48.        }
49.     });
50.   addButton(buttonPanel, "Close", new ActionListener()
51.     {
52.       public void actionPerformed(ActionEvent event)
53.       {
54.            System.exit(0);
55.       }
56.     });
57.   add(buttonPanel, BorderLayout.SOUTH);
58. }
59. /**
60.  * Добавляет кнопку в контейнер.
61.  * @param c Контейнер
62.  * @param title Заголовок кнопки
63.  * @param listener Слушатель действия кнопки
64.  */
65. public void addButton(Container c, String title, ActionListener listener)
66. {
67.  JButton button = new JButton(title);
68.  c.add(button);
69.  button.addActionListener(listener);
70. }
71.
72. /**
73.  * Добавляет прыгающий мяч к панели и заставляет его прыгать 1000 раз.
74.  */
75. public void addBall()
76. {
77.  try
78.  {
79.     Ball ball = new Ball();
80.     comp.add(ball);
81.     for (int i = 1; i <= STEPS; i++)
82.     {
83.        ball.move(comp.getBounds());
84.        comp.paint(comp.getGraphics());
85.        Thread.sleep(DELAY);
86.     }
87.  }
88.  catch (InterruptedException e)
89.  {
90.  }
91. }
92.
93. private BallComponent comp;
94. public static final int DEFAULT_WIDTH = 450;
95. public static final int DEFAULT_HEIGHT = 350;
96. public static final int STEPS = 1000;
97. public static final int DELAY = 3;
98. }
```

Листинг 14.2. Содержимое файла `Ball.java`

```
1. import java.awt.geom.*;
2.
3. /**
4.  * Мяч, перемещающийся и отскакивающий от границ прямоугольника.
5.  * @version 1.33 2007-05-17
6.  * @author Cay Horstmann
7.  */
8. public class Ball
9. {
10. /**
11.  * Перемещает мяч в следующую позицию, меняя направление
12.  * при столкновении с краем поля.
13.  */
14. public void move(Rectangle2D bounds)
15. {
16.  x += dx;
17.  y += dy;
18.  if (x < bounds.getMinX())
19.  {
20.    x = bounds.getMinX();
21.    dx = -dx;
22.  }
23.  if (x + XSIZE >= bounds.getMaxX())
24.  {
25.    x = bounds.getMaxX() - XSIZE;
26.    dx = -dx;
27.  }
28.  if (y < bounds.getMinY())
29.  {
30.    y = bounds.getMinY();
31.    dy = -dy;
32.  }
33.  if (y + YSIZE >= bounds.getMaxY())
34.  {
35.    y = bounds.getMaxY() - YSIZE;
36.    dy = -dy;
37.  }
38. }
39. /**
40.  * Получает форму мяча в текущей позиции.
41.  */
42. public Ellipse2D getShape()
43. {
44.  return new Ellipse2D.Double(x, y, XSIZE, YSIZE);
45. }
46.
47.private static final int XSIZE = 15;
48.  private static final int YSIZE = 15;
49. private double x = 0;
50. private double y = 0;
51. private double dx = 1;
52. private double dy = 1;
53.}
```

Листинг 14.3. Содержимое файла `BallComponent.java`

```
1. import java.awt.*;
2. import java.util.*;
3. import javax.swing.*;
4.
5. /**
6.  * Компонент, отображающий мячи.
7.  * @version 1.33 2007-05-17
8.  * @author Cay Horstmann
9.  */
10. public class BallComponent extends JPanel
11. {
12.  /**
13.   * Добавить мяч к компоненту.
14.   * @param b Добавляемый мяч
15.   */
16.  public void add(Ball b)
17.  {
18.   balls.add(b);
19.  }
20.
21.  public void paintComponent(Graphics g)
22.  {
23.   super.paintComponent(g); // очистка фона
24.   Graphics2D g2 = (Graphics2D) g;
25.   for (Ball b : balls)
26.   {
27.     g2.fill(b.getShape());
28.   }
29.  }
30.
31.  private ArrayList<Ball> balls = new ArrayList<Ball>();
32. }
```

API **`java.lang.Thread 1.0`**

- `static void sleep(long mills)`

  "Засыпает" на заданное количество миллисекунд.

  Параметры: `mills` Количество миллисекунд задержки.

Использование потоков для предоставления шанса другим задачам

Теперь попробуем сделать нашу программу с прыгающим мячом отзывчивой, запустив код перемещения мяча в отдельном потоке. Фактически, в результате вы сможете запускать множество мячей. Каждый из них будет двигаться в своем собственном потоке. В дополнение *поток диспетчера событий* AWT продолжит работать параллельно, взяв на себя заботу о событиях пользовательского интерфейса. Поскольку каждый поток получает возможность выполниться, поток диспетчера событий также получит возможность известить программу, когда пользователь щелкнет на кнопке Close в процессе движения мяча. Затем поток может "закрыть" действие.

Мы используем код прыгающего мяча для того, чтобы дать вам визуальное представление о необходимости параллелизма. Вообще всегда нужно относиться с осторожностью к всякого рода долго выполняющимся вычислениям. Такие вычисления всегда являются частью некоторого объемлющего каркаса, такого как графический пользовательский интерфейс или Web-приложение. Всякий раз, когда этот каркас вызывает ваши методы, ожидается быстрый возврат. Если ваша задача требует для своего выполнения длительного времени, вы должны использовать отдельный поток.

Ниже описана простая процедура для запуска задачи в отдельном потоке.

1. Поместить код задачи в метод run класса, реализующего интерфейс `Runnable`. Этот интерфейс очень прост, он включает единственный метод:

```
public interface Runnable
{
    void run();
}
```

   Вы просто реализуете класс вроде следующего:

```
class MyRunnable implements Runnable
{
    public void run()
    {
       код задачи
    }
}
```

2. Сконструировать объект вашего класса:

```
Runnable r = new MyRunnable();
```

3. Сконструировать объект `Thread` из `Runnable`:

```
Thread t = new Thread(r);
```

4. Запустить поток:

```
t.start();
```

Чтобы поместить нашу программу прыгающего мяча в отдельный поток, понадобится лишь реализовать класс `BallRunnable` и поместить код анимации в метод `run`, как показано ниже:

```
class BallRunnable implements Runnable
{
 . . .
 public void run()
 {
  try
  {
     for (int i = 1; i <= STEPS; i++)
     {
       ball.move(component.getBounds());
       component.repaint();
       Thread.sleep(DELAY);
     }
  }
  catch (InterruptedException exception)
  {
  }
 }
 . . .
}
```

Опять-таки, мы должны перехватить исключение `InterruptedException`, которое может сгенерировать метод `run`. Об исключениях мы поговорим в следующем разделе. Обычно это исключение применяется для того, чтобы запросить прерывание потока. Соответственно, при этом происходит выход из метода `run`. При каждом щелчке на кнопке Start метод `addBall()` запускает новый поток (рис. 14.2).

```
Ball b = new Ball();
panel.add(b);
Runnable r = new BallRunnable(b, panel);
Thread t = new Thread(r);
t.start();
```

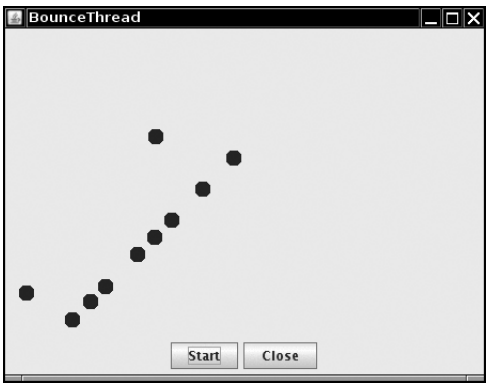


Рис. 14.2. Множество работающих потоков

Вот и все! Теперь вы знаете, как запустить задачи параллельно. Остальная часть этой главы будет посвящена вопросам взаимодействия между потоками.

Полный код приведен в листинге 14.4.



На заметку! Вы можете также определить потоки, создавая подклассы `Thread`, как показано ниже:

```
class MyThread extends Thread
{
    public void run()
    {
        код задачи
    }
}
```

Затем вы конструируете объект этого подкласса и вызываете его метод `start`. Однако такой подход теперь не рекомендуется. Следует отделять *задачу*, которая должна быть выполнена параллельно, от *механизма* ее выполнения. Если у вас есть множество задач, то будет слишком дорого создавать отдельный поток для каждой из них. Вместо этого вы можете использовать пул потоков. Об этом читайте в разделе “Класс `Executors`” настоящей главы.



Внимание! Не вызывайте метод `run()` класса `Thread` или объекта `Runnable`. Прямой вызов этого метода просто выполнит задачу в том же потоке — новый поток запущен не будет. Вместо этого вызывайте метод `Thread.start()`. Он создаст новый поток, который и выполнит метод `run()`.

Листинг 14.4. Содержимое файла `Ball.java`

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4. /**
5.  * Демонстрирует анимированный прыгающий мяч.
6.  * @version 1.33 2007-05-17
7.  * @author Cay Horstmann
8.  */
9. public class BounceThread
10. {
11.  public static void main(String[] args)
12.  {
13.   EventQueue.invokeLater(new Runnable()
14.   {
15.      public void run()
16.      {
17.        JFrame frame = new BounceFrame();
18.        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.        frame.setVisible(true);
20.      }
21.   });
22.  }
23. }
24. /**
25.  * Класс, реализующий Runnable для анимации прыгающего мяча.
26.  */
27. class BallRunnable implements Runnable
28. {
29.  /**
30.   * Конструктор Runnable.
31.   * @aBall Прыгающий мяч
32.   * @aPanel Компонент, в котором прыгает мяч
33.   */
34.  public BallRunnable(Ball aBall, Component aComponent)
35.  {
36.   ball = aBall;
37.   component = aComponent;
38.  }
39.  public void run()
40.  {
41.   try
42.   {
43.      for (int i = 1; i <= STEPS; i++)
44.      {
45.        ball.move(component.getBounds());
46.        component.repaint();
47.        Thread.sleep(DELAY);
48.      }
49.   }
50.   catch (InterruptedException e)
51.   {
52.   }
53.  }
54.  private Ball ball;
55.  private Component component;
56.  public static final int STEPS = 1000;
57.  public static final int DELAY = 5;
58. }
```

```
59. /**
60.  * Фрейм с панелью и кнопками.
61.  */
62. class BounceFrame extends JFrame
63. {
64.  /**
65.   * Конструирует фрейм с компонентом для отображения
66.   * прыгающего мяча и кнопками Start и Close.
67.   */
68.  public BounceFrame()
69.  {
70.   setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
71.   setTitle("BounceThread");
72.   comp = new BallComponent();
73.   add(comp, BorderLayout.CENTER);
74.   JPanel buttonPanel = new JPanel();
75.   addButton(buttonPanel, "Start", new ActionListener()
76.      {
77.        public void actionPerformed(ActionEvent event)
78.        {
79.             addBall();
80.        }
81.      });
82.   addButton(buttonPanel, "Close", new ActionListener()
83.      {
84.        public void actionPerformed(ActionEvent event)
85.        {
86.             System.exit(0);
87.        }
88.      });
89.   add(buttonPanel, BorderLayout.SOUTH);
90.  }
91.  /**
92.   * Добавляет кнопку в контейнер.
93.   * @param c Контейнер
94.   * @param title Метка кнопки
95.   * @param listener Слушатель событий кнопки
96.   */
97.  public void addButton(Container c, String title, ActionListener listener)
98.  {
99.   JButton button = new JButton(title);
100.  c.add(button);
101.  button.addActionListener(listener);
102. }
103. /**
104.  * Добавляет прыгающий мяч к панели и запускает поток.
105.  */
106. public void addBall()
107. {
108.  Ball b = new Ball();
109.  comp.add(b);
110.  Runnable r = new BallRunnable(b, comp);
111.  Thread t = new Thread(r);
112.  t.start();
113. }
114. private BallComponent comp;
115. public static final int DEFAULT_WIDTH = 450;
116. public static final int DEFAULT_HEIGHT = 350;
117. public static final int STEPS = 1000;
118. public static final int DELAY = 3;
119. }
```

API java.lang.Thread 1.0

- `Thread(Runnable target)`

  Конструирует новый поток, вызывающий метод `run()` указанной цели.

- `void start()`

  Запускает поток, инициируя при этом вызов метода `run()`. Этот метод немедленно возвращает управление. Новый поток выполняется параллельно.

- `void run()`

  Вызывает метод `run()` ассоциированного объекта `Runnable`.

API java.lang.Runnable 1.0

- `void run()`

  Метод должен быть переопределен и содержать инструкции для выполнения нужной задачи в потоке.

Прерывание потоков

Поток прерывается, когда его метод `run` возвращает управление, выполнив оператор `return`, после выполнения последнего оператора в теле метода, или если случается исключение, которое не перехвачено в методе. В начальном выпуске Java также присутствовал метод `stop()`, который мог быть вызван другим потоком, чтобы прервать выполнение данного. Однако этот метод теперь не рекомендуется к применению. Причину мы обсудим в разделе "Почему методы `stop()` и `suspend()` не рекомендованы к применению".

Существует способ *принудительного* прерывания потока. Однако метод `interrupt()` может применяться для *запроса* прерывания потока.

Когда метод `interrupt()` вызывается с потоком, устанавливается его *статус прерывания* (interrupted status). Это флаг типа `boolean`, имеющийся у каждого потока. Каждый поток должен периодически проверять значение этого флага, чтобы знать, когда он должен быть прерван.

Чтобы узнать, был ли установлен статус прерывания, вызывается статический метод `Thread.currentThread()` для получения текущего потока и последующего вызова метода `isInterrupted()`:

```
while (!Thread.currentThread().isInterrupted() && есть еще что делать)
{
   выполнять какую-то работу
}
```

Однако если поток блокирован, он не может проверить статус прерывания. И здесь на помощь приходит `InterruptedException`. Когда метод `interrupt` вызывается для потока, который заблокирован таким вызовом, как `sleep()` или `wait()`, то блокирующий вызов прерывается исключением `InterruptedException`. (Существуют блокирующие вызовы ввода-вывода, которые не могут быть прерваны; в этих случаях нужно искать прерываемые альтернативы. За подробностями обращайтесь во второй том.)

Не существует требования языка, чтобы прерванный поток прекратил работу. Прерывание потока просто привлекает его внимание. Прерванный поток может сам решить, как реагировать на прерывание. Некоторые потоки настолько важны, что

должны обработать исключение и продолжать работу. Но достаточно часто поток должен просто интерпретировать прерывание как запрос на прекращение. Метод `run()` такого потока имеет следующую форму.

```
public void run()
{
 try
 {
 . . .
  while (!Thread.currentThread().isInterrupted() && еще есть работа)
  {
    выполнять какую-то работу
  }
 }

 catch(InterruptedException e)
 {
  // поток был прерван во время ожидания
 }
 finally
 {
  при необходимости очистить
 }
 // выход из метода run завершает поток
}
```

Проверка `isInterrupted()` не обязательна и не удобна, если можно вызвать метод `sleep()` (или другой прерываемый метод) после каждой рабочей итерации. Если вы вызываете метод `sleep()`, когда установлен статус прерывания, поток не "засыпает". Вместо этого он очищает статус (!) и генерирует исключение `InterruptedException`. Таким образом, если ваш цикл вызывает `sleep()`, не проверяйте статус прерывания. Вместо этого перехватывайте `InterruptedException`, как показано ниже.

```
public void run()
{
 try
 {
  . . .
  while (еще есть работа)
  {
    выполнять какую-то работу
    Thread.sleep(delay);
  }
 }

 catch(InterruptedException e)
 {
  // поток был прерван во время ожидания
 }
 finally
 {
  при необходимости - очистить
 }
 // выход из метода run завершает поток
}
```



На заметку! Существуют два очень похожих метода — `interrupted()` и `isInterrupted()`. `interrupted()` — статический метод, проверяющий, был ли прерван *текущий* поток. Более того, вызов этого метода *очищает* статус прерывания потока. С другой стороны, `isInterrupted()` — метод экземпляра, который вы можете использовать для проверки того, был ли прерван любой поток. Его вызов не изменяет статуса прерывания.

Можно найти массу опубликованного кода, где на низком уровне прерывание `InterruptedException` игнорируется, примерно так:

```
void mySubTask()
{
  . . .
  try { sleep(delay); }
  catch (InterruptedException e) {} // НЕ ИГНОРИРУЙТЕ!
  . . .
}
```

Не поступайте так! Если вы не можете придумать ничего полезного, что можно было бы поместить в блок `catch`, у вас есть два разумных выбора.

- В блоке `catch` вызвать `Thread.currentThread().interrupt()`, чтобы установить статус прерывания. Затем вызывающий код может проверить его.

```
void mySubTask()
{
  . . .
  try { sleep(delay); }
  catch (InterruptedException e) { Thread.currentThread().interrupt(); }
  . . .
}
```

- Или, что лучше, снабдите сигнатуру метода "заклинанием" `throws InterruptedException` и удалите блок `try`. Тогда вызывающий код (или, в конечном итоге, метод `run`) сможет перехватить его.

```
void mySubTask() throws InterruptedException
{
  . . .
  sleep(delay);
  . . .
}
```



java.lang.Thread 1.0

- `void interrupt()`

  Посылает потоку запрос на прерывание. Статус прерывания потока устанавливается в `true`. Если поток в данный момент блокирован вызовом `sleep`, тогда генерируется `InterruptedException`.

- `static boolean interrupted()`

  Проверяет, был ли прерван текущий поток. Обратите внимание, что это статический метод. Его вызов имеет побочный эффект — статус прерывания текущего потока сбрасывается в `false`.

- `boolean isInterrupted()`

  Проверяет, был ли прерван поток. В отличие от статического метода `interrupted()`, этот вызов не изменяет статуса прерывания потока.

- `static Thread currentThread()`

  Возвращает объект `Thread`, представляющий текущий выполняющийся поток.

Состояния потока

Потоки могут пребывать в одном из шести состояний:

- новый;
- работоспособный;
- блокированный;
- ожидающий;
- временно ожидающий;
- завершенный.

Каждое из этих состояний мы объясним в последующих разделах.
Чтобы определить текущее состояние потока, просто вызовите метод `getState()`.

Новые потоки

Когда вы только создали поток операцией `new`, например, `new Thread(r)`, этот поток еще не запущен. Это означает, что он находится в состоянии "новый". Пока он находится в этом состоянии, это значит, что программа еще не запустила на выполнение код, находящийся внутри него. Прежде, чем поток будет запущен, нужно выполнить некоторый объем работы.

Работоспособные потоки

Как только вы вызовете метод `start()`, поток оказывается в работоспособном (runnable) состоянии. Работоспособный поток может в данный момент выполняться, а может и нет. Зависит от операционной системы — будет ли выделено потоку время на выполнение. (Спецификация Java, однако, не называет это отдельным состоянием. Работающий поток также находится в "работоспособном" состоянии.)

Если поток запущен, он не обязательно постоянно продолжает выполняться. Фактически, даже желательно, чтобы выполняющийся поток периодически приостанавливался, давая возможность поработать другим потокам. Детали планирования выполнения потоков зависят от служб, предоставляемых операционной системой. Системы с примитивным планированием выделяют каждому работоспособному потоку по отрезку времени для выполнения задачи. Когда этот отрезок истекает, операционная система *выгружает* поток и дает возможность поработать другому потоку (рис. 14.4). Выбирая следующий поток, операционная система принимает во внимание *приоритеты* потока — подробнее см. раздел "Приоритеты потока".

Все современные настольные и серверные операционные системы используют вытесняющее (приоритетное) планирование. Однако портативные устройства, такие как мобильные телефоны, могут использовать кооперативное планирование. В таких устройствах поток утрачивает управление, только когда вызывает метод `yield()`, блокируется или находится в состоянии ожидания.

На машинах с несколькими процессорами каждый процессор может выполнять поток, и можно иметь несколько потоков, работающих одновременно. Конечно, если потоков больше, чем процессоров, планировщик вынужден заниматься делением времени.

Следует всегда помнить, что работоспособный поток в любой момент времени может выполняться, а может и простаивать (вот почему состояние называется "работоспособным", а не "работающим").

Блокированные и ожидающие потоки

Когда поток заблокирован или находится в состоянии ожидания, он временно не активен. Он не выполняет никакого кода и потребляет минимум ресурсов. На планировщике потоков лежит обязанность повторно активизировать его. Подробности зависят от того, как было достигнуто неактивное состояние.

- Когда поток пытается получить внутренний объект блокировки (но не `Lock` из библиотеки `java.util.concurrent`), он становится *блокированным*. (Блокировки `java.util.concurrent` мы обсудим в разделе "Блокирующие объекты", а внутренние объекты блокировки — в разделе "Ключевое слово `synchronized`"). Поток разблокируется, когда все остальные потоки освобождают блокировку и планировщик потоков позволяет данному потоку захватить его.
- Когда поток ожидает другого потока для уведомления планировщика о наступлении некоторого условия, он входит в состояние *ожидания*. Эти состояния мы обсудим далее в разделе "Объекты условий". Это происходит вызовом методов `Object.wait()` или `Thread.join()`, либо ожиданием `Lock` или `Condition` в библиотеке `java.util.concurrent`. На практике разница между блокированным состоянием и состоянием ожидания не существенна.
- Несколько методов принимают параметр — длительность таймаута. Их вызов водит поток в состояние *временного ожидания*. Это состояние сохраняется до тех пор, пока либо не истечет указанное время таймаута, либо не будет получено соответствующее уведомление. К методам с таймаутом относятся `Object.wait()`, `Thread.join()`, `Lock.tryLock()` и `Condition.await()`.

На рис. 14.3 показаны состояния, в которых может находиться поток, и возможные переходы между ними. Когда поток заблокирован или находится в состоянии ожидания (или, конечно, когда он завершается), то другой поток планируется к запуску. Когда поток повторно активизируется (например, по истечении таймаута или когда ему удается захватить блокировку), то планировщик сравнивает его приоритет с приоритетом выполняющихся в данный момент потоков. Если приоритет данного потока ниже, он приостанавливается и запускается новый поток.

Завершенные потоки

Поток завершается по одной из следующих причин.

- Умирает естественной смертью при нормальном выходе из метода `run()`.
- Умирает внезапно, поскольку неперехваченное исключение прерывает метод `run()`.

В частности, вы можете уничтожить поток, вызвав его метод `stop()`. Этот метод генерирует объект ошибки `ThreadDeath`, который убивает поток. Однако метод `stop()` не рекомендован к применению, и вы должны избегать его вызовов в своем коде.

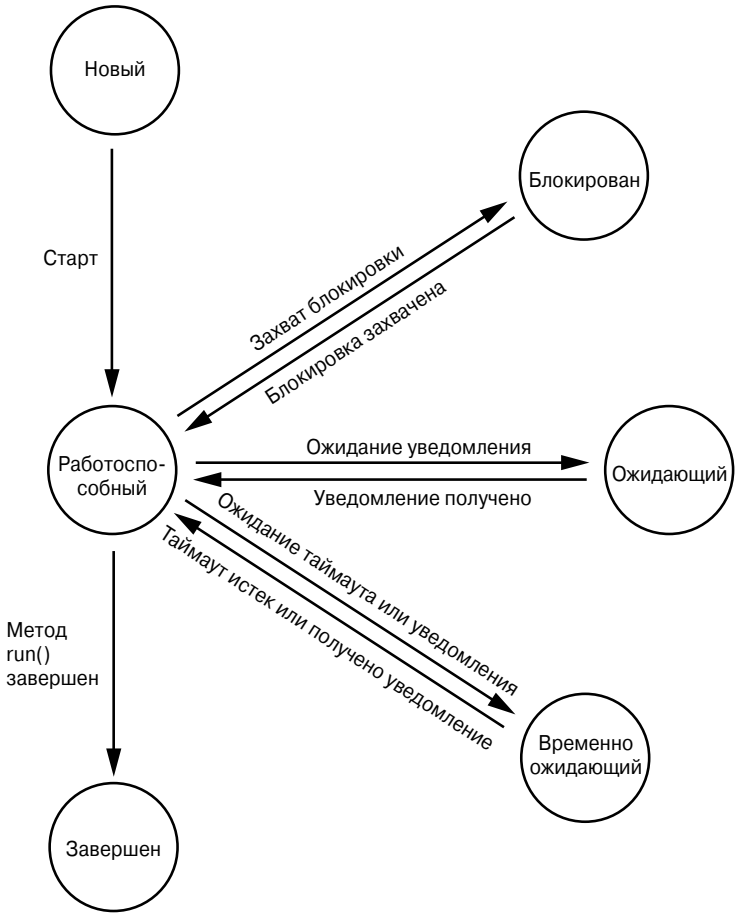


Рис. 14.3. Состояния потока

API **java.lang.Thread 1.0**

- `void join()`

  Ожидает завершения указанного потока.

- `void join(long mills)`

  Ожидает завершения указанного потока или истечения указанного периода времени.

- `Thread.State getState()` **5.0**

  Получает состояние данного потока; может принимать значения `NEW`, `RUNNABLE`, `BLOCKED`, `WAITING`, `TIMED_WAITING` или `TERMINATED`.

- `void stop()`

  Останавливает поток. Метод не рекомендован к применению.

- `void suspend()`

  Временно приостанавливает поток. Метод не рекомендован к применению.

- `void resume()`

  Возобновляет поток. Вызывается только после вызова `suspend()`. Метод не рекомендован к применению.

Свойства потока

В следующих разделах мы обсудим разнообразные свойства потоков: приоритеты потоков, потоки-демоны, группы потоков и обработчики неперехваченных исключений.

Приоритеты потоков

В языке программирования Java каждый поток имеет *приоритет*. По умолчанию поток наследует приоритет потока, который его создал. Вы можете повысить или понизить приоритет любого потока методом `setPriority()`. Можно установить приоритет в любое значение между `MIN_PRIORITY` (определен в классе `Thread` равным `1`) и `MAX_PRIORITY` (равен 10). `NORM_PRIORITY` определен равным 5.

Всякий раз, когда планировщик потоков выбирает новый поток для выполнения, он предпочитает потоки с более высоким приоритетом. Однако приоритеты потоков в значительной мере *зависимы от системы*. Когда виртуальная машина полагается на реализацию потоков платформы, на которой она выполняется, приоритеты потоков Java отображаются на уровни приоритетов этой платформы, которая может иметь больше или меньше уровней приоритетов.

Например, в Windows предусмотрено семь уровней приоритетов. Некоторые приоритеты Java отображаются на тот же уровень приоритета операционной системы. В виртуальной машине Sun для Linux приоритеты потоков игнорируются вообще. Все потоки имеют равный приоритет.

Начинающие программисты иногда злоупотребляют приоритетами потоков. Есть немного причин для того, чтобы "дергать" приоритеты. Вы никогда не должны строить свою программу так, чтобы ее правильное функционирование зависела от уровней приоритетов.



Внимание! Если вы используете приоритеты, вам следует знать о распространенных ошибках начинающих. Если есть несколько потоков с высоким приоритетом, которые никогда не бывают неактивными, то менее приоритетные потоки могут вообще *никогда* не выполниться. Всякий раз, когда планировщик решает запустить новый поток, он выбирает сначала среди наиболее высокоприоритетных потоков, даже если они полностью подавят низкоприоритетные потоки.



java.lang.Thread 1.0

- `void setPriority(int newPriority)`

  Устанавливает приоритет потока. Приоритет должен находиться между `Thread.MIN_PRIORITY` и `Thread.MAX_PRIORITY`. Для нормального приоритета указывайте `Thread.NORM_PRIORITY`.

- `static int MIN_PRIORITY`

  Минимальный приоритет, который может иметь `Thread`. Значение минимального приоритета равно `1`.

- `static int NORM_PRIORITY`

  Приоритет `Thread` по умолчанию. Значение приоритета по умолчанию равно `5`.

- `static int MAX_PRIORITY`

  Максимальный приоритет, который может иметь `Thread`. Значение максимального приоритета равно `10`.

- `static void yield()`

  Заставляет текущий выполняющийся поток уступить управление. Если есть другие работоспособные потоки с приоритетом не ниже приоритета данного потока, они будут запланированы для выполнения следующими. Обратите внимание, что этот метод — статический.

Потоки-демоны

Вы можете превратить поток в демон следующим вызовом:

```
t.setDaemon(true);
```

Правда, в таком потоке нет ничего "демонического". Демон — это просто поток, который не имеет других целей в жизни, кроме служения другим. Примерами могут служить потоки таймера, посылающие регулярные "тики" другим потокам, или потоки, очищающие состояние элементов кэша. Когда остаются только потоки-демоны, виртуальная машина завершает работу. Нет смысла продолжать выполнение программы, когда все оставшиеся потоки являются демонами.

Потоки-демоны иногда ошибочно используются начинающими программистами, которые не хотят думать о действиях по завершению. Однако это может быть опасно. Поток-демон никогда не должен обращаться к постоянному ресурсу? такому как файл или база данных, поскольку он может быть прерван в любой момент, даже посередине операции.

API **java.lang.Thread 1.0**

- `void setDaemon(boolean isDaemon)`

  Помечает данный поток как демон или пользовательский поток. Этот метод должен вызываться перед запуском потока.

Обработчики неперехваченных исключений

Метод `run()` потока не может генерировать никаких контролируемых исключений, но может быть прерван неконтролируемым исключением. В этом случае поток уничтожается.

Однако нет конструкции `catch`, куда может распространиться исключение. Вместо этого непосредственно перед смертью потока исключение передается обработчику неперехваченных исключений.

Обработчик должен относиться к классу, реализующему интерфейс `Thread.UncaughtExceptionHandler`. Этот интерфейс имеет единственный метод:

```
void uncaughtException(Thread t, Throwable e)
```

Начиная с Java SE 5.0, вы можете инсталлировать обработчик в любой поток методом `setUncaughtExceptionHandler`.

Вы можете также инсталлировать обработчик по умолчанию для всех потоков с помощью статического метода `setDefaultUncaughtExceptionHandler()` класса `Thread`. Заменяющий обработчик может использовать API протоколирования для отправки отчетов о необработанных исключениях в файл протокола.

Если вы не инсталлируете обработчик по умолчанию, то им является `null`. Однако если вы не инсталлируете обработчик для индивидуального потока, то обработчиком является объект потока `ThreadGroup`.



На заметку! Группа потоков — это коллекция потоков, которой можно управлять совместно. По умолчанию все потоки, которые вы создаете, относятся к одной и той же группе потоков, но можно устанавливать и другие группы. Поскольку в Java SE 5.0 появились лучшие средства для операций над коллекциями потоков, вам не нужно использовать группы потоков в ваших собственных программах.

Класс `ThreadGroup` реализует интерфейс `Thread.UncaughtExceptionHandler`. Его метод `uncaughtException()` выполняет следующие действия.

1. Если группа потоков не имеет родителя, вызывается метод `uncaughtException()` родительской группы.
2. В противном случае, если метод `Thread.getDefaultExceptionHandler` возвращает отличный от `null` обработчик, вызывается этот обработчик.
3. В противном случае, если `Throwable` является экземпляром `ThreadDeath`, то ничего не происходит.
4. В противном случае имя потока и трассировка стека `Throwable` печатаются в потоке ошибок `System.err`.

Именно такую трассировку стека вы, несомненно, видели много раз в ваших программах.

API java.lang.Thread 1.0

- `static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler)` **5.0**
- `static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()` **5.0**

  Устанавливают или получают обработчик по умолчанию для необработанных исключений.

- `void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler)` **5.0**
- `Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()` **5.0**

  Устанавливают или получают обработчик для необработанных исключений. Если никакой обработчик не инсталлирован, таким обработчиком является объект группы потоков.

API java.lang.Thread.UncaughtExceptionHandler 5.0

- `void uncaughtException(Thread t, Throwable e)`

  Определяется для протоколирования специального отчета при завершении потока с не обработанным исключением.

Параметры: `t` Поток, который был завершен из-за необработанного исключения.
`e` Объект не обработанного исключения.

API **java.lang.ThreadGroup 1.0**

- `void uncaughtException(Thread t, Throwable e)`

  Вызывает этот метод родительской группы потоков, если таковая имеется, или же вызывает обработчик по умолчанию класса `Thread`, если там есть обработчик по умолчанию, в противном случае выводит трассировку стека в стандартный поток ошибок. (Однако, если `e` — объект `ThreadDeath`, трассировка стека подавляется. Объекты `ThreadDeath` генерируются устаревшим методом `stop()`.)

Синхронизация

В большинстве практических многопоточных приложений двум или более потокам необходимо разделять доступ к одним и тем же данным. Что произойдет, если два потока имеют доступ к одному объекту и каждый из них вызывает метод, модифицирующий состояние этого объекта? Как и можно было предположить, потоки станут наступать друг другу на пятки. В зависимости от порядка обращения к данным, можно получить поврежденный объект. Такая ситуация часто называется *состоянием гонок* (race condition).

Пример состояния гонок

Чтобы избежать повреждения разделяемых данных множеством потоков, вам следует научиться *синхронизировать доступ*. В этом разделе вы увидите, что произойдет, если не использовать синхронизацию. В следующем разделе мы покажем, как синхронизировать обращение к данным.

В следующей тестовой программе мы эмулируем банк с множеством счетов. Будем случайно генерировать транзакции, переводящие деньги между счетами. Каждый счет владеет одним потоком. Каждая транзакция перемещает случайную сумму денег со счета, обслуживаемого текущим потоком, на другой произвольный счет.

Код эмуляции достаточно прост. У нас есть класс `Bank` с методом `transfer()`. Этот метод перемещает некоторую сумму денег с одного счета на другой. (Пока мы не будем беспокоиться об отрицательных остатках на счетах.) Ниже приведен код метода `transfer()` класса `Bank`.

```
public void transfer(int from, int to, double amount)
   // Внимание: небезопасно при вызове из нескольких потоков!
{
  System.out.print(Thread.currentThread());
  accounts[from] -= amount;
  System.out.printf(" %10.2f с %d на %d", amount, from, to);
  accounts[to] += amount;
  System.out.printf(" Общий баланс: %10.2f%n", getTotalBalance());
}
```

Далее рассмотрим код класса `TransferRunnable`. Его метод `run()` выполняет снятие денег с фиксированного банковского счета. В каждой итерации метод `run()` выбирает случайный целевой счет и случайную сумму, вызывает метод `transfer()` объекта `Bank`, затем засыпает.

```
class TransferRunnable implements Runnable
{
 . . .
 public void run()
 {
  try
  {
     int toAccount = (int) (bank.size() * Math.random());
     double amount = maxAmount * Math.random();
     bank.transfer(fromAccount, toAccount, amount);
     Thread.sleep((int) (DELAY * Math.random()));
  }
  catch(InterruptedException e) {}
 }
}
```

Когда этот эмулятор запускается, мы не знаем, какая сумма находится на любом банковском счете в любой момент времени. Но мы знаем, что общая сумма денег по всем счетам должна оставаться неизменной, поскольку мы переносим деньги с одного счета на другой, а не снимаем их окончательно.

В конце каждой транзакции метод `transfer()` заново вычисляет общую сумму и печатает ее. Эта программа никогда не прекращается. Чтобы ее уничтожить, просто нажмите <Ctrl+C>.

Ниже показан типичный вывод.

```
. . .
Thread[Thread-11,5,main] 588.48 с 11 на 44 Общий баланс: 100000.00
Thread[Thread-12,5,main] 976.11 с 12 на 22 Общий баланс: 100000.00
Thread[Thread-14,5,main] 521.51 с 14 на 22 Общий баланс: 100000.00
Thread[Thread-13,5,main] 359.89 с 13 на 81 Общий баланс: 100000.00
. . .
Thread[Thread-36,5,main] 401.71 с 36 на 73 Общий баланс: 99291.06
Thread[Thread-35,5,main] 691.46 с 35 на 77 Общий баланс: 99291.06
Thread[Thread-37,5,main] 78.64 с 37 на 3 Общий баланс: 99291.06
Thread[Thread-34,5,main] 197.11 с 34 на 69 Общий баланс: 99291.06
Thread[Thread-36,5,main] 85.96 с 36 на 4 Общий баланс: 99291.06
. . .
Thread[Thread-4,5,main]Thread[Thread-33,5,main] 7.31 с 31 на 32 Общий баланс:
99979.24
  627.50 с 4 на 5 Общий баланс: 99979.24
. . .
```

Как видим, что-то пошло совсем не так. В течение нескольких транзакций банковский баланс оставался в сумме $100 000, что совершенно верно — изначально было 100 счетов по $1000 на каждом. Но через некоторое время баланс слегка изменился. Когда вы запустите эту программу, то можете обнаружить, что ошибка возникнет быстро, или же общий баланс будет нарушен через довольно длительное время. Такая ситуация не внушает доверия, и вы, вероятно, не захотите положить ваши заработанные тяжким трудом денежки в такой банк.

В листингах с 14.5 по 14.7 представлен полный исходный код. Посмотрим, можно ли найти причину проблемы в коде. В следующем разделе мы распутаем эту головоломку.

Листинг 14.5. Содержимое файла `UnsynchBankTest.java`

```
1. /**
2.  * Демонстрирует повреждение данных при доступе множества потоков
3.  * к одной структуре данных.
4.  * @version 1.30 2004-08-01
5.  * @author Cay Horstmann
6.  */
7. public class UnsynchBankTest
8. {
9.  public static void main(String[] args)
10. {
11.  Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);
12.  int i;
13.  for (i = 0; i < NACCOUNTS; i++)
14.  {
15.     TransferRunnable r = new TransferRunnable(b, i, INITIAL_BALANCE);
16.     Thread t = new Thread(r);
17.     t.start();
18.  }
19. }
20.
21. public static final int NACCOUNTS = 100;
22. public static final double INITIAL_BALANCE = 1000;
23.}
```

Листинг 14.6. Содержимое файла `Bank.java`

```
1. /**
2.  * Банк со множеством счетов.
3.  * @version 1.30 2004-08-01
4.  * @author Cay Horstmann
5.  */
6. public class Bank
7. {
8.  /**
9.   * Конструктор банка.
10.  * @param n Количество счетов
11.  * @param initialBalance Начальный баланс каждого счета
12.  */
13. public Bank(int n, double initialBalance)
14. {
15.  accounts = new double[n];
16.  for (int i = 0; i < accounts.length; i++)
17.  accounts[i] = initialBalance;
18. }
19. /**
20.  * Перевод денег с одного счета на другой.
21.  * @param from Счет, с которого осуществляется перевод
22.  * @param to Счет, на который осуществляется перевод
23.  * @param amount Сумма перевода
24.  */
25. public void transfer(int from, int to, double amount)
26. {
27.  if (accounts[from] < amount) return;
28.  System.out.print(Thread.currentThread());
```

```
29.  accounts[from] -= amount;
30.  System.out.printf(" %10.2f с %d на %d", amount, from, to);
31.  accounts[to] += amount;
32.  System.out.printf(" Общий баланс: %10.2f%n", getTotalBalance());
33. }
34.
35. /**
36.  * Получить суммарный баланс.
37.  * @return Общий баланс
38.  */
39. public double getTotalBalance()
40. {
41.  double sum = 0;
42.  for (double a : accounts)
43.     sum += a;
44.  return sum;
45. }
46.
47. /**
48.  * Получить количество счетов в банке.
49.  * @return Количество счетов
50.  */
51. public int size()
52. {
53.  return accounts.length;
54. }
55. private final double[] accounts;
56.}
```

Листинг 14.7. Содержимое файла `TransferRunnable.java`

```
1. /**
2.  * Реализация Runnable, переводящая деньги в банке с одного счета на другой.
3.  * @version 1.30 2004-08-01
4.  * @author Cay Horstmann
5.  */
6. public class TransferRunnable implements Runnable
7. {
8.  /**
9.   * Конструктор Runnable.
10.  * @param b Банк, между счетами которого выполняется перевод
11.  * @param from Счет, с которого выполняется перевод
12.  * @param max Максимальная сумма одного перевода
13.  */
14. public TransferRunnable(Bank b, int from, double max)
15. {
16.  bank = b;
17.  fromAccount = from;
18.  maxAmount = max;
19. }
20. public void run()
21. {
22. try
23. {
```

```
24.  while (true)
25.  {
26.    int toAccount = (int) (bank.size() * Math.random());
27.    double amount = maxAmount * Math.random();
28.    bank.transfer(fromAccount, toAccount, amount);
29.    Thread.sleep((int) (DELAY * Math.random()));
30.   }
31.  }
32.  catch (InterruptedException e)
33.  {
34.  }
35. }
36. private Bank bank;
37. private int fromAccount;
38. private double maxAmount;
39. private int DELAY = 10;
40.}
```

Объяснение состояния гонок

В предыдущем разделе мы запустили программу, в которой несколько потоков обновляли балансы банковских счетов. По истечении некоторого времени в ней накапливаются ошибки, и в результате некоторая сумма либо теряется, либо добавляется ниоткуда. Такая проблема возникает, когда два потока пытаются одновременно обновлять один и тот же счет. Предположим, что два потока одновременно выполняют такую инструкцию:

```
accounts[to] += amount;
```

Проблема в том, что эта операция не *атомарна*. Такая инструкция может выполняться следующим образом:

1. Загрузить `accounts[to]` в регистр.
2. Добавить `amount`.
3. Перенести результат обратно в `accounts[to]`.

Теперь представим, что первый поток выполняет шаги 1 и 2, после чего приостанавливается. Предположим, второй поток просыпается и обновляет тот же элемент в массиве `accounts`. Затем просыпается первый поток и выполняет шаг 3. Такое действие уничтожает результат модификации второго потока. В результате общая сумма оказывается неверной (рис. 14.4).

Наша тестовая программа обнаруживает повреждение данных. (Конечно, есть шанс получить сигнал ложной тревоги, если поток будет прерван во время тестирования!)



На заметку! Вы можете просмотреть байт-коды, выполняемые виртуальной машиной в нашем классе. Запустите следующую команду для декомпиляции файла `Bank.class`:

```
javap -c -v Bank
```

Например, следующая строка:

```
accounts[to] += amount;
```

транслируется в приведенный ниже байт-код:

```
aload_0
getfield #2; //Поле accounts:[D
iload_2
dup2
daload
dload_3
dadd
dastore
```

Что именно означают эти коды — не важно. Важно то, что команда инкремента состоит из нескольких инструкций, и поток, выполняющий их, может быть прерван в любой точке.

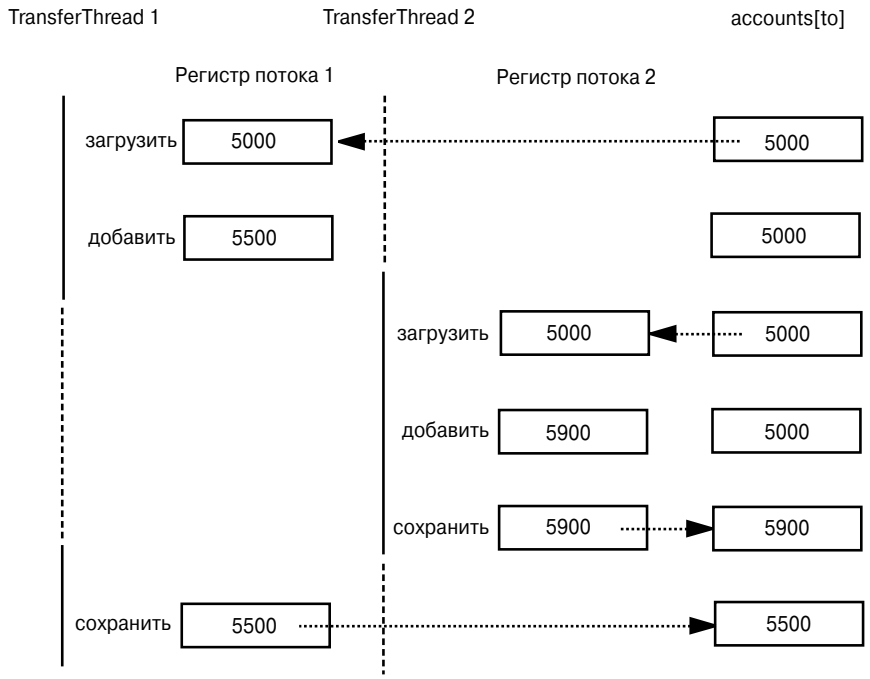


Рис. 14.4. Одновременный доступ из двух потоков

Каковы шансы повредить данные? Мы повысили вероятность проявления проблемы за счет того, что перемежали операторы печати операторами обновления баланса.

Если пропустить операторы печати, то риск повреждения немного снизится, поскольку каждый поток будет выполнять настолько мало работы прежде, чем заснуть, что прерывание на средине вычислений окажется маловероятным. Однако и в этом случае риск повреждения данных не исключается полностью. Если вы запустите достаточно много потоков на сильно загруженной машине, то программа даст сбой даже при отсутствии в программе операторов печати. Сбой произойдет через минуты, часы, или даже дни. Вообще говоря, нет ничего хуже для программиста, чем ошибка в программе, которая проявляется только раз за несколько дней.

Суть проблемы в том, что работа метода `transfer()` может быть прервана на полпути. Если мы сможем гарантировать завершение выполнения метода до того, как поток утратит управление, то состояние объекта банковского счета никогда не будет нарушено.

Блокирующие объекты

Начиная с Java SE 5.0, существуют два механизма для защиты блока кода от параллельного доступа. В языке Java для этой цели предусмотрено ключевое слово `synchronized`, а в Java SE 5.0 появился еще и класс `ReentrantLock`. Ключевое слово `synchronized` автоматически обеспечивает блокировку — так же, как ассоциированное "условие", которое удобно в большинстве случаев, когда требуется явная блокировка. Однако, увидев блокировки и условия в изоляции, мы полагаем, что проще понять ключевое слово `synchronized`. Каркас `java.util.concurrent` представляет отдельные классы для реализации этих фундаментальных механизмов, работу которых мы объясним в разделе "Объекты условий". Когда вы поймете, как устроены эти строительные блоки, мы перейдем к разделу "Ключевое слово `synchronized`".

Базовый эскиз защиты блока кода с помощью `ReentrantLock` выглядит так, как показано ниже.

```
myLock.lock(); // объект ReentrantLock
try
{
   критичный раздел
}
finally
{
   myLock.unlock(); // гарантировать разблокировку даже в случае
                    // возникновения исключения
}
```

Такая конструкция гарантирует, что только один поток в единицу времени сможет войти в критичный раздел кода. Как только один поток блокирует объект блокировки, никакой другой поток не сможет выполнить оператор `lock`. Когда другие потоки попытаются это сделать, они будут деактивизированы до тех пор, пока первый поток не снимает блокировку с этого объекта.



Внимание! Чрезвычайно важно, чтобы операция `unlock` помещалась в конструкцию `finally`. Если код критичного раздела сгенерирует исключение, блокировка должна быть снята. В противном случае другие потоки будут заблокированы навсегда.

Воспользуемся блокировкой для защиты нашего метода `transfer()` класса `Bank`.

```
public class Bank
{
public void transfer(int from, int to, int amount)
{
  bankLock.lock();
  try
  {
    System.out.print(Thread.currentThread());
    accounts[from] -= amount;
    System.out.printf(" %10.2f с %d на %d", amount, from, to);
    accounts[to] += amount;
    System.out.printf(" Общий баланс: %10.2f%n", getTotalBalance());
  }
  finally
  {
    bankLock.unlock();
  }
}
```

```
. . .
private Lock bankLock = new ReentrantLock(); // ReentrantLock реализует
                                             // интерфейс Lock
}
```

Предположим, что один поток вызывает `transfer()` и приостанавливается до его завершения, а второй поток также вызовет `transfer()`. Второй поток не сможет захватить блокировку и останется блокированным на вызове метода `lock()`. Он будет деактивизирован, и вынужден будет ждать, пока первый поток не завершит выполнение метода `transfer()`. Когда первый поток снимет блокировку, тогда только второй поток сможет продолжить работу (рис. 14.5).

Попробуем все это. Добавим блокирующий код в метод `transfer()` и запустим программу снова. Вы можете прогонять ее бесконечно, но общий баланс банка на этот раз не будет нарушен.

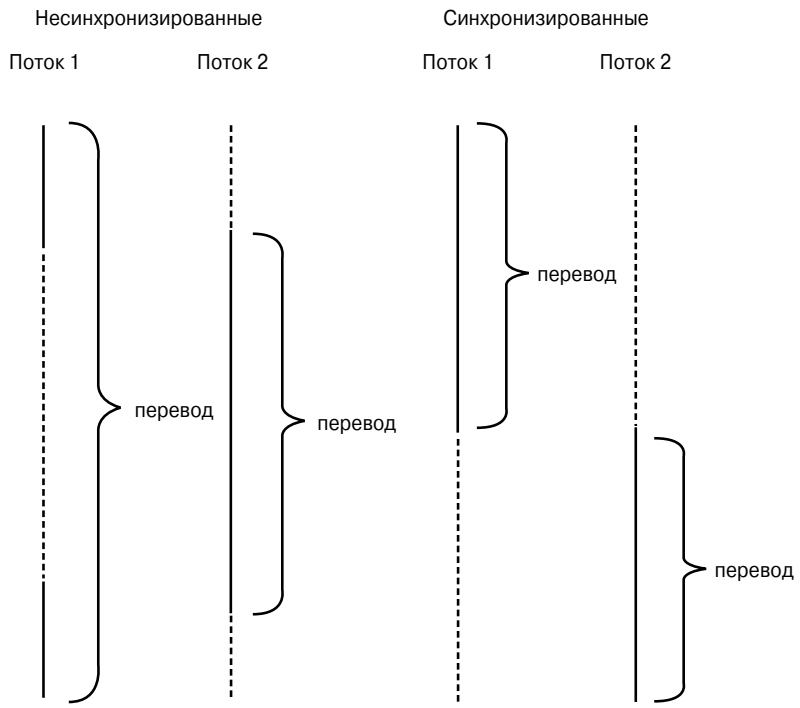


Рис. 14.5. Сравнение синхронизированных и не синхронизированных потоков

Отметим, что каждый объект `Bank` имеет свой собственный объект `ReentrantLock`. Если два потока попытаются обратиться к одному и тому же объект `Bank`, то этот объект блокировки сможет сериализовать доступ. Однако если два потока обращаются к разным объектам `Bank`, то каждый из них захватывает свою блокировку и ни один из потоков не блокируется. Так и должно быть, потому что потоки никак не перерываются друг с другом, работая с разными экземплярами `Bank`.

Блокировка называется *реентерабельной*, потому что поток может повторно захватывать блокировку, которой он уже владеет. Объект блокировки имеет счетчик блокировок, отслеживающий вложенные вызовы метода `lock()`. Поток должен вызвать `unlock()` для каждого вызова `lock()`, чтобы, в конце концов, снять блокировку.

Благодаря этому средству, код, защищенный блокировкой, может вызывать другой метод, использующий ту же самую блокировку.

Например, метод `transfer()` вызывает метод `getTotalBalance()`, который также блокирует объект `bankLock`, у которого после этого значения счетчика блокировок равно 2. Когда метод `getTotalBalance()` завершается, значение счетчика возвращается в 1. При выходе из метода `transfer()` счетчик имеет значение 0, и поток снимает блокировку.

Обычно вы захотите защищать блоки кода, обновляющие и проверяющие разделенный объект. Тогда можно быть уверенным, что эти операции полностью завершатся, прежде чем другой поток сможет использовать тот же объект.



Внимание! Следует быть внимательным, чтобы код в критичной секции программы не был пропущен из-за генерации исключения. Если исключение сгенерировано до конца секции, тогда оператор `finally` освободит блокировку, но объект может остаться в поврежденном состоянии.



java.util.concurrent.locks.Lock 5.0

- `void lock()`

  Захватывает блокировку; если она в данный момент захвачена другим потоком, текущий поток блокируется.

- `void unlock()`

  Снимает блокировку.



java.util.concurrent.locks.ReentrantLock 5.0

- `ReentrantLock()`

  Конструирует реентерабельную блокировку, которая может быть использована для защиты критичной части кода.

- `ReentrantLock(boolean fair)`

  Конструирует реентерабельную блокировку с заданной политикой равноправности. Равноправная блокировка отдает предпочтение потоку, ожидающему дольше всех. Однако такая равноправность может плохо отразиться на производительности. Поэтому по умолчанию равноправность от блокировок не требуется.



Внимание! Кажется, лучше быть равноправной, однако равноправные блокировки работают намного медленнее обычных. Вы должны включать равноправную блокировку только когда знаете, что делаете, и имеете на то особые причины — существенные для решения конкретной проблемы. Даже если вы используете равноправную блокировку, у вас все равно нет гарантий, что планировщик потоков будет также равноправным. Если планировщик потоков решит пренебречь потоком, который длительное время ожидает снятия блокировки, то никакая равноправность блокировки не спасет.

Объекты условий

Часто поток входит в критичную секцию только для того, чтобы обнаружить, что он не может работать, пока не будет выполнено определенное условие. В таких случаях вы будете использовать *объект условия* (condition object) для управления потоками, которые захватили блокировку, но не могут выполнить полезную работу. В этом разделе мы представим реализацию объектов условий в библиотеке Java (по историческим причинам *объекты условий* часто называют *условными переменными*).

Попробуем усовершенствовать наш эмулятор банка. Мы не будем перемещать деньги со счета, который не содержит достаточной суммы, чтобы покрыть нужный перевод. Отметим, что мы не можем использовать код следующего типа:

```
if (bank.getBalance(from) >= amount)
   bank.transfer(from, to, amount);
```

Вполне возможно, что текущий поток будет деактивизирован между успешным выполнением проверки и вызовом `transfer()`.

```
if (bank.getBalance(from) >= amount)
  // в этой точке поток может быть деактивизирован
bank.transfer(from, to, amount);
```

На момент возобновления работы потока баланс счета может измениться, т.е. уменьшиться ниже допустимого предела. Следует как-то гарантировать, что никакой другой поток не сможет модифицировать баланс между проверкой остатка и выполнением перевода. Это делается защитой и проверки и собственно перевода с помощью одной блокировки.

```
public void transfer(int from, int to, int amount)
{
  bankLock.lock();
  try
  {
    while (accounts[from] < amount)
    {
     // ждать
     . . .
    }
    // перевод
    . . .
  }
  finally
  {
    bankLock.unlock();
  }
}
```

Что мы делаем теперь, если на счету нет достаточной суммы? Мы ждем, пока какой-то другой поток не добавит денег. Но данный поток только что получил эксклюзивный доступ к `bankLock`, так что ни один другой поток не имеет шансов сделать взнос. И здесь на помощь приходит объект условия.

Объект блокировки может иметь один или более ассоциированных с ним объектов условий. Вы получаете объект условия методом `newCondition()`. Каждому объекту условия можно присвоить имя, напоминающее об условии, которое он представляет. Например, объект, представляющий условие “достаточной суммы” (sufficient funds), устанавливается следующим образом.

```
class Bank
{
  public Bank()
  {
  . . .
  sufficientFunds = bankLock.newCondition();
  }
  . . .
  private Condition sufficientFunds;
}
```

Если метод `transfer()` обнаружит, что сумма на счету недостаточно, он вызовет

```
sufficientFunds.await();
```

Теперь текущий поток деактивизирован и отдал блокировку. Это позволяет другому потоку, как мы надеемся, увеличить баланс текущего счета.

Имеется существенное отличие между потоком, ожидающим возможности захватить блокировку, и потоком, который вызвал `await()`. Как только поток вызвал метод `await()`, он входит в набор потоков, ожидающих данного условия. Поток не становится работоспособным, когда становится доступной блокировка. Вместо этого он остается деактивизированным, пока другой поток не вызовет метод `signalAll()` на том же условии.

Когда другой поток переведет деньги, он должен будет вызвать

```
sufficientFunds.signalAll();
```

Этот вызов активизирует все потоки, ожидающие данного условия. Когда потоки удаляются из набора ожидания, они опять становятся выполнимыми и в конечном итоге планировщик активизирует их снова. В этот момент они попытаются повторно захватить объект блокировки. Как только он окажется доступным, один из этих потоков захватит блокировку и продолжит работу с того места, где он остановился, получив управление после вызова `await()`.

В этот момент поток снова должен проверить условие. Нет никаких гарантий, что условие теперь выполнится — метод `signalAll()` просто сигнализирует ожидающим потокам о том, что условие теперь *может быть* удовлетворено, и что его стоит проверить заново.



На заметку! Вообще вызов метода `await()` должен быть помещен в цикл следующей формы:

```
while (!(можно продолжать))
  условие.await();
```

Чрезвычайно важно, чтобы *какой-то* другой поток вызвал, в конце концов, метод `signallAll()`. Когда поток вызывает `await()`, он не имеет возможности повторно активизировать самого себя. Здесь он полностью полагается на другие потоки. Если ни один из них не позаботится о повторной активизации ожидающего потока, его работа никогда не будет возобновлена. Это может привести к неприятной ситуации *взаимной блокировки* (deadlock). Если все прочие потоки будут блокированы, и последний активный поток вызовет `await()`, не разблокируя ни одного другого, он также будет заблокирован. И тогда не останется ни одного потока, который мог бы разблокировать других, а потому программа повиснет.

Когда следует вызывать `signalAll()`? Эмпирическое правило заключается в том, что вызывать этот метод нужно при каждом таком изменении состояния объекта, которое может быть выгодно ожидающим потокам. Например, всякий раз, когда изме-

няются балансы счетов, ожидающим потокам следует давать другой шанс для проверки балансов. В нашем примере мы вызываем `signalAll()`, когда завершаем перевод денег.

```
public void transfer(int from, int to, int amount)
{
  bankLock.lock();
  try
  {
  while (accounts[from] < amount)
    sufficientFunds.await();
    // перевод денег
    . . .
    sufficientFunds.signalAll();
  }
  finally
  {
    bankLock.unlock();
  }
}
```

Обратите внимание, что вызов `signalAll()` не влечет за собой немедленной активизации ожидающего потока. Он лишь разблокирует ожидающие потоки, чтобы они могли состязаться за объект блокировки после того, как текущий поток выйдет из синхронизированного метода.

Другой метод — `signal()` — разблокирует только один поток из набора ожидающих, выбирая его случайным образом. Это более эффективно, чем разблокировать все потоки, хотя здесь существует определенная опасность. Если случайно выбранный поток обнаружит, что он еще не может выполнять работу, он вновь заблокируется. И если никакой другой поток не вызовет вновь `signal()`, система попадет в состояние взаимной блокировки.



Внимание! Поток может вызывать только `await()`, `signalAll()` или `signal()` по условию, когда он владеет блокировкой этого условия.

Если вы запустите пример программы из листинга 14.8, то заметите, что теперь все идет хорошо. Общий баланс в $100 000 сохраняется неизменным. Ни один счет не имеет отрицательного баланса (Опять-таки, вам придется нажать <Ctrl+C>, чтобы прекратить выполнение программы.) Вы можете также заметить, что программа работает немного медленнее — это цена, которую приходится платить за добавление "бухгалтерии", связанной с механизмом синхронизации.

На практике использование условий может быть довольно непростым. Прежде чем вы попытаетесь реализовать собственные объекты условий, стоит рассмотреть применение конструкций, описанных в разделе "Синхронизаторы".

Листинг 14.8. Содержимое файла `Bank.java`

```
1. import java.util.concurrent.locks.*;
2. /**
3.  * Банк с множеством счетов, использующий блокировки для сериализации доступа.
4.  * @version 1.30 2004-08-01
5.  * @author Cay Horstmann
6.  */
```

```
7. public class Bank
8. {
9. /**
10. * Конструирует банк.
11. * @param n Количество счетов
12. * @param initialBalance Начальный баланс каждого счета
13. */
14.public Bank(int n, double initialBalance)
15.{
16.  accounts = new double[n];
17.  for (int i = 0; i < accounts.length; i++)
18.     accounts[i] = initialBalance;
19.  bankLock = new ReentrantLock();
20.  sufficientFunds = bankLock.newCondition();
21.}
22./**
23. * Переводит деньги с одного счета на другой.
24. * @param from Счет, с которого выполняется перевод
25. * @param to Счет, на который выполняется перевод
26. * @param amount Сумма перевода
27. */
28.public void transfer(int from, int to, double amount) throws
29.InterruptedException
30.{
31.  bankLock.lock();
32.  try
33.  {
34.     while (accounts[from] < amount)
35.       sufficientFunds.await();
36.     System.out.print(Thread.currentThread());
37.     accounts[from] -= amount;
38.     System.out.printf(" %10.2f с %d на %d", amount, from, to);
39.     accounts[to] += amount;
40.     System.out.printf(" Общий баланс: %10.2f%n", getTotalBalance());
41.     sufficientFunds.signalAll();
42.  }
43.  finally
44.  {
45.     bankLock.unlock();
46.  }
47.}
48./**
49. * Получает сумму всех счетов.
50. * @return Общий баланс
51. */
52.public double getTotalBalance()
53.{
54.  bankLock.lock();
55.  try
56.  {
57.     double sum = 0;
58.     for (double a : accounts)
59.       sum += a;
60.     return sum;
61.  }
```

```
62.  finally
63.  {
64.     bankLock.unlock();
65.  }
66. }
67. /**
68.  * Получает количество счетов в банке.
69.  * @return Количество счетов
70.  */
71. public int size()
72. {
73.  return accounts.length;
74. }
75. private final double[] accounts;
76. private Lock bankLock;
77. private Condition sufficientFunds;
78.}
```

API **java.lang.concurrent.locks.Lock 5.0**

- `Condition newCondition()`

Возвращает объект условия, ассоциированный с данной блокировкой.

API **java.lang.concurrent.locks.Condition 5.0**

- `void await()`

  Помещает данный поток в набор ожидания для этого условия.

- `void signalAll()`

  Разблокирует все потоки в наборе ожидания данного условия.

- `void signal()`

  Разблокирует один случайно выбранный поток в наборе ожидания данного условия.

Ключевое слово synchronized

В предшествующих разделах вы видели, как надо использовать объекты `Lock` и `Condition`. Прежде чем двинуться дальше, подведем итоги, перечислив ключевые моменты, касающиеся блокировок и условий.

- Блокировка защищает сегмент кода, позволяя только одному потоку в единицу времени выполнять этот код.
- Блокировка управляет потоками, которые пытаются войти в защищенный сегмент кода.
- Каждый объект условия управляет потоками, вошедшими в защищенный сегмент кода, но которые пока не в состоянии выполнять работу.

Интерфейсы `Lock` и `Condition` были добавлены в Java SE 5.0, чтобы предоставить программистам высокую степень контроля блокировок. Однако в большинстве ситуаций вам не понадобится такой контроль, и вы можете использовать механизм, построенный на средствах языка Java. Еще со времен версии 1.0 *каждый объект* в Java обладает внутренней блокировкой. Если метод объявлен с ключевым словом `synchronized`,

то блокировка объекта защищает весь этот метод. То есть для того, чтобы вызвать этот метод, поток должен захватить внутреннюю блокировку объекта.

Другими словами, следующее

```
public synchronized void method()
{
   тело метода
}
```

является эквивалентом

```
public void method()
{
  this.внутренняяБлокировка.lock();
  try
  {
    тело метода
  }
  finally { this.внутренняяБлокировка.unlock(); }
}
```

Например, вместо использования явной блокировки, мы можем просто объявить метод `transfer` класса `Bank`, как `synchronized`.

Внутренний объект блокировки имеет единственное ассоциированное с ним условие. Метод `wait()` добавляет поток в набор ожидания, а методы `notifyAll()`/`notify()` разблокируют ожидающие потоки. Другими словами, вызов `wait()` или `notifyAll()` — это эквиваленты следующего:

```
внутреннееУсловие.await();
внутреннееУсловие.signalAll();
```



На заметку! Методы `wait()`, `notifyAll()` и `notify()` являются финальными (`final`) методами класса `Object`. Методы `Condition` должны именоваться `await`, `signalAll` и `signal`, так что они не конфликтуют с этими методами.

Например, вы можете реализовать класс `Bank` следующим образом:

```
class Bank
{
  public synchronized void transfer(int from, int to, int amount)
       throws InterruptedException
  {
    while (accounts[from] < amount)
      wait(); //ожидать единственного условия внутреннего объекта блокировки
    accounts[from] -= amount;
    accounts[to] += amount;
    notifyAll(); // известить все потоки, ожидающие условия
  }
  public synchronized double getTotalBalance() { . . . }
  private double[] accounts;
}
```

Как видите, применение ключевого слова `synchronized` порождает намного более краткий код. Конечно, чтобы понять его, необходимо знать о том, что каждый объект обладает внутренней блокировкой, и что эта блокировка имеет внутреннее условие. Блокировка управляет потоками, которые пытаются войти в метод `synchronized`. Условие управляет потоками, вызвавшими `wait()`.



Совет. Синхронизированные методы относительно просты. Однако начинающие программисты часто испытывают затруднения с условиями. Прежде чем использовать `wait()/notifyAll()`, вы должны рассмотреть применение одной из конструкций, описанных в разделе "Синхронизаторы".

Также допустимо объявлять статические методы синхронизированными. Когда вызывается такой метод, он захватывает внутреннюю блокировку ассоциированного объекта класса. Например, если класс `Bank` имеет статический синхронизированный метод, тогда блокировка объекта `Bank.class` блокируется при ее вызове. В результате ее не может вызвать никакой другой поток и никакой другой синхронизированный статический метод того же класса.

Внутренние блокировки и условия имеют некоторые ограничения.

- Вы не можете прервать поток, который пытается захватить блокировку.
- Вы не можете специфицировать таймаут, пытаясь захватить блокировку.
- Наличие единственного условия на блокировку может быть неэффективным.

Что же следует использовать в вашем коде — объекты `Lock` и `Condition` или синхронизированные методы? Вот наши рекомендации.

- Лучше не использовать ни `Lock/Condition`, ни ключевое слово `synchronized`. Во многих ситуациях вы можете применять один из механизмов пакета `java.util.concurrent`, который сделает всю работу по блокировке за вас. Например, в разделе "Блокирующие очереди" будет показано, как использовать блокирующие очереди для синхронизации потоков, выполняющих общую задачу.
- Если ключевое слово `synchronized` подходит для вашей ситуации, обязательно используйте его. Вы напишете меньше кода, и у вас будет меньше шансов допустить ошибку. В листинге 14.9 показан пример с банком, реализованный на основе синхронизированных методов.
- Используйте `Lock/Condition`, если вы действительно нуждаетесь в дополнительных возможностях этих конструкций.

Листинг 14.9. Содержимое файла `Bank.java`

```
1. import java.util.concurrent.locks.*;
2. /**
3.  * Банк с множеством счетов, использующий блокировки для сериализации доступа.
4.  * @version 1.30 2004-08-01
5.  * @author Cay Horstmann
6.  */
7. public class Bank
8. {
9.  /**
10.  * Конструирует банк.
11.  * @param n Количество счетов
12.  * @param initialBalance Начальный баланс каждого счета
13.  */
14. public Bank(int n, double initialBalance)
15.{
16.  accounts = new double[n];
17.  for (int i = 0; i < accounts.length; i++)
18.     accounts[i] = initialBalance;
19. }
```

```
20. /**
21.  * Переводит деньги с одного счета на другой.
22.  * @param from Счет, с которого выполняется перевод
23.  * @param to Счет, на который выполняется перевод
24.  * @param amount Сумма перевода
25.  */
26. public void transfer(int from, int to, double amount) throws
27. InterruptedException
28. {
29.     while (accounts[from] < amount)
30.       wait();
31.     System.out.print(Thread.currentThread());
32.     accounts[from] -= amount;
33.     System.out.printf(" %10.2f from %d to %d", amount, from, to);
34.     accounts[to] += amount;
35.     System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
36.     sufficientFunds.signalAll();
37.     notufyAll();
38. }
39. /**
40.  * Получает сумму всех счетов.
41.  * @return Общий баланс
42.  */
43. public double getTotalBalance()
44. {
45.   double sum = 0;
46.   for (double a : accounts)
47.     sum += a;
48.   return sum;
49. }
50. /**
51.  * Получает количество счетов в банке.
52.  * @return Количество счетов
53.  */
54.  public int size()
55.  {
56.   return accounts.length;
57.  }
58.  private final double[] accounts;
59. }
```

API java.lang.Object 1.0

- `void notifyAll()`

  Разблокирует потоки, вызвавшие `wait()` на этом объекте. Этот метод может быть вызван только изнутри синхронизированного метода или блока. Метод генерирует исключение `IllegalMonitorStateException`, если поток не является владельцем блокировки объекта.

- `void notify()`

  Разблокирует один случайно выбранный поток среди потоков, вызвавших `wait()` на этом объекте. Этот метод может быть вызван только изнутри синхронизированного метода или блока. Метод генерирует исключение `IllegalMonitorStateException`, если поток не является владельцем блокировки объекта.

- `void wait(long mills)`
- `void wait(long mills, int nanos)`

  Заставляют поток ожидать уведомления в течение указанного периода времени. Эти методы вызываются только из синхронизированного метода или блока. Генерируют исключение `IllegalMonitorStateException`, если текущий поток не является владельцем блокировки объекта.

  Параметры: `mills` Количество миллисекунд.

  `nanos` Количество наносекунд, не более 1 000 000.

Синхронизированные блоки

Как мы уже упомянули, каждый объект Java имеет собственную блокировку. Поток может захватить эту блокировку, вызвав синхронизированный метод. Но есть и второй механизм захвата блокировки — входом в *синхронизированный блок*. Когда поток входит в блок в форме

```
synchronized (obj) // синтаксис синхронизированного блока
{
   критичный сегмент
}
```

то он захватывает блокировку объекта `obj`.

Иногда вы встретите специальные блокировки, вроде следующей:

```
public class Bank
{
 public void transfer(int from, int to, int amount)
 {
  synchronized (lock) // специальная блокировка
  {
    accounts[from] -= amount;
    accounts[to] += amount;
  }
  System.out.println(. . .);
 }
 . . .
 private double[] accounts;
 private Object lock = new Object();
}
```

Здесь объект `lock` создается только для использования блокировки, которая имеется в каждом объекте Java.

Иногда программисты используют блокировку объекта для реализации дополнительных атомарных операций — такая практика получила название *блокировки клиентской стороны*. Рассмотрим, к примеру, класс `Vector` — список, чьи методы синхронизированы. Теперь предположим, что мы поместили наши банковские балансы в `Vector<Double>`. Ниже показана "родная" реализация метода `transfer()`.

```
public void transfer(Vector<Double> accounts, int from, int to, int amount)
  // ОШИБКА
{
  accounts.set(from, accounts.get(from) - amount);
  accounts.set(to, accounts.get(to) + amount);
  System.out.println(. . .);
}
```

Методы get и set класса Vector синхронизированы, но это нам не поможет. Вполне возможно, что поток будет приостановлен в методе transfer() после завершения первого вызова get(). Другой поток может затем поместить другое значение в ту же позицию. Однако мы можем захватить блокировку:

```
public void transfer(Vector<Double> accounts, int from, int to, int amount)
{
  synchronized (accounts)
  {
    accounts.set(from, accounts.get(from) - amount);
    accounts.set(to, accounts.get(to) + amount);
  }
  System.out.println(. . .);
}
```

Такой подход работает, но он целиком зависит от того факта, что класс Vector использует внутреннюю блокировку для всех его изменяющих методов. Однако так ли это на самом деле? Документация по классу Vector этого не обещает. Вы должны тщательно изучать исходный код и надеяться, что будущие версии не будут включать не синхронизированных изменяющих данные методов. Как видим, блокировка клиентской стороны — вещь очень хрупкая и потому обычно не рекомендуемая к применению.

Концепция монитора

Блокировки и условия — мощный инструмент синхронизации потоков, но они не слишком объектно-ориентированы. В течение многих лет исследователи ищут способы обеспечения безопасности многопоточности, дабы избавить программистов от необходимости думать о явных блокировках. Одно из наиболее успешных решений — концепция *монитора*, которая была впервые предложена Пером Бринчем Хансеном (Per Brinch Hansen) и Тони Хоаром (Tony Hoare) в 70-х годах. В терминологии Java монитор обладает перечисленными ниже свойствами.

- Монитор — это класс, имеющий только приватные поля.
- Каждый объект этого класса имеет ассоциированную с ним блокировку.
- Все методы блокируются этой блокировкой. Другими словами, если клиент вызывает obj.method(), то при этом автоматически запирается блокировка объекта obj в начале метода и освобождается по его завершении. Поскольку все поля приватные, такой подход гарантирует, что ни один поток не сможет обратиться к ним, пока ими манипулирует какой-то другой поток.
- Блокировка может иметь любое количество ассоциированных условий.

Ранние версии мониторов имели единственное условие, с достаточно элегантным синтаксисом. Вы можете просто вызвать await accounts[from] >= balance, не используя явно условную переменную. Однако исследования показали, что неразборчивая повторная проверка условий может оказаться неэффективной. Проблема была решена применением явных переменных условий, каждая из которых управляет отдельным набором потоков.

Проектировщики Java вольно адаптировали концепцию монитора. *Каждый объект* в Java обладает внутренней блокировкой и внутренним условием. Если метод объявлен с ключевым словом synchronized, он работает как метод монитора. Переменная условия доступна через вызовы wait/notifyAll/notify.

Однако объекты Java отличаются от мониторов в трех важных отношениях, ослабляющих безопасность потоков.

- Поля не обязательно должны быть `private`.
- Методы не обязаны быть `synchronized`.
- Внутренняя блокировка доступна клиентам.

Это — явное пренебрежение требованиями безопасности, изложенными Пером Бринчем Хансеном. В уничижительном обозрении, посвященном примитивам многозадачности Java, он пишет: "Для меня является непостижимым тот факт, что небезопасный параллелизм столь серьезно принят сообществом программистов, и это спустя четверть века после изобретения мониторов и языка Concurrent Pascal. Этому нет оправданий." (Java's Insecure Parallelism, ACM SIGPLAN Notices 34:38–45, April 1999).

Поля `volatile`

Иногда плата за синхронизацию кажется непомерной, когда нужно просто читать или писать одно или два поля экземпляра. В конце концов, что страшного может произойти? К сожалению, современные процессоры и компиляторы, оставляют широкие возможности для появления ошибок.

- Компьютеры с несколькими процессорами могут временно удерживать значения из памяти в регистрах или локальных кэшах. Вследствие этого потоки, выполняющиеся на разных процессорах, могут видеть разные значения в одной и той же области памяти!
- Компиляторы могут изменять порядок инструкций для достижения максимальной отдачи. Они не меняют этот порядок так, что изменился бы смысл кода, но используют предположения о том, что значения в памяти меняются только явными инструкциями кода. Однако значение в памяти может быть изменено другим потоком!

Если вы используете блокировки для защиты кода, который может выполняться множеством потоков, то не столкнетесь с этими проблемами. Компиляторы обязаны уважать блокировки, при необходимости очищая локальные кэши и не изменяя порядок инструкций. Подробности объясняются в документе *Java Memory Model and Thread Specification*, разработанном JSR 133 (`http://www.jsp.org/en/jsr/detail?id=133`). Большая часть спецификации очень сложна и полна технических деталей, но этот документ также содержит ряд наглядных примеров. Более доступный обзор темы, принадлежащий перу Брайана Гоетца (Brian Goetz), доступен по адресу:

`http://www-106.ibm.com/developerworks/java/library/j-jtp02244.html`



На заметку! Брайан Гоетц предложил следующий "девиз синхронизации": если вы пишете переменную, которая затем может быть прочитана другим потоком, или же читаете переменную, которая была записана другим потоком, то обязаны использовать синхронизацию.

Ключевое слово `volatile` представляет неблокирующий механизм синхронизированного доступа к полю экземпляра. Если вы объявляете поле как `volatile`, то компилятор и виртуальная машина принимают во внимание тот факт, что поле может быть обновлено параллельно в другом потоке.

Например, представим объект, у которого есть boolean-флаг done, устанавливаемый в одном потоке и опрашиваемый в другом. Как мы уже говорили, вы можете использовать блокировку:

```
public synchronized boolean isDone() { return done; }
public synchronized void setDone() { done = true; }
private boolean done;
```

Возможно, применять внутреннюю блокировку объекта — не самая лучшая идея. Методы isDone и setDone могут блокироваться, если другой поток заблокировал объект. Если так случилось, можно использовать отдельный Lock только для этой переменной. Но это влечет за собой множество хлопот.

В данном случае имеет смысл объявить поле как volatile:

```
public boolean isDone() { return done; }
public void setDone() { done = true; }
private volatile boolean done;
```



Внимание! Переменные volatile не гарантируют никакой атомарности. Например, метод

```
public void flipDone() { done = !done; } // не атомарный!
```

не гарантирует смены значения поля на противоположное.

В очень простом случае существует третья возможность — использовать AtomicBoolean. Этот класс имеет методы get и set, атомарность которых гарантируется (как если бы они были синхронизированными). Реализация использует эффективные инструкции машинного уровня, гарантирующие атомарность без использования блокировок. В пакете java.util.concurrent.atomic предусмотрено множество классов атомарных целых чисел, чисел с плавающей точкой, массивов и т.п. Эти классы предназначены не для прикладных, а для системных программистов, которые разрабатывают параллельно работающие утилиты.

Подведем итог: безопасный параллельный доступ к полю обеспечивает выполнение следующих трех условий.

- Поле объявлено как final, и обращение к нему осуществляется после завершения работы конструктора.
- Каждое обращение к полю защищено обычной блокировкой.
- Поле объявлено как volatile.



На заметку! До появления Java SE 5.0 семантика слова volatile была довольно-таки либеральной. Проектировщики языка пытались дать его реализаторам свободу действий в том, что касалось оптимизации производительности кода, использующего поля volatile. Однако старая спецификация была настолько сложна, что реализаторы не всегда следовали ей, и кроме того, она допускала путаницу и нежелательное поведение вроде порождения неизменяемых объектов, которые на самом деле таковыми не являлись.

Взаимные блокировки

Блокировки и условия не могут решить всех проблем, которые возникают в многопоточных приложениях. Рассмотрим следующую ситуацию:

```
Счет 1: $200
Счет 2: $300
Поток 1: Перевод $300 со Счета 1 на Счет 2
Поток 2: Перевод $400 со Счета 2 на Счет 1
```

Как следует из рис. 14.6, потоки 1 и 2 четко блокируются. Ни один из них не работает, поскольку балансы счетов 1 и 2 недостаточны для выполнения транзакции.

Может ли так случиться, что все потоки будут блокированы, поскольку каждый из низ будет ожидать прихода денег? Такая ситуация называется *взаимной блокировкой* (deadlock).

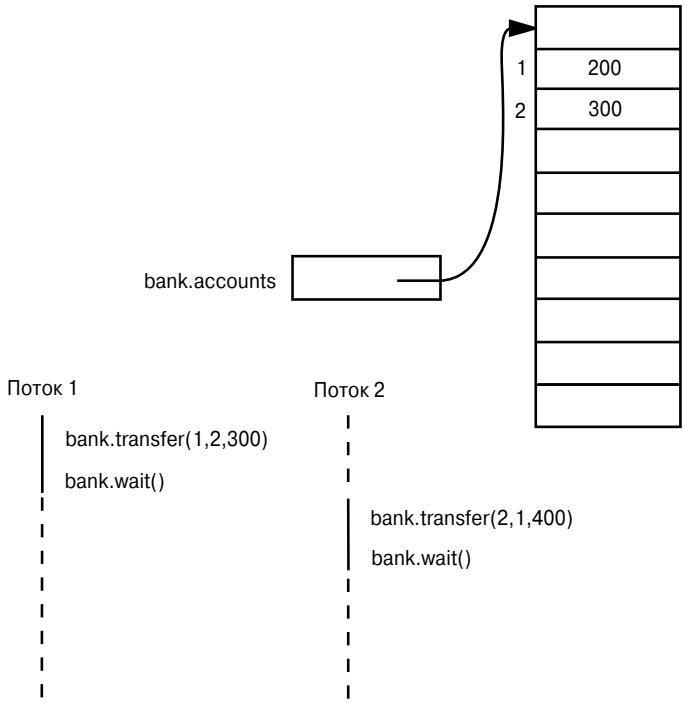


Рис. 14.6. Ситуация взаимной блокировки

В нашей программе взаимная блокировка не может произойти по простой причине. Сумма каждого перевода не превышает $1000. Поскольку есть всего 100 счетов с общим балансом $100 000, то, как минимум, один из счетов в любой момент времени должен иметь более $1000. Поэтому поток, переводящий деньги с этого счета, всегда должен отработать.

Но если вы измените метод `run()`, исключив лимит в $1000 на транзакцию, то взаимная блокировка произойдет очень скоро. Попробуйте и убедитесь. Установите `NACCOUNTS` равным 10. Сконструируйте объект `Runnable` со значением `max` равным `2*INITIAL_BALANCE` и запустите программу. Она будет работать довольно-таки долго, после чего зависнет.



Совет. Когда программа зависнет, нажмите <Ctrl+\>. При этом вы получите дамп потока с перечислением всех потоков. Каждый поток имеет трассировку стека, в которой видно, где именно произошла блокировка. В качестве альтернативы запустите `jconsole`, как описано в главе 11, и просмотрите панель Threads (Потоки), показанную на рис. 14.7.

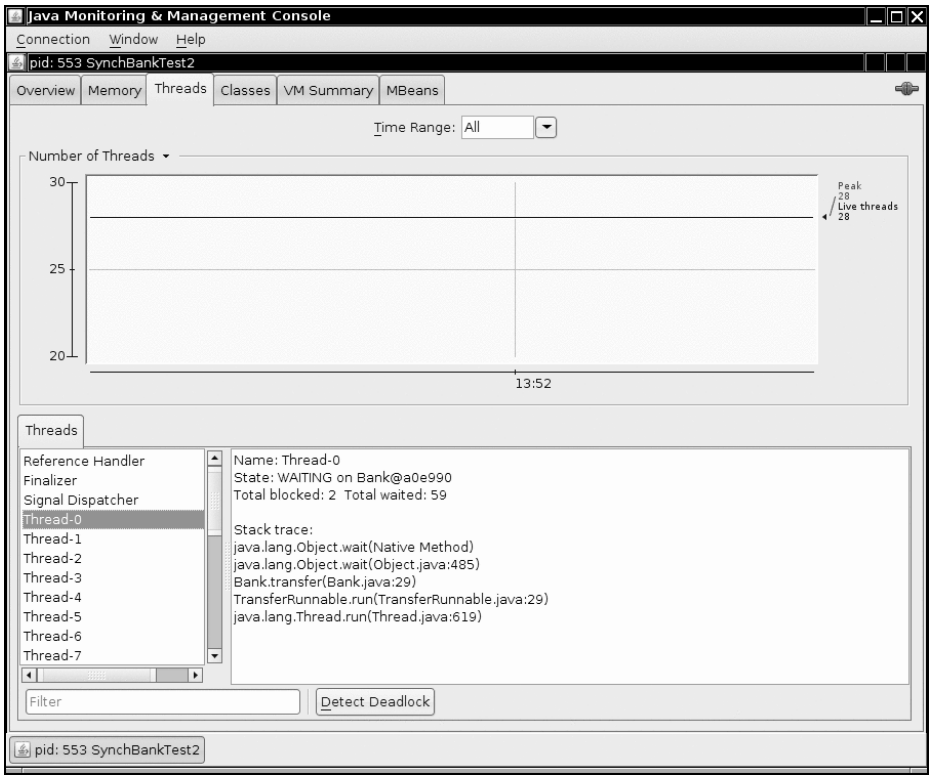


Рис. 14.7. Панель Threads в `jconsole`

Другой способ создать взаимную блокировку — сделать `i`-й поток ответственным за помещение денег на `i`-й счет вместо снятия их с `i`-го счета. В этом случае есть шанс, что все потоки накинутся на один и тот же счет, и каждый будет пытаться снять с него деньги. Попробуйте! В программе `SynchBankTest` обратитесь к методу `run()` класса `TransferRunnable`. В вызове `transfer` поменяйте местами `fromAccount` и `toAccount`. После запуска программы вы почти немедленно получите взаимную блокировку.

А вот другая ситуация, в которой может легко случиться взаимная блокировка: замените метод `signalAll` на `signall` в программе `SynchBankTest`. Вы обнаружите, что, в конце концов, программа зависнет. (Опять-таки, лучше установить `NACCOUNTS` в 10 для скорейшего получения эффекта.) В отличие от метода `signalAll()`, который извещает все потоки, ожидающие добавления денег на счет, метод `signal()` разблокирует только один поток. Если этот поток не может выполнять работу, то все потоки могут оказаться заблокированы. Рассмотрим следующий простой сценарий создания взаимной блокировки.

```
Счет 1: $1,990
Все прочие счета: $990 на каждом
Поток 1: Переводит $995 со счета 1 на счет 2
Все прочие потоки: Перевод $995 с их счета на другой
```

Ясно, что все потоки кроме 1 блокированы, поскольку на их счетах недостаточно денег.

Поток 1 выполняет перевод, после чего мы имеем следующую ситуацию:

```
Счет 1: $995
Счет 2: $1,985
Все прочие счета: $990 на каждом
```

Затем поток 1 вызывает `signal()`. Метод `signal()` случайным образом выбирает поток для разблокировки. Предположим, он выбирает поток 3. Этот поток просыпается, обнаруживает, что на его счете денег недостаточно, и вызывает опять `await()`. Но поток 1 все еще работает. Генерируется новая случайная транзакция, скажем такая:

```
Поток 1: Перевод $997 со счета 1 на счет 2
```

Теперь поток 1 также вызывает `await()` и *все* потоки блокируются. Система входит во взаимную блокировку. Виновником этого является вызов `signal()`. Он разблокирует только один поток, и таким потоком может оказаться совсем не тот, который позволит двигаться дальше. (В нашем сценарии поток 2 должен выполнить перевод денег со счета 2.)

К сожалению, в Java нет ничего, что позволило бы избегать или снимать взаимные блокировки. Вы должны так проектировать свою программу, чтобы исключить ситуации взаимных блокировок.

Тестирование блокировок и таймауты

Поток блокируется навечно, когда он вызывает метод `lock()` для захвата блокировки, которой владеет другой поток. Поэтому следует быть осторожным, пытаясь захватить блокировку. Метод `tryLock()` пытается захватить блокировку и возвращает `true`, если ему это удается. В противном случае он немедленно возвращает `false`, и тогда ваш поток может переключиться на какую-нибудь другую полезную работу.

```
if (myLock.tryLock())
  // теперь поток владеет блокировкой
  try { . . . }
  finally { myLock.unlock(); }
else
  // делать что-то другое
```

Вы можете вызвать `tryLock` с параметром — таймаутом, как показано ниже:

```
if (myLock.tryLock(100, TimeUnit.MILLISECONDS)) . . .
```

`TimeUnit` — значение перечисления из ряда `SECONDS`, `MILLISECONDS`, `MICROSECONDS` и `NANOSECONDS`.

Метод `lock()` не может быть прерван. Если поток прерван во время ожидания захвата блокировки, прерванный поток продолжает оставаться блокированным до тех пор, пока доступна блокировка. Если случится взаимная блокировка, то метод `lock()` никогда не завершится.

Однако, если вы вызовете `tryLock()` с таймаутом, то при попытке прерывания потока, находящегося в состоянии ожидания, будет сгенерировано исключение `InterruptedException`. Это, несомненно, полезное средство, поскольку позволяет программе прерывать взаимные блокировки.

Также вы можете вызвать метод `lockInterruptibly()`. Он имеет то же назначение, что и `tryLock()`, но с бесконечным таймаутом.

Когда вы ожидаете условия, то также можете применить таймаут:

```
myCondition.await(100, TimeUnit.MILLISECONDS))
```

Метод `await()` возвращает управление, если другой поток активизирует данный поток вызовом `signalAll()` или `signal()`, или по истечении таймаута, или же в связи с тем, что поток был прерван.

Метод `await()` генерирует исключение `InterruptedException`, если ожидающий поток прерывается. В том случае (наверное, маловероятном), если вы предпочитаете продолжить ожидание, используйте вместо него метод `awaitUninterruptibly()`.

API java.util.concurrent.locks.Lock 5.0

- `boolean tryLock()`

  Пытается захватить блокировку без блокирования; возвращает `true` в случае удачи. Этот метод захватывает блокировку, если она доступна, даже в случае применения "равноправной" политики блокирования и ожидания других потоков.

- `boolean tryLock(long time, TimeUnit unit)`

  Пытается захватить блокировку, блокируя не дольше заданного времени; возвращает `true` в случае успеха.

- `void lockInterruptibly()`

  Захватывает блокировку, устанавливая бесконечную блокировку. Если поток прерывается, генерирует исключение `InterruptedException`.

API java.util.concurrent.locks.Condition 5.0

- `boolean await(long time, TimeUnit unit)`

  Входит в набор ожидания данного условия, блокируя до тех пор, пока поток не будет удален из набора ожидания либо пока не истечет указанное время. Возвращает `false`, если метод завершился по истечении таймаута, и `true` — в противном случае.

- `boolean awaitUninterruptibly()`

  Входит в набор ожидания данного условия, блокируя до тех пор, пока поток не будет удален из набора ожидания. Если поток прерван, метод не генерирует исключения `InterruptedException`.

Блокировки чтения/записи

Пакет `java.util.concurrent.locks` определяет два класса блокировок — `ReentrantLock`, о котором мы уже говорили, и `ReentrantReadWriteLock`. Последний удобен, когда есть много потоков, читающих структуру данных, и поменьше потоков — модифицирующих ее. В такой ситуации имеет смысл разрешить разделенный доступ читателям. Конечно, поток-писатель должен по-прежнему иметь исключительный доступ.

Ниже описаны шаги, необходимые для использования блокировок чтения/записи.

1. Сконструировать объект `ReentrantReadWriteLock`:

```
private ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
```

2. Извлечь блокировки чтения и записи:

```
private Lock readLock = rwl.readLock();
private Lock writeLock = rwl.writeLock();
```

3. Использовать блокировку чтения для всех читателей:

```
public double getTotalBalance()
{
  readLock.lock();
  try { . . . }
  finally { readLock.unlock(); }
}
```

4. Использовать блокировку записи для всех, кто изменяет данные:

```
public void transfer(. . .)
{
  writeLock.lock();
  try { . . . }
  finally { writeLock.unlock(); }
}
```

API **java.util.concurrent.locks.ReentrantReadWriteLock 5.0**

- `Lock readLock()`

  Получает блокировку чтения, которая может быть захвачена многими читателями, но исключая всех писателей.

- `Lock writeLock()`

  Получает блокировку записи, исключающую доступ всех читателей и писателей.

Почему методы `stop()` и `suspend()` не рекомендованы к применению

В начальном выпуске Java был определен метод `stop()`, который просто останавливал поток, и метод `suspend()`, который блокировал поток до тех пор, пока другой поток не вызовет `resume()`. Методы `stop()` и `suspend()` имеют нечто общее: оба пытаются контролировать поведение данного потока без кооперации потоков.

Оба эти метода были объявлены как не рекомендованные, начиная с Java SE 1.2. Метод `stop()` по сути своей небезопасен, а что касается метода `suspend()`, то опыт показал, что он часто приводит к взаимным блокировкам. В этом разделе вы увидите, почему эти методы проблематичны, и что нужно делать, дабы избежать проблем.

Давайте сначала обратимся к методу `stop()`. Этот метод прекращает любые незавершенные методы, включая `run()`. Когда поток останавливается, он немедленно снимает блокировки со всех объектов, которые он блокировал. Это может привести к тому, что объекты останутся в несогласованном состоянии. Например, предположим, что `TransferThread()` остановлен на полпути перевода денег с одного счета на другой — после снятия с одного счета, но перед добавлением их на другой. В этом случае наш объект-банк *поврежден*. Поскольку блокировка снята, это повреждение будет видимо другим потокам, которые не были остановлены.

Когда поток желает остановить другой поток, он не имеет возможности узнать, когда вызов метода `stop()` безопасен, а когда он может привести к повреждению объектов. Поэтому данный метод был объявлен не рекомендованным к применению (deprecated). Когда вы хотите остановить поток, то должны его прервать. Прерванный поток может затем остановиться сам, когда это можно будет сделать безопасно.



На заметку! Некоторые авторы требовали объявления метода `stop()` нежелательным, поскольку он мог приводить к появлению объектов, заблокированных навсегда остановленным потоком. Однако подобное требование некорректно. Остановленный поток выходит из всех синхронизированных методов, который он вызывал, — посредством генерации исключения `ThreadDeath`. Как следствие, поток освобождает все внутренние объекты блокировок, которые он удерживает.

Теперь посмотрим, что не так с методом `suspend()`. В отличие от `stop()`, `suspend()` не повреждает объекты. Однако если вы приостановите поток, владеющий блокировкой, то эта блокировка останется недоступной до тех пор, пока поток не будет возобновлен. Если поток, вызвавший метод `suspend()`, попытается захватить ту же блокировку, программа войдет в состояние взаимной блокировки. Приостановленный поток ожидает, когда его возобновят, а поток, который приостановил его, ожидает освобождения блокировки.

Такая ситуация часто возникает в графическом интерфейсе пользователя. Предположим, что у вас есть графический эмулятор банка. Кнопка с меткой Pause (Пауза) приостанавливает потоки переводов, а кнопка с меткой Resume (Возобновление) возобновляет их.

```
pauseButton.addActionListener(new
ActionListener()
{
  public void actionPerformed(ActionEvent event)
  {
     for (int i = 0; i < threads.length; i++)
       threads[i].suspend(); // Не делайте так
  }
});
resumeButton.addActionListener(. . .); // вызов возобновления всех потоков,
                                       // выполняющих переводы
```

Представим, что метод `paintComponent` рисует график каждого счета, вызывая метод `getBalances()` для получения массива балансов.

Как вы увидите в разделе “Потоки и Swing”, действия обеих кнопок и перерисовка происходят в одном и том же потоке — *потоке диспетчера событий*. Рассмотрим следующий сценарий.

1. Один из потоков, выполняющих переводы, захватывает блокировку объекта `bank`.
2. Пользователь щелкает кнопку Pause.
3. Все потоки переводов приостанавливаются; один из них продолжает удерживать блокировку на объекте `bank`.
4. По той же причине график счета должен быть перерисован.
5. Метод `paintComponent` вызывает метод `getBalance()`.
6. Этот метод пытается захватить блокировку объекта `bank`.

Все, программа зависла.

Поток диспетчера событий не может работать, потому что блокировкой владеет один из приостановленных потоков. Поэтому пользователь не может щелкнуть на кнопке Resume, и поток не может возобновить работу.

Если вы хотите безопасно приостановить поток, введите переменную `suspendRequested` и проверяйте ее в безопасном месте вашего метода `run()` — там,

где ваш поток не блокирует объекты, необходимые другим потокам. Когда ваш поток обнаружит, что переменная suspendRequested установлена, он должен будет подождать до тех пор, пока она станет вновь доступной.

Следующий каркас кода реализует такой дизайн.

```
public void run()
{
  while (. . .)
  {
    . . .
    if (suspendRequested)
    {
       suspendLock.lock();
       try { while (suspendRequested) suspendCondition.await(); }
       finally { suspendLock.unlock(); }
    }
  }
}
public void requestSuspend() { suspendRequested = true; }
public void requestResume()
{
  suspendRequested = false;
  suspendLock.lock();
  try { suspendCondition.signalAll(); }
  finally { suspendLock.unlock(); }
}
private volatile boolean suspendRequested = false;
private Lock suspendLock = new ReentrantLock();
private Condition suspendCondition = suspendLock.newCondition();
```

Блокирующие очереди

Теперь вы знакомы со всеми низкоуровневыми строительными блоками, формирующими основы параллельного программирования в Java. Однако в практическом программировании вы предпочтете держаться насколько возможно, дальше от низкоуровневых конструкций. Гораздо проще и безопаснее работать со структурами более высокого уровня, реализованными экспертами в области параллелизма.

Многие проблемы, связанные с потоками, можно элегантно и безопасно сформулировать, применив одну или более очередей. Поток-поставщик вставляет элементы в очередь, а потоки-потребители извлекают их. Очередь позволяет безопасно передавать данные из одного потока в другой. Например, рассмотрим нашу программу банковских переводов. Вместо того чтобы обращаться к объекту банка напрямую, потоки, выполняющие переводы, вставляют объект-инструкцию перевода в очередь. Другой поток удаляет инструкцию из очереди и осуществляет сам перевод. Только этот поток имеет доступ к внутренностям объекта-банка. Никакая синхронизация не требуется. (Конечно, разработчики безопасных к потокам классов очередей должны позаботиться о блокировках и условиях, но это их проблема, а не ваша.)

Блокирующая очередь заставляет поток блокироваться, когда вы пытаетесь добавить элемент в момент переполнения очереди или когда пытаетесь удалить элемент из пустой очереди. Блокирующие очереди — удобный инструмент для координации работы множества потоков. Рабочие потоки могут периодически помещать промежуточные результаты в блокирующую очередь. Другие рабочие потоки удаляют промежуточные результаты и модифицируют их далее.

Очередь автоматически балансирует нагрузку. Если первая группа потоков работает медленнее, чем вторая, то вторая блокируется, ожидая результатов. Если же первая группа работает быстрее, она притормаживается, пока вторая не догонит. В табл. 14.1 перечислены методы блокирующих очередей.

Таблица 14.1. Методы блокирующих очередей

Метод	Нормальное действие	Действие в особых случаях
`add`	Добавляет элемент.	Генерирует исключение `IllegalStateException`, если очередь полна.
`element`	Возвращает элемент, находящийся в голове очереди.	Генерирует исключение `NoSuchElementException`, если очередь пуста.
`offer`	Добавляет элемент и возвращает `true`.	Возвращает `false`, если очередь полна.
`peek`	Возвращает элемент, находящийся в голове очереди.	Возвращает `null`, если очередь пуста.
`poll`	Возвращает элемент, находящийся в голове очереди, удаляя его из очереди.	Возвращает `null`, если очередь пуста.
`put`	Добавляет элемент.	Блокирует, если очередь полна.
`remove`	Возвращает элемент, находящийся в голове очереди, удаляя его из очереди.	Генерирует исключение `NoSuchElementException`, если очередь пуста.
`take`	Возвращает элемент, находящийся в голове очереди, удаляя его из очереди.	Блокирует, если очередь пуста.

Методы блокирующих очередей разделяются на три категории, в зависимости от поведения, когда очередь полна или пуста. Если вы используете очередь, как инструмент управления потоками, то вам понадобятся методы `put()` и `take()`. Операции `add`, `remove` и `element` генерируют исключение, когда вы пытаетесь добавить элемент в полную очередь либо получить головной элемент пустой очереди. Конечно, в многопоточной программе очередь может наполниться или опустеть в любой момент, так что вы можете пожелать использовать вместо них методы `offer`, `poll` и `peek`. Эти методы просто возвращают признак сбоя вместо исключения, если не могут выполнить свою работу.



На заметку! Методы `poll()` и `peek()` возвращают `null` для индикации неудачи. Поэтому недопустимо вставлять значения `null` в такие очереди.

Существуют варианты методов `offer()` и `poll()` с таймаутом. Например, следующий вызов

```
boolean success = q.offer(x, 100, TimeUnit.MILLISECONDS);
```

в течение 100 миллисекунд пытается вставить элемент в хвост очереди. Если ему это удается, он возвращает `true`; в противном случае, когда таймаут истекает, возвращается `false`.

Аналогично, следующий вызов

```
Object head = q.poll(100, TimeUnit.MILLISECONDS)
```

в течение 100 миллисекунд пытается удалить элемент из головы очереди. Если получается, он возвращает `true`, иначе — `false`, когда истекает таймаут.

Метод `put()` блокирует, когда очередь полна, а метод `take()` выполняет блокирование, когда очередь пуста. Существуют также эквиваленты `offer()` и `put()` без таймаутов.

Пакет `java.util.concurrent` предлагает несколько вариантов блокирующих очередей. По умолчанию `LinkedBlockingQueue` не имеет верхнего ограничения емкости, но такая верхняя граница емкости может быть указана. `LinkedBlockingDeque` — версия очереди с двумя концами. `ArrayBlockingQueue` конструируется с заданной емкостью и необязательным параметром — признаком равноправности. Если этот параметр указан, то предпочтение отдается очередям, дольше всего находящимся в состоянии ожидания. Как всегда, от этого существенно страдает производительность, и потому это средство нужно применять только в том, если ваша проблема действительно требует этого.

`PriorityBlockingQueue` — очередь с приоритетами, а не просто "первый вошел — первый вышел". Элементы удаляются в соответствии с их приоритетами. Эта очередь имеет неограниченную емкость, но при попытке извлечь элемент из пустой очереди происходит блокирование. (Подробнее об очередях с приоритетами читайте в главе 13.)

И, наконец, `DelayQueue` содержит объекты, реализующие интерфейс `Delayed`:

```
interface Delayed extends Comparable<Delayed>
{
   long getDelay(TimeUnit unit);
}
```

Метод `getDelay()` возвращает оставшееся время задержки для объекта. Отрицательное значение указывает, что время задержки истекло. Элементы могут быть удалены из `DelayQueue` только тогда, когда истекает время их задержки. Также потребуется реализовать метод `compareTo()`. `DelayQueue` использует этот метод для сортировки своих элементов.

Программа, представленная в листинге 14.10, демонстрирует использование блокирующей очереди для управления множеством потоков. Программа выполняет поиск среди всех файлов в каталоге и его подкаталогах, распечатывая строки, содержащие указанное ключевое слово.

Поток-поставщик перечисляет все файлы во всех подкаталогах и помещает их в блокирующую очередь. Эта операция выполняется быстро, и очередь быстро заполняется всеми файлами файловой системы, если не установлен предел ее емкости.

Также запускается огромное количество потоков поиска. Каждый такой поток берет файл из очереди, открывает его, печатает все строки, содержащие ключевую строку, затем берет следующий файл. Для прекращения работы приложения, когда никакой дальнейшей работы не требуется, мы используем трюк. Для того чтобы просигнализировать о завершении, перечисляющий поток помещает в очередь объект-заглушку. (Это похоже на муляж чемодана с надписью "последний чемодан" на багажной ленте.) Когда поисковый поток получает этот объект, он возвращает его обратно и завершается.

Отметим, что в данном случае не требуется никакой явной синхронизации потоков. В этом приложении мы используем саму структуру очереди в качестве синхронизирующего механизма.

Листинг 14.10. Содержимое файла `BlockingQueueTest.java`

```
1. import java.io.*;
2. import java.util.*;
3. import java.util.concurrent.*;
4. /**
5.  * @version 1.0 2004-08-01
6.  * @author Cay Horstmann
7.  */
8. public class BlockingQueueTest
9. {
10.  public static void main(String[] args)
11.  {
12.    Scanner in = new Scanner(System.in);
13.    System.out.print("Укажите базовый каталог (например, /usr/local/
14.    jdk1.6.0/src): ");
15.    String directory = in.nextLine();
16.    System.out.print("Введите ключевое слово (например, volatile): ");
17.    String keyword = in.nextLine();
18.    final int FILE_QUEUE_SIZE = 10;
19.    final int SEARCH_THREADS = 100;
20.    BlockingQueue<File> queue = new ArrayBlockingQueue<File>(FILE_QUEUE_SIZE);
21.    FileEnumerationTask enumerator = new FileEnumerationTask(queue,
22.      new File(directory));
23.    new Thread(enumerator).start();
24.    for (int i = 1; i <= SEARCH_THREADS; i++)
25.      new Thread(new SearchTask(queue, keyword)).start();
26.  }
27.}
28./**
29. * Эта задача перебирает все файлы в каталоге и его подкаталогах.
30. */
31.class FileEnumerationTask implements Runnable
32.{
33.  /**
34.   * Конструктор FileEnumerationTask.
35.   * @param queue Блокирующая очередь, в которую добавляются перечисленные файлы
36.   * @param startingDirectory Каталог, с которого начинается перечисление
37.   */
38.  public FileEnumerationTask(BlockingQueue<File> queue, File startingDirectory)
39.  {
40.    this.queue = queue;
41.    this.startingDirectory = startingDirectory;
42.  }
43.  public void run()
44.  {
45.    try
46.    {
47.       enumerate(startingDirectory);
48.       queue.put(DUMMY);
49.    }
50.    catch (InterruptedException e)
51.    {
52.    }
53.  }
```

```
54.  /**
55.   * Рекурсивно перебирает все файлы в данном каталоге и его подкаталогах.
56.   * @param directory Каталог, с которого начинается поиск
57.   */
58.  public void enumerate(File directory) throws InterruptedException
59.  {
60.     File[] files = directory.listFiles();
61.     for (File file : files)
62.     {
63.       if (file.isDirectory()) enumerate(file);
64.       else queue.put(file);
65.     }
66.  }
67.  public static File DUMMY = new File("");
68.  private BlockingQueue<File> queue;
69.  private File startingDirectory;
70.}
71./**
72. * Ищет в файлах заданное ключевое слово.
73. */
74.class SearchTask implements Runnable
75.{
76.  /**
77.   * Конструктор SearchTask.
78.   * @param queue Очередь, из которой извлекаются файлы
79.   * @param keyword Искомое ключевое слово
80.   */
81.  public SearchTask(BlockingQueue<File> queue, String keyword)
82.  {
83.    this.queue = queue;
84.    this.keyword = keyword;
85.  }
86.  public void run()
87.  {
88.  try
89.  {
90.    boolean done = false;
91.    while (!done)
92.    {
93.      File file = queue.take();
94.      if (file == FileEnumerationTask.DUMMY)
95.      {
96.        queue.put(file);
97.        done = true;
98.      }
99.      else search(file);
100.   }
101. }
102. catch (IOException e)
103. {
104.   e.printStackTrace();
105. }
106. catch (InterruptedException e)
107. {
108. }
109.}
```

```
110. /**
111.  * Ищет в файле заданное ключевое слово и печатает все строки, содержащие его.
112.  * @param file Файл для поиска
113.  */
114. public void search(File file) throws IOException
115. {
116.   Scanner in = new Scanner(new FileInputStream(file));
117.   int lineNumber = 0;
118.   while (in.hasNextLine())
119.   {
120.    lineNumber++;
121.    String line = in.nextLine();
122.    if (line.contains(keyword))
123.       System.out.printf("%s:%d:%s%n", file.getPath(), lineNumber, line);
124.   }
125.   in.close();
126.  }
127.  private BlockingQueue<File> queue;
128.  private String keyword;
129. }
```

API java.util.concurrent.ArrayBlockingQueue<E> 5.0

- `ArrayBlockingQueue(int capacity)`
- `ArrayBlockingQueue(int capacity, boolean fair)`

Конструируют блокирующую очередь заданной емкости и установкой равноправности. Очередь реализована в виде циклического массива.

API java.util.concurrent.LinkedBlockingQueue<E> 5.0

API java.util.concurrent.LinkedBlockingDeque<E> 6

- `LinkedBlockingQueue()`
- `LinkedBlockingDeque()`

Конструируют неограниченную блокирующую одностороннюю или двустороннюю очередь, реализованную в виде связного списка.

- `LinkedBlockingQueue(int capacity)`
- `LinkedBlockingDeque(int capacity)`

Конструируют ограниченную блокирующую одностороннюю или двустороннюю очередь, реализованную в виде связного списка.

API java.util.concurrent.DelayQueue<E extends Delayed> 5.0

- `DelayQueue()`

Конструирует неограниченную блокирующую очередь элементов `Delayed`. Из очереди могут быть удалены только элементы, чья задержка истекла.

API java.util.concurrent.Delayed 5.0

- `long getDelay (TimeUnit unit)`

  Получает задержку для данного объекта, измеренную в заданных единицах времени.

API java.util.concurrent.PriorityBlockingQueue<E> 5.0

- `PriorityBlockingQueue()`
- `PriorityBlockingQueue(int initialCapacity)`
- `PriorityBlockingQueue(int initialCapacity, Comparator<? super E> comparator)`

  Конструируют неограниченную блокирующую очередь с приоритетами, реализованную в виде кучи.

  Параметры: `initialCapacity` Начальная емкость очереди с приоритетами. По умолчанию равна `11`.

  `comparator` Компаратор, используемый для сравнения элементов. Если не указан, то элементы должны реализовывать интерфейс `Comparable`.

API java.util.concurrent.BlockingQueue<E> 5.0

- `void put(E element)`

  Добавляет элемент, при необходимости осуществляя блокировку.

- `E take()`

  Удаляет и возвращает элемент из головы очереди, при необходимости осуществляя блокировку.

- `boolean offer(E element, long time, TimeUnit unit)`

  Добавляет заданный элемент и возвращает `true` в случае успеха, при необходимости осуществляя блокировку на время добавления элемента или до истечения таймаута.

- `E poll(long time, TimeUnit unit)`

  Удаляет и возвращает элемент из головы очереди, при необходимости осуществляя блокировку, пока элемент доступен или пока не истечет таймаут. При сбое возвращает `null`.

API java.util.concurrent.BlockingDeque<E> 6

- `void putFirst(E element)`
- `void putLast(E element)`

  Добавляют элемент, при необходимости осуществляя блокировку.

- `E takeFirst()`
- `E takeLast()`

  Удаляют и возвращают элемент из головы очереди, при необходимости осуществляя блокировку.

- `boolean offerFirst(E element, long time, TimeUnit unit)`
- `boolean offerLast(E element, long time, TimeUnit unit)`

  Добавляют заданный элемент и возвращают `true` в случае успеха, при необходимости осуществляя блокировку на время добавления элемента или до истечения таймаута.

- `E pollFirst(long time, TimeUnit unit)`
- `E pollLast(long time, TimeUnit unit)`

  Удаляют и возвращают элемент из головы очереди, при необходимости осуществляя блокировку, пока элемент доступен, или пока не истечет таймаут. В случае сбоя возвращают `null`.

Коллекции, безопасные к потокам

Если множество потоков одновременно модифицируют структуру данных, такую как хеш-таблица, то эту структуру очень легко повредить. (Подробнее о хеш-таблицах читайте в главе 13.) Например, один поток может начать вставлять новый элемент. Предположим, что этот поток будет приостановлен в процессе переназначения ссылок между частями хеш-таблицы. Если другой поток начнет проход по тому же списку, он может проследовать по неправильным ссылкам и привести к разрушению, при этом, возможно, сгенерировав исключение либо войдя в бесконечный цикл.

Вы можете защитить разделяемую структуру данных, применив блокировку, но обычно проще выбрать безопасную к потокам реализацию коллекции. Блокирующие очереди, о которых шла речь в предыдущем разделе, конечно, являются безопасными в отношении потоков коллекциями. В следующих разделах мы рассмотрим другие безопасные к потокам коллекции, представленные в библиотеке Java.

Эффективные карты, наборы и очереди

Пакет `java.util.concurrent` предоставляет эффективные реализации карт (maps), сортированных наборов (sets) и очередей: `ConcurrentHashMap`, `ConcurrentSkipListMap`, `ConcurrentSkipListSet` и `ConcurrentLinkedQueue`.

Эти коллекции используют изощренные алгоритмы, минимизирующие вероятность конкуренции, обеспечивая параллельный доступ к разным частям структуры данных.

В отличие от большинства коллекций, метод `size` не обязательно выполняется за постоянное время. Определение размера одной из перечисленных коллекций обычно требует прохода по всем ее членам.

Коллекции возвращают *слабо согласованные* (weakly consistent) итераторы. Это означает, что итератор может отражать все модификации, внесенные после его создания, а может и не отражать их, но они никогда не возвращают одно и то же значение дважды, и не генерируют исключения `ConcurrentModificationException`.



На заметку! В отличие от этого, итератор коллекции из пакета `java.util` генерирует исключение `ConcurrentModificationException`, когда коллекция была модифицирована после создания этого итератора.

Конкурентная хеш-карта (hash map) может эффективно поддерживать большое количество читателей и фиксированное количество писателей. По умолчанию предполагается существование до 16 *параллельных* потоков-писателей. Может быть намного

больше таких потоков, но если более 16 выполняют запись одновременно, то остальные временно блокируются. Вы можете специфицировать большее количество в конструкторе, но вряд ли это вам понадобится.

Классы `ConcurrentHashMap` и `ConcurrentSkipListMap` включают полезные методы для атомарных вставок и удалений ассоциаций. Метод `putIfAbsent()` атомарно добавляет новую ассоциацию, которой не было ранее. Это удобно для организации кэша, к которому обращается множество потоков, чтобы гарантировать, что только один из них в единицу времени сможет добавлять элемент в кэш:

```
cache.putIfAbsent(key, value);
```

Обратной операцией является `remove()` (которая, возможно, должна вызвать `removeIfPresent()`).

Следующий вызов автоматически удаляет ключ и значение, если они присутствуют в коллекции:

```
cache.remove(key, value)
```

И, наконец, приведенный ниже вызов автоматически заменяет старое значение новым, находя его по ключу:

```
cache.replace(key, oldValue, newValue)
```

API java.util.concurrent.ConcurrentLinkedQueue<E> 5.0

- `ConcurrentLinkedQueue<E> ()`

  Конструирует неограниченную не блокирующую очередь, безопасно доступную множеству потоков.

API java.util.concurrent.ConcurrentSkipListSet<E> 6

- `ConcurrentSkipListSet <E> ()`
- `ConcurrentSkipListSet <E> (Comparator<? super E> comp)`

  Конструирует сортированный набор, безопасно доступный множеству потоков. Первый конструктор требует, чтобы элементы реализовали интерфейс `Comparable`.

API java.util.concurrent.ConcurrentHashMap<K, V> 5.0

API java.util.concurrent.ConcurrentSkipListMap<K, V> 6

- `ConcurrentHashMap <K, V> ()`
- `ConcurrentHashMap <K, V> (int initialCapacity)`
- `ConcurrentHashMap <K, V> (int initialCapacity, float loadFactor, int concurrencyLevel)`

  Конструируют хеш-карту, безопасно доступную множеству потоков.

Параметры		
	`initialCapacity`	Начальная емкость коллекции. По умолчанию равна 16.
	`loadFactor`	Управляет изменением размера: если средняя загрузка узла превышает этот фактор, размер таблицы изменяется.
	`concurrencyLevel`	Ожидаемое количество конкурирующих потоков, выполняющих запись.

- `ConcurrentSkipListMap<K, V>()`
- `ConcurrentSkipListSet<K, V>(Comparator<? super K> comp)`

  Конструируют сортированную карту, безопасно доступную множеству потоков. Первый конструктор требует, чтобы ключи реализовывали интерфейс `Comparable`.
- `V putIfAbsent(K key, V value)`

  Если ключ еще не присутствует в карте, ассоциирует заданное значение с данным ключом и возвращает `null`. В противном случае возвращает имеющееся значение, ассоциированное с ключом.
- `boolean remove(K key, V value)`

  Если ключ ассоциирован с данным значением, удаляет указанный ключ и указанное значение и возвращает `true`. В противном случае возвращает `false`.
- `boolean replace (K key, V oldValue, V newValue)`

  Если данный ключ ассоциирован с `oldValue`, то ассоциирует его с `newValue`. В противном случае возвращает `false`.

Копирующиеся при записи массивы

`CopyOnWriteArrayList` и `CopyOnWriteArraySet` — безопасные к потокам коллекции, в которых все изменения создают копию лежащего в основе массива. Такой подход удобен, если количество потоков, выполняющих итерацию по коллекции, значительно превышает количество потоков, изменяющих ее. Когда вы конструируете итератор, он содержит ссылку на текущий массив. Если массив позднее изменен, итератор по-прежнему ссылается на старую версию массива, а текущий массив коллекции заменяется новым. Как следствие, старый итератор имеет согласованный (хотя потенциально устаревший) вид и не нуждается в синхронизации.

Устаревшие безопасные к потокам коллекции

Еще в ранних выпусках Java классы `Vector` и `Hashtable` предоставляли безопасные к потокам реализации динамического массива и хеш-таблицы. В Java SE 1.2 эти классы были объявлены устаревшими и заменены классами `ArrayList` и `HashMap`. Эти классы не являются безопасными к потокам, но в библиотеке коллекций предусмотрен другой механизм для обеспечения безопасности потоков. Любой класс может быть сделан безопасным к потокам благодаря *синхронизирующей оболочке*:

```
List<E> synchArrayList = Collections.synchronizedList(new ArrayList<E>());
Map<K, V> synchHashMap = Collections.synchronizedMap(new HashMap<K, V>());
```

Методы полученной в результате коллекции защищены блокировкой, обеспечивая безопасный доступ потокам.

Вы должны гарантировать, что ни один поток не обращается к данным через оригинальные, не синхронизированные методы. Проще всего сделать это, не сохраняя никаких ссылок на исходный объект. Просто сконструируйте коллекцию и немедленно передайте ее оболочке, как показано в примерах.

Однако вы все же должны использовать блокировку “клиентской стороны”, если хотите выполнить итерацию по коллекции в то время, когда другой поток имеет возможность изменить ее:

```
synchronized (synchHashMap)
{
  Iterator<K> iter = synchHashMap.keySet().iterator();
  while (iter.hasNext()) . . .;
}
```

Такой же код следует использовать при использовании цикла “for each”, поскольку этот цикл применяет итератор. Заметим, что итератор дает сбой, генерируя исключение ConcurrentModificationException, когда другой поток пытается изменить коллекцию во время работы итератора. Синхронизация также необходима, чтобы легко и надежно обнаружить конкурирующее изменение данных.

Но обычно лучше использовать коллекции, определенные в пакете java.util.concurrent, вместо синхронизирующих оболочек. В частности, карта ConcurrentHashMap тщательно реализована с таким прицелом, чтобы множество потоков могли обращаться к ней, не блокируя друг друга, если они работают с разными узлами данных. Исключением является часто обновляемый массив-список. В этом случае синхронизированному ArrayList лучше предпочесть CopyOnWriteArrayList.

API java.util.Collections 1.2

- static <E> Collection<E> synchronizedCollection(Collection<E> c)
- static <E> List synchronizedList(List<E> c)
- static <E> Set synchronizedSet(Set<E> c)
- static <E> SortedSet synchronizedSortedSet(SortedSet<E> c)
- static <K, V> Map<K, V> synchronizedMap(Map<K, V> c)
- static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> c)

Конструируют представления коллекции с синхронизированными методами.

Интерфейсы Callable и Future

Интерфейс Runnable инкапсулирует задачу, выполняющуюся асинхронно; вы можете воспринимать это как асинхронный метод без параметров и возвращаемого значения. Callable подобен Runnable, но с возвратом значения. Интерфейс Callable является параметризованным типом, с единственным общедоступным методом call().

```
public interface Callable<V>
{
   V call() throws Exception;
}
```

Параметр представляет собой тип возвращаемого значения.

Например, Callable<Integer> представляет асинхронное вычисление, которое в конечном итоге возвращает объект Integer.

Future хранит результат асинхронного вычисления. Вы можете запустить вычисление, предоставив кому-либо объект Future, и забыть о нем. Владелец объекта Future может получить результат, когда он будет готов.

Интерфейс Future имеет следующие методы.

```
public interface Future<V>
{
  V get() throws . . .;
  V get(long timeout, TimeUnit unit) throws . . .;
  void cancel(boolean mayInterrupt);
  boolean isCancelled();
  boolean isDone();
}
```

Вызов первого метода get() устанавливает блокировку до тех пор, пока не завершится вычисление. Второй метод генерирует исключение TimeoutException, если

истекает таймаут до завершения вычислений. Если прерывается поток, выполняющий вычисление, оба метода генерируют исключение `InterruptedException`. Если вычисление уже завершено, `get()` немедленно возвращает управление.

Метод `isDone()` возвращает `false`, если вычисление продолжается, и `true` — если оно завершено.

Вы можете прервать вычисление, вызвав метод `cancel()`. Если вычисление еще не стартовало, оно отменяется и уже не будет запущено. Если же вычисление уже идет, оно прерывается в случае равенства `true` параметра `mayInterrupt`.

Класс-оболочка `FutureTask` представляет собой удобный механизм для превращения `Callable` одновременно в `Future` и `Runnable`, реализуя оба интерфейса. Например:

```
Callable<Integer> myComputation = . . .;
FutureTask<Integer> task = new FutureTask<Integer>(myComputation);
Thread t = new Thread(task); // это Runnable
t.start();
. . .
Integer result = task.get(); // это Future
```

Программа в листинге 14.11 демонстрирует эту концепцию в действии. Она подобна предыдущему примеру, который находит файлы, содержащие ключевое слово. Однако теперь мы просто будем подсчитывать количество файлов, соответствующих критерию поиска. Таким образом, у нас будет долго работающая задача, которая в результате даст целочисленное значение — пример `Callable<Integer>`.

```
class MatchCounter implements Callable<Integer>
{
  public MatchCounter(File directory, String keyword) { . . . }
  public Integer call() { . . . } // возвращает количество найденных файлов
}
```

Сконструируем объект `FutureTask` из `MatchCounter` и используем его для запуска потока:

```
FutureTask<Integer> task = new FutureTask<Integer>(counter);
Thread t = new Thread(task);
t.start();
```

И, наконец, напечатаем результат:

```
System.out.println(task.get() + " файлов найдено.");
```

Конечно, вызов `get()` устанавливает блокировку до тех пор, пока не будет готов результат.

Внутри метода `call()` мы используем тот же механизм рекурсивно. Для каждого подкаталога создадим новый `MathCounter` и запустим поток для него. Также мы спрячем объекты `FutureTask` в `ArrayList<Future<Integer>>`. И в конце сложим все результаты:

```
for (Future<Integer> result : results)
   count += result.get();
```

Каждый вызов `get()` устанавливает блокировку до тех пор, пока не будет готов результат. Конечно, потоки работают параллельно, так что есть шанс, что результаты будут готовы почти одновременно.

Листинг 14.11. Содержимое файла `FutureTest.java`

```
1. import java.io.*;
2. import java.util.*;
3. import java.util.concurrent.*;
4. /**
5.  * @version 1.0 2004-08-01
6.  * @author Cay Horstmann
7.  */
8. public class FutureTest
9. {
10.public static void main(String[] args)
11.{
12.  Scanner in = new Scanner(System.in);
13.  System.out.print("Укажите базовый каталог (например, /usr/local/jdk5.0/src): ");
14.  String directory = in.nextLine();
15.  System.out.print("Введите ключевое слово (например, volatile): ");
16.  String keyword = in.nextLine();
17.  MatchCounter counter = new MatchCounter(new File(directory), keyword);
18.  FutureTask<Integer> task = new FutureTask<Integer>(counter);
19.  Thread t = new Thread(task);
20.  t.start();
21.  try
22.  {
23.     System.out.println(task.get() + " файлов найдено.");
24.  }
25.  catch (ExecutionException e)
26.  {
27.     e.printStackTrace();
28.  }
29.  catch (InterruptedException e)
30.  {
31.  }
32.}
33.}
34./**
35. * Задача подсчитывает файлы в каталоге и всех его подкаталогах,
36. * которые содержат указанное ключевое слово.
37. */
38.class MatchCounter implements Callable<Integer>
39.{
40./**
41. * Конструктор MatchCounter.
42. * @param directory Каталог, с которого начинается поиск
43. * @param keyword Искомое ключевое слово
44. */
45.public MatchCounter(File directory, String keyword)
46.{
47.   this.directory = directory;
48.   this.keyword = keyword;
49.}
50.public Integer call()
51.{
52.   count = 0;
53.   try
54.   {
55.      File[] files = directory.listFiles();
56.      ArrayList<Future<Integer>> results = new ArrayList<Future<Integer>>();
```

```
57.       for (File file : files)
58.          if (file.isDirectory())
59.          {
60.             MatchCounter counter = new MatchCounter(file, keyword);
61.             FutureTask<Integer> task = new FutureTask<Integer>(counter);
62.             results.add(task);
63.             Thread t = new Thread(task);
64.             t.start();
65.          }
66.          else
67.          {
68.            if (search(file)) count++;
69.          }
70.       for (Future<Integer> result : results)
71.       try
72.       {
73.          count += result.get();
74.       }
75.       catch (ExecutionException e)
76.       {
77.          e.printStackTrace();
78.       }
79.    }
80.    catch (InterruptedException e)
81.    {
82.    }
83.    return count;
84. }
85. /**
86.  * Ищет в файле заданное ключевое слово.
87.  * @param file Файл, в котором идет поиск
88.  * @return true, если слово найдено в файле
89.  */
90. public boolean search(File file)
91. {
92.  try
93.   {
94.    Scanner in = new Scanner(new FileInputStream(file));
95.    boolean found = false;
96.    while (!found && in.hasNextLine())
97.    {
98.      String line = in.nextLine();
99.      if (line.contains(keyword)) found = true;
100.   }
101.   in.close();
102.   return found;
103.  }
104. catch (IOException e)
105.  {
106.   return false;
107.  }
108. }
109. private File directory;
110. private String keyword;
111. private int count;
112. }
```

API java.util.concurrent.Callable<V> 5.0

- `V call()`

  Запускает задачу, порождающую результат.

API java.util.concurrent.Future<V> 5.0

- `V get()`
- `V get(long time, TimeUnit unit)`

  Получает результат, устанавливая блокировку до момента его готовности или истечения указанного времени. Второй метод генерирует исключение `TimeoutException` в случае неудачи.

- `boolean cancel(boolean mayInterrupt)`

  Пытается отменить выполнение задачи. Если задача уже запущена и параметр `mayInterrupt` равен `true`, она прерывается. Возвращает `true` в случае успешной отмены.

- `boolean isCancel()`

  Возвращает `true`, если задача была отменена до ее завершения.

- `boolean isDone()`

  Возвращает `true`, если задача завершена нормально или отменена, либо по исключению.

API java.util.concurrent.FutureTask<V> 5.0

- `FutureTask(Callable<V> task)`
- `FutureTask(Runnable task, V result)`

  Конструирует объект, реализующий одновременно `Future<V>` и `Runnable`.

Класс Executors

Создание нового потока — довольно дорогостоящая операция, поскольку включает взаимодействие с операционной системой. Если ваша программа создает большое количество кратковременных потоков, то имеет смысл использовать *пул потоков*. Пул потоков содержит множество простаивающих потоков, готовых к запуску. Вы помещаете `Runnable` в пул, и один из потоков вызывает его метод `run()`. Когда метод `run()` завершается, поток не уничтожается и остается в пуле в готовности обслужить новый запрос.

Другая причина для использования пула потоков — необходимость ограничить количество конкурирующих потоков. Создание огромного числа потоков может пагубно отразиться на производительности, и даже привести к краху виртуальной машины. Если у вас есть алгоритм, создающий большое количество потоков, то вам следует установить "фиксированный" пул потоков, который ограничит общее количество конкурирующих потоков.

Класс `Executors` имеет ряд статических методов-фабрик для конструирования пулов потоков; их список приведен в табл. 14.2.

Таблица 14.2. Методы-фабрики класса `Executors`

Метод	Описание
`newCachedThreadPool`	Новые потоки создаются при необходимости; простаивающие потоки сохраняются в течение 60 секунд.
`newFixedThreadPool`	Пул содержит фиксированный набор потоков; простаивающие потоки сохраняются бесконечно.
`newSingleThreadExecutor`	Пул с единственным потоком, выполняющим переданные ему задачи последовательно (подобно потоку диспетчера событий Swing).
`newScheduledThreadPool`	Пул фиксированных потоков для планового запуска; замена `java.util.Timer`.
`newSingleThreadScheduledExecutor`	Пул с единственным потоком для планового запуска.

Пулы потоков

Рассмотрим первые три метода из табл. 14.2. Остальные методы обсудим позднее, в разделе "Плановое выполнение". Метод `newCachedThreadPool()` конструирует пул потоков, выполняющий каждую задачу немедленно, используя существующий простаивающий поток, если он доступен, либо создавая новый в противном случае. Метод `newFixedThreadPool()` конструирует пул потоков фиксированного размера. Если количество задач превышает количество простаивающих потоков, то лишние ставятся в очередь, чтобы быть запущенными, когда завершатся текущие выполняющиеся задачи. Метод `newSingleThreadExecutor()` создает пул размером в 1 поток. Единственный поток выполняет все отправленные ему задачи — одну за другой. Перечисленные три метода возвращают объект класса `ThreadPoolExecutor`, реализующий интерфейс `ExecutorService`.

Вы можете послать `Runnable` или `Callable` объекту `ExecutorService` одним из следующих методов:

```
Future<?> submit(Runnable task)
Future<T> submit(Runnable task, T result)
Future<T> submit(Callable<T> task)
```

Пул запустит установленную таким образом задачу при первой возможности. Когда вызывается `submit()`, вы получаете объект `Future`, который можно использовать для опроса состояния задачи.

Первый метод `submit()` возвращает странно выглядящий `Future<?>`. Вы можете использовать этот объект для вызова `isDone()`, `cancel()` или `isCancelled()`. Однако метод `get()` по завершении просто возвращает `null`.

Вторая версия `submit()` также принимает `Runnable`, а метод `get()` объекта `Future` вернет по завершении заданный объект `result`.

Третья версия принимает `Callable`, а возвращенный `Future` получает результат вычисления по его готовности.

Завершив работу с пулом потоков, вызывайте `shutdown()`. Этот метод инициирует последовательность выключения пула. После этого объект пула не принимает новых задач. Когда все задачи завершены, потоки в пуле уничтожаются. В качестве альтернативы можно вызвать `shutdownNow()`. В этом случае пул отменяет все задачи, которые еще не стартовали, и пытается прервать работающие потоки.

Подведем итог: что надо сделать, чтобы использовать пул потоков.

1. Вызвать статический метод `newCachedThreadPool()` или `newFixedThreadPool()` класса `Executors`.
2. Вызвать `submit()` для установки объектов `Runnable` или `Callable`.
3. Если хотите иметь возможность прерывать задачу, или если устанавливаете объекты `Callable`, обратитесь к возвращенным объектам `Future`.
4. Вызвать `shutdown()`, когда больше не собираетесь запускать новых задач.

Например, программа в предыдущем примере создает огромное количество кратковременных потоков — по одному на каталог. Программа в листинге 14.12 использует вместо этого для запуска задач пул потоков.

Для информации эта программа печатает максимальный размер пула, достигнутый во время выполнения. Эта информация недоступна через интерфейс `ExecutorService`. По этой причине приходится выполнять приведение объекта пула к классу `ThreadPoolExecutor`.

Листинг 14.12. Содержимое файла `ThreadPoolTest.java`

```
1. import java.io.*;
2. import java.util.*;
3. import java.util.concurrent.*;
4. /**
5.  * @version 1.0 2004-08-01
6.  * @author Cay Horstmann
7.  */
8. public class ThreadPoolTest
9. {
10.  public static void main(String[] args) throws Exception
11.  {
12.    Scanner in = new Scanner(System.in);
13.    System.out.print("Укажите базовый каталог (например, /usr/local/jdk5.0/src): ");
14.    String directory = in.nextLine();
15.    System.out.print("Введите ключевое слово (например, volatile): ");
16.    String keyword = in.nextLine();
17.    ExecutorService pool = Executors.newCachedThreadPool();
18.    MatchCounter counter = new MatchCounter(new File(directory), keyword, pool);
19.    Future<Integer> result = pool.submit(counter);
20.    try
21.    {
22.      System.out.println(result.get() + " файлов найдено.");
23.    }
24.    catch (ExecutionException e)
25.    {
26.      e.printStackTrace();
27.    }
28.    catch (InterruptedException e)
29.    {
30.    }
31.    pool.shutdown();
32.    int largestPoolSize = ((ThreadPoolExecutor) pool).getLargestPoolSize();
33.    System.out.println("максимальный размер пула = " + largestPoolSize);
34.  }
35. }
```

```
36. /**
37.  * Задача подсчитывает файлы в каталоге и всех его подкаталогах,
38.  * которые содержат указанное ключевое слово.
39.  */
40. class MatchCounter implements Callable<Integer>
41. {
42.    /**
43.     * Конструктор MatchCounter.
44.     * @param directory Каталог, с которого начинается поиск
45.     * @param keyword Искомое ключевое слово
46.     * @param pool Пул потоков для запуска задач
47.     */
48.   public MatchCounter(File directory, String keyword, ExecutorService pool)
49.   {
50.    this.directory = directory;
51.    this.keyword = keyword;
52.    this.pool = pool;
53.   }
54.   public Integer call()
55.   {
56.    count = 0;
57.    try
58.    {
59.     File[] files = directory.listFiles();
60.     ArrayList<Future<Integer>> results = new ArrayList<Future<Integer>>();
61.     for (File file : files)
62.        if (file.isDirectory())
63.        {
64.           MatchCounter counter = new MatchCounter(file, keyword, pool);
65.           Future<Integer> result = pool.submit(counter);
66.           results.add(result);
67.        }
68.        else
69.        {
70.           if (search(file)) count++;
71.        }
72.     for (Future<Integer> result : results)
73.        try
74.        {
75.           count += result.get();
76.        }
77.        catch (ExecutionException e)
78.        {
79.           e.printStackTrace();
80.        }
81.    }
82.    catch (InterruptedException e)
83.    {
84.    }
85.    return count;
86.   }
87.   /**
88.    * Ищет в файле заданное ключевое слово.
89.    * @param file Файл, в котором идет поиск
90.    * @return true, если слово найдено в файле
91.    */
```

```
92.  public boolean search(File file)
93.  {
94.   try
95.   {
96.    Scanner in = new Scanner(new FileInputStream(file));
97.    boolean found = false;
98.    while (!found && in.hasNextLine())
99.    {
100.     String line = in.nextLine();
101.     if (line.contains(keyword)) found = true;
102.   }
103.   in.close();
104.   return found;
105.  }
106.  catch (IOException e)
107.  {
108.   return false;
109.  }
110. }
111. private File directory;
112. private String keyword;
113. private ExecutorService pool;
114. private int count;
115.}
```

API java.util.concurrent.Executors 5.0

- `ExecutorService newCachedThreadPool()`

  Возвращает кэшированный пул потоков, создающий потоки по мере необходимости и закрывающий потоки, которые простаивают более 60 секунд.

- `ExecutorService newFixedThreadPool(int threads)`

  Возвращает пул потоков, использующий заданное количество потоков для запуска задач.

- `ExecutorService newSingleThreadExecutor()`

  Возвращает исполнитель, запускающий задачи последовательно в одном потоке.

API java.util.concurrent.ExecutorService 5.0

- `Future<T> submit(Callable<T> task)`
- `Future<T> submit(Runnable task, T result)`
- `Future<?> submit(Runnable task)`

  Устанавливают заданную задачу на выполнение.

- `void shutdown()`

  Останавливает службу, завершая все запущенные задачи и не принимая новых.

API java.util.concurrent.ThreadPoolExecutor 5.0

- `int getLargestPoolSize()`

  Возвращает максимальный размер пула потоков, который был достигнут за время его жизни.

Плановое выполнение

Интерфейс `ScheduledExecutorService` включает методы для планирования или многократного выполнения задач. Это — обобщение `java.util.Timer`, позволяющее создавать пул потоков. Методы `newScheduledThreadPool()` и `newSingleThreadScheduledExecutor()` класса `Executors` возвращают объекты, реализующие интерфейс `ScheduledExecutorService`.

Вы можете запланировать однократный запуск `Runnable` или `Callable` после истечения некоторого времени. Также можно запланировать периодический запуск `Runnable`. Подробности — ниже, во врезках, посвященных API.

API java.util.concurrent.Executors 5.0

- `ScheduledExecutorService newScheduledThreadPool(int threads)`

  Возвращает пул потоков, использующий заданное их количество для планирования запуска задач.

- `ScheduledExecutorService newSingleThreadScheduledExecutor()`

  Возвращает исполнитель, планирующий запуск задач последовательно в одном потоке.

API java.util.concurrent.ScheduledExecutorService 5.0

- `ScheduledFuture<V> schedule(Callable<V> task, long time, TimeUnit unit)`
- `ScheduledFuture<?> schedule(Runnable task, long time, TimeUnit unit)`

  Планируют запуск заданной задачи по истечении заданного времени.

- `ScheduledFuture<?> scheduleAtFixedRate(Runnable task, long initialDelay, long period, TimeUnit unit)`

  Планирует периодический запуск заданной задачи, каждые `period` единиц времени после истечения начального времени задержки.

- `ScheduledFuture<?> scheduleWithFixedDelay(Runnable task, long initialDelay, long delay, TimeUnit unit)`

  Планирует периодический запуск заданной задачи, с задержкой `delay` единиц времени между запусками после истечения указанного времени задержки.

Управление группами задач

Мы показали вам, как использовать службу исполнителей — пул потоков для повышения эффективности выполнения задач. Иногда исполнитель применяется больше по причинам тактического характера, просто для управления группами взаимосвязанных задач. Например, вы можете прервать все задачи в исполнителе, вызвав его метод `shutdownNow()`.

Метод `invokeAny()` посылает все объекты `Callable` из коллекции и возвращает результат выполненной задачи. Вы не знаете, результат какой именно задачи возвращен — предположительно, это будет результат задачи, которая завершилась быстрее всех. Вы будете использовать этот метод, имея дело с проблемами, любое решение которых вас устраивает. Например, предположим, что вам нужно разложить на множители большие целые числа — вычисление, необходимое для взлома шифра RSA. Вы

можете запустить множество задач, каждая из которых пытается найти множители из разных диапазонов. Как только одна из задач найдет ответ, все вычисления можно прекращать.

Метод `invokeAll()` запускает все объекты `Callable` из коллекции и возвращает список объектов `Future`, представляющих результаты всех задач. Вы можете обрабатывать результаты вычислений, когда они будут готовы:

```
List<Callable<T>> tasks = . . .;
List<Future<T>> results = executor.invokeAll(tasks);
for (Future<T> result : results)
   processFurther(result.get());
```

Недостаток этого подхода состоит в том, что, может быть, придется ждать слишком долго, если на выполнение первой задачи уйдет слишком много времени. Возможно, больше смысла имеет получать результаты по мере их готовности. Это можно сделать с помощью `ExecutorCompletionService`.

Запустите исполнитель, полученный обычным способом. Затем сконструируйте `ExecutorCompletionService`. Передайте задачи этому экземпляру службы. Служба управляет блокирующей очередью объектов `Future`, содержащих результаты переданных задач, пополняющейся по мере их готовности. Таким образом, вычисления можно организовать более эффективно:

```
ExecutorCompletionService service = new ExecutorCompletionService(executor);
for (Callable<T> task : tasks) service.submit(task);
for (int i = 0; i < tasks.size(); i++)
   processFurther(service.take().get());
```

API java.util.concurrent.ExecutorService 5.0

- `T invokeAny(Collection<Callable<T>> tasks)`
- `T invokeAny(Collection<Callable<T>> tasks, long timeout, TimeUnit unit)`

  Выполняют указанные задачи и возвращают результат одной из них. Второй метод генерирует исключение `TimeoutException` по истечении таймаута.

- `List<Future<T>> invokeAll(Collection<Callable<T>> tasks)`
- `List<Future<T>> invokeAll(Collection<Callable<T>> tasks, long timeout, TimeUnit unit)`

  Выполняют указанные задачи и возвращают результаты их всех. Второй метод генерирует исключение `TimeoutException` по истечении таймаута.

API java.util.concurrent.ExecutorCompletionService 5.0

- `ExecutorCompletionService(Executor e)`

  Конструирует службу завершения исполнителя, которая собирает результаты заданного исполнителя.

- `Future<T> submit(Callable<T> task)`
- `Future<T> submit(Runnable task, T result)`

  Посылают задачу лежащему в основе исполнителю.

- `Future<T> take()`

  Удаляет следующий готовый результат, устанавливая блокировку, если завершенных результатов нет.

- `Future<T> poll()`
- `Future<T> poll(long time, TimeUnit unit)`

  Удаляют следующий готовый результат или `null`, если таковых нет. Второй метод ожидает в течение указанного времени.

Синхронизаторы

Пакет `java.util.concurrent` содержит несколько классов, которые облегчают управление наборами совместно работающих задач — см. табл. 14.3. Эти механизмы имеют "консервированную функциональность" для часто встречающихся сценариев взаимодействия потоков. Если у вас есть набор совместно работающих потоков, которые следуют одному из этих сценариев поведения, вы должны просто использовать соответствующий библиотечный класс вместо попыток соорудить самодельную коллекцию блокировок и условий.

Таблица 14.3. Синхронизаторы

Класс	Что делает	Когда использовать
`CyclicBarrier`	Позволяет набору потоков ожидать, пока их счетчик достигнет общего барьера, а затем необязательно выполнить некоторое граничное действие.	Когда должно быть завершено определенное количество потоков, прежде чем можно будет использовать их результаты.
`CountDownLatch`	Позволяет набору потоков ожидать, когда значение счетчика уменьшится до 0.	Когда один или более потоков должны ожидать происхождения заданного количества событий.
`Exchanger`	Позволяет двум потокам обмениваться объектами, когда оба будут готовы.	Когда два потока работают с двумя экземплярами одной и той же структуры данных, причем один из них заполняет свой экземпляр, а второй — опустошает свой.
`Semaphore`	Позволяет набору потоков ожидать разрешения на работу.	Для ограничения общего числа потоков, имеющих доступ к ресурсу. Если допустимое количество потоков — один, то все остальные потоки блокируются, пока владеющий ресурсом поток не даст им разрешения.
`SynchronousQueue`	Позволяет потоку передавать объект другому потоку.	Чтобы посылать объект из одного потока в другой, когда оба готовы, без явной синхронизации.

Семафоры

Концептуально семафоры управляют множеством *разрешений*. Чтобы пройти семафор, поток запрашивает разрешение вызовом `acquire()`. Доступно только фиксированное число разрешений, ограничивающее количество потоков, которые могут пройти семафор. Другие потоки могут отдавать разрешения, вызывая `release()`. Объектов разрешений в действительности не существует. Семафор просто хранит счетчик. Более того, разрешение не может быть отдано потоком, который получил его. Фактически любой поток может отдать любое число разрешений. Но если их количество превысит допустимый максимум, то семафор просто установит свой счетчик

в максимальное значение. Все это придает семафорам высокую степень гибкости, но может также привести к путанице.

Семафоры были предложены Эдсгером Дейкстрой (Edsger Dijkstra) в 1968 г. для использования в качестве *примитивов синхронизации*. Дейкстра доказал, что семафоры могут быть эффективно реализованы и что они достаточно мощны, чтобы решать множество часто встречающихся проблем синхронизации. Почти в любой книге по операционным системам вы найдете реализации ограниченных очередей, использующих семафоры. Разумеется, прикладные программисты не должны заново изобретать ограниченные очереди. Мы советуем использовать семафоры только тогда, когда их поведение хорошо отображается на вашу проблему синхронизации, и не нужно "заплетать мозги", подгоняя их к вашему случаю.

Простейшим примером семафора является семафор со счетчиком, равным 1. Такой семафор можно использовать в качестве "ворот" для одного потока, открывающихся и закрывающихся другим потоком. В разделе "Пример: приостановка и возобновление анимации" вы увидите такой пример, где рабочий поток выполняет анимацию. Иногда рабочий поток ожидает щелчка на кнопке пользователем. Он пытается получить разрешение, и вынужден ожидать, когда щелчок на кнопке приведет к его выдаче.

Защелки с обратным отсчетом

`CountDonLatch` позволяет набору потоков ожидать, пока значение счетчика достигнет нуля. Защелка с обратным отсчетом — одноразовая. Как только счетчик достигнет 0, вы не можете увеличить его снова.

Полезным специальным случаем является защелка со счетчиком, равным 1. Она реализует одноразовые "ворота". Поток задерживается у ворот, пока другой поток не установит счетчик в 0.

Вообразим, например, набор потоков, которому для выполнения своей работы нужны некоторые начальные данные. Рабочий поток стартует и ожидает открытия "ворот". Другой поток подготавливает данные. Когда данные готовы, он вызывает `countDown()` и тем открывает "ворота" для всех рабочих потоков.

Вы можете затем использовать другую защелку, чтобы узнать, когда все рабочие потоки завершатся. Инициализируйте защелку количеством потоков. Каждый рабочий поток перед своим завершением уменьшает значение счетчика. Другой поток, который потребляет результат, ожидает на защелке, и начинает свою работу, как только все рабочие потоки завершатся.

Барьеры

Класс `CyclicBarrier` реализует рандеву, называемое *барьером*. Рассмотрим множество потоков, каждый из которых выполняет некоторую часть общего вычисления. Когда все части готовы, их результаты должны быть комбинированы. Когда поток завершает свою часть работы, мы разрешаем ему двигаться до барьера. Как только все потоки достигнут барьера, барьер открывается, и потоки могут работать дальше.

Поясним на примере. Сначала сконструируем барьер, передав ему количество участвующих потоков:

```
CyclicBarrier barrier = new CyclicBarrier(nthreads);
```

Каждый поток выполняет некоторую работу и вызывает `await()` на барьере до завершения:

```
public void run()
{
  doWork();
  barrier.await();
  . . .
}
```

Метод `await()` принимает необязательный параметр — таймаут:

```
barrier.await(100, TimeUnit.MILLISECONDS);
```

Если любой из потоков, ожидающих барьера, покинет барьер, то барьер *ломается*. (Поток может покинуть, если он вызвал `await()` со значением таймаута или из-за прерывания.) В этом случае метод `await()` для всех прочих потоков генерирует исключение `BrokenBarrierException`. Вызов `await()` в уже ожидающих потоках немедленно прекращается.

Вы можете применить необязательное *барьерное действие*, которое выполняется, когда все потоки достигают барьера:

```
Runnable barrierAction = . . .;
CyclicBarrier barrier = new CyclicBarrier(nthreads, barrierAction);
```

Действие может собрать результаты всех индивидуальных потоков.

Барьер называется циклическим, потому что может быть повторно использован после того, как все ожидающие потоки будут отпущены. В этом отношении он отличается от `CountDownLatch`, который может быть использован лишь однократно.

Обменники

Обменник `Exchanger` используется, когда два потока работают с двумя экземплярами одинакового буфера данных. Как правило, один из потоков наполняет буфер, а другой потребляет его содержимое. Когда оба заканчивают свою функцию, происходит обмен буферами.

Синхронные очереди

Синхронная очередь — это механизм, объединяющий поток-производитель и поток-потребитель. Когда поток вызывает `put()` на `SynchronousQueue`, он блокируется до тех пор, пока другой поток не вызовет `take()`, и наоборот. В отличие от случая с `Exchanger`, данные передаются только в одном направлении — от производителя к потребителю.

Несмотря на то что класс `SynchronousQueue` реализует интерфейс `BlockingQueue`, концептуально он не является очередью. Он не содержит никаких элементов — его метод `size()` всегда возвращает 0.

Пример: приостановка и возобновление анимации

Рассмотрим программу, которая выполняет некоторую работу, обновляет экран дисплея и затем ожидает, когда пользователь просмотрит результат и щелкнет на кнопке для продолжения, после чего выполняет следующую единицу работы.

Семафор со счетчиком разрешений, равным 1, может быть использован для синхронизации рабочего потока и потока диспетчера событий. Рабочий поток вызывает `acquire()` всякий раз, когда он готов к паузе. Поток графического пользовательского интерфейса вызывает `release()` при всяком нажатии пользователем кнопки Continue (Продолжить).

Что случится, если пользователь щелкнет кнопку несколько раз, в то время как рабочий поток находится в состоянии готовности? Поскольку доступно лишь одно разрешение, счетчик разрешений всегда остается равным 1.

В листинге 14.13 представлена программа, реализующая эту идею. Программа анимирует алгоритм сортировки. Рабочий поток сортирует массив, периодически останавливаясь и ожидая от пользователя разрешения продолжить. Пользователь может насладиться отображением текущего состояния алгоритма и щелкнуть на кнопке Continue, позволяя рабочему потоку перейти к следующему шагу.

Мы не станем утомлять вас деталями алгоритма сортировки, так что просто вызовем `Array.sort()` — метод, реализующий алгоритм сортировки слиянием. Чтобы приостановить алгоритм, применим объект `Comparator`, ожидающий открытия семафора. Таким образом, анимация будет приостановлена всякий раз, когда алгоритм сравнивает два элемента. Мы нарисуем текущие значения массива и выделим сравниваемые элементы (рис. 14.8).



На заметку! Анимация показывает слияние мелких сортированных диапазонов в более крупные диапазоны, но не совсем точно. Алгоритм сортировки слиянием использует второй массив для удержания временных значений, которые не видны. Но цель этого примера не в том, чтобы углубиться в алгоритмы сортировки, а в том, чтобы показать использование семафора для приостановки рабочего потока.

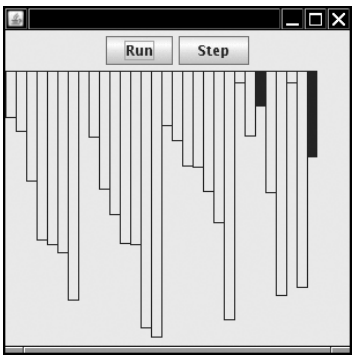


Рис. 14.8. Анимация алгоритма сортировки

Листинг 14.13. Содержимое файла `AlgorithmAnimation.java`

```
1. import java.awt.*;
2. import java.awt.geom.*;
3. import java.awt.event.*;
4. import java.util.*;
5. import java.util.concurrent.*;
6. import javax.swing.*;
7. /**
8.  * Программа для анимации алгоритма сортировки.
9.  * @version 1.01 2007-05-18
10. * @author Cay Horstmann
11. */
12.public class AlgorithmAnimation
13.{
```

```
14.  public static void main(String[] args)
15.  {
16.    EventQueue.invokeLater(new Runnable()
17.      {
18.        public void run()
19.        {
20.          JFrame frame = new AnimationFrame();
21.          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22.          frame.setVisible(true);
23.        }
24.      });
25.  }
26.  }
27.  /**
28.   * Этот фрейм показывает сортируемый массив вместе с кнопками
29.   * для пошаговой анимации и для запуска без остановок.
30.   */
31.  class AnimationFrame extends JFrame
32.  {
33.   public AnimationFrame()
34.   {
35.    ArrayComponent comp = new ArrayComponent();
36.    add(comp, BorderLayout.CENTER);
37.    final Sorter sorter = new Sorter(comp);
38.    JButton runButton = new JButton("Run");
39.    runButton.addActionListener(new ActionListener()
40.       {
41.         public void actionPerformed(ActionEvent event)
42.         {
43.              sorter.setRun();
44.         }
45.       });
46.       JButton stepButton = new JButton("Step");
47.       stepButton.addActionListener(new ActionListener()
48.       {
49.         public void actionPerformed(ActionEvent event)
50.         {
51.           sorter.setStep();
52.         }
53.       });
54.    JPanel buttons = new JPanel();
55.    buttons.add(runButton);
56.    buttons.add(stepButton);
57.    add(buttons, BorderLayout.NORTH);
58.    setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
59.    Thread t = new Thread(sorter);
60.    t.start();
61.   }
62.   private static final int DEFAULT_WIDTH = 300;
63.   private static final int DEFAULT_HEIGHT = 300;
64.  }
65.  /**
66.   * Этот исполняемый класс реализует алгоритм сортировки. При сравнении
67.   * двух элементов приостанавливается и обновляет компонент.
68.   */
```

```
69. class Sorter implements Runnable
70. {
71. /**
72.  * Конструктор Sorter.
73.  * @param values Сортируемый массив
74.  * @param comp Компонент, отображающий ход сортировки
75.  */
76. public Sorter(ArrayComponent comp)
77. {
78.   values = new Double[VALUES_LENGTH];
79.   for (int i = 0; i < values.length; i++)
80.     values[i] = new Double(Math.random());
81.   this.component = comp;
82.   this.gate = new Semaphore(1);
83.   this.run = false;
84. }
85. /**
86.  * Устанавливает режим "run". Вызывается в потоке диспетчера событий.
87.  */
88. public void setRun()
89. {
90.   run = true;
91.   gate.release();
92. }
93. /**
94.  * Устанавливает режим "step". Вызывается в потоке диспетчера событий.
95.  */
96. public void setStep()
97. {
98.   run = false;
99.   gate.release();
100. }
101. public void run()
102. {
103.   Comparator<Double> comp = new Comparator<Double>()
104.     {
105.     public int compare(Double i1, Double i2)
106.     {
107.       component.setValues(values, i1, i2);
108.       try
109.       {
110.         if (run) Thread.sleep(DELAY);
111.         else gate.acquire();
112.       }
113.       catch (InterruptedException exception)
114.       {
115.         Thread.currentThread().interrupt();
116.       }
117.       return i1.compareTo(i2);
118.       }
119.     };
120.   Arrays.sort(values, comp);
121.   component.setValues(values, null, null);
122. }
123. private Double[] values;
124. private ArrayComponent component;
```

```
125. private Semaphore gate;
126. private static final int DELAY = 100;
127. private volatile boolean run;
128. private static final int VALUES_LENGTH = 30;
129. }
130. /**
131.  * Этот компонент рисует массив и помечает два его элемента.
132.  */
133. class ArrayComponent extends JComponent
134. {
135. /**
136.  * Устанавливает значения для отрисовки. Вызывается потоком сортировки.
137.  * @param values Массив значений для отображения
138.  * @param marked1 Первый помеченный элемент
139.  * @param marked2 Второй помеченный элемент
140.  */
141. public synchronized void setValues(Double[] values, Double marked1, Double marked2)
142. {
143.   this.values = values.clone();
144.   this.marked1 = marked1;
145.   this.marked2 = marked2;
146.   repaint();
147. }
148. public synchronized void paintComponent(Graphics g) // Вызывается в потоке
149.                                                     // диспетчера событий.
150. {
151.   if (values == null) return;
152.   Graphics2D g2 = (Graphics2D) g;
153.   int width = getWidth() / values.length;
154.   for (int i = 0; i < values.length; i++)
155.   {
156.     double height = values[i] * getHeight();
157.     Rectangle2D bar = new Rectangle2D.Double(width * i, 0, width, height);
158.     if (values[i] == marked1 || values[i] == marked2) g2.fill(bar);
159.     else g2.draw(bar);
160.   }
161.  }
162.  private Double marked1;
163.  private Double marked2;
164.  private Double[] values;
165. }
```

API java.util.concurrent.CyclicBarrier 5.0

- `CyclicBarrier(int parties)`
- `CyclicBarrier(int parties, Runnable barrierAction)`

  Конструируют циклический барьер для заданного количества участников. `BarrierAction` выполняется, когда все части вызвали `await()` на барьере.

- `int await()`
- `int await(long time, TimeUnit unit)`

  Ожидают, пока все участники не вызовут `await()` на барьере, либо пока не истечет заданный таймаут — в этом случае генерируется исключение `TimeoutException`. При успехе возвращается индекс участника. Первый участник имеет индекс `parties-1`, а последний — `0`.

API java.util.concurrent.CountDownLach 5.0

- `CountDownLach(int count)`

  Конструирует защелку с заданным значением счетчика обратного отсчета.

- `void await()`

  Ожидает, пока значение счетчика не достигнет 0.

- `void await(long time, TimeUnit unit)`

  Ожидает, пока значение счетчика не достигнет 0, либо пока не истечет заданный таймаут. Возвращает `true`, если счетчик достиг 0, и `false` — по истечении таймаута.

- `public void countDown()`

  Уменьшает на единицу счетчик данной защелки.

API java.util.concurrent.Exchanger<V> 5.0

- `V exchange(V item)`
- `V exchange(V item, long time, TimeUnit unit)`

  Устанавливают блокировку, пока другой поток не вызовет этот метод, затем обмениваются элементом с другим потоком и возвращают элемент другого потока. Второй метод генерирует исключение `TimeoutException` по истечении таймаута.

API java.util.concurrent.SynchronousQueue<V> 5.0

- `SynchronousQueue()`
- `SynchronousQueue(boolean fair)`

  Конструируют синхронную очередь, позволяющую потокам передавать элементы. Если `fair` равно `true`, очередь отдает предпочтение потокам, ожидающим дольше всего.

- `void put(V item)`

  Устанавливает блокировку, пока другой поток не вызовет `take()`, чтобы взять данный элемент.

- `V take()`

  Устанавливает блокировку, пока другой поток не вызовет `put()`. Возвращает элемент, представленный другим потоком.

API java.util.concurrent.Semaphore 5.0

- `Semaphore(int permits)`
- `Semaphore(int permits, boolean fair)`

  Конструируют семафор с заданным количеством разрешений. Если `fair` равно `true`, отдается предпочтение потокам, ожидающим дольше всего.

- `void acquire()`

  Ожидает получения разрешения.

- `void tryAcquire()`

  Пытается получить разрешение; возвращает `false`, если нет доступных разрешений.

- `void tryAcquire(long time, TimeUnit unit)`

  Пытается получить разрешение в течение заданного времени; возвращает `false`, если нет доступных разрешений.

- `void release()`

  Освобождает разрешение.

Потоки и Swing

Как мы уже упоминали во введении к этой главе, одной из причин использования потоков в программе является необходимость сделать ее более реактивной. Когда ваша программа должна выполнять какую-то длительную работу, следует запускать новый рабочий поток вместо того, чтобы блокировать пользовательский интерфейс.

Однако следует быть осторожным с тем, что вы делаете в рабочем потоке, поскольку, как ни удивительно, библиотека Swing не является *безопасной к потокам*. Если попытаться манипулировать элементами пользовательского интерфейса из нескольких потоков, то можно повредить этот пользовательский интерфейс.

Чтобы продемонстрировать проблему, запустите текстовую программу из листинга 14.14. Когда вы щелкаете на кнопке Bad (Плохо), запускается новый поток, чей метод `run()` "истязает" выпадающий список, случайным образом добавляя и удаляя элементы.

```
public void run()
{
 try
 {
  while (true)
  {
    int i = Math.abs(generator.nextInt());
    if (i % 2 == 0)
       combo.insertItemAt(new Integer(i), 0);
    else if (combo.getItemCount() > 0)
       combo.removeItemAt(i % combo.getItemCount());
    sleep(1);
  }
 catch (InterruptedException e) {}
 }
}
```

Попробуйте запустить это. Щелкните на кнопке Bad, затем несколько раз щелкните на выпадающем списке. Переместите линейку прокрутки. Переместите окно. Еще раз щелкните на кнопке Bad. Продолжайте щелкать на выпадающем списке. В конце концов, вы увидите отчет об исключении (рис. 14.9).

Что происходит? Когда элемент вставляется в выпадающий список, последний инициирует событие для обновления дисплея. Затем вступает в действие код отображения, читается текущий размер выпадающего списка и подготавливается отображение значений. Но рабочий поток продолжает работать — в некоторый момент он уменьшает количество элементов в выпадающем списке. Код отображения полагает, что значений в модели больше, чем уже есть на самом деле, запрашивает несуществующие значения и генерирует исключение `ArrayIndexOutOfBounds`.

Такой ситуации можно избежать, позволив программистам блокировать объект выпадающего списка на время его отображения. Однако по двум причинам проектировщики Swing решили не тратить усилий на то, чтобы обеспечить безопасность к потокам Swing. Во-первых, синхронизация требует времени, а никто не хочет еще более замедлять Swing. Но что более важно, так это то, что участники команды Swing изучили опыт других команд разработчиков, которые имели дело с безопасными к потокам наборами инструментов пользовательского интерфейса. То, что они обнаружили, было неутешительным. Программисты, использующие безопасные к потокам наборы инструментов, путались в требованиях синхронизации и часто создавали программы, попадающие в состояние взаимной блокировки.

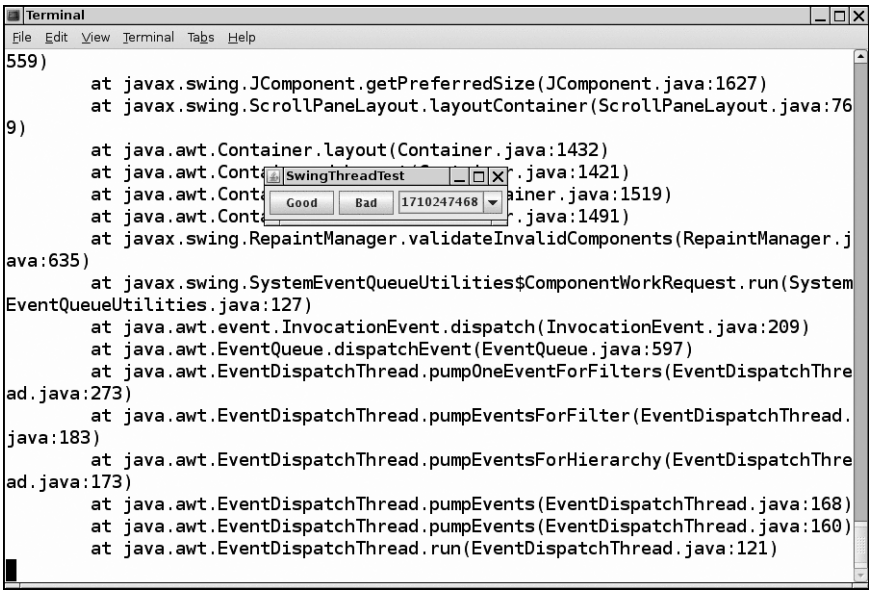


Рис. 14.9. Отчет об исключении на консоли

Выполнение задач, требующих много времени

Когда вы используете потоки вместе со Swing, то должны следовать двум простым правилам.

- Если действие требует много времени, выполняйте его в отдельном рабочем потоке и никогда — в потоке диспетчера событий.
- Не трогайте никакие компоненты Swing ни в одном потоке, за исключением потока диспетчера событий.

Причину появления первого правила понять легко. Если вы надолго загрузите работой поток диспетчера событий, то пользователю покажется, что приложение зависло, поскольку оно не сможет реагировать ни на какие события. В частности, поток диспетчера событий никогда не должен выполнять вызовов ввода-вывода, которые могут блокировать его навсегда, и никогда не должен вызывать метод `sleep()`. (Если вам нужно подождать некоторое время, применяйте события таймера.)

Второе правило часто называют правилом *единственного потока* для программирования Swing. Речь о нем пойдет позже.

Может показаться, что эти два правила противоречат друг другу. Предположим, что вы запустили отдельный поток для выполнения трудоемкой задачи. Обычно вы хотите обновлять пользовательский интерфейс для отражения хода работы в этом потоке. Когда задача завершится, вам опять понадобится обновить графический пользовательский интерфейс. Но вы не можете трогать компоненты Swing из вашего потока. Например, если вы захотите обновить индикатор выполнения или текст метки, то вы не можете просто установить их нужное значение из вашего потока.

Чтобы решить эту проблему, можно использовать два служебных метода в любом потоке для добавления произвольных действий в очередь событий. Например, предположим, что вы хотите периодически обновлять метку в потоке, чтобы отразить ход работы. Вызывать `label.setText()` из вашего потока нельзя. Вместо этого используйте методы `invokeLater()` и `invokeAndWait()` класса `EventQueue`, чтобы осуществить этот вызов в потоке диспетчера событий.

Вот что нужно сделать. Вы помещаете код Swing в метод `run()` класса, реализующего интерфейс `Runnable`. Затем создаете объект этого класса и передаете его статическому методу `invokeLater()` или `invokeAndWait()`. Например, вот как можно обновить текст метки:

```
EventQueue.invokeLater(new
Runnable()
{
  public void run()
  {
     label.setText(percentage + "% выполнено");
  }
});
```

Метод `invokeLater()` вернет управление немедленно, когда событие будет отправлено в очередь событий. Метод `run()` выполняется асинхронно. Метод `invokeAndWait()` ожидает, пока действительно не будет выполнен `run()`.

В ситуации с обновлением метки хода работ подходит метод `invokeLater()`. Пользователям важнее, чтобы рабочий поток выполнил больше работы, чем иметь более точный индикатор выполнения.

Оба метода выполняют `run()` в потоке диспетчера событий. Никакого дополнительного потока не создается.

В листинге 14.14 демонстрируется применение метода `invokeLater()` для безопасной модификации содержимого выпадающего списка. Если вы щелкнете на кнопке Good (Хорошо), поток вставит и удалит числа. Однако реальная модификация произойдет в потоке диспетчера событий.

Листинг 14.14. Содержимое файла `SwingThreadTest.java`

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.util.*;
4. import javax.swing.*;
5. /**
6.  * Эта программа демонстрирует, что поток, работающий параллельно с потоком
7.  * диспетчера событий, может вызвать ошибки в компонентах Swing.
8.  * @version 1.23 2007-05-17
9.  * @author Cay Horstmann
10. */
11.public class SwingThreadTest
12.{
```

```
13. public static void main(String[] args)
14. {
15.  EventQueue.invokeLater(new Runnable()
16.     {
17.       public void run()
18.       {
19.            SwingThreadFrame frame = new SwingThreadFrame();
20.            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21.            frame.setVisible(true);
22.       }
23.     });
24. }
25.}
26./**
27. * Этот фрейм содержит две кнопки для наполнения выпадающего списка
28. * из отдельного потока. Кнопка Good использует очередь событий,
29. * а кнопка Bad модифицирует выпадающий список напрямую.
30. */
31.class SwingThreadFrame extends JFrame
32.{
33. public SwingThreadFrame()
34. {
35.  setTitle("SwingThreadTest");
36.  final JComboBox combo = new JComboBox();
37.  combo.insertItemAt(Integer.MAX_VALUE, 0);
38.  combo.setPrototypeDisplayValue(combo.getItemAt(0));
39.  combo.setSelectedIndex(0);
40.  JPanel panel = new JPanel();
41.  JButton goodButton = new JButton("Good");
42.  goodButton.addActionListener(new ActionListener()
43.     {
44.        public void actionPerformed(ActionEvent event)
45.        {
46.           new Thread(new GoodWorkerRunnable(combo)).start();
47.        }
48.     });
49.  panel.add(goodButton);
50.  JButton badButton = new JButton("Bad");
51.  badButton.addActionListener(new ActionListener()
52.     {
53.       public void actionPerformed(ActionEvent event)
54.       {
55.          new Thread(new BadWorkerRunnable(combo)).start();
56.       }
57.     });
58.  panel.add(badButton);
59.  panel.add(combo);
60.  add(panel);
61.  pack();
62. }
63.}
64./**
65. * Этот класс, реализующий Runnable, модифицирует случайным образом
66. * выпадающий список, добавляя и удаляя числа. Это может вызвать ошибки,
67. * потому что методы выпадающего списка не синхронизированы, и к списку
68. * одновременно имеют доступ и диспетчер событий и рабочий поток.
```

```
69.  */
70. class BadWorkerRunnable implements Runnable
71. {
72. public BadWorkerRunnable(JComboBox aCombo)
73. {
74.   combo = aCombo;
75.   generator = new Random();
76. }
77. public void run()
78. {
79.   try
80.   {
81.      while (true)
82.      {
83.         int i = Math.abs(generator.nextInt());
84.         if (i % 2 == 0) combo.insertItemAt(i, 0);
85.         else if (combo.getItemCount() > 0)
86.             combo.removeItemAt(i % combo.getItemCount());
87.         Thread.sleep(1);
88.      }
89.   }
90.   catch (InterruptedException e)
91.   {
92.   }
93. }
94. private JComboBox combo;
95. private Random generator;
96. }
97. /**
98.  * Этот класс, реализующий Runnable, модифицирует случайным образом выпадающий
99.  * список, добавляя и удаляя числа. Чтобы обеспечить целостность выпадающего
100. * списка, операции редактирования отправляются в поток диспетчера событий.
101. */
102. class GoodWorkerRunnable implements Runnable
103. {
104. public GoodWorkerRunnable(JComboBox aCombo)
105. {
106.   combo = aCombo;
107.   generator = new Random();
108. }
109. public void run()
110. {
111.   try
112.   {
113.      while (true)
114.      {
115.        EventQueue.invokeLater(new Runnable()
116.        {
117.             public void run()
118.             {
119.                int i = Math.abs(generator.nextInt());
120.                if (i % 2 == 0) combo.insertItemAt(i, 0);
121.                else if (combo.getItemCount() > 0) combo.removeItemAt(i
122.                           % combo.getItemCount());
123.             }
124.        });
```

```
125.       Thread.sleep(1);
126.     }
127.   }
128.   catch (InterruptedException e)
129.   {
130.   }
131. }
132. private JComboBox combo;
133. private Random generator;
134.}
```

API **java.awt.EventQueue 1.1**

- `static void invokeLater(Runnable runnable)` **1.2**

  Заставляет метод `run()` объекта `runnable` выполниться в потоке диспетчера событий, после того, как будут обработаны все ожидающие события.

- `static void invokeAndWait (Runnable runnable)` **1.2**

  Заставляет метод `run()` объекта `runnable` выполниться в потоке диспетчера событий, после того, как будут обработаны все ожидающие события. Этот вызов устанавливает блокировку до тех пор, пока не завершится метод `run()`.

- `static boolean isDispatchThread()` **1.2**

  Возвращает `true`, если поток, выполняющий этот метод, является потоком диспетчера событий.

Использование класса `SwingWorker`

Когда пользователь запускает команду, для выполнения которой требуется много времени, вы запускаете новый поток, чтобы выполнить эту работу. Как вы видели в предыдущем разделе, этот поток должен использовать метод `EventQueue.invokeLater()` для обновления пользовательского интерфейса.

Несколько авторов предложили классы, предназначенные для облегчения этой программистской задачи, и один из таких классов был включен в Java SE 6. В этом разделе мы опишем этот класс — `SwingWorker`.

Программа в листинге 14.15 включает команды для загрузки текстового файла и для отмены процесса загрузки. Попробуйте запустить ее с большим файлом, таким как полный англоязычный текст "Графа Монте-Кристо", находящийся в каталоге `gutenberg` в рамках загруженного кода для этой книги. Файл загружается в отдельном потоке. Пока он читается, пункт меню Open (Открыть) остается недоступным, а пункт Cancel (Отмена) — наоборот, активен (рис. 14.10). После прочтения каждой строки обновляется счетчик в строке состояния. По завершении процесса чтения пункт меню Open становится доступным, Cancel — недоступным, а в строке состояния появляется текст "Done" (Сделано).

Этот пример демонстрирует типичные действия пользовательского интерфейса для фоновой задачи.

- После каждой единицы работы обновить пользовательский интерфейс, чтобы показать ее продвижение.
- После завершения всей работы выполнить окончательные изменения в пользовательском интерфейсе.

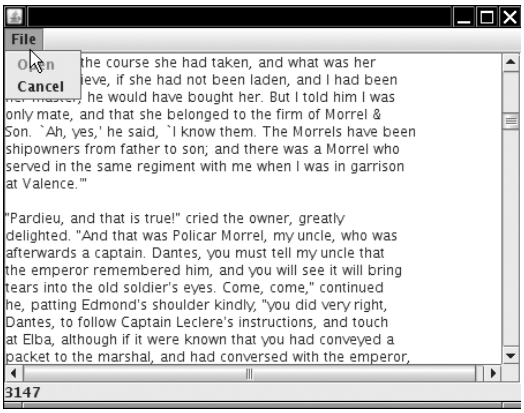


Рис. 14.10. Загрузка файла в отдельном потоке

Класс `SwingWorker` облегчает реализацию этой задачи. Вы переопределяете метод `doInBackground()`, чтобы он выполнял вашу трудоемкую задачу и периодически вызывал `publish()` для отображения хода работы. Этот метод выполняется в рабочем потоке. Метод `publish()` вызывает выполнение метода `process()`, чтобы заставить поток диспетчера событий обработать данные о ходе работ. По завершении работы в потоке диспетчера событий вызывается метод `done()`, который выполняет завершающее обновление пользовательского интерфейса.

Всякий раз, когда вы хотите выполнить какую-то работу в рабочем потоке, конструируйте новый экземпляр `SwingWorker`. (Каждый такой объект предназначен для однократного использования.) Затем вызывайте метод `execute()`. Обычно вы будете вызывать этот метод в потоке диспетчера событий, но это не рекомендуется.

Предполагается, что объект-работник (worker) производит некоторого рода результат; а потому `SwingWorker<T, V>` реализует `Future<T>`. Этот результат может быть получен методом `get()` интерфейса `Future`. Поскольку метод `get()` устанавливает блокировку до тех пор, пока не будет доступен результат, вы не станете вызывать его немедленно после вызова `execute()`. Хорошей идеей будет вызвать его тогда, когда вы уже знаете, что работа завершена. Обычно вы будете вызывать `get()` из метода `done()`. (Вызывать `get()` не обязательно. Иногда обработка данных о ходе работ — это все, что вам понадобится.)

Как промежуточные данные хода работ, так и конечный результат могут иметь произвольные типы. Класс `SwingWorker` получает эти типы через параметры типа. `SwingWorker<T, V>` генерирует результат типа `T` и данные о прогрессе типа `V`.

Чтобы прервать работу, используйте метод `cancel()` интерфейса `Future`. Когда работа отменена, метод `get()` генерирует исключение `CancellationException`.

Как уже упоминалось, вызов `publish()` из рабочего потока повлечет за собой вызов `process()` в потоке диспетчера событий. Для эффективности результаты нескольких вызовов `publish()` могут быть причиной только одного вызова `process()`. Метод `process()` принимает `List<V>`, содержащий все промежуточные результаты.

Попробуем заставить этот механизм работать на примере чтения текстового файла. Как известно, `JTextArea` — достаточно медленный компонент. Добавление строк из длинного текстового файла (вроде "Графа Монте-Кристо") потребует значительного времени.

Чтобы показать пользователю ход работы, мы хотим отображать в строке состояния количество прочитанных строк. Таким образом, данные о ходе работ будут состо-

ять из текущего номера строки и собственно самой текущей строки текста. Упакуем это в тривиальный внутренний класс.

```
private class ProgressData
{
  public int number;
  public String line;
}
```

Конечный результат — текст, прочитанный в `StringBuilder`. Таким образом, нам понадобится `SwingWorker<StringBuilder, ProgressData>`.

В методе `doInBackground()` мы читаем файл, строку за строкой. После каждой строки вызываем `publish()` для передачи номера прочитанной строки и ее содержимого.

```
@Override public StringBuilder doInBackground()
throws IOException, InterruptedException
{
  int lineNumber = 0;
  Scanner in = new Scanner(new FileInputStream(file));
  while (in.hasNextLine())
  {
     String line = in.nextLine();
     lineNumber++;
     text.append(line);
     text.append("\n");
     ProgressData data = new ProgressData();
     data.number = lineNumber;
     data.line = line;
     publish(data);
     Thread.sleep(1); // для проверки cancellaction; в ваших программах
                      // не понадобится
  }
  return text;
}
```

После каждой прочитанной строки мы выполняем приостановку, чтобы без проблем можно было проверить возможность отмены, но в ваших программах вы, скорее всего, не захотите замедлять работу задержками подобного рода. Если закомментировать эту строку, то мы обнаружим, что текст “Графа Монте-Кристо” загружается достаточно быстро, всего за несколько пакетных обновлений пользовательского интерфейса.



На заметку! Вы можете “сгладить” поведение этой программы, если обновлять текстовую область из рабочего потока, но такое невозможно для большинства других компонентов Swing. Мы показываем вам общий подход, при котором все обновления компонента происходят в потоке диспетчера событий.

В методе `process()` мы игнорируем все номера строк, за исключением последней, и объединяем все прочитанные строки для единого обновления текстовой области.

```
@Override public void process(List<ProgressData> data)
{
  if (isCancelled()) return;
  StringBuilder b = new StringBuilder();
  statusLine.setText("" + data.get(data.size() - 1).number);
  for (ProgressData d : data) { b.append(d.line); b.append("\n"); }
  textArea.append(b.toString());
}
```

В методе `done()` текстовая область обновляется полным текстом, а пункт меню Cancel становится недоступным. Обратите внимание, как рабочий поток стартует в слушателе события пункта меню Open.

Эта простая техника позволит вам выполнять "долгоиграющие" задачи, сохраняя возможность пользовательского интерфейса реагировать на действия пользователя.

Листинг 14.15. Содержимое файла `SwingWorkerTest.java`

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.io.*;
4. import java.util.*;
5. import java.util.List;
6. import java.util.concurrent.*;
7. import javax.swing.*;
8. /**
9.  * Эта программа демонстрирует рабочий поток, который
10. * выполняет потенциально длительную задачу.
11. * @version 1.1 2007-05-18
12. * @author Cay Horstmann
13. */
14.public class SwingWorkerTest
15.{
16.public static void main(String[] args) throws Exception
17.{
18.  EventQueue.invokeLater(new Runnable()
19.     {
20.       public void run()
21.       {
22.          JFrame frame = new SwingWorkerFrame();
23.          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24.          frame.setVisible(true);
25.       }
26.     });
27.}
28.}
29./**
30. * Этот фрейм содержит текстовую область для отображения текстового файла, меню
31. * для открытия файла и прерывания процесса открытия, а также строку состояния
32. * для отображения хода загрузки файла.
33. */
34.class SwingWorkerFrame extends JFrame
35.{
36.public SwingWorkerFrame()
37.{
38.  chooser = new JFileChooser();
39.  chooser.setCurrentDirectory(new File("."));
40.  textArea = new JTextArea();
41.  add(new JScrollPane(textArea));
42.  setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
43.  statusLine = new JLabel(" ");
44.  add(statusLine, BorderLayout.SOUTH);
45.  JMenuBar menuBar = new JMenuBar();
46.  setJMenuBar(menuBar);
47.  JMenu menu = new JMenu("File");
48.  menuBar.add(menu);
49.  openItem = new JMenuItem("Open");
```

```
50.  menu.add(openItem);
51.  openItem.addActionListener(new ActionListener()
52.    {
53.       public void actionPerformed(ActionEvent event)
54.       {
55.         // Отобразить диалог для выбора файла.
56.         int result = chooser.showOpenDialog(null);
57.         // Если файл выбран, установить его как пиктограмму метки.
58.         if (result == JFileChooser.APPROVE_OPTION)
59.         {
60.              textArea.setText("");
61.              openItem.setEnabled(false);
62.              textReader = new TextReader(chooser.getSelectedFile());
63.              textReader.execute();
64.              cancelItem.setEnabled(true);
65.         }
66.       }
67.    });
68.  cancelItem = new JMenuItem("Cancel");
69.  menu.add(cancelItem);
70.  cancelItem.setEnabled(false);
71.  cancelItem.addActionListener(new ActionListener()
72.     {
73.       public void actionPerformed(ActionEvent event)
74.       {
75.            textReader.cancel(true);
76.       }
77.     });
78. }
79. private class ProgressData
80. {
81.  public int number;
82.  public String line;
83. }
84. private class TextReader extends SwingWorker<StringBuilder, ProgressData>
85. {
86.  public TextReader(File file)
87.  {
88.    this.file = file;
89.  }
90.  // Следующий метод выполняется в рабочем потоке,
91.  // он не затрагивает компоненты Swing.
92.  @Override
93.  public StringBuilder doInBackground() throws IOException,
94.  InterruptedException
95.  {
96.    int lineNumber = 0;
97.    Scanner in = new Scanner(new FileInputStream(file));
98.    while (in.hasNextLine())
99.    {
100.     String line = in.nextLine();
101.     lineNumber++;
102.     text.append(line);
103.     text.append("\n");
104.     ProgressData data = new ProgressData();
105.     data.number = lineNumber;
```

```
106.      data.line = line;
107.      publish(data);
108.      Thread.sleep(1); //для проверки отмены; в ваших программах не понадобится
109.    }
110.    return text;
111.  }
112.  // Следующий метод выполняется в потоке диспетчера событий.
113.  @Override
114.  public void process(List<ProgressData> data)
115.  {
116.    if (isCancelled()) return;
117.    StringBuilder b = new StringBuilder();
118.    statusLine.setText("" + data.get(data.size() - 1).number);
119.    for (ProgressData d : data)
120.    {
121.       b.append(d.line);
122.       b.append("\n");
123.    }
124.    textArea.append(b.toString());
125.  }
126.  @Override
127.  public void done()
128.  {
129.    try
130.    {
131.      StringBuilder result = get();
132.      textArea.setText(result.toString());
133.      statusLine.setText("Done");
134.    }
135.    catch (InterruptedException ex)
136.    {
137.    }
138.    catch (CancellationException ex)
139.    {
140.      textArea.setText("");
141.      statusLine.setText("Cancelled");
142.    }
143.    catch (ExecutionException ex)
144.    {
145.      statusLine.setText("" + ex.getCause());
146.    }
147.    cancelItem.setEnabled(false);
148.    openItem.setEnabled(true);
149.  }
150.  private File file;
151.  private StringBuilder text = new StringBuilder();
152. };
153. private JFileChooser chooser;
154. private JTextArea textArea;
155. private JLabel statusLine;
156. private JMenuItem openItem;
157. private JMenuItem cancelItem;
158. private SwingWorker<StringBuilder, ProgressData> textReader;
159. public static final int DEFAULT_WIDTH = 450;
160. public static final int DEFAULT_HEIGHT = 350;
161. }
```

API java.swing.SwingWorker<T, V> 6

- `abstract T doInBackground()`

  Переопределите этот метод, чтобы он выполнял фоновую задачу и возвращал результат работы.

- `void process(List<V> data)`

  Переопределите этот метод для обработки промежуточных данных о ходе работы в потоке диспетчера событий.

- `void publish(V... data)`

  Отправляет промежуточные данные о прогрессе в поток диспетчера ошибок. Вызывайте этот метод из `doInBackground()`.

- `void execute()`

  Планирует запуск данного "рабочего" в рабочем потоке.

- `SwingWorker.StateValue getState()`

  Получает состояние рабочего потока; может быть одним из значений `PENDING`, `STARTED` или `DONE`.

Правило единственного потока

Каждое приложение Java запускается с метода `main()`, который работает в главном потоке. В программе Swing главный поток живет кратковременно. Он планирует конструирование пользовательского интерфейса в потоке диспетчера событий и завершает свою работу. После создания пользовательского интерфейса поток диспетчера событий обрабатывает уведомления о событиях наподобие вызовов `actionPerformed()` или `paintComponent()`. Другие потоки, такие как поток, отправляющий события в очередь, работают "за кулисами", но эти потоки невидимы прикладному программисту.

Ранее в этой главе мы уже представили правило единственного потока: "не затрагивать компоненты Swing из любого потока, кроме потока диспетчера событий". Здесь мы глубже исследуем это правило.

Существует несколько исключений из этого правила.

- Вы можете совершенно безопасно добавлять и удалять слушатели событий в любом потоке. Конечно, методы слушателей будут вызваны из потока диспетчера событий.
- Небольшое количество методов Swing являются безопасными к потокам. Они специально помечены в документации по API предложением "This method is thread safe, although most Swing methods are not" (Этот метод безопасный к потокам, хотя большинство методов Swing таковыми не являются). Наиболее полезные из таких безопасных к потокам методов следующие:

```
JTextComponent.setText()
JTextArea.insert()
JTextArea.append()
JTextArea.replaceRange()
JComponent.repaint()
JComponent.revalidate()
```



На заметку! В этой книге мы многократно использовали метод `repaint()`, но метод `revalidate()` применяется реже. Его назначение — инициировать компоновку компонента после того, как изменилось содержимое. Традиционная библиотека AWT включает метод `validate()` для инициации перекомпоновки компонента. Для компонентов Swing вы должны вместо него вызывать `revalidate()`. (Однако чтобы инициировать компоновку `JFrame`, вам придется вызывать `validate()`, потому что `JFrame` — это `Component`, а не `JComponent`.)

Исторически правило единственного потока было более либеральным. Любой поток мог конструировать компоненты, устанавливать их свойства и добавлять в контейнеры – до тех пор, пока ни один из компонентов не был *реализован*. Компонент реализуется, когда начинает принимать события перерисовки и проверки достоверности. Это происходит, как только выполняется вызов `serVisible(true)` или `pack()` для этого компонента, либо когда компонент добавляется в уже реализованный контейнер.

Эта версия правила одного потока была удобной. Она позволяла создавать графический пользовательский интерфейс в методе `main()` и затем вызывать `serVisible(true)` на фрейме верхнего уровня приложения. Не было необходимости в утомительном планировании запуска через `Runnable` в потоке диспетчера событий.

К сожалению, некоторые реализаторы компонентов не уделили должного внимания тонкостям исходного правила одного потока. Они запускали действия в потоке диспетчера событий, не заботясь даже о проверке того, реализован ли компонент. Например, если вызывается `setSelectionStart()` или `setSelectionEnd()` на `JTextComponent`, то перемещения знака вставки планируются в потоке диспетчера событий, даже если компонент невидим.

Обнаружить и исправить эти проблемы совсем несложно, но проектировщики Swing пошли по пути наименьшего сопротивления. Они объявили, что обращаться к компонентам из любого потока, который не является потоком диспетчера событий, небезопасно. А поэтому вам нужно конструировать пользовательский интерфейс в потоке диспетчера событий, используя вызов `EventQueue.invokeLater()`, который вы уже не раз видели в наших примерах программ.

Конечно, существует масса программ, которые не соблюдают осторожности и руководствуются старыми представлениями о правиле одного потока, инициализируя пользовательский интерфейс в главном потоке. Эти программы подвержены некоторому риску, связанному с тем, что определенные действия по инициализации пользовательского интерфейса вызовут конфликт с действиями главного потока. Как было указано в главе 7, вы вряд ли захотите оказаться одним из немногих “счастливчиков”, которые столкнутся с неприятностями и будут вынуждены тратить время на отладку нерегулярно воспроизводящихся ошибок, связанных с потоками. Поэтому лучше просто строго следовать правилу единственного потока.

Итак, вы добрались до конца первого тома. Этот том был посвящен основам языка программирования Java и тем частям стандартной библиотеки, которые нужны в большинстве программных проектов. Мы надеемся, что вам понравилась эта экскурсия по основам Java, и что вы нашли здесь много полезной информации. Освещение более сложных тем, таких как сетевое программирование, расширенные средства AWT/Swing, безопасность и интернационализация, вы найдете во втором томе.

Приложение

КЛЮЧЕВЫЕ СЛОВА JAVA

Ключевое слово	Значение	См. главу
`abstract`	Абстрактный класс или метод	5
`assert`	Используется для поиска внутренних ошибок программы	11
`boolean`	Булевский тип	3
`break`	Прерывание `switch` или цикла	3
`byte`	8-битовый целочисленный тип	3
`case`	Переключатель `switch`	3
`catch`	Конструкция блока try для перехвата исключений	11
`char`	Символьный тип Unicode	3
`class`	Определяет тип класса	4
`const`	Не используется	
`continue`	Продолжить выполнение в конце цикла	3
`default`	Ветвь по умолчанию в `switch`	3
`do`	Начало цикла `do/while`	3
`double`	Тип числа с плавающей точкой двойной точности	3
`else`	Ветвь `else` оператора `if`	3
`enum`	Перечислимый тип	3
`extends`	Определяет родитель класса	4
`final`	Константа, либо класс или метод, который не может переопределяться	5
`finally`	Часть блока `try`, выполняемая всегда	11

Ключевое слово	Значение	См. главу
float	Тип числа с плавающей точкой одинарной точности	3
for	Тип цикла	3
goto	Не используется	
if	Условный оператор	3
implements	Определяет интерфейс (интерфейсы), реализуемые классом	6
import	Импортирует пакет	4
instanceof	Проверка, является ли объект экземпляром класса	5
int	32-битный целочисленный тип	3
interface	Абстрактный тип с методами, который класс может реализовать	6
long	64-битный целочисленный тип	3
native	Метод, реализуемый операционной системой	11 (том 2)
new	Выделяет память для нового объекта или массива	3
null	Нулевая ссылка	3
package	Пакет классов	4
private	Модификатор доступа только в пределах методов данного класса	4
protected	Модификатор доступа только в пределах методов данного класса и его наследников	5
public	Модификатор открытого доступа методам всех классов	4
return	Возврат из метода	3
short	16-битный целочисленный тип	3
static	Средство, уникальное для класса, а не объектов данного класса	3
strictfp	Использует строгие правила для вычислений с плавающей точкой	2
super	Объект или конструктор суперкласса	5
switch	Оператор переключения	3
synchronized	Метод или блок кода, атомарный для потока	14
this	Неявный аргумент метода или конструктора класса	4
throw	Генерирует исключение	11
throws	Исключения, которые может генерировать метод	11
transient	Пометка данных, которые не должны быть постоянными	1 (том 2)
try	Блок кода, в котором могут возникнуть перехватываемые исключения	11
void	Обозначает метод, не возвращающий значений	3
volatile	Указывает, что поле может быть доступно нескольким потокам	14
while	Цикл	3

Предметный указатель

A

Application Programming Interface (API), 79

E

Eclipse, 46
Enterprise Edition (EE), 38

F

File Transfer Protocol (FTP), 24

G

General Public License (GPL), 34
Graphics User Interface (GUI), 300

H

HTML, 32; 53
HTTP (Hypertext Transfer Protocol), 24

I

Internet Foundation Classes (IFC), 300

J

Java
  дерево каталогов Java, 42
Java 2 (J2), 38
Java Archive (JAR), 503
Java Development Kit (JDK)
  инсталляция, 37
Java Foundation Classes (JFC), 301
Java Network Launch Protocol (JNLP), 510
Java Runtime Environment (JRE), 38
Java Specification Requests (JSR), 620
Java Web Start, 510
JUnit, 600
Just-in-time (JIT), 202

M

Micro Edition (ME), 38

N

NetBeans, 38

S

Software Development Kit (SDK), 38
Standard Edition (SE), 38
Swing, 300

T

TCP/IP (Transmission Control Protocol/
    Internet Protocol), 24

U

UML (Unified Modeling Language), 130
Unicode, 63
Update, 38
URL (Uniform Resource Location), 24
UTF-16, 64

V

W

WYSIWYG (What You See Is What You Get), 379

X

XML, 32

А

Алгоритм, 126
  решето Эратосфена, 713
Аплет, 28; 52; 503; 524
  AppletContext, 540
  AppletFrame, 542
  WelcomeApplet, 52
  контекст аплета, 539
  простой, 525
Аргумент, *См.* Параметр, 60

Б

Барьер, 785
Безопасность
  сертификат безопасности, 515
Библиотека
  AWT, 373
  IFC, 300
  Java, 655
  Swing, 301
  коллекции из библиотеки Java, 661; 690
  рефлексии, 235
Бинарный поиск, 705
Блок, *См.* Оператор, составной, 59; 94
  finally, 570
  вложенный, 94
  инициализационный, 168
  синхронизированный, 753
Блокировка
  взаимная, 756
  чтения/записи, 760

В

Ввод данных, 85
Виртуальная машина, 627

Всплывающая подсказка, 433
Выпадающий список (combo box), 409

Г

Группа, 671

Д

Действие (action), 357
Делегат, 253
Демон, 734
Дерево каталогов Java, 42
Дескриптор (tag), 182
  <applet>, 529
  <object>, 532
Диагностическое утверждение (assertion), 559; 578
Диалоговое окно, 474
  About, 474
  Eclipse, 47
  для ввода пароля, 478
  для выбора опций, 463
  для выбора файлов, 485
    с возможностью предварительного просмотра, 489
  для выбора цвета, 496
  модальное, 463
  немодальное, 463
  создание, 473
Динамическое связывание, 194; 198
Диспетчер
  безопасности, 513
  компоновки, 383; 436; 457
    BorderLayout, 384
    GridBagLayout, 437
    GridLayout, 386
    GroupLayout, 447
Доклет, 186

З

Защелка с обратным отсчетом, 785

И

Индекс, 111
Инкапсуляция, 127; 148
Интерлиньяж, 330
Интерфейс, 253; 260
  Callable, 773
  Collection, 656
  Comparable, 641
  Condition, 749
  Enumeration, 653; 709
  Future, 773
  Iterator, 656
  List, 691
  ListIterator, 692
  Lock, 749
  Preferences API, 552
  Queue, 655
  Runnable, 773
  Set, 695
  коллекции, 653
  очереди, 653
Исключение, 238
  ArrayIndexOutBoundsException, 562
  EOFException, 564
  FileNotFoundException, 562
  IllegalStateException, 658
  IOException, 562
  NullPointerException, 562
  RuntimeException, 562
  UnsupportedOperationException, 683
  иерархия исключений в Java, 561
  контролируемое (checked), 562
  неконтролируемое (unchecked), 562
  обработка исключений, 238; 559
  обработчик исключения, 560
  перехват исключений, 566
Итератор, 656
  перемещение итератора, 658

К

Каркас (framework), 690
  коллекций, 690
Карта (map), 682
  свойств, 547; 710
Клавиши быстрого доступа, 425
Класс, 127; 190
  -адаптер, 353
  -оболочка, 229
  -перечисление, 233
  AboutPanel, 506
  AbstractQueue, 655
  Action, 425
  Applet, 525
  Array, 249
  ArrayDeque, 655
  ArrayList, 222; 620
  BasicButtonUI, 382
  BigInteger, 109
  BitSet, 711
  BorderFactory, 405
  ButtonGroup, 401
  ButtonPanel, 344
  ButtonUIListener, 382
  Calendar, 134
  Class, 236; 645
  ClassNotFoundException, 569
  Collections, 696; 704; 706
  Color, 325

ColorAction, 344
ConsoleHandler, 591
ConsoleWindow, 605
CountDownLatch, 784
Cursor, 366
CyclicBarrier, 784; 785
Date, 134
Employee, 142; 178; 190
EnumSet, 688
EOFException, 565
EventHandler, 349
EventObject, 372
EventTracer, 607
Exchanger, 784
Executors, 777
FileHandler, 591
FileInputStream, 562
FileOpenService, 518
Graphics, 314; 335
Graphics2D, 324
GraphicsEnvironment, 327
GregorianCalendar, 134
Handler, 590
Hashtable, 709
IdentityHashMap, 689
Integer, 229; 295
InvocationTargetException, 569
JButton, 341; 382
JColorChooser, 496
JComboBox, 409
JComponent, 433
JDialog, 473
JFileChooser, 463; 485
JFrame, 304
JLabel, 393
JOptionPane, 463; 473
JTextComponent, 391
LinkedList, 655; 664; 697
Manager, 190
Math, 71
MouseHandler, 368
MouseMotionHandler, 368
Object, 210
Pair, 622; 647
Preferencs, 548
Properties, 552
RectangularShape, 320
ReentrantLock, 742
Robot, 610
RuntimeException, 561; 562; 569
Scanner, 86
Semaphore, 784
StackTraceElement, 573
StreanHandler, 590
StrictMath, 72
String, 75; 79; 641
StringBuffer, 76
SwingWorker, 797
SynchronousQueue, 784
Thread, 718
ThreadGroup, 735
Toolkit, 308
TreeMap, 692
TreeSet, 673; 692
Type, 648
Vector, 670; 753
WeakHashMap, 686
абстрактный, 204; 267
вложенный, 276; 291
  анонимный, 288
  локальный, 285
  статический, 290
загрузчик класса, 294
идентификация классов, 128
импортирование, 173
интерфейс класса, 139
конструктор класса, 146
обобщенный, 622
подкласс, 190
поле класса, 153
прокси, 293
путь к классу, 179
суперкласс, 190
финальный, 201
Клон, 150
Ключевое слово
abstract, 805
assert, 579; 805
boolean, 805
break, 805
byte, 805
case, 805
catch, 571; 805
char, 805
class, 57; 805
const, 68; 805
continue, 805
default, 805
do, 805
double, 805
else, 805
enum, 805
extends, 190; 805
final, 67; 194; 201; 287; 805
finally, 570; 805
float, 806
for, 806

goto, 806
if, 806
implements, 806
import, 173; 806
instanceof, 806
int, 806
interface, 806
long, 806
native, 806
new, 806
null, 806
package, 806
private, 177; 806
protected, 209; 806
public, 57; 177; 200; 806
return, 806
short, 806
static, 67; 153; 806
strictfp, 68; 806
super, 192; 806
switch, 806
synchronized, 749; 806
this, 148; 806
throw, 806
throws, 806
transient, 806
try, 806
void, 806
volatile, 755; 806
while, 806
Код
обобщенный, 627
Кодировка
Unicode, 64
UTF-16, 64
Кодовая единица, 78
Кодовая точка, 78
Коллекция
из библиотеки Java, 661; 690
упорядоченная, *См.* Список, связный, 663
Комментарии, 60
извлечение комментариев, 185
к классу, 182
к методам, 183
к полям, 184
обзорные, 185
общего характера, 184
пакетные, 185
Компилятор, 230
оперативный, 202
Конкатенация строк, 75
Константа
класса, 67
Конструктор
по умолчанию, 165
Контейнер, 383
Куча, 681

М

Массив, 111
анонимный, 113
двумерный, 123
динамически созданный, 115
зубчатый, 122
инициализация, 113
копирование массивов, 114
многомерный, 119
переменная массива, 114; 115
сортировка, 116
Меню, 419
всплывающее, 423
Метка, 393
Метод, 58; 127
-фабрика, 155
доступа (accessor), 136
модифицирующий (mutator), 136
обобщенный, 624
приватный, 151
статический, 154
указатели на методы, 253
Многопоточность, 717
Модификатор доступа, 57

Н

Набор (set), 672; 682
Наследование (inheritance), 128; 130; 190
иерархия наследования, 196
множественное, 267
цепочка наследования, 196

О

Обобщения (generics), 620
Обработка событий, 340
Обработчик исключения, 560
Объект, 128
клонирование объектов, 268
сериализованный, 531
Объектно-ориентированное программирование, 126
Объектно-ориентированное проектирование, 24
Окно
диалоговое, 463
модальное, 463
немодальное, 463
создание, 473

Оператор
  break, 94; 106
  continue, 108
  do/while, 99
  for, 94; 102
  if, 95
  if/else if, 97
  import, 173
  switch, 94; 104
  while, 98
  составной, 94
  условный, 95
Операция, 68
  !=, 69
  &, 70
  &&, 70
  *, 68
  +, 68
  ++, 69
  -, 68
  --, 69
  /, 68
  <, 69
  <<, 70
  <=, 69
  ==, 69
  =>, 69
  >, 69
  >>, 70
  >>>, 70
  ^, 70
  |, 70
  ||, 70
  битовая, 70
  декрементирования, 69
  иерархия операций, 73
  инкрементирования, 69
  логическая, 69
  отношения, 69
  приоритеты операций, 74
  сдвига, 70
Очередь, 653
  блокирующая, 763
  синхронная, 786
Ошибка
  ввода, 560
  обработка ошибок, 559
  программирования, 560

П

Пакет, 172
  Swing, 300
  герметизация пакета, 178
  по умолчанию, 175
  ресурсов, 586
Панель, 312
  инструментов, 431
  прокрутки (scroll pane), 395
Параметр, 60
  неявный, 147
  явный, 147
Перегрузка (overloading), 164
Переключатель (radio button), 401
Переменная, 65
  инициализация, 66
  объектная, 133
  статическая, 153
Подстрока, 75
Поиск
  бинарный, 705
Поле
  класса, 153
  статическое, 152
Полиморфизм, 194; 197
Поток, 717
  -демон, 734
  прерывание, 727
  приоритет, 733
  управления, 94
    прерывание, 106
Приведение типов (casting), 73; 202
Приложение WebStartCalculator, 519
Программа
  appletviewer, 52; 527; 540
  Eclipse, 46
  jar, 504
  jconsole, 604
  JUnit, 600
  Welcome, 50
Программирование
  обобщенное, 620
  объектно-ориентированное, 126
Прокси-класс, 293
Протокол
  FTP, 24
  HTTP, 24
  JNLP, 510
  TCP/IP, 24
  регистратор протокола (logger), 583
Протоколирование, 582
  базовое, 583
  расширенное, 583
Профилировщик, 605
Процесс, 717

Р

Рамка (border), 405
Рефлексия, 190; 235; 645
  библиотека рефлексии, 235

С

Связывание
  динамическое, 194; 198
Семафор, 784
Сертификат безопасности, 515
Сигнатура, 164
Символ
  специальный, 64
Синхронизаторы, 784
Синхронизация, 736
Список
  -массив, 670
  связный, 662
    двунаправленный, 662
Стек, 711
  трассировка стека, 573
Строка, 75
  конкатенация строк, 75
  неизменность строк, 76
  построение строк, 84

Т

Таблица
  хеш, 670
    коллизия хеш-таблицы, 671
Технология Java Web Start, 510
Тип данных, 61
  boolean, 65
  byte, 61
  char, 63
  double, 62
  float, 62
  int, 61
  long, 61
  short, 61
  перечислимый, 74
  преобразование числовых типов, 72
  приведение типов (casting), 202
  приведение числовых типов, 73
  с плавающей точкой, 62
  целочисленный, 61

У

Утверждение (assertion)
  диагностическое, 559; 578
Утилита
  jar, 505
  javadoc, 182
  MRJAppBuilder, 506

Ф

Файл
  ActionTest.java, 362
  AlgorithmAnimation.java, 787
  AnonymousInnerClassTest.java, 289
  AppletApplication.java, 546
  AppletFrame.java, 543
  ArrayListTest.java, 226
  Ball.java, 721; 725
  BallComponent.java, 722
  Bank.java, 738; 747; 751
  BigIntegerTest.java, 109
  BlockingQueueTest.java, 766
  BorderTest.java, 406
  Bounce.java, 719
  BuggyButtonTest.java, 615
  ButtonTest.java, 344
  Calculator.java, 387
  CalendarTest.java, 139
  Chart.java, 535
  CheckBoxTest.java, 399
  CircleLayoutTest.java, 458
  CloneTest.java, 272
  ColorChooserText.java, 498
  ComboBoxTest.java, 410
  CompoundInterest.java, 121
  ConsoleWindow.java, 605
  ConstructorTest.java, 170
  DataExchangeTest.java, 480
  DialogTest.java, 475
  DrawTest.java, 321
  Employee.java, 177
  EmployeeSortTest.java, 263
  EmployeeTest.java, 144
  EnumTest.java, 234
  EqualsTest.java, 218
  EventTracer.java, 608
  FileChooserTest.java, 490
  FontTest.java, 331
  FutureTest.java, 775
  GBC.java, 444
  GenericReflectionTest.java, 648
  GridBagLayoutTest.java, 442
  GroupLayoutTest.java, 452
  ImageTest.java, 336
  ImageViewer.java, 50
  InnerClassTest.java, 280
  InputTest.java, 86
  JAR, 503
  LinkedListTest.java, 667
  LoggingImageWindow.java, 592
  LotteryArray.java, 124

LotteryDrawing.java, 117
LotteryOdds.java, 104
ManagerTest.java, 195
MANIFEST.MF, 504
MapTest.java, 684
MenuTest.java, 428
MethodPointerTest.java, 255
MouseTest.java, 368
NotHelloWorld.java, 315
NotHelloWorldApplet.java, 526
ObjectAnalyzerTest.java, 247
OptionDialogTest.java, 466
PackageTest.java, 176
PairTest.java, 623
PairTest2.java, 626
PairTest3.java, 643
ParamTest.java, 162
PersonTest.java, 207
PlafTest.java, 352
PreferencesTest.java, 554
PriorityQueueTest.java, 681
PropertiesTest.java, 548
ProxyTest.java, 296
RadioButtonTest.java, 402
ReflectionTest.java, 241
ResourceTest.java, 508
Retirement.java, 100
RobotTest.java, 612
SetTest.java, 672
ShuffleTest.java, 704
Sieve.cpp, 714
Sieve.java, 713
SimpleFrameTest.java, 304
SizedFrameTest.java, 309
SliderTest.java, 415
src.sip, 41
StackTraceTest.java, 573
StaticInnerClassTest.java, 292
StaticTest.java, 156
SwingThreadTest.java, 794
SwingWorkerTest.java, 800
TextComponentTest.java, 396
ThreadPoolTest.java, 779
TimerTest.java, 275
ToolBarTest.java, 433
TransferRunnable.java, 739
TreeSetTest.java, 677
UnsynchBankTest.java, 738
WebStartCalculator.java, 520
Welcome.java, 45
WelcomeApplet.html, 53
WelcomeApplet.java, 54
манифеста, 504
Фильтр, 591
Форматирование выходных данных, 87
Фрейм, 304

Х

Хеш-карта, 686
Хеш-код, 214; 670; 674
Хеш-набор, 670
  связный, 687
Хеш-таблица, 215; 670
  коллизия хеш-таблицы, 671

Ц

Цвет
  системный, 325
Цикл
  do/while, 98
  for, 101
  for each, 112
  while, 98
  выход из цикла, 106
  неопределенный, 98
  определенный, 101
Цифровая подпись, 514

Э

Экземпляр, 127

Я

Язык
  HTML, 32; 53
  Java, 32
  JavaScript, 35
  UML, 130
  XML, 32
  C#, 33

Научно-популярное издание

Кей С. Хорстманн, Гари Корнелл

Java 2. Библиотека профессионала
Том 1. Основы
8-е издание

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

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

Подписано в печать 14.03.2013. Формат 70×100/16
Гарнитура Times. Печать офсетная
Усл. печ. л. 65,79. Уч.-изд. л. 45,9
Доп. тираж 1000 экз. Заказ № 0000

Первая Академическая типография "Наука"
199034, Санкт-Петербург, 9-я линия, 12/28

VD_CoreJava-v1.indd 814 20.02.2008 15:40:18

ПОЛНЫЙ СПРАВОЧНИК ПО JAVA™ 7-Е ИЗДАНИЕ

Герберт Шилдт



www.williamspublishing.com

Книга известного “гуру” в области программирования посвящена новой версии одного из наиболее популярных и совершенных языков – Java. Построенная в виде учебного и справочного пособия, она является превосходным источником исчерпывающей информации по последней версии платформы Java, Java SE 6, и позволяет практически с нуля научиться разрабатывать приложения и аплеты производственного качества. Помимо синтаксиса самого языка и фундаментальных принципов программирования, в книге подробно рассматриваются такие сложные вопросы, как ключевые библиотеки Java API, каркас коллекций, создание аплетов и сервлетов, AWT, Swing и Java Beans. Немалое внимание уделяется вводу-выводу, работе в сети, регулярным выражениям и обработке строк.

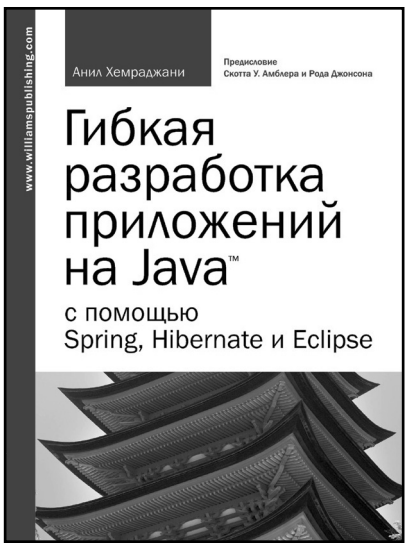
ISBN 978-5-8459-1168-1

в продаже

REKLAMA_CoreJava-v1.indd 815 17.02.2008 13:30:42

ГИБКАЯ РАЗРАБОТКА ПРИЛОЖЕНИЙ НА JAVA
С ПОМОЩЬЮ SPRING, HIBERNATE И ECLIPSE

Анил Хемраджани



www.williamspublishing.com

В этой книге основное внимание уделено разработке и в меньшей степени инфраструктуре. Другими словами, больше внимания уделено технологиям разработки приложений, таким как Spring, Hibernate и Eclipse, а не программным продуктам, таким как серверы приложений или базы данных. Все, что представлено в этой книге, опробовано в реальных приложениях, которые успешно работают (некоторые в кластеризуемой среде сервера приложений). Одна из задач этой книги заключается в краткости и конкретности, поэтому автор решил практически полностью сосредоточиться на разработке хорошо масштабируемого приложения. В данной книге, кроме технологий Spring, Hibernate и Eclipse, также описаны альтернативные и конкурирующие технологии.

ISBN 978-5-8459-1375-3 **в продаже**

REKLAMA_CoreJava-v1.indd 816 17.02.2008 13:30:43