Ведь это так просто!

5-е издание

Java®

для ЧАЙНИКОВ®



Научитесь:

- объединять небольшие компоненты в одну большую программу
- использовать библиотеки стандартных классов Java
- создавать объекты и повторно использовать методы базовых классов
- обрабатывать исключения и реагировать на события

Барри Берд

Java®

FOR DUMMIES®

5th Edition

by Barry Burd



Wiley Publishing, Inc.

Java®

для чайников®

5-е издание

Барри Берд



ДИАЛЕКТИКА

Москва • Санкт-Петербург • Киев
2013

ББК 32.973.26-018.2.75
Б48
УДК 681.3.07

Компьютерное издательство "Диалектика"
Главный редактор *С.Н. Тригуб*
Зав. редакцией *В.Р. Гинзбург*
Перевод с английского и редакция канд. техн. наук *А.Г. Сысонюка*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Берд, Барри.
Б48 Java для чайников, 5-е изд. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2013. — 368 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1769-0 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

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

Copyright © 2013 by Dialektika Computer Publishing.

Original English language edition Copyright © 2011 by Wiley Publishing, Inc.

All rights reserved including the right of reproduction in whole or in part in any `form`. `This` translation is published by arrangement with Wiley Publishing, Inc.

Научно-популярное издание

Барри Берд

Java для чайников, 5-е издание

В издании использованы карикатуры американского художника Рича Теннанта

Литературный редактор *Л.Н. Красножон*
Верстка *Л.В. Чернокозинская*
Художественный редактор *Е.П. Дынник*
Корректор *Л.А. Гордиенко*

Подписано в печать 30.04.2013. Формат 70x100/16
Гарнитура Times. Печать офсетная
Усл. печ. л. 29,67. Уч.-изд. л. 19,49
Доп. тираж 1500 экз. Заказ № 915.

Первая Академическая типография "Наука"
199034, Санкт-Петербург, 9-я линия, 12/28

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1769-0 (рус.)

ISBN 978-0-470-37173-2 (англ.)

© Компьютерное изд-во "Диалектика", 2013, перевод, оформление, макетирование

© Wiley Publishing, Inc., 2011

Оглавление

Введение 15

Часть I. Основы Java 21
Глава 1. Знакомство с Java 23
Глава 2. Разработка программного обеспечения 35
Глава 3. Базовые компоненты Java 49

Часть II. Создание программы на Java 69
Глава 4. Переменные и значения 71
Глава 5. Управляющие инструкции 95
Глава 6. Циклы 119

Часть III. Объектно-ориентированное программирование 131
Глава 7. Классы и объекты 133
Глава 8. Повторное использование кода 157
Глава 9. Конструкторы 181

Часть IV. Передовые методики программирования 199
Глава 10. Правильное применение переменных и методов 201
Глава 11. Массивы и коллекции 225
Глава 12. Как сохранить хорошую мину при плохой игре 253
Глава 13. Область видимости 279
Глава 14. Реагирование на события клавиатуры и мыши 299
Глава 15. Аплеты 315
Глава 16. Соединение с базой данных 325

Часть V. Великолепные десятки 333
Глава 17. Десять способов избежать ошибок 335
Глава 18. Десять сайтов, посвященных Java 341

Часть VI. Приложения 343
Приложение А. Установка интегрированной среды разработки 345
Приложение Б. Использование Eclipse 353
Предметный указатель 359

Содержание

Об авторе 13
Введение 15
Как работать с этой книгой 15
Соглашения, используемые в книге 15
Что можно не читать 16
Несколько предположений 17
Структура книги 18
Часть I. Основы Java 18
Часть II. Создание программы на Java 18
Часть III. Объектно-ориентированное программирование 18
Часть IV. Передовые методики программирования 19
Часть V. Великолепные десятки 19
Часть VI. Приложения 19
Пиктограммы, используемые в книге 19
Что дальше 20
Ждем ваших отзывов! 20

Часть I. Основы Java 21

Глава 1. Знакомство с Java 23
Что можно делать с помощью Java 24
Зачем писать программы 25
Немного истории 25
Объектно-ориентированное программирование 28
Объектно-ориентированные языки 28
Объекты и классы 30
Преимущества объектно-ориентированного подхода 31
Наглядное представление классов и объектов 32
Что дальше 33

Глава 2. Разработка программного обеспечения 35
Быстрое начало 35
Что должно быть установлено на компьютере 37
Что такое компилятор 38
Что такое виртуальная машина Java 40
Процесс разработки 45
Интегрированная среда разработки 46

Глава 3. Базовые компоненты Java 49
Поговорим на языке Java 49
Грамматика и общие имена 50
Слова в программе Java 51

Ваша первая программа на Java 53
Как работает ваша первая программа 54
Классы 54
Методы 55
Главный метод программы 57
Как приказать компьютеру выполнить нужную операцию 58
Фигурные скобки 60
Не говорите "Без комментариев..." 62
Добавление комментариев в код 63
Не будьте слишком строги к старине Барри 66
Использование комментариев для экспериментов с кодом 66

Часть II. Создание программы на Java 69

Глава 4. Переменные и значения 71
Переменчивая переменная 71
Оператор присваивания 74
Типы значений и переменных 75
Отображение текста 77
Числа без десятичной точки 78
Инициализация при объявлении 79
Примитивные типы Java 80
Тип `char` 81
Тип `boolean` 82
Ссылочные типы 83
Объявление импорта 86
Создание новых значений с помощью операторов 87
Инициализировать можно раз, а присваивать — много раз 89
Операторы инкремента и декремента 89
Операторы присваивания 93

Глава 5. Управляющие инструкции 95
Принятие решения с помощью инструкции `if` 95
Угадайте число 96
Ввод текста с клавиатуры 96
Генерация случайных чисел 98
Инструкция `if` 98
Двойной знак равенства 100
Блоки 100
Отступы в инструкции `if` 101
Сломанная вилка (`if` без `else`) 101
Условия с операторами сравнения и логическими операторами 102
Сравнение чисел и символов 102

Сравнение объектов 103
Импортируем все за один раз 105
Логические операторы 106
Чем "нуль" отличается от "ничто" 108
Условия в скобках и скобки в условиях 109
Вложение инструкций `if` 111
Переключатель `switch` 113
Выбор варианта 113
Не забывайте вставлять `break`! 115
Строковый аргумент — новинка в Java 7 117

Глава 6. Циклы 119
Цикл `while` 119
Цикл `for` 122
Структура цикла `for` 124
Премьера хита "Эл под дождем" 125
Цикл `do` 126
Считывание одного символа с клавиатуры 129
Манипулирование файлами 130
Объявление переменной в блоке 130

Часть III. Объектно-ориентированное программирование 131

Глава 7. Классы и объекты 133
Определение класса 133
Открытый класс 135
Объявление переменных и создание объектов 135
Инициализация переменной 138
Одна программа — это много классов 139
Определение метода в классе 139
Счет, отображающий сам себя 140
Заголовок метода 141
Передача параметров методу и получение значения от метода 142
Передача значения в метод 144
Значение, возвращаемое методом 145
Как сделать, чтобы числа выглядели красиво 147
Модификаторы доступа 151
Правильный стиль программирования 152
Как сделать поле недоступным 154
Проверка правил с помощью методов доступа 156

Глава 8. Повторное использование кода 157
Определение класса 158
Класс, определяющий служащего 158

Правильное использование класса 159
Создание платежного чека 161
Работа с файлами (небольшое отступление) 162
Хранение данных в файле 162
Копирование и вставка кода 163
Чтение из файла 164
Куда подевался мой файл? 166
Добавление имен папок в имя файла 166
Построчное чтение 167
Наследование 169
Создание производного класса 171
Использование производных классов 174
Соответствие типов 175
Использование класса `PartTimeEmployee` 176
Переопределение существующих методов 177
Аннотации 179
Вызов методов базовых и производных классов 179

Глава 9. Конструкторы 181
Определение конструктора 181
Что такое температура 182
Что такое температурная шкала 182
Так что же такое температура? 183
Что можно сделать с температурой 185
Поиск нужного конструктора 186
Некоторые вещи никогда не изменяются 189
Конструктор базового класса в производном классе 190
Усовершенствованный класс температуры 190
Конструкторы производных классов 191
Использование усовершенствованного класса температуры 192
Конструктор, выполняемый по умолчанию 193
Конструктор может не только заполнять поля 194
Классы и методы Java API 196
Аннотация `@SuppressWarnings` 198

Часть IV. Передовые методики программирования 199

Глава 10. Правильное применение переменных и методов 201
Определение класса 201
Еще один способ сделать числа красивыми 202
Использование класса `Player` 203
Девять вызовов конструктора 205
Графический интерфейс пользователя 205
Отфутболим исключение другому методу 207

Статические поля и методы 207
Зачем столько статики 209
Статическая инициализация 210
Отображение общей статистики команды 210
Статический импорт 212
Осторожно, статика! 213
Поэкспериментируем с переменными 214
Переменная на своем месте 215
Переменные в разных местах 217
Передача параметров 219
Передача по значению 219
Возвращение результата 221
Передача по ссылке 221
Возвращение объекта из метода 223
Эпилог 224

Глава 11. Массивы и коллекции 225
Как расположить предметы в ряд 225
Создание массива в два этапа 227
Сохранение значений 228
Табулостопы и другие специальные символы 230
Инициализация массива 230
Расширенный цикл `for` 231
Поиск 233
Массивы объектов 236
Использование класса `Room` 237
Еще один способ украшения чисел 240
Тернарный условный оператор 241
Аргументы командной строки 241
Использование аргументов командной строки в коде 242
Проверка количества аргументов командной строки 244
Коллекции 245
Классы коллекций 246
Класс `ArrayList` 247
Использование обобщенных типов 250
Проверка, есть ли еще данные 250

Глава 12. Как сохранить хорошую мину при плохой игре 253
Обработка исключений 254
Параметр блока `catch` 258
Типы исключений 259
Кто должен перехватить исключение 261
Блок `catch` с несколькими типами исключений 267

Не будем чрезмерно осторожничать 267
Восстановление работы программы после исключения 268
Наши друзья — хорошие исключения 269
Обработайте исключение или передайте его дальше 270
Блок `finally` 274
Закрытие файлов 276
Как закрыть файл 276
Ресурсы в заголовке блока `try` 276

Глава 13. Область видимости 279
Модификаторы доступа к членам классов 280
Классы, доступ и части программы 280
Классы и члены классов 281
Правила доступа к членам класса 281
Пример с рисунком во фрейме 283
Структура папок 285
Создание фрейма 286
Как изменить программу, не изменяя классы 287
Доступ, установленный по умолчанию 289
Как проникнуть в пакет 292
Защищенный доступ 292
Включение не производного класса в тот же пакет 294
Модификаторы доступа к классам 295
Открытые классы 296
Не открытые классы 296

Глава 14. Реагирование на события клавиатуры и мыши 299
Реагирование на щелчок мышью 299
События и обработка событий 301
Интерфейсы Java 302
Потоки 303
Ключевое слово `this` 304
Тело метода `actionPerformed()` 305
Идентификатор версии 305
Реагирование на другие события 306
Внутренние классы 311

Глава 15. Аплеты 315
Пример простого аплета 315
Выполнение аплета 316
Открытый класс 317
Классы Java API 317
Анимированный аплет 318

Методы, используемые в аплете 319
Содержимое методов аплета 320
Реагирование на события в аплете 322

Глава 16. Соединение с базой данных 325
JDBC и Java DB 325
Создание записей базы данных 326
Использование команд SQL 328
Подключение и отключение базы данных 328
Извлечение данных 330

Часть V. Великолепные десятки 333

Глава 17. Десять способов избежать ошибок 335
Правильное использование регистра букв 335
Выход из блока `switch` 336
Сравнение двух значений 336
Добавление элемента в графический интерфейс 336
Добавление приемников событий 337
Определение конструкторов 337
Исправление нестатических ссылок 337
Соблюдение границ массива 337
Указатели на `null` 338
Помогите виртуальной машине Java найти классы 338

Глава 18. Десять сайтов, посвященных Java 341
Веб-сайты этой книги 341
Сайты Java 341
Новости, обзоры, примеры кодов 342
Работа 342
Сайты для каждого 342

Часть VI. Приложения 343

Приложение А. Установка интегрированной среды разработки 345
Загрузка и установка JDK 345
Загрузка и установка Eclipse 349
Конфигурирование Eclipse 351
Приложение Б. Использование Eclipse 353
Работа с примерами книги 353
Создание собственного проекта 355

Предметный указатель 359

Об авторе

Барри Берд получил диплом магистра информатики в Университете Рутгерса и степень доктора философии в области математики в Университете Иллинойса. С 1980 года др. Берд является профессором кафедры математики и информатики в Университете Дрю в Мэдисоне, штат Нью-Джерси. Он также преподает на курсах по программированию и часто выступает с докладами на конференциях в США, Австралии, Европе и Азии. Кроме того, он автор многих статей и книг по программированию на Java. Живет в Мэдисоне, штат Нью-Джерси, с женой и двумя детьми.

Введение

Java — прекрасный язык! Я программирую на нем уже много лет. Особенно мне нравится синтаксис Java — аккуратный и хорошо структурированный. Почти все в нем подчиняется простым правилам. Правда, некоторые правила на первый взгляд выглядят устрашающими, но данная книга для того и предназначена, чтобы помочь вам понять их. Так что если вы планируете программировать на Java, но не хотите читать скучные толстые учебники, прочитайте эту книгу, и вы найдете все, что нужно, чтобы освоиться с новым языком программирования.

Как работать с этой книгой

Я очень хотел бы сказать "Откройте наугад любую страницу книги и начните писать код Java. Не оглядывайтесь назад; вам всего лишь нужно заполнить пустые поля формы". В некоторых случаях это действительно так. Вы ничего не испортите, если начнете писать код на Java, не умея этого делать, поэтому можете смело экспериментировать с любыми примерами.

Однако нужно честно признать, что приведенное выше утверждение в большинстве случаев неправильное. Если вы не видите всей картины, написать корректную программу будет слишком тяжело. Это справедливо не только для Java, но и для любого языка программирования. Если вы введете код, не зная, что он делает, а затем этот код не будет работать, вы окажетесь в тупике.

Поэтому я разделил материал книги на небольшие порции. Каждая порция — это приблизительно одна глава. Можете начать чтение с любой главы, а иногда даже с середины главы. Я приложил немало усилий, чтобы примеры были понятными без обращения к другим главам и главы были как можно более независимыми одна от другой. Если в какой-то главе вводится сложная концепция, объясняемая в другой главе, я всегда привожу ссылку на другую главу.

В целом, читая данную книгу, воспользуйтесь следующими советами.

- ✓ Если вы уже знаете что-либо, можете об этом не читать.
- ✓ Если вы любопытны, не бойтесь забежать вперед. Всегда можно вернуться к предыдущей главе, чтобы возобновить чтение с места, на котором остановились, или найти объяснение непонятной концепции.

Коды примеров, используемых в книге, можно загрузить с сайта издательства по такому адресу:

`http://www.dialektika.com/books/978-5-8459-1769-0.html`

Соглашения, используемые в книге

Почти каждая техническая книга начинается с упоминания о шрифтовых выделениях, и данная книга не является исключением. Ниже приведено краткое описание применяемых здесь типографских соглашений.

- ✓ Новые термины выделяются *курсивом*.
- ✓ Коды Java, имена файлов и адреса URL приводятся моноширинным шрифтом.
- ✓ Если при чтении книги вам нужно будет что-нибудь ввести с клавиатуры, вводимый текст выделяется полужирным моноширинным шрифтом, например "В текстовое поле введите слово **`MyNewProject`**".
- ✓ Надписи на элементах интерфейса (флажках, кнопках, пунктах меню, переключателях и т.п.) приводятся рубленым шрифтом, например "Щелкните на кнопке Open".
- ✓ Когда при вводе с клавиатуры нужно что-либо вставить в код, это обозначается курсивом:

  ```
  public class любое_имя {...
  ```

  Это означает, что в качестве имени класса нужно ввести любой допустимый идентификатор. Поскольку идентификатор не должен содержать пробелов, в подстановке используется символ подчеркивания.
- ✓ Для обозначения последовательности щелчков при работе с многоуровневыми меню, вкладками и кнопками используется стрелочка (⇨), например "Выберите команду File⇨Open (Файл⇨Открыть)". Иногда цепочка команд может содержать разнородные элементы, например имя команды в меню, имя вкладки в открывшемся окне, имя кнопки, на которой нужно щелкнуть, и т.п.

Что можно не читать

Откройте главу 1 и посмотрите, знаком ли вам излагаемый материал. Если не знаком, прочитайте главу, в противном случае переходите к главе 2. Если вам знаком и этот материал, переходите к главе 3 и т.д. В некоторых случаях принятие решения (остановиться на главе или перейти к следующей) может вызвать затруднение, поэтому ниже дан ряд советов по этому поводу.

- ✓ Если вы уже знаете, что такое Java и для чего нужен этот язык, пропустите главу 1 и сразу переходите к главе 2.
- ✓ Если вы можете запустить Java-программу на выполнение и вам безразлично, что происходит "за кулисами", пропустите главу 2 и переходите к главе 3.
- ✓ Если вы зарабатываете на жизнь программированием, но на каком-либо другом языке (например, на C++ или C#), пропустите главу 1 и переходите к главе 2 или 3. Когда вы достигнете глав 5 и 6, концепции языка начнут проясняться. Если вы успешно дошли до главы 7, значит, уже можете считаться Java-разработчиком.
- ✓ Если вы хорошо знакомы с языком C (но не C++), начните с глав 2–4 и пропустите главы 5 и 6.

- ✓ Если вы работаете на C++, просмотрите главы 2 и 3, пропустите главы 4–6 и приступайте к серьезному чтению начиная с главы 7. Java немного отличается от C++ трактовкой классов и объектов.
- ✓ Если вы зарабатываете на жизнь программированием на Java, обращайтесь ко мне и помогите написать очередное, 6-е издание данной книги.

Несколько предположений

Данная книга посвящена языку программирования Java, но несколько слов все же нужно сказать и о вас, читателе этой книги.

- ✓ **Я предполагаю, что у вас есть беспрепятственный доступ к компьютеру.** Коды, приведенные в данной книге в качестве примеров, могут выполняться практически на любом компьютере, на котором можно установить операционную систему Windows, Mac, Linux или Unix. Подойдет даже компьютер десятилетней давности (плюс-минус несколько лет), лишь бы у него были монитор, клавиатура, мышка и, естественно, коврик для мышки.
- ✓ **Я предполагаю, что вы умеете работать со стандартными элементами пользовательского интерфейса, такими как меню и диалоговые окна.** Вам не обязательно быть опытным пользователем Windows, Linux или Mac, но вы должны уметь запустить программу, найти файл, создать каталог, скопировать файл в каталог, т.е. выполнить элементарные операции. Большую часть времени при работе с примерами данной книги вы будете вводить текст с клавиатуры и щелкать мышкой.

  В редких случаях, когда вам придется выполнять более сложные операции, такие как перетаскивание или копирование в буфер обмена, я буду рассказывать вам, как это делается (конечно, только в начальных главах). Однако учтите, что ваш компьютер может быть сконфигурирован миллионами разных способов, и мои указания не могут быть совершенно точными во всех случаях. Поэтому рекомендую применять следующую методику: в первую очередь точно выполните мои указания; если что-либо не получилось, попытайтесь откорректировать их в соответствии со спецификой вашей системы; если и в этом случае не удается достичь желаемых результатов, ознакомьтесь с инструкциями к вашей системе.
- ✓ **Я предполагаю, что вы умеете мыслить логически.** Логическое мышление — главный элемент программирования на любом языке, включая Java. Надеюсь, эта книга поможет вам раскрыть неведомые ранее логические способности.
- ✓ **Я не делаю никаких предположений о вашем предыдущем опыте программирования (или о его отсутствии).** Работая над данной книгой, я попытался сделать невозможное — создать книгу, интересную для опытного программиста и в то же время доступную для человека, не имеющего

никакого опыта программирования. Это означает, что я не предполагаю наличия у вас какого-либо опыта или знаний в области программирования.

Если вы до этого момента ни разу не сталкивались с понятиями цикла или массива, можете смело приступать к чтению данной книги. С другой стороны, если вы уже встречались с ними в других языках (возможно, в Visual Basic, C++ или COBOL), вы обнаружите ряд интересных особенностей их реализации в Java.

Разработчики Java взяли из концепции объектно-ориентированного программирования лучшие идеи, упростили их и создали на их основе мощную технологию решения алгоритмических задач. В Java вы найдете много новых средств, поощряющих творческий подход к программированию. Многие из них сначала покажутся вам довольно сложными, но со временем они станут для вас вполне естественными. В любом случае вам понравится программировать на Java.

Структура книги

Наименьшая структурная единица этой книги — подразделы, из которых состоят разделы, из которых, в свою очередь, состоят главы. И наконец, из глав состоят части книги. Вся книга состоит из пяти частей плюс приложение. Ниже дано краткое описание каждой части.

Часть I. Основы Java

В этой части описаны инфраструктура и язык Java, включая основные технические концепции, применяемые инструменты и синтаксические правила. Вы узнаете, что такое виртуальная машина Java, как исходный код превращается в байтовый и зачем это нужно.

Часть II. Создание программы на Java

В главах 4–6 рассматриваются основы программирования и раскрываются понятия, которые нужно знать, чтобы заставить компьютер делать то, что от него требуется. Если вы писали программы на Visual Basic, C++ или любом другом языке, материал части II покажется вам знакомым. В этом случае можете пропустить некоторые разделы или прочитать их быстро. Но не читайте слишком быстро, чтобы не пропустить информацию об отличиях Java от других языков программирования (особое внимание этому вопросу уделяется в главе 4).

Часть III. Объектно-ориентированное программирование

В этой части находятся мои любимые главы, потому что в них рассматриваются важнейшие понятия объектно-ориентированного программирования. Из нее вы узнаете, как решаются фундаментальные задачи программирования. Примеры данной части небольшие, но они хорошо иллюстрируют серьезные идеи. Читая часть III, вы узнаете, как создаются классы и объекты и как можно повторно использовать существующие классы.

В настоящее время на рынке имеется немало книг, в которых принципы объектно-ориентированного программирования рассматриваются довольно расплывчато, так сказать, в общих чертах. Могу с уверенностью утверждать, что данная книга не принадлежит к их числу. Каждая концепция объектно-ориентированного программирования в книге проиллюстрирована простым и конкретным примером.

Часть IV. Передовые методики программирования

Если вы уже пробовали программировать Java и хотите узнать побольше, в этой части вы найдете для себя много интересного. Она посвящена тонкостям программирования на Java, которые не заметны непосвященным. Из этой части вы узнаете об использовании массивов и коллекций, реагировании на события, создании аплетов Java и взаимодействии с базами данных.

Часть V. Великолепные десятки

В этой части даны полезные советы о том, как избежать распространенных ошибок и найти дополнительные источники информации.

Часть VI. Приложения

В этой части подробно рассматриваются вопросы загрузки и установки программного обеспечения, необходимого для разработки программ на Java. Данная тема является вспомогательной, поэтому она оформлена в виде приложений.

Пиктограммы, используемые в книге

Если бы вы могли видеть меня в процессе написания книги, вы заметили бы, что я сижу перед компьютером и разговариваю сам с собой. Я мысленно произношу каждое предложение. Когда в моем сознании возникают новые мысли, комментарии или что-то, что не соответствует основной теме, я немного изгибаю шею и наклоняю голову. По этому признаку любой человек, который меня слушает (хотя обычно никого рядом нет), может понять, что я отвлекся от основной темы.

Конечно, читая книгу, вы не увидите, как я наклоняю голову. Следовательно, нужно как-то обозначить то, что я отклонился в сторону. Таким обозначением служат перечисленные ниже пиктограммы.



Дополнительная порция информации, которая пригодится вам в практической работе.



Каждому человеку свойственно ошибаться. На протяжении своей профессиональной карьеры программиста я сам допустил огромное количество ошибок. Как преподаватель я знаю, в каких именно местах склонны совершать ошибки начинающие программисты. Такие места отмечены данной пиктограммой.



Данной пиктограммой обозначены концепции, которые я рекомендую запомнить. Пиктограмма помогает мысленно выделить их в тексте книги.



Я не предполагаю, что вы прочитаете эту книгу от корки до корки и все запомните. Я старался подбирать такие примеры, чтобы для их понимания нужны были только сведения, приводимые в текущей главе. Но в некоторых случаях для понимания примера или рассматриваемой концепции вам все же придется обратиться к другой главе. Такие места отмечены данной пиктограммой.



Я не смог удержаться от включения в текст любопытных фактов или подробностей, не обязательных для понимания принципов работы на Java. В некоторых случаях такие подробности помогут вам понять, о чем думали разработчики языка или инструментов Java, создавая данное средство. Читать абзацы, отмеченные данной пиктограммой, не обязательно, однако иногда они могут быть полезными, и уж во всяком случае — интересными.

Что дальше

Если вы дошли до этого места, значит, вы готовы приступить к чтению книги о Java. Я ваш гид, босс и помощник одновременно. Выполняйте мои указания и рассчитывайте на мою помощь. Я сделал все возможное, чтобы чтение книги увлекло вас и, что еще важнее, помогло понять принципы работы на Java. Итак, читайте и наслаждайтесь!

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. 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 интересны любые ваши замечания в наш адрес.

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

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

Наши электронные адреса:

E-mail: `info@dialektika.com`

WWW: `http://www.dialektika.com`

Наши почтовые адреса:

в России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1

в Украине: 03150, Киев, а/я 152

Часть I

Основы Java

The 5th Wave — Рич Теннант



В этой части...

Познакомьтесь с Java и узнайте, что это такое и почему вам нужно (или не нужно) научиться программировать на этом языке. Если вы что-то когда-то слышали о Java, но не уверены в том, что правильно понимаете базовые концепции этого языка, данная часть будет для вас полезной. Если вы программировали на других языках, то найдете здесь важную информацию об особенностях программирования на Java. Если вы сказали кому-то, что являетесь крупным специалистом по Java, и теперь лихорадочно пытаетесь что-нибудь узнать об этом языке, чтобы поддержать свою репутацию, эта часть тоже будет для вас полезной.

Глава 1

Знакомство с Java

В этой главе...

- Что можно делать с помощью Java
- Зачем писать программы
- Немного истории
- Объектно-ориентированное программирование

Подумайте о том, чего вы хотите от своего компьютера. Я считаю, что компьютеры полезны главным образом по следующим двум причинам.

- ✓ **Работая для нашего блага, компьютер не упрямится, не скучает, не подвергается стрессам и не устает.** Компьютер — это ваш электронный раб. Мой компьютер работает 24 часа в сутки, обрабатывая радиоастрономические данные для программы поиска внеземных цивилизаций SETI@home. Сочувствую ли я ему потому, что он так тяжело трудится? Жалуется ли компьютер на свою нелегкую участь? Требует ли он защиты своих прав у профсоюзов и Национальной комиссии по защите наемных работников? Нет.

  Я отдаю команды, щелкаю кнутом и делаю с ним все что хочу. Чувствую ли я угрызения совести? Нет.

- ✓ **Компьютеры переносят идеи, а не бумагу.** В былые времена, когда человек хотел передать кому-либо сообщение, он нанимал гонца, который садился на лошадь и доставлял сообщение из рук в руки. Текст сообщения был записан на бумаге, пергаменте, глиняной табличке или любом другом физическом носителе, доступном в то время.

  Сейчас этот процесс кажется нам весьма расточительным, но только потому, что нам комфортно в нашей электронной эпохе. Сообщения — это идеи, а физические предметы — чернила, бумага, лошадь — не имеют никакого отношения к содержанию реальной идеи, они лишь ее временные физические носители (даже если люди использовали их для переноса идей в будущее на несколько столетий). Тем не менее сами по себе идеи безбумажны, "безлошадны" и "безгонцовы".

  Компьютеры хороши тем, что по сравнению с другими физическими носителями переносят идеи чрезвычайно эффективно. Они не переносят почти ничего, кроме идеи (всего лишь несколько фотонов радиоизлучения и несколько наноампер в импульсе электрического тока), причем переносят ее со скоростью света и точно в то место, в которое нужно.

При эффективной работе с идеями происходит нечто удивительное: неожиданно исчезают все накладные расходы. Больше не нужно валить деревья и возить бумагу, нанимать курьеров и содержать канцелярию. Нам осталось лишь нажимать кнопки. Без накладных расходов можно не только делать нужную работу намного быстрее, но и решать намного более сложные задачи.

Что можно делать с помощью Java

Как хорошо было бы, если бы к компьютеру можно было обратиться на естественном языке, например приказать: "Вычисли мне сумму всех ячеек в данном столбце таблицы!" Но, к сожалению, мы не можем этого сделать. Мы вынуждены писать эту инструкцию на специальном языке, который называется *языком программирования*. В частности, Java — это один из языков программирования.

В научно-фантастических фильмах и романах часто встречаются сюжеты, в которых человек на естественном языке приказывает роботу выполнить какую-либо простую работу, и в результате получает неожиданные, катастрофические результаты. Английский, русский и другие естественные языки не подходят для коммуникации с компьютером по следующим причинам.

- ✓ **Предложение на естественном языке чаще всего двусмысленное.** Слушающий человек правильно понимает вкладываемый смысл на основе текущей ситуации, своего жизненного опыта и других факторов, но компьютер лишен жизненного опыта человека и не видит текущую ситуацию. Например, когда слышен звонок и человек говорит: "Открой дверь!", другой человек понимает, что нужно открыть входную дверь, а не дверь шкафа или кухни, хотя в предложении "Открой дверь!" нет информации о том, какую дверь нужно открыть.
- ✓ **На естественном языке очень тяжело правильно сформулировать сложную команду.** Эта особенность естественного языка наглядно проявляется в юридических документах. Читая длинные, витиеватые фразы, мы неодобрительно называем такой стиль "канцелярщиной", хотя пишущий всего лишь пытался точно и полно выразить требование, которое нельзя сформулировать по-другому.
- ✓ **Предложение на естественном языке всегда перегружено побочными смыслами и дополнительной, ненужной информацией.** Даже человек иногда не может отделить в предложении нужное от ненужного.
- ✓ **Инструментов интерпретации команд на естественном языке не существует.** На данный момент уже есть программы, принимающие команды с микрофона, однако они воспринимают только простейшие команды наподобие "Открыть вкладку" или "Установить флажок", к тому же произнесенные громко и четко. Работать с ними настолько неудобно, что нужную операцию намного легче выполнить традиционными способами с помощью мыши и клавиатуры.

Дабы сообщить компьютеру, что он должен сделать, нужно пообщаться с ним на специальном языке. Программист должен написать на этом языке краткие,

недвусмысленные инструкции. Такой язык, как уже было упомянуто, называется *языком программирования*, а последовательность инструкций на языке программирования называется *программой*. Совокупность всех программ, выполняющихся на компьютере, называется *программным обеспечением*. Таким образом, программа представляет собой некоторый текст, называемый *кодом*. Код — это последовательность символов. В качестве примера ниже приведен код программы на языке Java.

```
class PayBarry {
  public static void main(String args[]) {
    double checkAmount = 1257.63;
    System.out.print("Оплатить заказ Барри Берда, ");
    System.out.print("сумма: ");
    System.out.print(checkAmount);
    System.out.println(" рублей.");
  }
}
```

Зачем писать программы

Предположим, на завтра у вас запланирована вечеринка и вы хотите приготовить на десерт что-нибудь вкусненькое. У вас есть два варианта: либо купить готовый торт или пирог, либо приготовить его самому, используя различные ингредиенты. Первый вариант займет у вас меньше времени, и, возможно, вы даже потратите меньше денег, но вы получите продукт со многими нежелательными ингредиентами, такими как пропиленгликоль, глутамат натрия, красители, ароматизаторы и т.п. Готовить торт самому сложнее и дольше, но у вас будет именно тот продукт, который вам нужен.

Сказанное выше вполне справедливо и по отношению к компьютерным программам. Вы можете либо использовать программу, написанную другими людьми, либо написать собственную. Второй вариант сложнее первого, зато у вас будет программа, хорошо приспособленная для ваших нужд.

Создание компьютерных кодов — большой глобализованный бизнес, в котором заняты сотни миллионов людей во всех уголках земного шара. Кроме того, написанием программ занимаются одиночки-профессионалы, небольшие компании, любители (в качестве хобби), а также люди, работающие в других областях и создающие несложные программы для своих нужд. Типичная большая компания состоит из отделов и команд, которые работают на компанию и создают для нее части большой программы. Однако быть членом какой-либо команды необязательно; вы можете писать программы для себя, для своих знакомых или просто для развлечения. Представьте себе что-либо полезное, что может делать компьютер. Затратив определенное время, вы сможете написать программу, заставляющую его делать это. Конечно, требуемое для этого время может быть очень большим, возможно, даже нереальным, однако многие интересные и полезные программы можно написать в течение нескольких дней или даже часов.

Немного истории

Ниже приведена краткая история современного компьютерного программирования. Справедливости ради историю программирования следовало бы начать задолго

до появления компьютеров: с Ады Лавлейс, которая создала первую в мире программу на железных пластинках для механической счетной машинки, или с И.С. Баха, который разработал алгоритм создания музыки на основе результатов бросания игральных костей. Однако нас интересует главным образом программирование современных компьютеров, а не история культуры, поэтому включим в нашу историю только высокоуровневые языки программирования, используемые на электронных компьютерах, а не в механических счетных устройствах.

- ✓ **1954–1957 годы: разработка языка FORTRAN** (FORmula TRANslation — преобразование формул).

  FORTRAN был первым высокоуровневым языком программирования (термин *высокоуровневый* означает, что язык состоит не из машинных команд, а из инструкций, понятных человеку и близких к естественному языку). Долгое время FORTRAN был единственным языком, применяемым для научных вычислений. Его долговечность удивительна: он применялся еще долгое время после того, как ушли со сцены многие более поздние языки. Этим он напоминает легендарный АН-2 ("кукурузник"), созданный более 50 лет тому назад, но продолжавший летать, когда устарели и были сняты с производства многие реактивные самолеты. Даже в наше время трансляторы FORTRAN существуют для всех современных операционных систем. Объясняется это не только уважением к почтенному прошлому, но и тем, что в исследовательских учреждениях до сих пор используется большое количество научных программ на FORTRAN.
- ✓ **1959 год: создан язык COBOL.** Буква "B" означает *business* (бизнес), и неспроста, потому что это главное, для чего был создан COBOL. Главной задачей COBOL была последовательная обработка записей. В течение нескольких лет после выпуска транслятора COBOL этот язык широко использовался для обработки коммерческих данных, однако вскоре появились реляционные базы данных, в которых были реализованы более совершенные средства обработки записей. В результате COBOL практически полностью сошел со сцены.
- ✓ **1972 год: ныне покойный Деннис Ритчи, работая на AT&T Bell Labs, создал язык C.** Этот язык оказал огромное влияние на синтаксис и структуру многих более поздних языков программирования. В частности, C++ и C# — его прямые "наследники", что видно даже из их названия, а Java и JavaScript внешне настолько напоминают C, что часто их тяжело отличить. Как и в Java, в коде C используются фигурные скобки для создания блоков, точки с запятыми для разделения инструкций, операторы `if` и `for` и т.п.

  С точки зрения функциональных возможностей на языке C можно решать все задачи, решаемые на FORTRAN, Java и других современных языках программирования. Различие между ними не в возможностях, а в удобстве и простоте работы с большими программами. В этом отношении Java значительно превосходит C.
- ✓ **1986 год: Бьярне Страуструп, опять же в AT&T Bell Labs, разработал язык C++.** В отличие от своего непосредственного предка C, язык C++ поддерживает модель объектно-ориентированного программирования (см. главу 2). Это был огромный шаг вперед.

- ✓ **23 мая 1995 года: компания Sun Microsystems выпустила первую официальную версию Java.** Фактически Java стал усовершенствованной версией C++. Главный девиз Java — "Написано раз, работает везде". Это делает Java языком, идеальным для распространения кодов через Интернет.

  Кроме того, Java оказался прекрасным языком общего назначения. С его помощью можно создавать приложения с графическим интерфейсом пользователя, управлять базами данных, писать приложения для мобильных устройств и т.п. Через пять лет после появления Java на этом языке работали уже более 2,5 миллионов разработчиков во всем мире (с тех времен у меня сохранилась футболка, на которой запечатлен этот факт).
- ✓ **Ноябрь 2000 года: Совет колледжей США объявил, что начиная с 2003 года экзамены по компьютерным технологиям будут основаны на Java.** Хотите знать, какой язык изучают университетские умники? Ну конечно же, Java!
- ✓ **2002 год: компания Microsoft объявила о создании языка C#.** Многие средства C# взяты из Java, в результате чего внешне они почти не отличаются друг от друга. Однако дальнейшее развитие этих двух конкурентов пошло разными путями. Microsoft добавила и продолжает добавлять в C# огромное количество новых модных средств, утверждая, что их наличие является преимуществом по сравнению с Java. Но в результате этого поднять со стола учебник по C# теперь может только студент атлетического телосложения. В то же время разработчики Java придерживаются консервативной стратегии развития и не торопятся добавлять новые средства, справедливо, как мне кажется, считая, что не следует засорять синтаксис языка средствами, без которых можно легко обойтись.
- ✓ **Июнь 2004 года: согласно отчету Sys-Con Media количество программистов на Java превысило количество программистов на C++ на 50%.** Даже более того — количество программистов на Java на 8% превысило их количество на C++ и C# вместе взятых! Правда, если добавить к C++ и C# программистов на Visual Basic (еще один язык от Microsoft, конкурент Java), их общее количество превзойдет число программистов на Java, но все же Java обгоняет Visual Basic по этому показателю почти в два раза.
- ✓ **Январь 2010 года: компания Oracle Corporation приобрела Sun Microsystems и включила Java в линейку продуктов Oracle.**
- ✓ **Май 2011 года: программы на Java выполняются более чем на 1,1 миллиарда настольных компьютеров во всем мире.** К этому следует добавить, что Java-программы выполняются на более чем трех миллиардах мобильных телефонов и портативных устройств. Согласно индексу TIOBE Programming Community в настоящее время Java является наиболее популярным языком программирования.

Как видите, успехи Java неоспоримы, следовательно, начав изучать Java, вы не прогадаете.

Объектно-ориентированное программирование

Однажды в три часа ночи мне снился сон об экзамене по истории, который я провалил в колледже. Преподаватель говорил мне с гипнотизирующей интонацией: "У вас есть два дня на подготовку к экзамену, но вы забудете, что нужно готовиться. Вы будете чувствовать себя виноватым, но не вспомните, по какой причине. Вина будет угнетать вас, угнетать, угнетать..."

Неожиданно зазвонил телефон, и я резко проснулся. Мне не нравятся сны об экзамене по истории, но просыпаться от телефонного звонка мне не нравится еще больше. В первый момент я хотел в резкой форме напомнить звонящему о необходимости сначала поинтересоваться, какое сейчас время в моем часовом поясе, но уронил телефон на пол и забыл сделать это. Когда я наконец нашел телефон и поднял трубку, человек на другом конце линии сообщил, что он репортер какой-то газеты. Он получил задание написать статью о Java и хочет, чтобы я в двух словах рассказал ему, что это такое.

Я сказал ему несколько слов о Java (решив, что так быстрее отвяжусь от него, чем если буду скандалить) и опять погрузился в сон. Утром я уже не мог вспомнить, что я ему сказал и не приснилось ли мне это. Но, как оказалось, не приснилось, потому что через несколько дней я получил экземпляр газеты, в которой мне бросился в глаза написанный крупными буквами заголовок: **"Барри Берд утверждает, что Java — это замечательный объектно-ориентированный язык"**.

Объектно-ориентированные языки

Существует много типов языков программирования. Приводить их классификацию здесь нет необходимости, упомяну лишь, что два наиболее распространенных типа — процедурные и объектно-ориентированные языки. Процедурные языки (например, FORTRAN) ориентируются на последовательность выполнения команд. Программа на таком языке представляет собой последовательность четких приказов компьютеру: "Сделай то, сделай это". В то же время программа на объектно-ориентированном языке состоит из объектов. Каждый объект — это набор данных и методов их обработки. Конечно, объектно-ориентированная программа также содержит последовательности команд (они находятся в методах), но при создании программы разработчик начинает с организации данных, а команды появляются позже.

Концепция объектов облегчает организацию данных и позволяет выполнять обработку части данных, не затрагивая другие данные. При изменении способов обработки данных можно воспользоваться существующими объектами, дополняя их новыми методами, а не начинать разработку "с нуля". Особенно сильно преимущества объектно-ориентированного языка по сравнению с процедурным проявляются в работе большой команды программистов при создании крупной программы. Каждый участник проекта может работать над своим объектом, не опасаясь, что кто-то изменит его данные или он сам по неосторожности испортит данные другого человека.



В середине 1950-х годов команда разработчиков создала язык программирования FORTRAN. Он был основан на том, что компьютеру давались четкие приказы, записанные в коде программы. Приказы, называемые командами или инструкциями, были, с одной стороны, похожи на слова английского языка (`IF`, `FOR`, `BEGIN` и т.п.), а с другой — были намного более четкими и однозначными, чем слова естественного языка.

На протяжении двадцати последующих лет разработчики создали много новых компьютерных языков. Многие из них были основаны на той же модели, что и FORTRAN, т.е. программа на таком языке представляла собой последовательность четких инструкций компьютеру. Такие языки называются *процедурными*. Наиболее популярным в этом ряду был язык C, название которого по странному стечению обстоятельств состояло из одной буквы. Разрабатывались также языки, отклоняющиеся от командного принципа. Например, в языках SIMULA и Smalltalk команды были отодвинуты на второй план, а главное внимание было сосредоточено на описании данных. В этих языках не принято писать "Найти заказы стоимостью более тысячи долларов. Распечатать найденные заказы". Вместо этого нужно определить, что такое заказ, какие заказы нужны и какого рода операция над ними должна быть выполнена. Такие языки в большей мере не командные, а описательные (их принято называть *декларативными*).

Приблизительно в это же время возникла и другая ветвь языков, со временем победившая в конкурентной борьбе, — объектно-ориентированные языки. Как и в процедурных языках, программа на объектно-ориентированном языке представляет собой последовательность команд, однако на этом их сходство заканчивается. В объектно-ориентированной программе все вращается вокруг данных. Объекты — это наборы данных и методов работы с ними. Программа состоит из объектов, а команды, хоть и выполняются, играют вспомогательную роль. Со временем опыт показал, что объектно-ориентированные языки — прекрасный инструмент для создания сложных программ большими коллективами разработчиков. Это объясняется следующими причинами.

- ✓ Программист должен больше внимания уделять не командам, а данным, потому что проблема всегда скрывается в данных, а команды — это всего лишь инструмент решения задачи.
- ✓ Объекты позволяют разделить работу над большой программой. Каждый программист может работать над своим объектом, не мешая другим программистам.
- ✓ Принципы объектно-ориентированного программирования позволяют многократно использовать одни и те же объекты, часто даже в разных программах. Конечно, FORTRAN также позволяет использовать одну и ту же процедуру в разных местах, но процедура, в отличие от объекта, слишком "хрупкая". Попав в другое место, она может "не вписаться в коллектив" и потерпеть крах.

В 1970-е годы декларативные языки, такие как SIMULA и Smalltalk, сошли со сцены и остались лишь в компьютерных журналах, обсуждаемые своими упрямыми сторонниками. Между тем процедурные языки типа FORTRAN размножались, как кролики, пока в 1986 году Бьярне Страуструп не создал C++, который стал первым мощным объектно-ориентированным языком. Этот язык быстро завоевал популярность благодаря тому, что в нем использовались синтаксис и терминология распространенного в то время языка C. К старому доброму C была лишь добавлена объектно-ориентированная структура. Многие компании немедленно отвернулись от традиционных FORTRAN и C, приняв в качестве стандарта программирования язык C++.

Однако C++ обладает рядом существенных недостатков. Используя C++, можно легко обойти стороной все объектно-ориентированные принципы и написать программу в стиле FORTRAN или C.

Начиная писать на C++ программу для решения упомянутой выше задачи выбора заказов, можно пойти по одному из двух путей.

- ✓ Можно начать непосредственно с команды поиска заказов стоимостью более тысячи долларов, а затем создать команду распечатки найденных заказов.
- ✓ Можно начать с определения объекта "заказ". Это и будет объектно-ориентированный подход. Вы должны определить, какие данные характеризуют заказ (стоимость, время размещения, время выполнения, налоговые отчисления и т.п.) и как обрабатываются эти данные.

Одни программисты считают, что C++ предоставляет лучшие средства обоих подходов, другие же возражают (и я отношусь к ним), что современные программы не должны создаваться в стиле FORTRAN или C, поэтому нужно исключить возможность написания программ в этом стиле. Простую программу легче и быстрее написать в процедурном стиле, и многие программисты так и делали бы. Но такая программа плохо вписывается в современную объектно-ориентированную среду.

В 1995 году Джеймс Гослинг, работая в то время на Sun Microsystems, создал язык Java. Он позаимствовал из C++ синтаксис и терминологию, но безжалостно выбросил все командные средства. Затем он добавил в Java средства, облегчающие применение принципов объектно-ориентированного программирования. В результате получился чистый и строгий язык. При программировании на Java у вас нет выбора: язык вынуждает вас работать только с объектами, как и должно быть в современном объектно-ориентированном мире компьютерных программ.

Объекты и классы

В объектно-ориентированном языке для организации данных используются объекты и классы.

Предположим, вам нужно написать программу, сопровождающую строительство коттеджей для нового жилищного товарищества. Коттеджи немного отличаются друг от друга. Разными могут быть такие параметры, как цвет фасада, внутренняя отделка, стиль кухни и т.п. В вашей объектно-ориентированной компьютерной программе каждый коттедж — это объект.

Однако коттеджи не только отличаются друг от друга. Они обладают одним и тем же набором изменяемых параметров, например каждый коттедж характеризуется цветом фасада, внутренним оформлением, стилем кухни и т.п. В объектно-ориентированной программе необходим главный список, содержащий все характеристики коттеджей, т.е. объектов. Такой список называется *классом*.

Возможно, лучше было бы назвать такой стиль программирования не объектно-ориентированным, а программированием на основе классов и объектов. Слово "классы" я поместил на первое место не случайно, потому что понятие класса важнее понятия объекта. Представьте себе строительную площадку. В ее углу посреди большой лужи стоит трейлер, в котором находятся контора прораба и чертеж коттеджа. Чертеж — это класс, поскольку он содержит исчерпывающую информацию о каждом коттедже и список изменяемых параметров.

Аналогия между чертежом и классом не ограничивается наличием списка параметров. Подобие между ними намного более глубокое. Чертеж создается один раз. На его основе подрядчик строит, например, десять коттеджей. То же самое справедливо и

для классов. Сначала программист один раз создает код класса, а затем компьютер по мере выполнения программы многократно создает объекты данного класса.

В этом заключена суть взаимоотношений между классами и объектами: программист определяет класс, а затем на основе определения класса компьютер создает индивидуальные объекты. Часто объекты называются *экземплярами класса*.

Преимущества объектно-ориентированного подхода

Предположим, собрание акционеров строящегося городка решило, что одна половина коттеджей должны быть трехкомнатными (весь второй этаж — одна комната), а другая половина — четырехкомнатными (на втором этаже две комнаты).

Если проводить аналогию с процедурными языками FORTRAN и C, вам пришлось бы иметь два комплекта чертежей и спецификаций, два списка комплектующих, два набора сопроводительных документов и т.д. Если понадобится что-то изменить на первом этаже, придется вносить одинаковые изменения в оба набора чертежей, спецификаций и документов.

В то же время классы позволяют применить принцип *наследования*, заключающийся в следующем. Вы создаете *базовый* класс — чертежи первого этажа и крыши, одинаковые для трех- и четырехкомнатных коттеджей. Затем вы создаете два *производных* класса и включаете в них только то, чего нет в базовом классе, т.е. чертеж второго этажа. Производный класс наследует все характеристики базового класса, поэтому оба производных класса будут содержать все чертежи, необходимые для построения коттеджа.

Назовем базовый класс `Коттедж`, а два производных класса — `ТрехкомнатныйКоттедж` и `ЧетырехкомнатныйКоттедж`. Согласно терминологии объектно-ориентированного программирования классы `ТрехкомнатныйКоттедж` и `ЧетырехкомнатныйКоттедж` наследуют класс `Коттедж` (рис. 1.1).

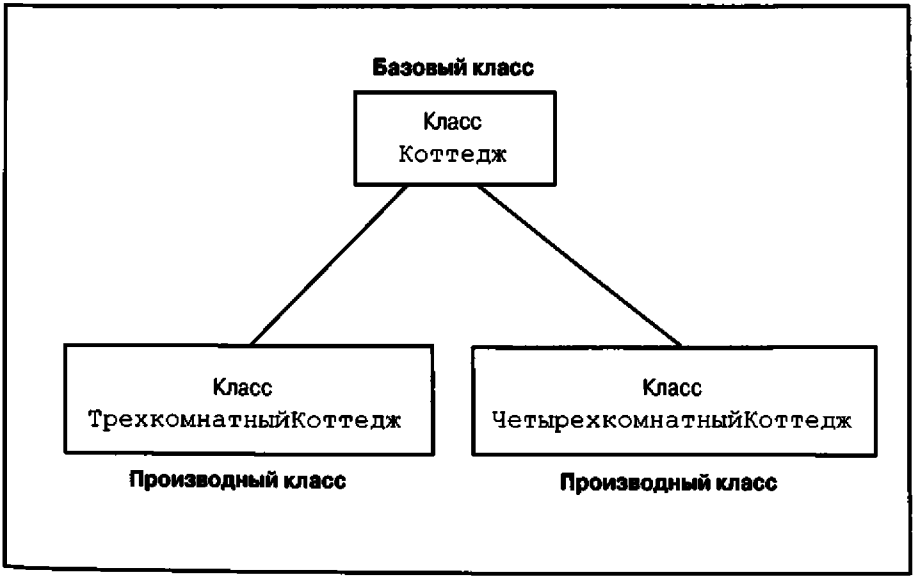


Рис. 1.1. Принцип наследования

Конечно, ваши коллеги-строители хотят узнать о вашей замечательной идее относительно того, как сэкономить бумагу для чертежей, и в тот момент, когда они собрались вокруг вас, чтобы услышать подробности, вы бросаете очередную идею: "Создавая производные классы на основе базового, мы можем в будущем повторно использовать все его средства. Если в будущем кто-то захочет спроектировать пятикомнатный коттедж, он может сэкономить 90% рабочего времени, создав проект на базе класса `Коттедж`".

"Но что делать, — спрашивает один из ваших коллег, — если понадобится другой план первого этажа? Нам придется выбросить весь класс `Коттедж` и начать чертить с нуля, а это стоит немалых денег".

Вы сообщаете ему уверенным тоном: "Нам не придется изменять класс `Коттедж`. Если кто-то захочет иметь джакузи в комнате первого этажа, мы создадим производный класс, который наследует класс `Коттедж` и добавляет джакузи в требуемое место". Назовем этот класс `КоттеджДжакузи`. Его единственное отличие от классов `ТрехкомнатныйКоттедж` и `ЧетырехкоманатныйКоттедж` состоит в том, что он не только наследует и дополняет базовый класс, но и *переопределяет* одно из его средств, а именно — структуру комнаты на первом этаже.

Объектно-ориентированные языки появились не по чьей-то прихоти. Их появление было обусловлено тем, что в индустрии программирования наступил глубокий, затяжной кризис. Сначала программисты долго писали код, а затем, когда обнаруживалось, что нужно что-то изменить, им приходилось выбрасывать написанный код и начинать все с нуля. Оказалось, что, когда в написанный код необходимо внести малейшее изменение, его невозможно использовать повторно, а легче написать код с самого начала. Это была не единственная причина кризиса в программировании. Другая, не менее важная причина состояла в том, что при создании большой программы люди неумышленно переопределяли данные друг друга, причем выяснить, какая инструкция ошибочно изменила некоторое значение, было очень тяжело, потому что это могло произойти в любом месте огромной программы. Для создания большой программы жизненно необходимы средства изоляции ее частей друг от друга. Все эти проблемы были решены в объектно-ориентированных языках.

Наглядное представление классов и объектов

Создавая программу на Java, вы постоянно будете работать с классами и объектами. Эти две концепции очень важны, поэтому в данной главе я подвожу вас к ним с разных сторон с помощью ряда аналогий.

Закройте на минутку глаза и подумайте о том, что означает фраза "некоторый предмет является стулом".

У стула есть сиденье, спинка и ножки. Каждое сиденье имеет форму, цвет, степень мягкости и т.п. Это свойства, которыми обладает стул. Слово "стул", как и класс — это не сам стул, а мысленная модель или понятие стула. В терминах объектно-ориентированного программирования класс `Стул` — это не предмет, а описание или представление класса предметов.

Теперь оторвите взгляд от книги и посмотрите вокруг себя. Вы видите в комнате несколько стульев, каждый из которых является конкретным объектом. Каждый объект — это реализация идеи стула, а говоря языком объектно-ориентированного программирования — экземпляр класса `Стул`.



Класс — это не набор предметов, а абстрактная идея, отображающая конкретные предметы. Когда я говорю о классе стульев, расположенных в вашей комнате, я имею в виду, что в комнате есть предметы со спинками, сиденьями и ножками, обладающие некоторым цветом, мягкостью и другими характеристиками, определенными в классе `Стул`. Разные стулья обладают разным цветом, но в классе `Стул` определен не цвет, а параметр цвета, который у разных экземпляров может иметь разные значения, т.е. конкретные цвета (зеленый, синий и т.д.).

В большинстве случаев имеет смысл представлять себе объект как конкретный экземпляр класса. Фактически официальная терминология совпадает с этим представлением. Когда вы пишете на Java программу, в которой используется класс `Стул`, каждый конкретный объект стула является экземпляром класса `Стул`.

Рассмотрим еще одно наглядное представление понятия класса. Предположим, в табл. 1.1 содержится информация о ваших банковских счетах.

Таблица 1.1. Банковские счета

Номер счета	Тип счета	Баланс
16-13154-22864-7	Чек	174,87
1011 1234 2122 0000	Кредит	–471,03
15-17238-13344-7	Депозит	247,38

Представляйте себе заголовки столбцов как класс, а каждую строку — как объект, т.е. экземпляр класса. Класс `Счет` исчерпывающе описывается заголовками столбцов данной таблицы.

Как вы помните, класс — это список параметров объекта. Согласно табл. 1.1 каждый счет имеет следующие параметры: номер, тип и баланс. Говоря терминами объектно-ориентированного программирования, у каждого объекта `Счет` (т.е. экземпляра класса `Счет`) есть значения номера, типа и баланса. Например, нижняя строка таблицы — это объект `Счет` с номером счета, равным `15-17238-13344-7`. Баланс этого же объекта равен `247,38`. Открывая новый счет, вы создаете новый объект этого же класса, и в таблице появляется еще одна строка. Новый счет будет еще одним экземпляром класса `Счет`.

Что дальше

В данной главе я привел самое общее описание языка Java и принципов объектно-ориентированного программирования. Читать общее описание имеет смысл, только начав знакомиться с предметом, однако оно не даст настоящего понимания предмета до тех пор, пока вы не углубитесь в подробности. Им посвящены следующие главы. Переверните страницу и читайте дальше.

Глава 2

Разработка программного обеспечения

В этой главе...

- Быстрое начало
- Что должно быть установлено на компьютере

Наилучший способ ознакомиться с Java — создать программу на компьютере. Работая с Java, вы пишете, тестируете и выполняете программу. В этой главе мы рассмотрим общий процесс разработки программного обеспечения на любом компьютере и под управлением любой операционной системы, будь то Windows, Mac, Linux или Solaris. Специфические особенности работы в конкретной операционной системе в данной главе не рассматриваются.



В приложении А приведены подробные инструкции по загрузке и установке виртуальной машины Java, пакета инструментов разработки и рабочей среды Eclipse в разных операционных системах.

Быстрое начало

Если вы опытный пользователь (что бы это ни означало) и слишком нетерпеливы, чтобы читать подробные инструкции, приведенные в приложениях А и Б, можете установить программное обеспечение, необходимое для разработки программ на Java, пользуясь приведенной ниже краткой инструкцией. Она пригодна для большинства компьютеров и операционных систем, хотя в некоторых операционных системах отдельные операции могут выполняться немного иначе.

Чтобы подготовить компьютер к созданию программ на Java, выполните следующие операции.

✓ **Посетите официальный сайт Oracle.**

Адрес официального сайта — `http://oracle.com/technetwork/java/javase/downloads/index.html`. Загрузите и установите пакет JDK (Java Development Kit — пакет инструментов разработки Java). Обратите внимание: вам нужен именно JDK, а не JRE (Java Runtime Environment — среда выполнения Java). Пакет JRE входит в JDK. Он предназначен только для выполнения программ Java, но не для их разработки.

Не посещайте сайт `http://java.com/ru`, потому что с него можно загрузить только JRE.

✓ **(Необязательный этап.) Откройте сайт `http://java.sun.com/javase/downloads/`.**

Загрузите и установите документацию Java SE, содержащую довольно полезный справочник по библиотечным классам Java.

✓ **Посетите сайт Eclipse.**

Открыв страницу `http://eclipse.org/downloads`, загрузите и установите программу Eclipse (это интегрированная среда разработки приложений на Java). Для этого скопируйте на свой жесткий диск архив, содержащий программу Eclipse. Установка состоит всего лишь в распаковке архива в произвольное место на диске, желательно в папку `C:\Program Files`. Запустите программу `eclipse.exe` (в Windows) — и на экране появится рабочая среда Eclipse.



Более подробная инструкция по работе с Eclipse приведена в приложении Б.



Страница загрузки Eclipse предлагает несколько пакетов, включая Eclipse Classic, Eclipse for Java EE, Eclipse for JavaScript и др. Для работы с примерами данной книги вам необходим сравнительно небольшой пакет Eclipse IDE for Java Developers. Пакет Eclipse for Java EE содержит довольно много дополнительных инструментов, которые вам сейчас не нужны. Можете загрузить и его, но на панели инструментов Eclipse будет много "лишних" кнопок, которые будут отвлекать вас.

✓ **Протестируйте среду разработки.**

- Запустите программу Eclipse.
- Создайте проект Java.
- В проекте создайте класс `Displayer`.
- В окне редактора Eclipse введите в файл `Displayer.java` код, приведенный в листинге 3.1.
- Запустите файл `Displayer.java` на выполнение и убедитесь в том, что на консоли появилось сообщение `Мне нравится Java!`. Если оно появилось, значит, рабочая среда Eclipse функционирует нормально.

При первом запуске Eclipse появится окно с предложением установить рабочее пространство. Рабочим пространством является указанная вами папка, в которой находятся проекты Eclipse. Можете установить в качестве рабочего пространства папку с примерами данной книги, которую можно загрузить с веб-страницы книги на сайте издательства. Вложенная папка `.metadata` указывает на то, что родительская папка является рабочим пространством Eclipse. Выбрав эту папку в качестве рабочего пространства, вы сможете легко запустить любой пример книги, открыв его проект в окне Eclipse.

Что должно быть установлено на компьютере

Однажды я встретил слесаря, и он показал мне инструменты, которые он использует для создания других инструментов. Я был в восторге, потому что сразу заметил аналогию с программированием.

Программист использует для создания программ другие, уже существующие программы в качестве инструментов. Существующие и создаваемые программы решают совершенно разные задачи. Например, программа на Java (создаваемая вами) может быть предназначена для отслеживания заказов. Для ее создания вы используете (вернее, будете использовать в ближайшем будущем) существующую программу, которая помогает вам вводить код, находить ошибки, запускать создаваемую программу на выполнение и т.д. С ее помощью нельзя отслеживать заказы, но можно создать программу, отслеживающую заказы.

Сколько инструментов необходимо для создания программ на Java? Как новичку вам пока что необходимы всего три инструмента.

- ✓ **Компилятор.** Это программа, которая принимает код на Java (его называют *исходным кодом*) и превращает его в *байтовый код*, который может выполняться на компьютере с помощью виртуальной машины Java. Компилятор входит в пакет JDK, который вы загрузили и установили в начале главы.
- ✓ **Виртуальная машина Java.** Это программа, которая выполняет байтовый код. Виртуальная машина находится в пакете JDK или JRE (см. выше). Еще раз напоминаю, что для разработки программ нужен пакет JDK, потому что с помощью пакета JRE можно только выполнять программы Java.
- ✓ **Интегрированная среда разработки.** Это программа, которая помогает редактировать, отлаживать и запускать исходный код Java. В принципе, это можно делать с помощью любого текстового редактора (например, блокнота Windows), но в специальной рабочей среде все намного удобнее, легче и быстрее. В данной книге в качестве интегрированной среды разработки используется программа Eclipse.

В Интернете есть довольно много платных и бесплатных инструментов, предназначенных для разработки программ на Java. Наиболее мощные платные инструменты предоставляют графическую среду разработки, в которой можно компоновать пользовательский интерфейс разрабатываемой программы путем перетаскивания элементов управления с помощью мыши. Программа Eclipse не предоставляет графическую среду, однако в ней есть много средств отладки, которых нет в графических средах. Например, в Eclipse есть точки прерывания, а в JDeveloper (это бесплатная графическая среда разработки) их нет. Поэтому часто наблюдается парадоксальная ситуация: большинство профессиональных разработчиков используют более простую среду Eclipse, считая, что средства тестирования и отладки важнее, чем возможность перетаскивать элементы интерфейса мышью, а многие начинающие программисты предпочитают графические среды, потому что не умеют самостоятельно организовывать прослушивание событий или создавать гибкую компоновку элементов интерфейса.

Оставшаяся часть главы посвящена описанию компиляторов, интегральных средств разработки и виртуальной машины Java.

Что такое компилятор

Компилятор принимает написанный вами код Java и преобразует его в байтовый код, который может выполняться на компьютере с помощью виртуальной машины Java.

Предположим, программа должна, просмотрев список комнат в гостинице и количество постояльцев в каждой комнате, найти свободную комнату. В листинге 2.1 приведен фрагмент программы на Java, решающий эту задачу. Обратите внимание на то, что в идентификаторах можно использовать кириллические буквы, что значительно облегчает понимание программы.

Листинг 2.1. Поиск пустой комнаты

```
//Это фрагмент программы на Java
номерКомнаты = 1;
while (номерКомнаты < 100) {
  if(количествоПостояльцев[номерКомнаты] == 0) {
     System.out.println("Комната " + номерКомнаты + " свободна.");
     System.exit(0);
     } else {
         номерКомнаты++;
     }
}
System.out.println("Свободных комнат нет.");
```

Код, показанный в листинге 2.1, ищет свободную комнату в небольшой гостинице, в которой есть только комнаты 1–99. Выполнить код в том виде, в котором он записан в листинге 2.1, нельзя. Нужно добавить еще несколько строк, начинающих и завершающих программу. Но пока что будем считать, что эти строки не нужны, дабы они не отвлекали вас от основной задачи. Сейчас наша задача состоит в том, чтобы понять назначение кода. Для этого запишем все операции, которые он выполняет, на русском языке.

```
Присвоить номеру текущей комнаты значение 1.
Просмотреть комнаты с номерами меньше 100.
  Посмотреть, сколько постояльцев сейчас проживает
       в текущей комнате.
  Если количество постояльцев равно 0,
      сообщить о том, что комната свободна,
      и завершить выполнение программы.
  В противном случае
      увеличить номер комнаты на 1
      и перейти в начало цикла.
Сообщить об отсутствии свободных комнат.
```

Если вы пока что ничего не поняли из листинга 2.1, не беспокойтесь: спустя короткое время вы сможете читать код Java почти так же легко, как текст на русском языке. Код, приведенный в листинге 2.1, называется *исходным кодом Java*.

Однако компьютер не понимает и не может выполнить исходный код. Понять этот код может только компилятор, который преобразует исходный код в *байтовый код*. В листинге 2.2 показан байтовый код, полученный в результате трансляции исходного кода, приведенного в листинге 2.1.

Листинг 2.2. Байтовый код, полученный из кода в листинге 2.1

```
aload_0
iconst_1
putfield Hotel/номерКомнаты - I
goto 32
aload_0
getfield Hotel/количествоПостояльцев [I
aload_0
getfield Hotel/номерКомнаты - I
iaload
ifne 26
getstatic java/lang/System/out Ljava/io/PrintStream;
new java/lang/StringBuilder
dup
ldc "Комната "
invokespecial java/lang/StringBuilder/<init>(Ljava/lang/String;)V
aload_0
getfield Hotel/номерКомнаты - I
invokevirtual java/lang/StringBuilder/append(I)Ljava/lang/
StringBuilder;
ldc " свободна."
invokevirtual
   java/lang/StringBuilder/append(Ljava/lang/String;)Ljava/lang/
StringBuilder;
invokevirtual java/lang/StringBuilder/toString()Ljava/lang/String;
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
iconst_0
invokestatic java/lang/System/exit(I)V
goto 32
aload_0
dup
getfield Hotel/номерКомнаты - I
iconst_1
iadd
putfield Hotel/номерКомнаты - I
aload_0
getfield Hotel/номерКомнаты - I
bipush 100
if_icmplt 5
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "Свободных комнат нет."
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
return
```

Как видите, роли поменялись: компьютер понимает байтовый код, а человек не понимает. Но вам и не нужно понимать байтовый код. Его понимают только разработчики компиляторов. Когда вы создаете программу Java, вы пишете исходный код как в листинге 2.1. Затем вы пропускаете его через компилятор, на выходе которого получаете байтовый код. Иными словами, компилятор принимает код, который вы понимаете (листинг 2.1), и преобразует его в код, понятный для компьютера (листинг 2.2).



Предположим, вы поместили исходный код, приведенный в листинге 2.1, в файл `Hotel.java`. Тогда компилятор поместит байтовый код в файл `Hotel.class`. Если вы любопытны, можете посмотреть содержимое файла `Hotel.class` с помощью какого-нибудь текстового редактора, но вы не увидите ничего похожего на листинг 2.2. Вы увидите странные символы — квадратики, точки, стрелочки и т.п. Дело в том, что компилятор не только преобразует исходный код в байтовый, но и упаковывает байтовый код. Получается уже не байтовый, а *двоичный* код. Но поскольку этот факт ни на что не влияет (двоичный код автоматически преобразуется в байтовый), принято считать, что в файлах `.class` находится байтовый код. К тому же в данном случае двоичный код — всего лишь представление байтового кода. Чтобы увидеть байтовый код в том виде, в каком он показан в листинге 2.2, откройте файл `.class` с помощью программы Java Bytecode Editor (`http://cs.ioc.ee/~ando/jbe`).



Ни один разработчик программ Java не пишет байтовый код. С ним работают только разработчики компиляторов. Единственная причина, по которой можно посмотреть содержимое файла `.class`, — из любопытства. Каждому человеку интересно взглянуть на то, что делается под капотом или за кулисами, т.е. на то, что от него пытаются скрыть.

Что такое виртуальная машина Java

В предыдущем разделе говорилось, что компьютер не понимает исходный код и понимает только байтовый код. В данном разделе пришло время уточнить, что имеется в виду. Ваш компьютер, конечно же, понимает как исходный, так и байтовый код, потому что на нем установлен компилятор. Но компилятор установлен только на вашем компьютере, а на компьютерах пользователей его нет. На них установлена *виртуальная машина Java*. Кроме того, важно учитывать, что компьютер выполняет байтовый код не сам, а с помощью виртуальной машины Java. В разных компьютерах могут быть установлены процессоры разных типов. Кроме того, на компьютерах используются разные операционные системы, каждая из которых выполняет программы по-своему.

Строго говоря, сам компьютер выполняет не исходный и не байтовый, а *машинный* код, содержащий команды процессора, установленного в компьютер. Таким образом, программа проходит следующие этапы преобразования: вы пишете исходный код, компилятор преобразует его в байтовый код, виртуальная машина преобразует байтовый код в машинный, и, наконец, процессор выполняет машинный код.

Зачем столько преобразований? Рассмотрим следующую гипотетическую ситуацию. Предположим, вы работаете на компьютере с устаревшим процессором Pentium и операционной системой Linux. Ваш друг работает с Linux, но с процессором PowerPC.

В листинге 2.3 приведен машинный код программы, выводящей на экран фразу `Привет всем!`. Этот машинный код выполняется на процессоре Pentium в операционной системе Linux.

Листинг 2.3. Машинный код для процессора Pentium

```
.data

msg:
   .ascii "Привет всем!\n"
   len = . - msg

.text
.global _start

_start:
   movl $len,%edx
   movl $msg,%ecx
   movl $1,%ebx
   movl $4,%eax
   int $0x80

   movl $0,%ebx
   movl $1,%eax
   int $0x80
```

В листинге 2.4 содержится машинный код этой же программы, но предназначенной для процессора PowerPC под управлением Linux.

Листинг 2.4. Машинный код для процессора PowerPC

```
.data

msg:
   .string "Привет всем!\n"
   len = . - msg

.text
.global _start

_start:
   li 0,4
   li 3,1

   lis 4,msg@ha
   addi 4,4,msg@l
   li 5,len
   sc

   li 0,1
   li 3,1
   sc
```

Код листинга 2.3 без проблем выполняется на процессоре Pentium, но на процессоре PowerPC он не работает. И наоборот, код листинга 2.4 работает на PowerPC, но не работает на Pentium. Следовательно, если бы вы создали не байтовый, а машинный код, вам пришлось бы создать две аналогичные программы: одну — для своего компьютера, а другую — для компьютера вашего друга.

Но и это еще не все. Зайдя в гости к своей кузине, вы увидите, что в ее компьютере установлен процессор Pentium (как и в вашем), но компьютер работает под управлением операционной системы не Linux (как у вас), а Windows. Поэтому ни та, ни другая программа на ее компьютере не заработают, т.е. нужны уже три программы.

Но и это еще не все! Вы ведь пишете программу не только для своего друга и кузины, а для всего мира, в котором намного больше типов процессоров и операционных систем. Как же избежать полного хаоса?

Оказывается, с помощью байтового кода можно не только избежать хаоса, но и реализовать главный принцип Java: "Написано раз, работает везде". Согласно этому принципу один и тот же байтовый код пригоден для выполнения на любом процессоре и под управлением любой операционной системы. После того как вы напишете исходный код на Java и скомпилируете его в байтовый код, он без проблем будет выполняться на любом компьютере, на котором установлена виртуальная машина Java, даже на крошечном мобильном устройстве (естественно, если создаваемое вашей программой окно поместится на небольшом экране мобильного устройства).



Каким бы важным ни был байтовый код, вы можете прожить счастливую жизнь, ни разу на него не взглянув. Он вам не нужен. Компилятор создает байтовый код, виртуальная машина Java выполняет его, а вам с ним делать нечего. В следующих главах будет показано, что большую программу, состоящую из многих классов, рекомендуется упаковать в архивный файл JAR, в результате чего байтовый код будет спрятан от любопытных взглядов еще глубже.

Предположим, у вас есть файл с байтовым кодом, созданный на компьютере Windows. Вы можете скопировать этот файл на компьютер под управлением любой операционной системы (Mac, Linux, Solaris) и выполнить его без малейшего изменения. Такая программа называется *кроссплатформенной*. Кроссплатформенность — огромное преимущество программ Java, которого лишены многие другие системы программирования. Например, программы Windows работают только в операционной системе Windows, а программы Linux — только в операционной системе Linux. В некоторой степени кроссплатформенность присуща программам .NET, но переносить в другую операционную систему без изменений можно только простые программы .NET. При усложнении программы .NET она перестает быть кроссплатформенной (в значительной степени это объясняется зловредными происками компании Microsoft, которая хочет, чтобы все работали только в Windows).

Что делает байтовый код Java таким универсальным? Виртуальная машина Java. Это один из трех инструментов, которые нужно иметь на компьютере, чтобы разрабатывать программы Java, и единственный инструмент, необходимый для выполнения программ Java.

Представьте себе пленарное заседание Генеральной Ассамблеи ООН. На нем по очереди выступают делегаты от приблизительно 150 стран, причем каждый на своем языке. Чтобы перевести речь каждого выступающего непосредственно на каждый другой язык, нужно иметь в штате более двадцати тысяч переводчиков! Конечно, даже ООН не может позволить себе содержать такой огромный штат переводчиков. Чтобы уменьшить их количество, перевод выполняется в два этапа: сначала речь переводится на английский язык, а с английского — на язык слушателя. Такой простой прием позволяет уменьшить количество переводчиков в десятки раз. Этот же прием используется и в инфраструктуре Java (рис. 2.1). На какой бы платформе вы ни разрабатывали программу, результатом будет байтовый код Java. В свою очередь, на какой бы платформе ни выполнялась программа, байтовый код будет преобразован в машинный код данной платформы.

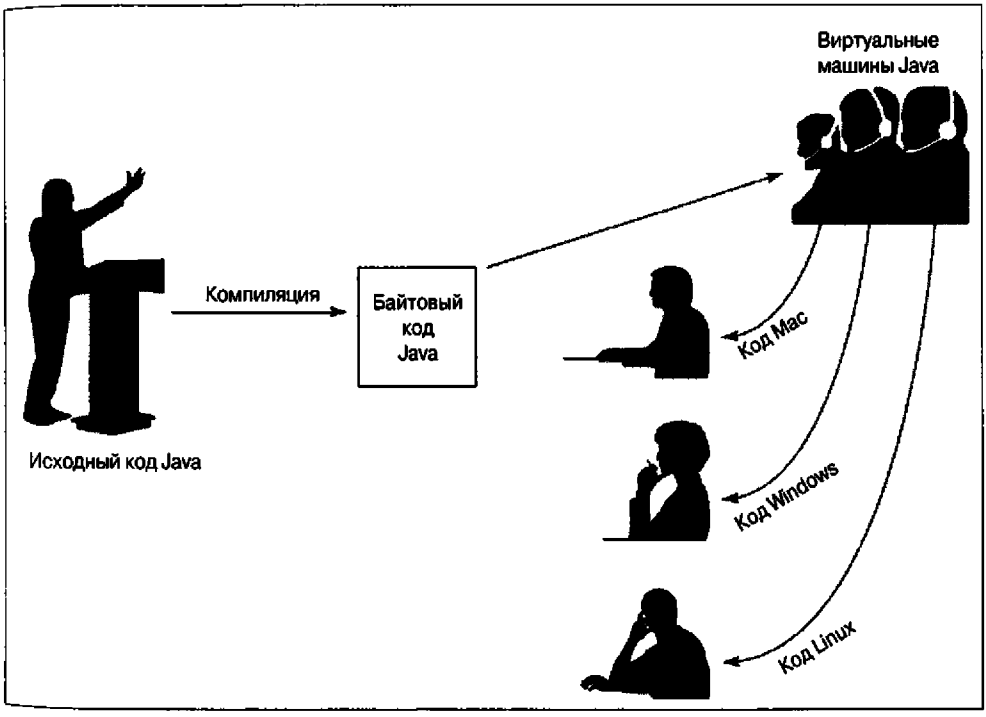


Рис. 2.1. Воображаемое пленарное заседание Генеральной Ассамблеи ООН

Еще раз взгляните на рис. 2.1. Ни выступающий, ни слушающие не обязаны знать промежуточный язык (английский). Каждому из них достаточно знать лишь собственный язык. Аналогично этому компьютеру необязательно понимать байтовый код, потому что виртуальная машина Java преобразует его в машинный код данной платформы. Поэтому программа Java может выполняться на любом компьютере, но для этого нужно установить на нем виртуальную машину Java.

Посетив сайт Oracle, чтобы найти нужные инструменты, вы увидите на нем множество странных названий типа JDK, JRE, SDK, JavaFX и т.д. Однако не пугайтесь, вам сейчас нужен только пакет JDK.

- ✓ **JDK** (Java Development Kit — набор инструментов разработки Java) содержит виртуальную машину, компилятор и другие средства разработки. SDK (Software Development Kit — пакет инструментов разработки программного обеспечения) — это устаревшее название пакета JDK. На других платформах термин "SDK" продолжает широко применяться.
- ✓ **JRE** (Java Runtime Environment — среда выполнения Java). Это, собственно, и есть виртуальная машина Java.

Чтобы выполнять на компьютере программы Java, нужно установить на нем пакет JRE, а чтобы разрабатывать программы Java, нужно установить на компьютер пакет JDK. Кроме инструментов разработки, он содержит виртуальную машину Java (т.е. JRE). Следовательно, установив JDK, вы можете как разрабатывать, так и выполнять программы Java.

Нумерация версий Java немного запутанная. Неизменным остается только название "Java", а названия пакетов по непонятным причинам изменяются. Рассмотрим хронологию версий по порядку.

- ✓ Java JDK 1.0 (1996).
- ✓ Java JDK 1.1 (1997).
- ✓ Java 2 SDK, 1.2 (1998).
  В 1998 году компания Sun Microsystems добавила цифру 2, которая означает новую версию языка Java. При этом нумерация версий пакета инструментов была продолжена без изменений. Зачем "JDK" переименовали в "SDK", нам неизвестно. Наверное, для того, чтобы потом можно было переименовать обратно в "JDK".
- ✓ Java 2 SDK, 1.3 (2000).
- ✓ Java 2 SDK, 1.4 (2002).
- ✓ Java 2 JDK, 5.0 (2004).
  В 2004 году Sun вернулась к названию "JDK" и частично отказалась от нумерации 1.*x*. Слово "частично" означает, что, кроме двойки в Java 2, пакет JDK теперь имеет два номера версии: номер версии продукта 5.0 и номер разработки 1.5.0. Поэтому, говоря о версии JDK, можно называть ее версией 5.0 или 1.5.0. Но эта путаница продолжалась недолго, и уже в следующей версии ненужные нули, единицы и точки были отброшены.
- ✓ Java 6 JDK (2006).
  В 2006 году компания Sun удалила лишнее из номера версии, но во многих местах продолжала использоваться старая нумерация, т.е. эта версия называлась Java 1.6.0.
- ✓ Java 6 Update 1 (2007).
  В 2010 году компания Oracle Corporation приобрела компанию Sun Microsystems. Нумерация обновлений продолжалась и на момент написания данной книги дошла до Java 6u24.
- ✓ Java 7 (2011).
  Пока что версия 7 еще "сырая", поэтому для работы с примерами книги рекомендую использовать Java 6.

Большинство примеров книги выполняются только в версии Java 5.0 и более поздних. Однако не беспокойтесь о версиях. Синтаксис языка остался практически неизменным с момента появления Java 2. Изменения касаются главным образом стандартных библиотечных классов, причем настолько специфичных, что в данной книге они не упоминаются.

Продолжим аналогию с переводом на пленарном заседании Генеральной Ассамблеи ООН. У каждой делегации есть свой переводчик с промежуточного языка. Аналогично этому на компьютере Windows есть компилятор байтового кода в машинный код Windows, на компьютере Linux — в код Linux, на компьютере Solaris — в код Solaris и т.д.

Таким образом, для каждой платформы (типа операционной системы и модели процессора) нужна своя виртуальная машина Java. Она служит посредником между универсальным байтовым кодом и конкретной платформой. Виртуальная машина читает байтовый код, преобразует его в машинный код данной платформы и отдает машинный код операционной системе, которая немедленно его выполняет.

Процесс разработки

При создании программы Java вы многократно проходите по одним и тем же этапам, показанным на рис. 2.2.

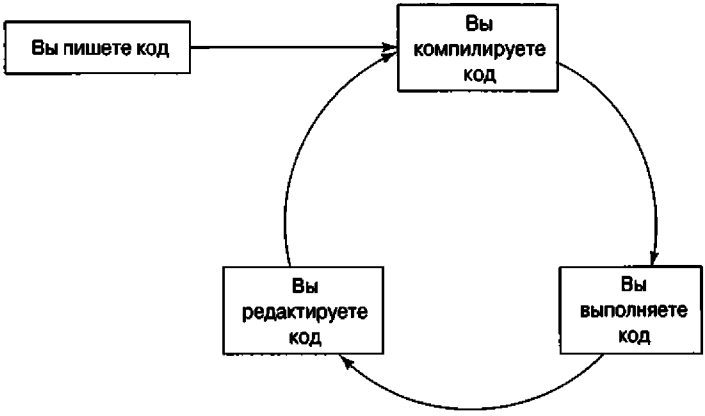


Рис. 2.2. Этапы разработки программного обеспечения

В первую очередь вы пишете код будущей программы. Написав первый черновик, вы компилируете код, выполняете его и смотрите, что получилось. Конечно, с первого раза ничего хорошего не получилось. Поэтому вы изменяете код и опять компилируете и выполняете его. Во время работы над программой вы повторяете этот процесс многократно.

В интегрированной среде разработки Eclipse этапы компиляции и выполнения — это один щелчок мышью на кнопке Run (Выполнить). Однако редактирование кода — это не один щелчок мышью, а сложная работа, требующая времени и концентрации внимания.



Никогда не впадайте в уныние, если код не работает. Даже профессиональные программисты проходят цикл, показанный на рис. 2.2, сотни раз, пока код не будет работать, как нужно. Неустанно экспериментируйте с кодом, пробуйте разные варианты, проверяйте работу кода с разными исходными данными. Научиться работать с кодом — ваша самая важная задача.

На рис. 2.2 написано: "Вы компилируете код, вы выполняете код..." Однако на самом деле это делаете не вы, а компьютер по вашей команде. Ваша работа состоит в редактировании кода. Компьютер не сделает это вместо вас, даже если вы ему прикажете. Уточненный цикл разработки показан на рис. 2.3.

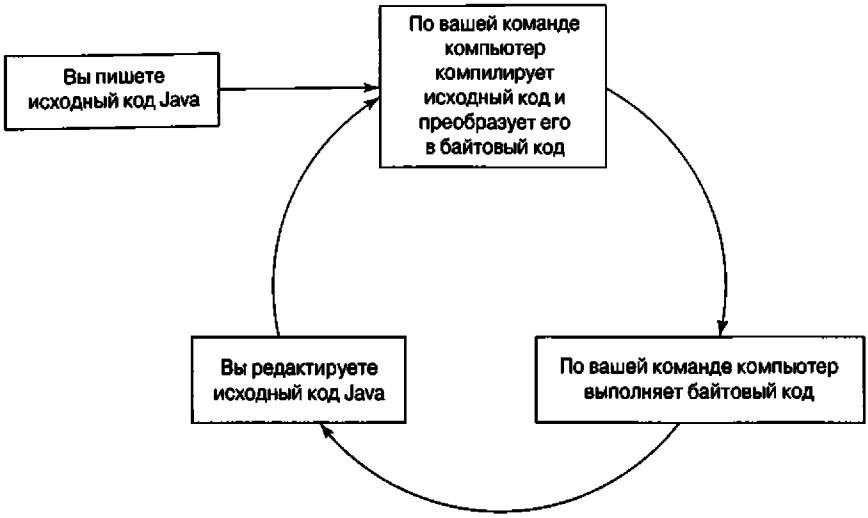


Рис. 2.3. Уточним, кто и что делает в цикле разработки



Для большинства людей рис. 2.3 содержит слишком много информации. Когда я щелкаю на кнопке Run, я не должен помнить о том, что компьютер компилирует код, создает байтовый код, запускает виртуальную машину Java и т.д. Все, что мне нужно, — выполнить код, который я вижу в окне редактирования. Поэтому подробности, показанные на рис. 2.3, не так уж важны. Они понадобятся лишь в том случае, если туманные выражения на рис. 2.2 сбивают вас с толку. Если рис. 2.2 вам понятен, можете проигнорировать рис. 2.3.

Интегрированная среда разработки

Интегрированная среда разработки — это программа (например, Eclipse), в окне которой вы видите исходный код Java и которая предоставляет удобные средства редактирования, компиляции, отладки и запуска создаваемой программы.

До появления интегрированных сред разработки для создания программы нужно было открыть несколько отдельных окон: одно — для редактирования программы, другое — для ее выполнения и третье — для того, чтобы увидеть в нем результаты выполнения. Вы и сейчас можете отказаться от Eclipse и создать программу Java с помощью проводника, блокнота и консоли (рис. 2.4).

Интегрированная среда разработки предоставляет все эти функции в одном хорошо организованном окне (рис. 2.5).

Программа Eclipse — не единственная интегрированная среда разработки приложений Java. Другие наиболее популярные программы — IntelliJ IDEA, JDeveloper и NetBeans. Некоторые из них даже предоставляют графический редактор, в окне которого можно видеть пользовательский интерфейс разрабатываемой программы (рис. 2.6). Однако многие профессиональные разработчики предпочитают текстовый редактор Eclipse, потому что графический редактор автоматически генерирует код и

часто портит код, написанный разработчиком. У автора программы как бы появляется соавтор, причем часто он упрямый и непонятливый. Вы написали, как вы хотите, а он все переделал по-своему. Вы переделали его код, как вам нужно, а он опять переделал все по-своему, и так будет продолжаться, пока вы не откажетесь от его услуг.

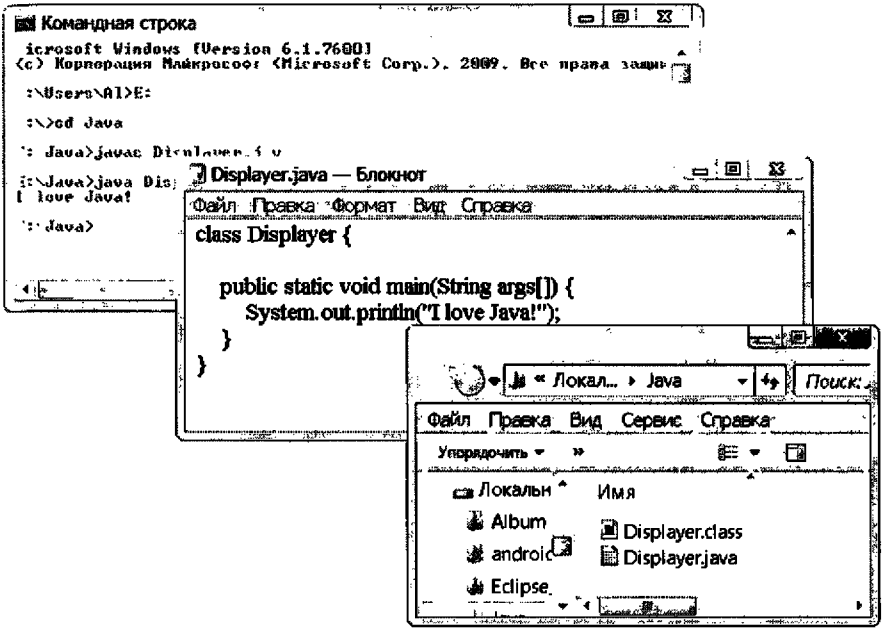


Рис. 2.4. Разработка кода Java "подручными средствами"

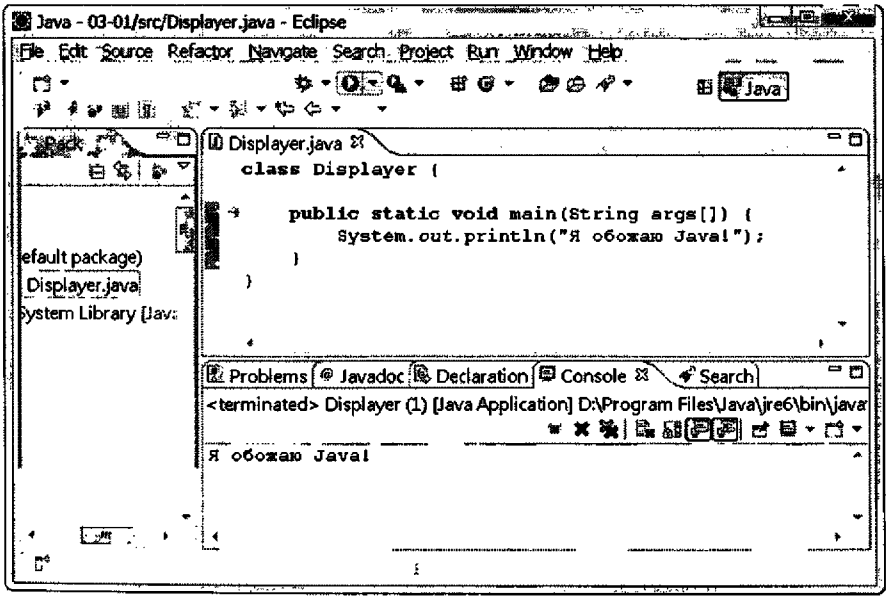


Рис. 2.5. Разработка кода Java в окне программы Eclipse

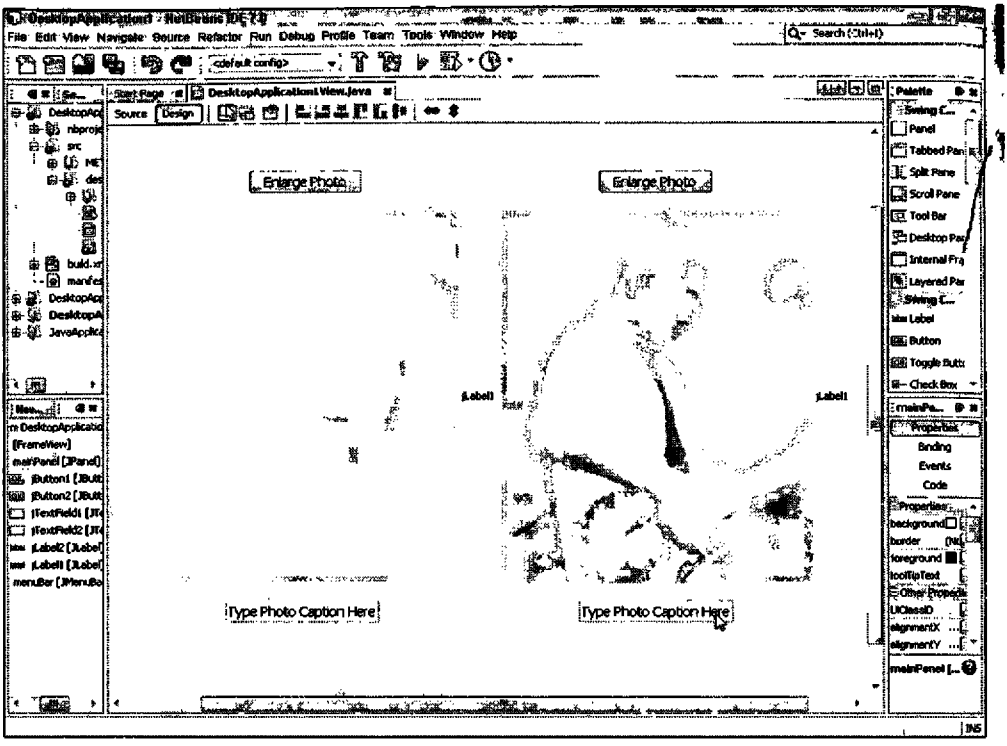


Рис. 2.6. Программа Java в окне графического редактора

Вы редактируете код программы Java в окне редактора Eclipse (см. рис. 2.5). После этого код нужно скомпилировать, запустить на выполнение и посмотреть, что получилось. Для этого щелкните на кнопке Run (Выполнить), расположенной на панели инструментов Eclipse. Результат выполнения можно увидеть в окне консоли, расположенной в нижней части главного окна Eclipse.



Более подробное описание интегрированной среды разработки Eclipse приведено в приложении Б.

Глава 3

Базовые компоненты Java

В этой главе...

- Поговорим на языке Java
- Ваша первая программа на Java
- Как работает ваша первая программа
- Не говорите "Без комментариев..."

"Все мысли, которые имеют
огромные последствия, всегда просты".
Лев Толстой

Этот принцип относится практически ко всем областям человеческой жизни, в том числе к компьютерному программированию. Поэтому в данной главе применяется многоуровневый подход. Сначала вы ознакомитесь с некоторыми компонентами программы Java, а затем, узнав множество подробностей, увидите, что они подчиняются простым принципам.

Поговорим на языке Java

Попытайтесь представить себе весь русский язык одновременно. Что вы видите? Слова, слова, слова... Огромное количество слов! Рассматривая русский язык "под микроскопом", вы отчетливо видите каждое слово. Однако отойдите на шаг назад, и вы увидите еще кое-что, кроме слов:

- ✓ грамматика языка;
- ✓ имена, слова и устойчивые группы слов.

Первая категория (грамматика) содержит правила наподобие "подлежащее должно быть того же рода и числа, что и сказуемое". Вторая категория содержит названия конкретных и абстрактных объектов. Например, четыре буквы "стол" обозначают любой предмет с четырьмя ножками, на котором что-нибудь лежит или может лежать, а шесть букв "Берлин" обозначают конкретный город, столицу Германии.

Язык программирования Java обладает многими свойствами естественного языка. В нем есть слова, грамматика, часто используемые имена, стилистические идиомы и ряд других компонентов естественных языков.

Грамматика и общие имена

Люди в Sun Microsystems, которые создавали Java, считали, что язык должен состоять из двух частей. Как и естественный язык, он должен иметь грамматику и набор слов.

- ✓ **Грамматика Java** определяется спецификацией языка, т.е. официальным документом, в котором написано, как используются компоненты языка. В принципе, вы можете почитать спецификацию Java, но для вас она будет бесполезной, потому что она создавалась главным образом для разработчиков компиляторов. Вам лучше подойдут учебники по Java. Из них вы узнаете все нужные вам правила, такие как "Каждой открывающей скобке должна соответствовать одна закрывающая" или "Чтобы умножить два числа, поместите между ними звездочку".

  Наряду с термином *грамматика* часто можно встретить термин *синтаксис*. Строго говоря, синтаксис — это раздел грамматики, но этот вопрос чисто академический. Даже профессиональные программисты не очень беспокоятся о том, какая разница между грамматикой и синтаксисом.

- ✓ **Классы API** (Application Programming Interface — интерфейс программирования приложений). Библиотека стандартных классов API, входящая в состав каждой виртуальной машины Java, содержит тысячи инструментов, позволяющих выполнять практически любые операции, которые может выполнять компьютер. Например, библиотека API содержит класс `Math`, который предоставляет большое количество математических функций, таких как синус, косинус, арксинус, арккосинус и т.п. Другой пример — стандартный класс `JFrame`, отображающий на экране окно, в которое можно поместить элементы интерфейса (кнопки, флажки, списки и т.п.).

  Полный официальный справочник по классам API можно найти на сайте `http://java.sun.com/javase/reference/api.jsp`, но учитывайте, что на момент публикации этой книги адрес может измениться. В таком случае откройте сайт Oracle и найдите ссылку на справочник по библиотечным классам.

Первая часть Java — грамматика языка — сравнительно небольшая. Это характерная особенность Java, довольно приятная. В других языках программирования количество правил в несколько раз больше, а в некоторых языках — в десятки раз.

Вторая часть Java — набор стандартных классов API — выглядит устрашающе большой. Стандартная библиотека содержит 5 тысяч инструментов, причем их количество продолжает увеличиваться с каждой новой версией JDK. Однако не пугайтесь, вам не нужно помнить все классы API. Даже профессиональные программисты помнят не более десятка наиболее "ходовых" классов, а с остальными знакомятся по необходимости, открыв справочник по классам. Вы также будете помнить только то, что часто используете.



Ни один человек в мире не знает все о Java API. Если вы будете разрабатывать программы с графическим интерфейсом пользователя для настольных компьютеров, вы будете часто использовать класс `JFrame` и, соответственно, помнить его структуру. Если же вы редко будете писать программы,

открывающие окна, но в некоторый момент вам понадобится окно, вы почитаете описание класса `JFrame` и узнаете, как это делается. Если программисту Java (даже самому крутому профессионалу) запретить пользоваться справочником по стандартным классам, он не сможет написать на Java почти ничего.

Каждый раз, когда вы пишете программу на Java, даже самую маленькую, вы создаете классы, которые обладают теми же правами, что и стандартные классы Java API. Интерфейс API — это набор классов и других инструментов, созданных программистами — участниками официальных проектов JCP (Java Community Process — процесс сообщества Java) и OpenJDK Project (Открытый проект пакета инструментов Java). Вы пока что, как я понимаю, не участвуете в JCP или OpenJDK, но, когда прочитаете эту книгу, вполне можете предложить свои услуги и получить часть работы.

Если вас интересует деятельность JCP, посетите сайт `http://jcp.org`. С проектом OpenJDK можно ознакомиться на сайте `http://openjdk.java.net`.



Ребята из JCP не пытаются держать в секрете свою работу над официальной библиотекой Java API. При желании вы также можете ознакомиться с исходными кодами официальных стандартных библиотек. При установке Java на ваш компьютер инсталляционная программа поместила на ваш диск архивный файл `src.zip`. В нем находятся все исходные коды Java API. Можете распаковать его с помощью любой программы архивирования.

Слова в программе Java

Принято считать, что в исходном коде Java используются слова двух категорий: *идентификаторы* и *ключевые слова*. Однако без дополнительных разъяснений такая классификация может ввести в заблуждение. Поэтому лучше считать, что в Java используются слова трех категорий: ключевые слова, идентификаторы API и пользовательские идентификаторы. *Пользовательские идентификаторы* — это слова, придумываемые программистом во время написания программы, а *идентификаторы API* — это имена переменных и классов, используемые в стандартных библиотечных классах Java.

Отличие между этими тремя категориями приблизительно такое же, как между словами естественного языка. В предложении "Сергей купил тетрадь" слово "купил" играет ту же роль, что и ключевое слово Java. Независимо от того, кто и в каком месте его использует, оно всегда означает приблизительно одно и то же — акт обмена денег на продукт. Слово "тетрадь" похоже на идентификатор API тем, что оно везде обозначает предмет, состоящий из нескольких сшитых листов. Конечно, в естественном языке из этого правила есть много исключений. Некоторые слова, называемые "синонимами", могут обозначать разные предметы, но мы не будем обращать на это внимания, потому что язык программирования более строг, чем естественный. В нем нет синонимов, и каждое ключевое слово всегда означает одно и то же.

В упомянутом выше предложении слово "Сергей" похоже на пользовательский идентификатор Java, потому что оно является именем конкретного объекта и в разных местах может обозначать разные предметы. В русском языке слова "Витя", "Коля", "Сидоров" и подобные им не имеют предустановленного, общего для всех значения. Значение такого слова зависит от того, где оно используется, ведь, как вы понимаете, есть много Сергеев, и это не один и тот же человек.

Теперь рассмотрим предложение "Юлий Цезарь завоевал Египет". На первый взгляд, идентификатор "Юлий Цезарь" здесь такой же, как в предыдущем примере — он обозначает человека. Однако его значение не зависит от того, кто и где его использует. Оно всегда означает римского императора, причем каждому читателю понятно, какого именно императора. Если бы русский язык был языком программирования, имя "Юлий Цезарь" было бы идентификатором API.

Итак, резюмируем вышесказанное: в исходном коде Java используются следующие слова.



- **Ключевые слова.** Это слова, имеющие специальное значение и определенные в грамматике Java. Значение ключевого слова всегда одно и то же в любой программе. Примеры ключевых слов — `if`, `else`, `while` и т.п.

  Количество и написание всех ключевых слов определено в спецификации Java, утверждаемой комитетом JCP. Никакой придуманный вами идентификатор не должен совпасть ни с одним ключевым словом Java.

  Полный список ключевых слов Java можно увидеть в шпаргалке, прилагаемой к данной книге.

- **Идентификаторы.** Идентификатор — это имя чего-либо (класса, объекта, метода, события и т.д.). В спецификации языка идентификаторы не определены (определено лишь общее понятие "идентификатор"), поэтому в разных программах они могут иметь разные значения.

- **Пользовательские идентификаторы.** Создавая программу на Java, вы придумываете новые имена для классов и других сущностей, встречающихся в исходном коде. Предположим, вы назвали что-то именем `Prime`, а программист в соседнем отделе случайно, не сговариваясь с вами, тоже назвал что-то именем `Prime`. Что произойдет в результате этого? Ничего! Слово `Prime` не имеет в Java никакого предустановленного значения. Кроме того, ваши программы не контактируют друг с другом. Поэтому имена `Prime` в них никак не затрагивают друг друга. Впрочем, иногда имена пересекаются. Такая ситуация называется *конфликтом имен*. Это может произойти, например, если вы и ваш коллега работаете над одним проектом и случайно назовете два разных класса одним и тем же именем.

- **Идентификаторы API.** Участники проекта JCP создали классы Java API, в которых используется более 40 тысяч имен. Классы API поставляются в дистрибутиве JDK, поэтому все эти имена доступны для вас. Можете использовать их в своей программе. Примеры идентификаторов API — `String`, `Integer`, `JWindow`, `JButton`, `JTextField` и т.п.

Строго говоря, значения идентификаторов API не увековечены в камне или бронзе. Вы можете использовать любой идентификатор API (например, `JButton`) в своем коде не по назначению. Конечно, он будет обозначать вашу сущность, а не ту, для обозначения которой он предназначен в API. Программа будет работать правильно, но не делайте так, потому что сами запутаетесь и собьете с толку другого программиста, который будет работать с вашей программой, потому что он привык к тому, что `JButton` — это класс кнопки. Кроме того, вы не сможете использовать в своей

программе стандартную кнопку JButton. Программисты Sun Microsystems, JCP и OpenJDK проделали огромную работу, чтобы предоставить вам простую, удобную кнопку, а вы презрительно ее отбросите. Впрочем, если вам понадобится кнопка, вы и без моих увещеваний не будете называть именем JButton другую сущность в своей программе.

Ваша первая программа на Java

Когда вы впервые посмотрите на программу Java, написанную другим человеком, вы испытаете легкое головокружение. Вы с ужасом осознаете, что ничего не понимаете в этом коде. За свою жизнь я написал сотни программ на Java, но до сих пор, читая чужой исходный код, чувствую себя неуверенно.

Чтобы понимать некоторый код, нужно знать назначение каждого идентификатора, структуру алгоритма, компоновку интерфейса и многое другое. Увидеть все это с первого взгляда невозможно даже для опытного программиста. Чтобы понять код, он также вынужден рассматривать его очень долго, может быть, даже много дней. Не верьте человеку, который утверждает, что без труда понимает любой код. Даже опытные программисты подходят к новому проекту медленно и аккуратно.

Поэтому начнем с простейшей программы, приведенной в листинге 3.1. В этом коде я проиллюстрирую вам несколько важных концепций, которые мы рассмотрим в следующем разделе. Это такие концепции, как классы, методы, инструкции Java и структура программы.

Листинг 3.1. Исходный код простейшей программы на Java

```
class Displayer {
  public static void main(String args[]) {
    System.out.println("Мне нравится Java!");
  }
}
```

Эта программа всего лишь выводит на консоль фразу Мне нравится Java!. Запустите программу в интегрированной среде разработки Eclipse, и вы увидите эту фразу в окне консоли Eclipse (рис. 3.1). Такой результат может показаться обескураживающим. Вы прочитали уже несколько глав, и все это лишь для того, чтобы вывести на экран простую фразу. Однако это только начало. Через некоторое время (не такое уж долгое) ваша программа Java будет делать все, чего вы от нее захотите.

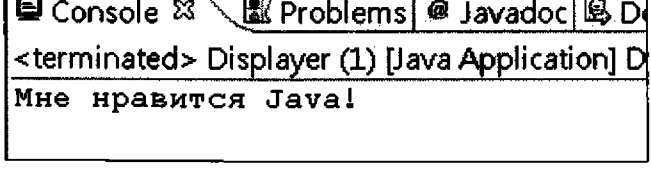


Рис. 3.1. Результат выполнения программы в Eclipse



Подробные инструкции по работе с Eclipse приведены в приложении Б.

Как работает ваша первая программа

В этом разделе приведено описание и объяснение программы, показанной в листинге 3.1.

Классы

Поскольку Java является объектно-ориентированным языком программирования, ваша главная задача — определить классы и создать объекты (т.е. экземпляры классов). Если вы этого еще не знаете, почитайте еще раз главу 1.

В особые дни, когда я становлюсь сентиментальным, я рассказываю людям о том, что в Java концепция объектно-ориентированного программирования реализована более чисто и полно, чем во многих других так называемых "объектно-ориентированных" языках. Я говорю это потому, что в Java вы не сможете ничего сделать, пока не создадите соответствующий класс. Это настолько сильно войдет у вас в привычку, что, когда ваша супруга попросит вас сходить в магазин за хлебом, вы ответите: "Извини, дорогая, но ты должна вложить эту просьбу в класс".

В программе Java есть только классы и ничего кроме классов. Буквально все (данные, методы, идеи, структура пользовательского интерфейса, ошибки, гениальные решения и досадные промахи) находится в классах. Для каждого класса нужно придумать имя. Программа, показанная в листинге 3.1, состоит из одного класса, который я назвал `Displayer`. Поэтому исходный код класса начинается словами `class Displayer` (рис. 3.2).

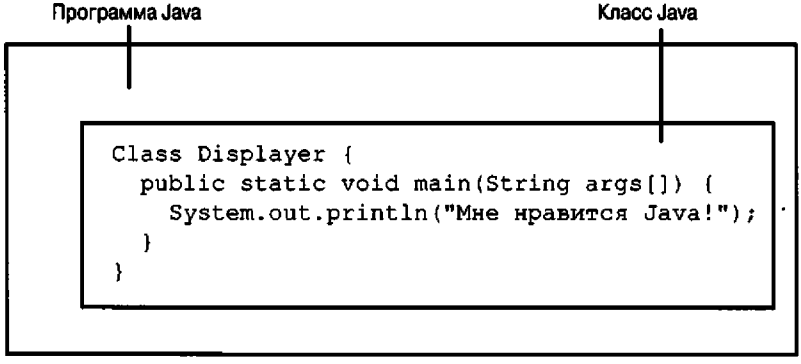


Рис. 3.2. Эта программа состоит из одного класса

Первое слово, `class`, — это ключевое слово Java. Независимо от того, кто пишет программу и что она будет делать, слово `class` всегда используется одинаково: оно обозначает определение класса и находится перед его именем. Слово `Displayer` — это пользовательский идентификатор, служащий именем класса. Я придумал это имя, когда создавал класс.



Исходный код Java чувствителен к регистру букв. Это означает, что `class` и `Class` являются разными словами. Если заменить первое слово вторым, программа не будет компилироваться, пока вы не исправите ошибку.

К регистру букв чувствительны как ключевые слова, так и идентификаторы. Вы можете назвать класс `displayer`, но, если в другом месте вы сошлетесь на него как на `Displayer`, компилятор сообщит, что не нашел этого класса. Согласно принятым в сообществе Java соглашениям, имена классов принято начинать с большой буквы, а имена объектов (т.е. экземпляров классов) — с малой. Если вы нарушите это правило, компилятор не отметит имя как ошибочное. Ему все равно, однако ваши коллеги, глядя на код, будут удивляться, почему вы используете такие странные имена, и вы не найдете, что им ответить.

Если назвать класс, например, `DogAndPony`, программа Eclipse поместит его в файл `DogAndPony.java`. Компилятор создаст байтовый код класса и поместит его в файл `DogAndPony.class`.

Методы

Предположим, вы работаете механиком в гараже. Ваш босс, который всегда спешит, имеет привычку отдавать приказы скороговоркой: "Нужно починитьГенератор в вон-том старомСеромФорде". Мысленно вы составляете список задач: "Загнать машину на яму, открыть капот, отвинтить крепление, залезть в яму, снять генератор, открыть корпус генератора и т.д.". Во всем этом можно выделить три компонента.

- ✓ **Имя задания, которое вы должны выполнить.** В данном случае имя задания — `починитьГенератор`.
- ✓ **Список задач, которые для этого нужно решить.** Задачи решаются строго в той последовательности, в которой они указаны в списке. Например, нельзя сначала отвинтить крепление генератора, а затем открыть капот.
- ✓ **Сердитый босс, который дал вам задание.** Ваш босс произнес имя задания `починитьГенератор`. Этого достаточно, потому что вы знаете, как это делается, т.е. имеете упомянутый выше список задач.

В этом сценарии уместно будет ввести понятие *метод*. Вы знаете, как починить генератор. На языке программирования это означает, что у вас есть метод починки генератора. Ваш босс посредством словесного приказа приводит этот метод в действие, а вы выполняете все операции, заложенные в метод.

Если вы согласны с таким пониманием термина "метод", значит, готовы к пониманию того, что такое метод в языках программирования. В Java метод — это список инструкций, которые должен выполнить компьютер. У каждого метода есть персональное имя, которое в отличие от человеческих имен (Сергей, Лена и т.п.) должно быть уникальным в своем классе. Однако в отличие от вашего босса программист не приказывает голосом, а пишет конструкцию языка, которая называется *вызов метода*. Это означает следующее: чтобы компьютер выполнил список инструкций, заложенных в методе, программа должна "вызвать метод".

Я никогда не писал программу для робота, ремонтирующего генератор, но если бы мне пришлось делать это, я назвал бы метод `починитьГенератор`. Список инструкций в теле метода `починитьГенератор` выглядел бы приблизительно так, как код в листинге 3.2.

Листинг 3.2. Определение метода

```
void починитьГенератор(Машина машина) {
  загнатьМашинуНаЯму(машина, яма);
  поднятьКапот(капот);
  отвинтитьГенератор(гайка, гайка, гайка, гайка);
  ...
}
```



Не рассматривайте листинги 3.2 и 3.3 слишком придирчиво. Это воображаемый код. Компьютер не может поднять капот или отвинтить гайку. Единственное достоинство этого кода в том, что он выглядит точно так же, как реальный код Java. В принципе, это синтаксически правильный код Java, но он никогда не будет выполнен (сейчас уже есть роботы, умеющие отвинчивать гайки, но в них не используется код, приведенный в листинге 3.2).

Итак, у нас есть метод Java, который нужно запустить на выполнение. Для этого в другом месте кода Java необходимо вставить инструкцию вызова метода `починитьГенератор`. Инструкция вызова метода представляет собой имя метода, после которого в скобках приведен список параметров. В данном случае нужно сообщить методу, в какой машине нужно отремонтировать генератор. Для этого укажем в списке единственный параметр — имя машины (листинг 3.3).

Листинг 3.3. Вызов метода

```
починитьГенератор(старыйСерыйФорд);
```

Теперь, когда вы знаете, что такое метод и как он работает, мы можем глубже "копнуть" терминологию.

- ✓ **Определение метода.** В русском языке слово "определение" имеет два разных значения: во-первых, вычисление, выяснение, обнаружение чего-либо, и, во-вторых, задание, постулирование, указание на то, что это такое. Возможно, вы помните, как на уроке геометрии в школе учили "определение треугольника" и "определение окружности". В математике и программировании слово "определение" всегда используется только во втором смысле. В данном случае определение метода — это список инструкций, т.е. вся информация о том, что компьютер должен сделать.

  В других книгах часто используется термин *объявление метода*. Это синоним, т.е. то же самое, что определение метода.
- ✓ **Заголовок метода.** Первая строка в определении, в которой указано имя метода и приведен список принимаемых параметров.
- ✓ **Тело метода.** Содержит список инструкций.
- ✓ **Вызов метода.** Инструкция, расположенная в другом месте программы и запускающая метод на выполнение (см. листинг 3.3).



Определение метода указывает компьютеру, что он должен сделать при вызове метода. Инструкция вызова метода приказывает компьютеру сделать это. Определение не инициирует выполнение метода, оно лишь содержит информацию о том, что должно быть сделано. Определение и вызов метода — это две разные части программы. В принципе, вызов можно разместить в теле этого же метода. Такой метод будет называться *рекурсивным*. Однако вам пока что не придется использовать такую хитрую конструкцию.



В тексте статей, книг и описаний, посвященных программированию, принято при упоминании метода ставить после его имени пару скобок, чтобы читатель с первого взгляда видел, что это метод, а не какая-либо другая сущность языка, например `починитьГенератор()`.

Главный метод программы

На рис. 3.3 приведена копия кода, показанного в листинге 3.1. Код содержит определение метода `main` (данное имя приведено в заголовке метода). Пока что не беспокойтесь о назначении ключевых слов `public`, `static`, `void`, `String` и др. Я объясню их в следующих главах.

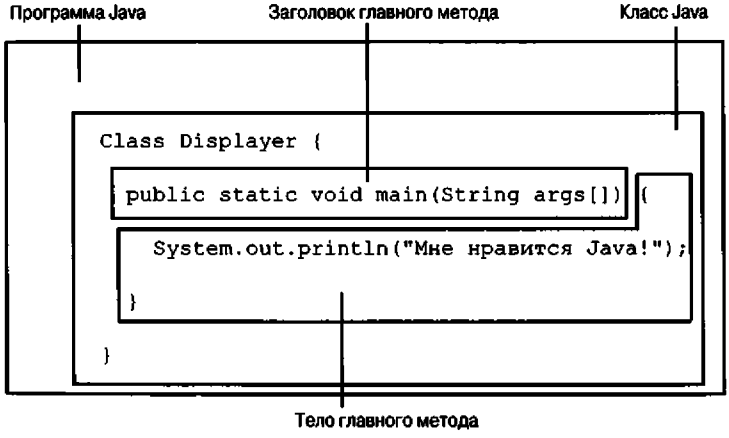


Рис. 3.3. Главный метод

Как и любой метод Java, главный метод — это рецепт.

```
Как приготовить пирог {
  разогреть печь;
  раскатать тесто;
  приготовить начинку;
  ...
}
```

В нашем случае рецепт выглядит так.

```
Как выполнить главный метод {
  вывести на консоль фразу "Мне нравится Java!";
}
```

Ключевое слово `main` (главный) играет в Java специальную роль. В частности, в коде программы нельзя написать инструкцию вызова метода `main()`. Имя `main` обозначает метод, который автоматически вызывается операционной системой при запуске программы. Следовательно, работа любой программы начинается с выполнения тела метода `main()`.

Взгляните еще раз на рис. 3.1. Когда вы запускаете программу `Displayer`, операционная система автоматически запускает метод `main()` и выполняет расположенные в нем инструкции. В программе `Displayer` метод `main()` содержит только одну инструкцию — приказ компьютеру вывести на консоль фразу `Мне нравится Java!` (см. рис. 3.1).



Все инструкции любой программы расположены только в методах. Компьютер делает только то, что ему приказывают инструкции, а инструкция выполняется только после вызова метода, в котором она расположена. Следовательно, любой метод, который нужно выполнить, должен быть вызван в коде программы. Но это не относится к методу `main()`, который вызывается операционной системой автоматически при запуске программы.



Практически в каждом языке программирования есть некоторый аналог метода Java. Если вы работали с другими языками, вам должны быть знакомы такие понятия, как "процедура", "функция", "подпрограмма", "оператор `PERFORM`", "оператор `call`". Все это фактически методы и вызовы методов, но в других языках они называются по-другому. Однако в любом языке программирования метод (или как он там называется) обладает именем и содержит список инструкций.

Как приказать компьютеру выполнить нужную операцию

Очень просто: написать инструкцию и вставить ее в тело метода. В листинге 3.1 есть только одна инструкция, заключенная в рамку на рис. 3.4. Она приказывает компьютеру вывести на консоль фразу `Мне нравится Java!`.

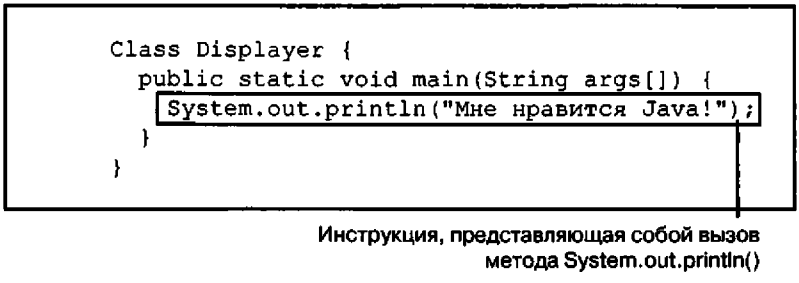


Рис. 3.4. Инструкция Java

Конечно, в Java есть много разных видов инструкций. Один из них — вызов метода. В листинге 3.3 показано, как выглядит вызов метода, а на рис. 3.4 содержится следующая инструкция вызова метода:

```
System.out.println("Мне нравится Java");
```

Выполняя эту инструкцию, компьютер вызывает стандартный библиотечный метод `System.out.println()`, который выводит на консоль заданную фразу.



В Java имена могут содержать точки. Что они означают, мы рассмотрим в главе 9.

На рис. 3.5 проиллюстрирован вызов метода `System.out.println()`. Фактически в программе `Displayer` используются два метода. Вот как они работают.

- ✓ **Программа содержит определение метода `main()`.** Я сам написал метод `main()`. Он вызывается операционной системой автоматически при запуске программы.
- ✓ **Программа содержит вызов метода `System.out.println()`.** Это единственная инструкция метода `main()`. Иными словами, вызов метода `System.out.println()` — это единственное, что делает метод `main()`.

  Определение метода `System.out.println()` находится в стандартной библиотеке Java API, устанавливаемой при инсталляции на компьютер пакета JDK или JRE.

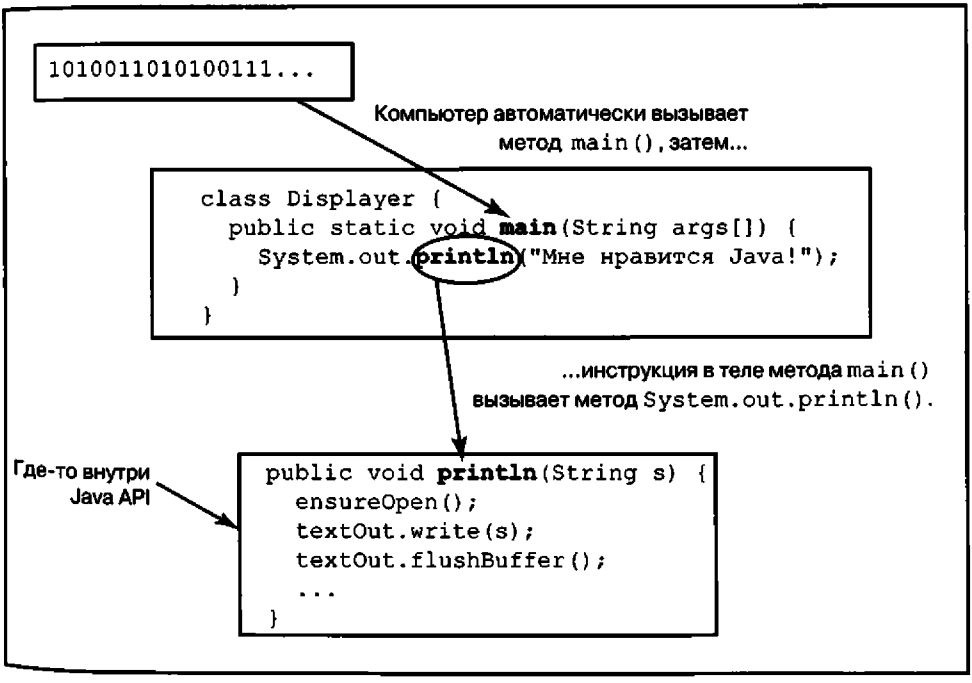


Рис. 3.5. Вызов метода `System.out.println()`



Если вы любопытны, можете посмотреть определение метода `System.out.println()` в библиотеке Java API. Это всего лишь еще один метод, написанный на языке Java. Однако пользы от ознакомления с кодами

стандартных библиотечных методов мало, вернее, совсем никакой. Лучше сосредоточьте внимание на применении метода. Вы должны знать, что он делает и какие параметры ему нужно передать.

Каждая инструкция Java заканчивается точкой с запятой (например, заключенная в рамку инструкция на рис. 3.4). Другие строки на рис. 3.4 не заканчиваются точками с запятыми, потому что это не инструкции. Например, заголовок метода не является инструкцией, потому что он не приказывает компьютеру сделать что-либо, а всего лишь информирует его о том, как называется метод и какие параметры ему нужно передать. Инструкции можно размещать по строкам произвольно — по нескольку инструкций в одной строке или одну инструкцию в нескольких строках. Границами между инструкциями служат точки с запятыми, а разрывы строк никак не влияют на них. Тем не менее в сообществе Java принято размещать по одной инструкции в каждой строке, чтобы код было легче читать.



Каждая инструкция Java должна заканчиваться точкой с запятой.

Фигурные скобки

Вспомните, как много лет назад школьный учитель убеждал вас в том, что во всем важен порядок и что нужно правильно организовать, разложить по полочкам свои дела, идеи, мысли, планы и знания. Фактически программа Java — это и есть хорошо структурированная, тщательно организованная информация, разложенная по полочкам. В листинге 3.1 программа начинается с заголовка класса `Displayer`, сообщающего о том, что дальше расположено определение класса. В класс вложен метод `main()`. Его заголовок сообщает, как он называется и какие параметры ему нужно передать, а также о том, что непосредственно после него начинается тело метода.

Вы должны строго придерживаться этой структуры. Нельзя написать заголовок метода внутри его тела или определение метода за пределами определения класса. Программа должна иметь древовидную структуру, в которой одни элементы вложены в другие. Инструкции вложены в метод. Методы вложены в класс. Классы вложены в программу.

Однако программа представляет собой текст, т.е. последовательность символов. Как же отобразить в ней древовидную структуру, не прибегая к помощи рисунка? Это делается с помощью следующих средств.

- ✓ **Фигурные скобки.** В коде Java они обозначают элементы программы. Например, тело метода (как и тело класса) должно быть заключено в фигурные скобки.
- ✓ **Отступы.** В принципе, программу Java можно записать в произвольной форме, например ввести все подряд, без пробелов и перевода строк. Синтаксически это будет правильная программа. Она будет работать точно так же, как хорошо структурированная. Но ее будет тяжело читать визуально. Чтобы облегчить понимание структуры программы, необходимо использовать отступы.

В древовидной, иерархической структуре программы Java высший элемент — класс. Тело класса заключено в фигурные скобки. Следовательно, после заголовка класса должна располагаться открывающая фигурная скобка, а после тела класса должна быть закрывающая фигурная скобка (листинг 3.4).

Листинг 3.4. Тело класса заключено в фигурные скобки

```
class Displayer {
  public static void main(String args[]) {
    System.out.println("Мне нравится Java!");
  }
}
```

Следующий, более низкий уровень иерархии — метод. Как и класс, он начинается с заголовка, и его тело должно быть заключено в фигурные скобки (листинг 3.5). Чтобы было лучше видно, что метод вложен в класс, код метода введен с отступом.

Листинг 3.5. Тело метода заключено в фигурные скобки

```
class Displayer {
  public static void main(String args[]) {
    System.out.println("Мне нравится Java!");
  }
}
```

Самый нижний элемент иерархической структуры программы — инструкция. Чтобы ее было лучше видно, она введена с самым большим отступом (см. рис. 3.5).



Никогда не забывайте о древовидной иерархической структуре программы Java.

Если ошибочно разместить фигурную скобку не там, где нужно, или опустить ее, программа, скорее всего, не будет работать. Компилятор сгенерирует сообщение о синтаксической ошибке. Но может быть и хуже: программа будет работать, но не так, как нужно, компилятор не сообщит об ошибке, вы ее не заметите, и она обнаружится только у заказчика, что повлечет для вас большие неприятности.

Если написать код без отступов, информирующих о структуре программы, она будет работать правильно, компилятор не будет возражать, но вам и тем более другому программисту будет тяжело понять, как работает программа.

Можете представить себе структуру программы как древовидную иерархическую структуру оглавления книги: книга разделена на части, в которых размещены главы, в которых, в свою очередь, размещены разделы (рис. 3.6). Существует также еще более причудливое представление структуры программы (рис. 3.7).

Многие среды разработки, в том числе Eclipse, создают отступы автоматически. Вы можете отредактировать величину отступов и правила их расстановки или вообще отключить их создание.

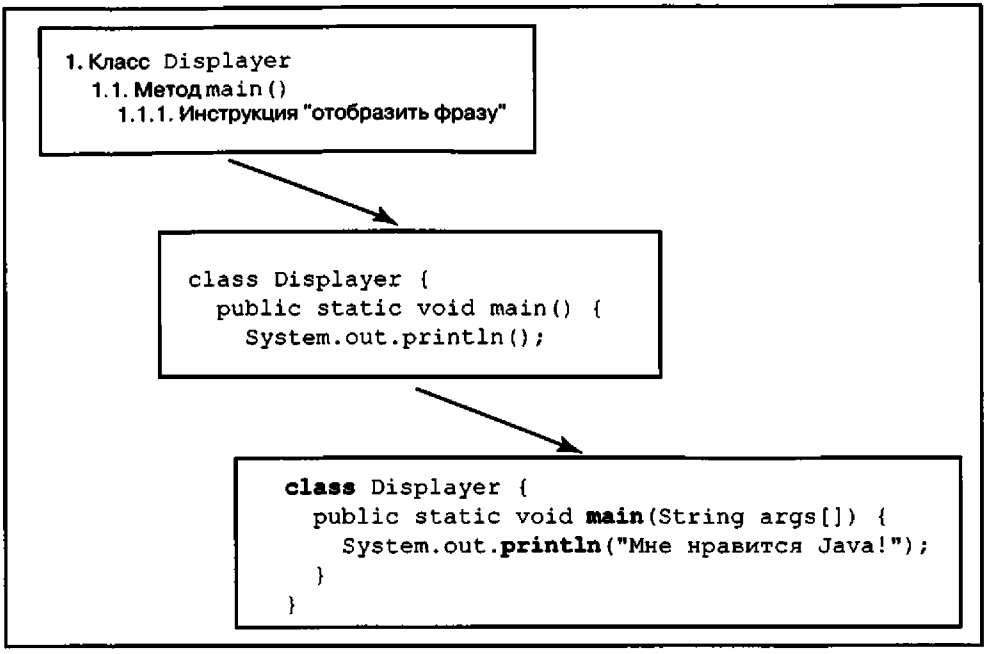


Рис. 3.6. Представление иерархической структуры программы

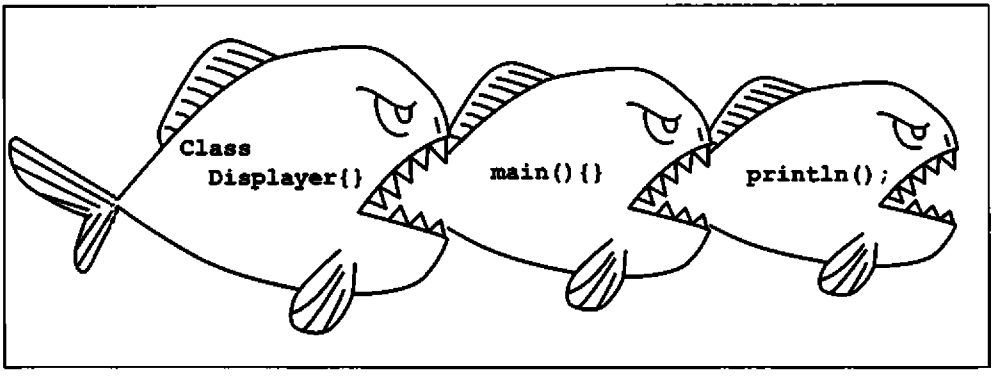


Рис. 3.7. Класс больше, чем метод; метод больше, чем инструкция

Не говорите "Без комментариев..."

Люди собрались вокруг костра, чтобы послушать древнюю легенду о талантливой программистке, ленившейся вставлять комментарии. Чтобы не стать объектом судебных преследований, назовем ее Джейн Про. Много лет Джейн работала над "Святым Граалем" компьютерных технологий — программой, которая могла бы самостоятельно думать. Если бы эту программу удалось сделать, она могла бы без участия и помощи человека выполнять нужную работу, изучать меняющуюся ситуацию и даже саму себя совершенствовать. День за днем долгими бессонными ночами Джейн продвигалась к

цели, и уже близок был момент, когда она рассчитывала увидеть в работе программы искру самостоятельной мысли.

Но в один прекрасный день произошло следующее событие. Когда она наконец-то почти высекла долгожданную искру, в ее почтовый ящик (традиционный, не электронный) пришло бумажное письмо от ее страховой компании по поводу ее здоровья. Не пугайтесь, в письме было не сообщение об ужасном заболевании, а всего лишь рутинная просьба позвонить в офис компании для уточнения некоторых формальностей. В стандартной форме страховой компании был пункт о ее дате рождения (как будто он мог измениться со времени заключения договора о социальном страховании). В последний раз она, будучи в состоянии глубокой задумчивости (Святой Грааль требует полной самоотдачи), в графе "Год рождения" написала "2011". Несовершенная программа (ей было очень далеко до Святого Грааля), установленная на компьютере компании, автоматически подсчитала сумму очередного страхового платежа, и он оказался отрицательным. Другая несовершенная программа, перечисляющая деньги на счет Джейн, не поняла, что нужно делать в таком случае, и все застопорилось в ожидании вмешательства человека.

Джейн набрала телефонный номер компании. После 20 минут блуждания по голосовому меню она наконец услышала человеческий голос. После 20 минут переговоров менеджер компании сообщила: "Мне очень жаль, но для решения этого вопроса вы должны позвонить по другому номеру". Звонок по другому номеру привел к тому же результату, но фраза немного отличалась от прежней: "Мне очень жаль, но оператор дал вам неправильный номер, вы должны обратиться к нему еще раз".

Через несколько месяцев, после 800 часов телефонных переговоров, ее ухо опухло, но она все же добилась обещания, что компания исправит графу "Год рождения" в ее форме. Воодушевленная таким успехом, она немедленно вернулась к своему Святому Граалю, но — о ужас! — вдруг увидела, что не может вспомнить, что делают и зачем нужны все эти буковки и закорючки, которые она сама же написала в огромном количестве.

Она смотрела на свою работу, все больше и больше казавшуюся ей сном, который она не может вспомнить утром после пробуждения. Код ничего не означал для нее. Она написала тысячи строк кода, и ни в одной строке не было комментария, объясняющего, как работает программа. Она не оставила ни единого намека, который помог бы ей вспомнить, о чем она думала, когда писала эти строки. В полной прострации Джейн выключила компьютер. Ей пришлось оставить работу над проектом.

Добавление комментариев в код

В листинге 3.6 показана версия программы, приведенной в листинге 3.1, но с комментариями. Кроме ключевых слов, идентификаторов и знаков пунктуации, предназначенных для компилятора, код теперь содержит текст, предназначенный для чтения человеком.

Листинг 3.6. Код программы с комментариями

```
/*
 * Листинг 3.6 книги "Java для чайников, 5-е издание"
 *
 * Copyright 2011, Издательство Диалектика.
 * Все права защищены.
*/
```

```
/**
 * Текст класса Displayer, который отображает заданный
 * фрагмент текста на консоли.
 *
 * @author  Barry Burd
 * @version 1.0 10/24/11
 * @see     java.lang.System
 */
class Displayer {
 /**
  * Метод main(), содержащий выполняющийся код.
  *
  * @param args (см. главу 11)
  */
  public static void main(String args[]) {
    System.out.println("Мне нравится Java!");
  } //Конец метода main().
}
```

Комментарий — это специальный раздел в исходном коде, предназначенный для того, чтобы облегчить человеку понимание программы, и считающийся частью документации программы.

В языке Java существуют комментарии трех видов.



- ✓ **Традиционные комментарии.** Первые шесть строк в листинге 3.6 представляют собой один традиционный комментарий. Комментарий должен начинаться символами `/*` и заканчиваться символами `*/`. Вся информация между этими символами предназначена только для человека, компилятор игнорирует ее. Часто комментарии такого типа называются *многострочными* (в отличие от однострочных, о которых говорится далее).

  Компиляторы рассматривались в главе 2.

  Строки 2–4 листинга 3.6 начинаются с дополнительных звездочек. Они не обязательные, их принято добавлять для украшения комментария. Лично я не вижу, стал ли комментарий красивее, но, раз все программисты на Java так делают, то лучше не быть белой вороной.

- ✓ **Однострочные комментарии.** Пример однострочного комментария — текст `//Конец метода main()` в листинге 3.6. В программе на Java все содержимое строки после символов `//` является комментарием, т.е. компилятор игнорирует этот текст.

- ✓ **Комментарии Javadoc.** Начинается с косой черты и двух звездочек (`/**`). В листинге 3.6 есть два комментария Javadoc — над заголовком класса `Displayer` и в теле этого класса.

  Комментарий Javadoc — это специальная разновидность традиционного комментария, но предназначенного для чтения людьми, которые не увидят код Java. "Что за чепуха?" — скажете вы. — "Как можно читать комментарий листинга 3.6, не видя его?"

Тем не менее можно. Специальная утилита `javadoc.exe` преобразует комментарии Javadoc в стандартную документацию класса. На рис. 3.8 показана веб-страница, автоматически сгенерированная на основе листинга 3.6. Таким способом задокументированы все библиотечные классы, что легко можно увидеть, читая справочник по классам Java.

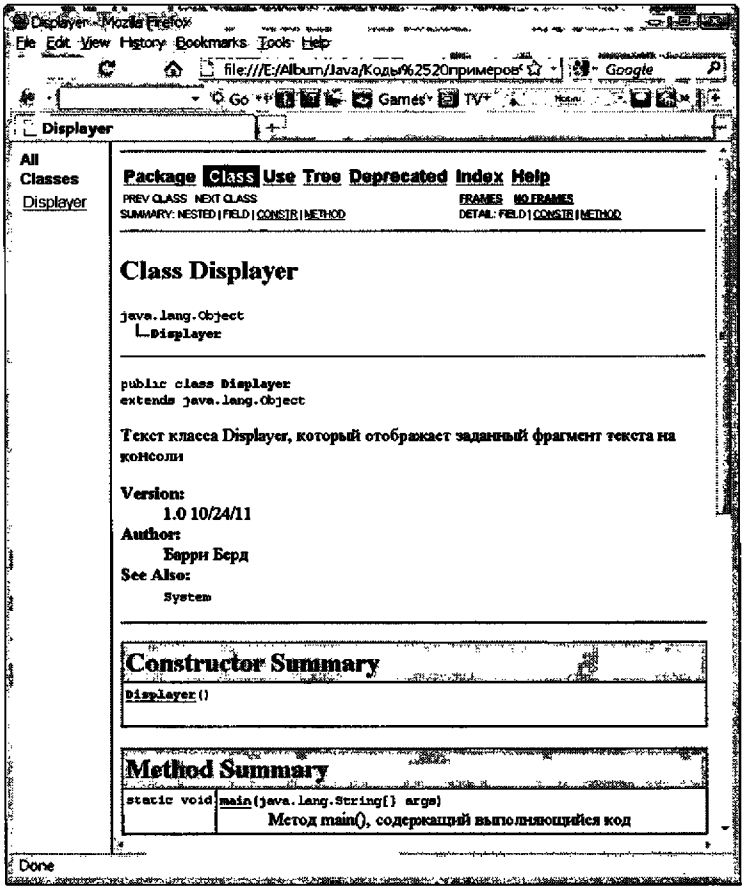


Рис. 3.8. Документация класса, приведенного в листинге 3.6; сгенерирована автоматически на основе комментариев Javadoc

Комментарии Javadoc очень полезны по следующим причинам.

- ✓ Часто единственный человек, читавший код Java, — это программист, писавший этот код. Все остальные применяют этот код, а для этого им нужно знать, что он делает. Как он работает, их не интересует. В комментариях Javadoc подробно описано, что делает данный код и как его следует использовать.
- ✓ Другие люди не смотрят в ваш код и не вносят в него изменений. Поэтому им лучше вообще не иметь к нему доступа, чтобы не внести в него ошибок.

- ✓ Некоторые люди могут не анализировать подробности работы кода. Все, что им нужно, есть в справочнике, автоматически сгенерированном утилитой `javadoc.exe`.
- ✓ Программист не создает две отдельные сущности: код Java — в одном месте и документацию к нему — в другом. Вместо этого он в одном и том же месте создает код и добавляет в него комментарии Javadoc. Он лучше всего знает код непосредственно во время его создания.
- ✓ Генерация веб-страницы на основе комментариев Javadoc выполняется автоматически. Поэтому вся документация продуцируется в одном и том же формате. Независимо от того, кто создавал код и каковы его личные предпочтения, вы увидите документацию в формате, аналогичном показанному на рис. 3.8. Этот формат документации хорошо знаком всем разработчикам приложений Java.

Чтобы сгенерировать веб-страницу документации на основе комментариев Javadoc, выберите в меню Eclipse команду Project⇨Generate Javadoc (Проект⇨Сгенерировать Javadoc). В открывшемся окне щелкните на кнопке Configure (Сконфигурировать). Найдите и выберите файл `javadoc.exe` (обычно он находится в каталоге `jdk` или `jre`). Задайте место расположения документации (по умолчанию предлагается каталог `doc`) и щелкните на кнопке Finish (Готово).

Не будьте слишком строги к старине Барри

В течение многих лет я учу студентов добавлять комментарии в код и убеждаю их не лениться делать это. В то же время многие годы я создаю учебные примеры кодов (как в листинге 3.1) без комментариев. Почему?

Три волшебных слова: "Знайте свою аудиторию". Когда вы пишете сложный код для решения реальных задач, ваша аудитория — программисты, менеджеры компаний, разрабатывающих программное обеспечение, и люди, которым зачем-то нужно понять, что и как вы сделали. Когда же я пишу примеры для книги, моя аудитория — это вы, начинающие программисты. Вместо того чтобы читать комментарии, вам лучше внимательно смотреть на инструкции Java. Более подробным комментарием служит текст книги.

Кроме того, я и сам немного ленив.

Использование комментариев для экспериментов с кодом

Если вы присутствовали при разговоре программистов, значит, вам довольно часто приходилось слышать специфические словечки *закомментировать* и *раскомментировать*. Когда вы пишете программу и что-то работает не так, как нужно, часто бывает полезно для эксперимента удалить часть кода, а затем вернуть его на место. Это поможет сравнить, как код работает с данным фрагментом и без него. Конечно, вам может не понравиться то, что код делает без фрагмента, поэтому удалять фрагмент кода навсегда нежелательно. Нужно иметь возможность вернуть его на место точно в том же виде, в каком он был до удаления. Проще всего это сделать, добавив две косые черты (`//`). Этим вы закомментируете код строки, т.е. превратите его в комментарий. Удалив две косые черты, вы раскомментируете код.

Например, инструкция

```
System.out.println("Мне нравится Java!");
```

превращается в комментарий

```
// System.out.println("Мне нравится Java!");
```

Компилятор Java игнорирует комментарий, поэтому закомментированный код ведет себя так, как будто его нет. В то же время его легко вернуть на место.

Традиционные (многострочные) комментарии не очень полезны для этой цели. Главная проблема состоит в том, что один многострочный комментарий может попасть внутрь другого. Предположим, вы хотите закомментировать следующие инструкции.

```
System.out.println("Родители!");
System.out.println("Ссорьтесь");
/*
 * Фраза умышленно размещена в четырех строках
 */
System.out.println("очень");
System.out.println("осторожно!");
```

Если вы окружите данный фрагмент символами многострочных комментариев, то получите следующий код.

```
/*
System.out.println("Родители!");
System.out.println("Ссорьтесь");
/*
 * Фраза умышленно размещена в четырех строках
 */
System.out.println("очень");
System.out.println("осторожно!");
*/
```

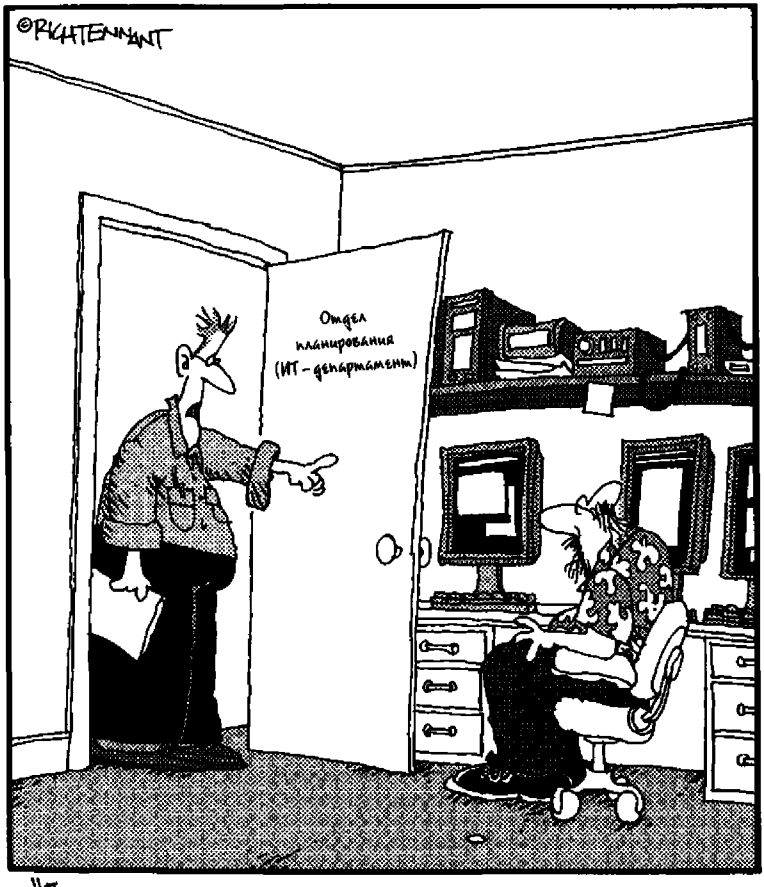
Первое вхождение символов `*/` завершает многострочный комментарий. В результате две последние инструкции оказываются не закомментированными, и, что хуже всего, символы `*/` сбивают с толку компилятор. Он не может понять, что нужно делать, и останавливается, сгенерировав сообщение об ошибке. Следовательно, чтобы закомментировать данный фрагмент, нужно раскомментировать внутренний фрагмент, а это настолько неудобно (ведь его нужно будет найти и опять закомментировать), что лучше отказаться от данного приема.

Большинство современных интегрированных сред разработки (в том числе Eclipse) предоставляют средства автоматического комментирования крупных блоков кода. Здесь я не описываю их только для того, чтобы не делать книгу слишком толстой.

Часть II

Создание программы на Java

The 5th Wave — Рич Теннант



"Программу по расчету приоритетов парковки придется свернуть. Сейчас нужно оперативно внедрить программу расчета маршрутов от офиса до ближайших кафе".

В этой части...

В этой части вы впервые напишете собственную программу на Java и на практике почувствуете, что это такое. Материал данной части излагается с учетом специфики Java, но, по сути, это учебник по программированию на любом языке. Из этой части вы узнаете об элементах программирования, таких как типы данных, переменные, логика выполнения, поток программы и т.п. Прочитав эту часть и попрактиковавшись с рассмотренными примерами и методиками программирования, вы сможете самостоятельно создавать довольно сложные программы.

Глава 4

Переменные и значения

В этой главе...

- Переменчивая переменная
- Оператор присваивания
- Типы значений и переменных
- Отображение текста
- Числа без десятичной точки
- Инициализация при объявлении
- Примитивные типы Java
- Ссылочные типы
- Объявление импорта
- Создание новых значений с помощью операторов

Приведенного ниже разговора между Джеком, ведущим передачи "Шоу на миллион", и Чарльзом, участником конкурса, никогда не было.

Чарльз: "При нехватке пищи асцидия поедает свой мозг и из животного превращается в растение".

Джек: "Это ваш окончательный ответ, Чарльз?"

Чарльз: "Э-м-м-мда. Я думаю, э-э-э-да-а-а..."

Джек: "Сколько денег сейчас на вашем счету, Чарльз?"

Чарльз: "Пятьдесят долларов и двадцать два цента".

Джек: "Ну что ж, вам придется обратиться в налоговую службу, потому что ваш ответ об асцидии совершенно правильный, и на ваш счет добавляется 1 миллион долларов!!! Что вы скажете об этом, Чарли?"

Чарльз: "Это справедливая награда за проявленные мною упорство, мужество и честность".

Некоторые элементы этого диалога можно представить несколькими строками кода на Java.

Переменчивая переменная

Независимо от того, как вы получили миллион долларов, данную величину можно обозначить с помощью переменной. Назовем эту переменную `amountInAccount` (сумма на счете). В принципе, в имени переменной можно использовать кириллические буквы. Компилятор Java не возражает, но почему-то так никто не делает. Возможно, потому что ключевые слова состоят только из латинских букв, и постоянно переключать раскладку клавиатуры было бы слишком хлопотно. Или потому, что программисты рассматривают это как еще один повод попрактиковаться в английском.

Или потому, что крепко держится давняя, прочная традиция, появившаяся во времена, когда компиляторы не принимали кириллицу. Но, какова бы ни была причина этого, не будем и мы нарушать традицию, по крайней мере в начале главы. Код, отображающий выигрыш Чарльзом миллиона долларов, показан в листинге 4.1.

Листинг 4.1. Использование переменной `amountInAccount`

```
amountInAccount = 50.22;
amountInAccount = amountInAccount + 1000000.00;
```

В коде используется переменная `amountInAccount`. Ее можно рассматривать, с одной стороны, как "заместитель" числа. Когда она равна `50.22`, в коде можно записать либо `50.22`, либо `amountInAccount`, результат будет один и тот же. С другой стороны, переменную можно рассматривать как имя ячейки в оперативной памяти компьютера. Сейчас в эту ячейку записано число `50.22`, но в любой момент и в любой строке кода в нее можно записать другое число. Естественно, когда вы запишете в переменную новое число, старое число будет удалено. Его больше не существует, и вы никогда не узнаете, что там было; способов сделать это не существует.

Необходимо также упомянуть о десятичном разделителе. В коде Java (как и в любом другом языке программирования) в качестве десятичного разделителя всегда используется точка. В других местах может быть по-разному. Программа Eclipse выводит на консоль точку. Если написать в Eclipse программу с графическим интерфейсом, то в русифицированной операционной системе число `50.22` будет отображено в текстовом поле как `50,22`. Впрочем, все зависит от параметров операционной системы, установленных в окне локализации. Единственное, в чем можно быть твердо уверенным, — это в том, что в коде Java всегда используется точка.

Содержимое ячейки памяти, обозначенной переменной, называется *значением* переменной. Иногда говорят "содержимое переменной", это синоним термина "значение".

На рис. 4.1 показано значение переменной до и после выполнения второй инструкции в листинге 4.1. После выполнения первой инструкции переменная `amountInAccount` содержит значение `50.22`, а в результате выполнения второй инструкции значение переменной стало равным `1000050.22`. Наглядно представьте себе значение переменной как маленькую коробочку, в которой находится бумажка с надписью `50.22` и которая окружена миллионами других таких же коробочек.

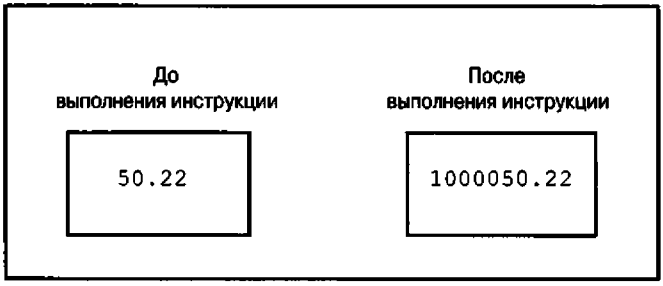


Рис. 4.1. Переменная `amountInAccount` до и после выполнения инструкции `amountInAccount = amountInAccount + 1000000`

Во время выполнения программы значение переменной изменяется (например, когда Джек перечисляет на счет Чарли миллион долларов, в ячейке памяти компьютера, стоящего в банке, появляется число, отображающее этот факт). Важно отметить, что в переменной может храниться не только число, но и строка символов, один символ или объект любого вида. Вид или формат объекта, хранящегося в переменной, называется его *типом*. Тип объекта может немного отличаться от типа переменной, в которой он хранится (обратите внимание на слово "немного": за ним много чего кроется, но об этом позже).

Типы объектов и переменных — чрезвычайно важное понятие в программировании. Они определяют, какие операции могут выполняться над переменными и как они выполняются. Этот вопрос подробнее рассматривается далее.



Понятия "переменная" и "имя переменной" — это не совсем одно и то же. Между ними существует ряд тонких различий. Однако часто эти понятия взаимозаменяемые. Слово "переменная" часто используется там, где, строго говоря, нужно написать "имя переменной", и наоборот. В рассмотренном выше примере `amountInAccount` — это имя переменной, обозначающее переменную. Впрочем, эти отличия больше академические. Даже профессиональные программисты пишут и говорят "изменить `amountInAccount`" — и никто не придирается, потому что всем понятно, что нужно изменить не имя, а значение переменной.

Каждое имя переменной — это *идентификатор*. Идентификатор — это строка символов, подчиняющаяся ряду правил. В частности, идентификатор должен состоять только из букв, цифр и символов подчеркивания (_), он не может содержать пробелов и т.п.

Прежде чем оставить листинг 4.1 и двинуться дальше, важно сделать еще несколько замечаний. Вы видите в нем числа `50.22` и `1000000`. Любой здравомыслящий человек не усомнится в том, что это числа, но программисты Java не относятся к этой категории. Они считают, что это не числа, а *литералы*.

Что же такого особенного в этих числах, и чем они заслужили такое название? Дело в том, что число — это абстрактное понятие, величина или значение чего-либо, а литерал — это объект программы. Он может иметь тип, как и переменная, в частности, быть числового или строкового типа. Например, `"Мне нравится Java!"` (см. листинг 3.1 в предыдущей главе) — это строковый литерал. Его-то уж никто не назовет числом.



Значение переменной может измениться, а значение литерала всегда остается одним и тем же.

Начиная с JRE 7 в числовые литералы можно добавлять символы подчеркивания. Вместо `1000000.00`, как в листинге 4.1, можно записать `1_000_000.00`. К сожалению, нельзя написать, как все привыкли, `1,000,000.00` — в США, `1.000.000,00` — в Германии или `1 000 000,00` — в России. Я не считаю это новшество очень ценным, потому что в коде программы литералы используются редко. Хранить данные следует в файлах и базах данных, а не в исходном коде Java. Тем не менее, если вам очень хочется написать `1 000 000,00`, можете отформатировать литерал (форматирование рассматривается в главах 10 и 11).

Оператор присваивания

В чем разница между инструкцией и оператором? Инструкция — это фрагмент программы, а *оператор* — это ключевое слово или символ, обозначающий некоторую операцию. Если инструкция не является блоком (см. главу 5), она заканчивается точкой с запятой. В Java есть приблизительно два десятка операторов, например оператор цикла `while`, оператор суммирования + и т.д. В данном разделе рассматривается оператор присваивания (символ =), который, как нетрудно догадаться по названию, присваивает чему-либо (чаще всего переменной) некоторое значение.

Я рекомендую вам выработать привычку читать инструкцию присваивания справа налево, как на рис. 4.2.

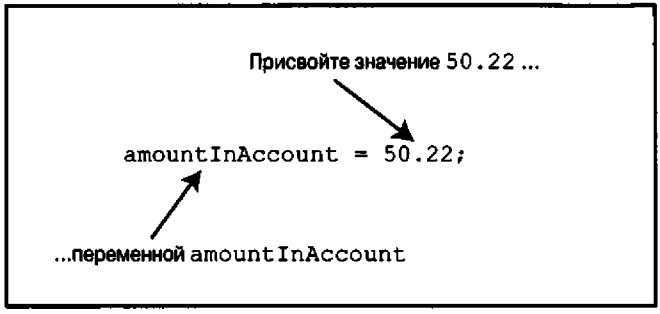


Рис. 4.2. Операция, выполняемая первой строкой листинга 4.1

Вторая инструкция листинга 4.1 немного сложнее (рис. 4.3). В одной инструкции выполняются сразу два оператора: сначала — оператор суммирования, а затем — оператор присваивания.

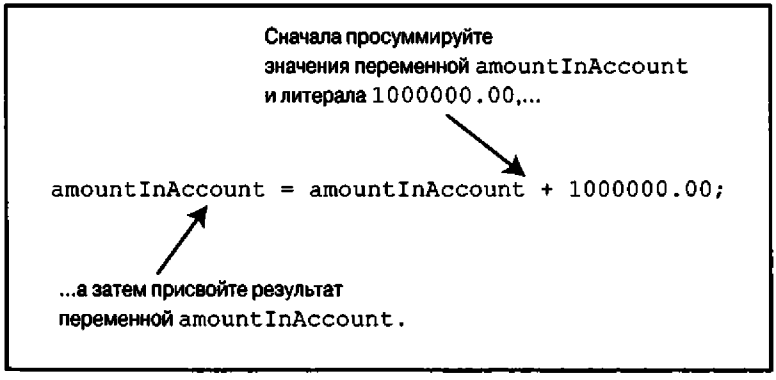


Рис. 4.3. Операция, выполняемая второй строкой листинга 4.1



При использовании оператора присваивания объект, которому что-либо присваивается, всегда находится слева от знака равенства. Операция присваивания — это запись некоторого значения в ячейку памяти, обозначенную переменной, расположенной слева от знака равенства.

Типы значений и переменных

Скорее всего, вам приходилось видеть научно-популярные фильмы, в которых вы летаете внутри компьютера и рассматриваете цепочки нулей и единиц, бегущие по цепям и записанные в квадратики ячеек памяти. Фактически компьютер работает только с нулями и единицами. Когда вы видите на экране букву J, в памяти находится не буква J, а цепочка 01001010. Процессор ничего не умеет делать с буквой J, он может работать только с последовательностями нулей и единиц. Соответственно, объем памяти измеряется в *битах*. Один бит — это одна позиция в цепочке, т.е. нуль или единица, а длина цепочки (количество позиций) — это количество битов. *Байт* — это 8 бит. По некоторым причинам объем памяти часто принято выражать в байтах, а не в битах.

Последовательность 01001010, обозначающая букву J, может также обозначать число 74. Эта же последовательность может обозначать число $1.0369608636003646 \times 10^{-43}$. Если же эта последовательность интерпретируется как пиксели на экране, она представляет изображение, состоящее из пикселей и показанное на рис. 4.4. Следовательно, значение 01001010 зависит от того, как оно интерпретируется программой.

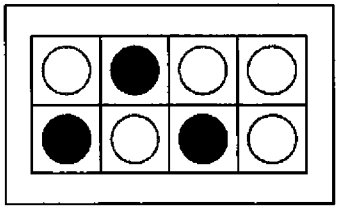


Рис. 4.4. Изображение из восьми пикселей на черно-белом экране

Программист должен каким-то образом сообщить программе о том, как ей следует интерпретировать цепочку 01001010. Для этого служит понятие типа. *Тип* — это категория значения. Он определяет, сколько битов компьютер выделяет для хранения переменной, что означают эти биты и как они будут обработаны. Например, если слева и справа от оператора + находятся переменные числового типа, компьютер выполнит операцию арифметического суммирования, а если переменные строкового типа, то результатом будет строка, полученная путем присоединения правой строки к левой.

Я скопировал строки из листинга 4.1 и поместил их в программу Java, приведенную в листинге 4.2. При выполнении этой программы на консоль был выведен результат, показанный на рис. 4.5.

Листинг 4.2. Программа, в которой используется переменная `amountInAccount`

```
class Millionaire {
  public static void main(String args[]) {
    double amountInAccount;
    amountInAccount = 50.22;
    amountInAccount = amountInAccount + 1000000.00;
    System.out.print("На вашем счету сейчас ");
```

```
    System.out.print(amountInAccount);
    System.out.println(" долларов.");
  }
}
```

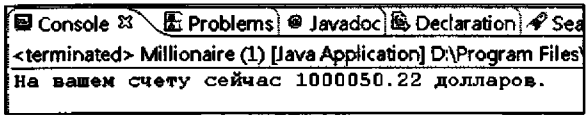


Рис. 4.5. Результат выполнения программы, показанной в листинге 4.2

Обратите внимание на первую строку в теле метода `main()`.

```
double amountInAccount;
```

Эта инструкция называется *объявлением переменной*. Когда вы помещаете объявление в код, вы как бы говорите программе: "В данном методе (или классе) будет использоваться переменная `amountInAccount`. Прошу любить и жаловать". Прочитав эту строку, компилятор поймет и запомнит, что `amountInAccount` — это имя переменной.

В данной строке используется ключевое слово `double` (двойной). Оно означает тип переменной `amountInAccount`. В переменной типа `double` может храниться числовое значение с плавающей точкой в диапазоне от -1.8×10^{308} до 1.8×10^{308}. Верхняя граница диапазона — огромное число. Если бы мы захотели записать его не в экспоненциальном, а обычном формате, нам пришлось бы написать 307 нулей и заполнить ими полстраницы. Это число намного больше, чем количество атомов в видимой части Вселенной. Оно специально выбрано таким большим, чтобы быть достаточным для любой ситуации, которая может встретиться в нашей жизни. Можно быть твердо уверенным, что на Земле никогда не появится богач, имеющий столько долларов.

Кроме `double`, в Java есть еще один числовой тип с плавающей точкой — `float`. От `double` он отличается тем, что имеет вдвое меньшую точность и занимает вдвое меньше места. Собственно, слово `double` (двойной) означает лишь то, что это значение вдвое длиннее, чем значение `float`. Какой тип лучше применять в программе? Применяйте `float`, только если в программе используются огромные массивы чисел (больше 10 Мбайт). Во всех остальных случаях используйте `double`.

Миллион долларов в листинге 4.2 впечатляет, но применять типы `double` и `float` в финансовых расчетах не рекомендуется. Они обладают следующей неприятной особенностью. После нескольких операций может оказаться, что на счете не `4`, а `3.9999999` доллара. Кроме того что это некрасиво выглядит, в расчет может вкрасться ошибка величиной в 1 цент, а это уже большая неприятность. Согласно бухгалтерским канонам, когда есть расхождение хотя бы в один цент, расчет неправильный, и бухгалтеру придется провести много дней над бумагами в поисках злосчастного цента. Поэтому в финансовых расчетах рекомендуется использовать тип `BigDecimal`, определенный в Java API специально для этих целей.

Плавающая десятичная точка

В Java есть два числовых типа с плавающей точкой — `double` и `float`. Чем они отличаются друг от друга? Объявляя переменную типа `double`, вы приказываете компьютеру выделить для нее 64 бит памяти, а для переменной `float` компьютер выделяет 32 бит.

Вы можете изменить листинг 4.2, объявив переменную `amountInAccount` типа `float`, а не `double`.

```
float amountInAccount;
```

Казалось бы, для хранения числа `50.22` тридцати двух битов более чем достаточно. Однако не совсем. Число 50 можно сохранить даже в 6 бит, но, кроме всего прочего, при сохранении числа важна точность. Если в листинге 4.2 объявить переменную `amountInAccount` с типом `float`, на консоль будет выведен результат `100050.25`, а не `100050.22`, как на рис. 4.5. Попробуйте и убедитесь в этом.

Это явно ошибочный результат. Чарльз получил на свой счет на три цента больше, чем ему причитается. Почему так получилось? Потому что тип `float` обеспечивает точность приблизительно 7 десятичных значащих цифр, а `double` — 14. В результате последняя цифра в значении типа `float` искажена.

Еще одна проблема с типом `float` состоит в следующем. Согласно синтаксическим правилам Java, литерал в формате, например, `50.22` имеет тип `double`. Чтобы литерал имел тип `float`, нужно добавить суффикс `f`, например `50.22f`. Если попытаться присвоить литерал типа `double` переменной типа `float`, произойдет потеря точности. Для хранения числа `50.22` нужна точность всего 4 значащие цифры, и точности типа `float`, казалось бы, достаточно, однако компилятор не знает, какая точность нужна. Он видит, что была точность 14, а стала 7 значащих цифр, и делает вывод, что точность потеряна. Поэтому компилятор откажется создать байтовый код и сгенерирует сообщение об ошибке.

Отображение текста

Последние три инструкции в листинге 4.2 выводят на консоль результат работы программы. В данном случае нужно отобразить на экране в одной строке несколько разных значений. Они отображаются тремя отдельными инструкциями. Первые две инструкции представляют собой вызовы библиотечного метода `System.out.print()`, а третья инструкция — вызов метода `System.out.println()`. Метод `print()` отображает текстовый фрагмент и оставляет курсор в текущей строке справа от последнего символа. Поэтому следующий метод `print()` или `println()` продолжает отображать символы в текущей строке консоли. Метод `println()` делает то же самое, но создает новую строку и переносит курсор в нее. Этот метод должен стоять последним, иначе последующие системные сообщения "приклеятся" к нашей фразе.



Метод `System.out.print()` выводит на консоль текст и оставляет курсор справа от последнего символа. Метод `System.out.println()` делает то же самое, но дополнительно создает новую строку и помещает в нее курсор вывода.

Числа без десятичной точки

"В 1995 году средняя американская семья имела 2,3 детей". Ха! С первыми двумя понятно, но покажите мне, как выглядит этот самый 0,3 ребенка! Очевидно, в конкретной семье количество детей может быть только целым, даже если последний ребенок очень маленький. То же самое справедливо и по отношению ко многим другим величинам. Только целым может быть количество планет в Солнечной системе, учеников в классе, домов на улице, книг на полке, телевизоров, проданных за месяц, в общем, всего, что измеряется в штуках, а не метрах или килограммах. Целыми числами принято выражать даже многие величины, которые по своей природе не являются целыми, например возраст человека. Следовательно, целые числа играют огромную роль в жизни людей. Поэтому в Java есть ряд специальных целочисленных типов (`byte`, `short`, `int` и `long`). В листинге 4.3 приведен пример использования целых чисел типа `int`.

Листинг 4.3. Использование типа `int`

```
class Лифт {
  public static void main(String args[]) {
    int весЧеловека;
    int грузоподъемностьЛифта;
    int количествоЧеловек;

    весЧеловека = 80;
    грузоподъемностьЛифта = 1000;
    количествоЧеловек = грузоподъемностьЛифта / весЧеловека;

    System.out.print("Лифт может поднять ");
    System.out.print(количествоЧеловек);
    System.out.println(" человек.");
  }
}
```

Рассмотрим, что делает эта программа. В отеле установлен пассажирский лифт с максимальной грузоподъемностью 1000 кг. Необходимо вычислить, сколько человек может войти в лифт одновременно. Предполагаем, что один человек весит 80 кг. Конечно, есть люди, которые весят больше 80 кг, но большинство людей все же весят меньше, поэтому будем считать, что заложен достаточный запас прочности. Программа присваивает исходным данным (`весЧеловека` и `грузоподъемностьЛифта`) необходимые значения и вычисляет значение `количествоЧеловек` путем деления двух переменных. После этого программа выводит на консоль фразу, сообщающую о количестве человек, которые могут войти в лифт (рис. 4.6).

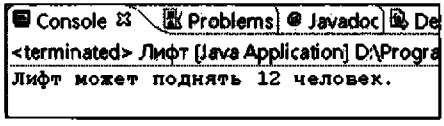


Рис. 4.6. Результат работы программы

Четыре целочисленных типа

В Java определены четыре типа переменных, в которых можно хранить целые числа — `byte`, `short`, `int` и `long`. Друг от друга они отличаются только количеством битов, занимаемых в памяти, и, следовательно, максимальным числом, которое можно сохранить в переменной данного типа. Например, в переменной типа `byte` нельзя сохранить число больше 127. Конечно, этого мало, но зато переменная `byte` занимает в памяти всего 8 бит — меньше, чем переменные других типов. Более подробное описание целочисленных типов приведено в табл. 4.1.

Проверим результат на калькуляторе, поделив `1000` на `80`. Однако в итоге мы получим не `12`, как сообщает нам программа, а `12.5`. Почему так получилось? Дело в том, что результатом деления целого на целое (`грузоподъемностьЛифта` на `весЧеловека`) в Java всегда является опять целое число. Важно помнить, что дробная часть при этом отбрасывается, а не округляется до ближайшего целого. Если нужно учитывать дробную часть, объявите переменные с типом `double`, а не `int`.

Инициализация при объявлении

Вернемся к листингу 4.3. В нем вы видите объявления трех переменных типа `int` с помощью трех инструкций. То же самое можно было сделать с помощью одной инструкции:

```
int весЧеловека, грузоподъемностьЛифта, количествоЧеловек;
```



Если переменные имеют разные типы, объявить их в одной инструкции невозможно. Например, объявить переменную `возрастЧеловека` типа `int` и переменную `суммаНаСчету` типа `double` можно только с помощью двух отдельных инструкций.

В листинге 4.3 одна инструкция объявляет переменную, а другая присваивает ей значение. Это можно сделать в одной инструкции.

```
int весЧеловека=80, грузоподъемностьЛифта=1000,
    количествоЧеловек=грузоподъемностьЛифта/весЧеловека;
```

В этом случае мы не можем сказать, что присвоили переменным значения, хотя оператор присваивания (знак =), казалось бы, присутствует. На самом деле мы *инициализировали* переменные во время их объявления. Не забывайте, что инициализация и присвоение — это разные операции.

Как и все в этом мире, инициализация переменных имеет не только преимущества, но и недостатки.

- ✓ **Когда вы объедините шесть инструкций листинга 4.3 в одну, код станет более кратким.** В одних случаях его будет легче читать, а в других — наоборот, тяжелее. Все зависит от того, как вы разместите инструкции на экране.
- ✓ **Инициализация переменных позволяет избежать некоторых ошибок.** Подробнее об этом сказано в главе 7.

✓ **В некоторых ситуациях у вас нет выбора: вы будете вынуждены инициализировать переменные вследствие специфики кода.** Такие ситуации рассматриваются в главе 6.

Примитивные типы Java

Типы `int` и `double`, которые мы рассматривали ранее, являются *примитивными типами*. Приятная особенность Java по сравнению с другими языками программирования состоит в небольшом количестве примитивных типов. Всего их 8, их полный список приведен в табл. 4.1.

Таблица 4.1. Примитивные типы Java

Имя типа	Объем занимаемой памяти, бит	Пример литерала	Диапазон значений
Целочисленные типы			
`byte`	8	`(byte)17`	От -128 до 127
`short`	16	`(short)17`	От -32768 до 32767
`int`	32	`17`	От -2147483648 до 2147483647
`long`	64	`17L` или `17l`	От -9223372036854775807 до 9223372036854775807
Типы с плавающей точкой			
`float`	32	`17.0f` или `17.0F`	От $-3{,}4\times10^{38}$ до $3{,}4\times10^{38}$
`double`	64	`17.0`	От -1.8×10^{308} до 1.8×10^{308}
Символьный тип			
`char`	16	`'a'`	Сотни тысяч символов в формате UTF-16
Булев тип			
`boolean`	8	`true`	Два значения — `false` и `true`

Наиболее полезны типы `int`, `double`, `char` и `boolean`. Без них вы не сможете работать. Тип `float` применяется, только когда нужно хранить в памяти огромные массивы (объемом более 10 Мбайт) чисел с плавающей точкой. Тип `long` используется, когда целые числа могут быть больше двух миллиардов. Это случается довольно редко. В практических задачах не нужно считать количество песчинок в пустыне или звезд на небе. Чаще всего подсчитывается количество предметов, таких как телевизоры, яблоки, дома и др., а для них два миллиарда — это огромное количество. Типы `byte` и `short` применяются очень редко в очень специфических случаях, например в процедурах шифрования данных при использовании специальных математических алгоритмов. Ранее мы уже рассматривали типы `int` и `double`. Теперь обсудим типы `char` и `boolean`.

Тип char

Не так давно люди думали, что компьютеры пригодны только для работы с числами и формулами. Теперь же, когда текстовые процессоры получили широкое распространение, никто больше так не думает. Сейчас компьютеры широко используются для обработки текстов, т.е. знаков пунктуации, букв и других символов.

Переменная типа `char` предназначена для хранения одного символа. В листинге 4.4 приведена простая программа, в которой используется тип `char`. Результат ее работы показан на рис. 4.7.

Листинг 4.4. Использование типа char

```
class CharDemo {
  public static void main(String args[]) {
    char myLittleChar = 'b';
    char myBigChar = Character.toUpperCase(myLittleChar);
    System.out.println(myBigChar);
  }
}
```

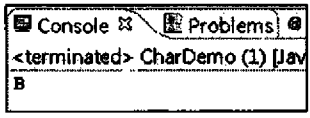


Рис. 4.7. Программа изменила регистр буквы

Переменной `myLittleChar` типа `char` присвоено значение `b`. Обратите внимание на то, что в программе на Java каждый литерал типа `char` заключен в одинарные кавычки.



Литерал типа `char` в исходном коде Java должен быть заключен в одинарные кавычки.

Следующая инструкция вызывает стандартный библиотечный метод `Character.toUpperCase()` и передает ему переменную `myLittleChar`. Этот метод возвращает букву, полученную при вызове, но теперь уже в верхнем регистре. Результат присваивается переменной `myBigChar` и отображается на консоли (см. рис. 4.7) с помощью метода `println()`.



Библиотечные функции Java API рассматривались в главе 3.

Можно ли сохранить в переменной типа `char` текстовую строку, как в следующей инструкции?

```
char myLittleChars = 'barry'; // Неправильно!
```

Нельзя. В переменной `char` в каждый момент времени может храниться только один символ. Сохранить более одного символа можно в переменной типа `String`, но это уже не примитивный тип. Мы рассмотрим тип `String` далее.



Если вы писали программы на других языках, вам должна быть знакома кодировка ASCII, которая используется в большинстве языков программирования. Однако в Java используется не ASCII, а Unicode. В кодировке ASCII каждый символ содержит 8 бит, а в Unicode — 8, 16 или 32 бит. Соответственно, в Unicode существует три вида кодирования — UTF-8, UTF-16 и UTF-32. В Java используется исключительно UTF-16. Это означает, что каждая переменная типа `char` занимает в памяти 2 или 4 байт. Впрочем, четыре байта используются только для хранения иероглифов, а для хранения букв практически любого языка (в том числе кириллических букв) используются два байта.

Тип `boolean`

В переменной типа `boolean` может храниться только одно из двух значений — `true` (истина) или `false` (ложь). Часто эти значения отображают одно из двух состояний, носящих разные названия: "Включено" или "Выключено", "Да" или "Нет", "0" или "1", "Истина" или "Ложь" и т.п. Использование типа `boolean` продемонстрировано в листинге 4.5. Результат работы этой программы показан на рис. 4.8.

Листинг 4.5. Использование переменной типа `boolean`

```
class Лифт2 {
  public static void main(String args[]) {
    System.out.println("Верно ли,");
    System.out.println("что лифт может поднять ");
    System.out.println("десять человек?");
    System.out.println();
    int весЧеловека = 80;
    int грузоподъемностьЛифта = 1000;
    int количествоЧеловек = грузоподъемностьЛифта / весЧеловека;
    boolean можетПоднять = количествоЧеловек >= 10;
    System.out.println("можетПоднять=" + можетПоднять);
  }
}
```

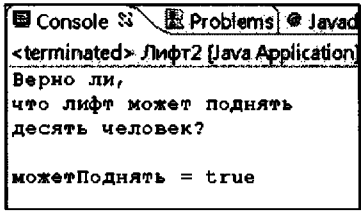


Рис. 4.8. Результат выполнения листинга 4.5

Переменная `можетПоднять` имеет тип `boolean`. Чтобы вычислить значение переменной `можетПоднять`, программа сравнивает значение `количествоЧеловек` с литералом `10`. Результат выполнения оператора сравнения `>=` представляет собой булево значение, которое присваивается переменной `можетПоднять`.

Поговорим немного о терминологии, потому что появилась необходимость представить новый термин. В Java, как и в любом другом языке программирования, *выражение* — это одно из трех: либо переменная, либо литерал, либо результат некоторой операции. Результатом выражения всегда служит некоторое значение. Следовательно, *выражение1+выражение2* — это тоже выражение. В приведенном выше примере подробнее рассмотрим такую инструкцию:

```
весЧеловека = 80;
```

В этой инструкции литерал `80` — это выражение. Его результат равен `80`. В инструкции

```
int количествоЧеловек = грузоподъемностьЛифта / весЧеловека;
```

результат выражения `грузоподъемностьЛифта/весЧеловека` присваивается переменной `количествоЧеловек`.



В коде Java выражением является все, что имеет какое-либо значение.

Это же справедливо и для фрагмента `количествоЧеловек>=10`. Его результат — булево значение (истина или ложь). Следовательно, это выражение, и его результат можно присвоить переменной булева типа. Попробуйте ввести не `10`, а `20`, и вы увидите, что результатом будет значение `false`.



В листинге 4.5 есть метод `System.out.println()`, которому не передается ни один параметр. Данная инструкция ничего не выводит, а только создает пустую строку, чтобы визуально отделить результат от вводной фразы (см. рис. 4.8).

Ссылочные типы

Выше мы рассмотрели примитивные типы Java. Теперь настало время ознакомиться с другой категорией типов — *ссылочными типами*. В листинге 4.6 продемонстрировано использование ссылочного типа `JFrame`, а на рис. 4.9 показан результат выполнения этой программы.

Листинг 4.6. Использование ссылочного типа

```
import javax.swing.JFrame;

class ShowAFrame {
  public static void main(String args[]) {
    JFrame myFrame = new JFrame();
    String myTitle = "Пустой фрейм";
    myFrame.setTitle(myTitle);
    myFrame.setSize(300, 200);
    myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    myFrame.setVisible(true);
  }
}
```

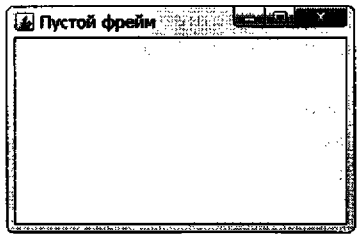


Рис. 4.9. Так выглядит пустой фрейм

Переменная типа `String` — это последовательность символов, т.е. значений типа `char`. Класс `String` определен в стандартной библиотеке Java API. В листинге 4.6 переменной `myTitle` типа `String` присвоена текстовая строка `Пустой фрейм`. Она отображается в заголовке фрейма, потому что переменная `myTitle` передана методу `setTitle()`.



Литерал типа `String` всегда должен быть заключен в двойные кавычки.

Стандартный библиотечный класс `JFrame` создает пустой фрейм, показанный на рис. 4.9. Фактически *фрейм* — это то, что в Windows называется окном, а в C# — формой. Чтобы сделать листинг 4.6 как можно проще, я не поместил во фрейм ничего — ни кнопок, ни надписей, но даже при этом фрейм является довольно сложной конструкцией. Поэтому пока что не пытайтесь понять все инструкции в листинге 4.6. Обратите внимание лишь на две переменные ссылочного типа: переменную `myTitle` типа `String` и переменную `myFrame` типа `JFrame`.

Описание типов `String` и `JFrame` можно найти в справочнике Java API. Вы увидите в справочнике, что `String` и `JFrame` — это имена классов.



Каждый класс (как библиотечный, так и созданный вами) — это ссылочный тип, а имя каждого класса — это имя ссылочного типа.

Чем же отличается ссылочный тип от примитивного? Чтобы создать переменную примитивного типа, достаточно объявить ее так:

```
double суммаНаСчету;
```

Или так:

```
double суммаНаСчету = 50.22;
```

А для создания объекта ссылочного типа объявить переменную недостаточно. Нужно еще и создать объект с помощью оператора `new`. Например, следующая инструкция объявляет переменную `myFrame` типа `JFrame`:

```
JFrame myFrame;
```

Но кроме этого, нужно создать объект класса `JFrame` и присвоить его объявленной выше переменной.

```
myFrame = new JFrame();
```

Эти две операции можно выполнить в одной инструкции.

```
JFrame myFrame = new JFrame();
```

Каждый класс Java — это ссылочный тип. Следовательно, если в коде есть объявление переменной ссылочного типа, значит, где-то в другом месте должно быть соответствующее определение класса с именем этого типа.

Когда вы объявляете переменную примитивного типа, например `int`, это означает, что где-то в компьютере для хранения значения этой переменной резервируется 4 байт памяти. Однако это справедливо только для примитивных типов, таких как `int` или `double`. Для ссылочного типа объявление означает нечто иное.

Поскольку существует класс `JFrame`, вы можете создавать объекты этого класса. Объект (т.е. экземпляр) класса `JFrame` — это фрейм, отображаемый на экране, например, при выполнении листинга 4.6. Объявив переменную `myFrame`, вы резервируете имя (но не область памяти, как в случае примитивных типов) для объекта типа `JFrame`. Иными словами, `myFrame` становится именем окна, видимого на экране (см. рис. 4.9). Однако не путайте имя и заголовок окна. Заголовок — это текстовая строка, отображаемая на экране. В частности, заголовок может содержать пробелы, а имя — нет. В программе можно создать произвольное количество объектов класса `JFrame`, т.е. окон (рис. 4.10). В них можно поместить разные элементы интерфейса и отобразить на них разные заголовки.

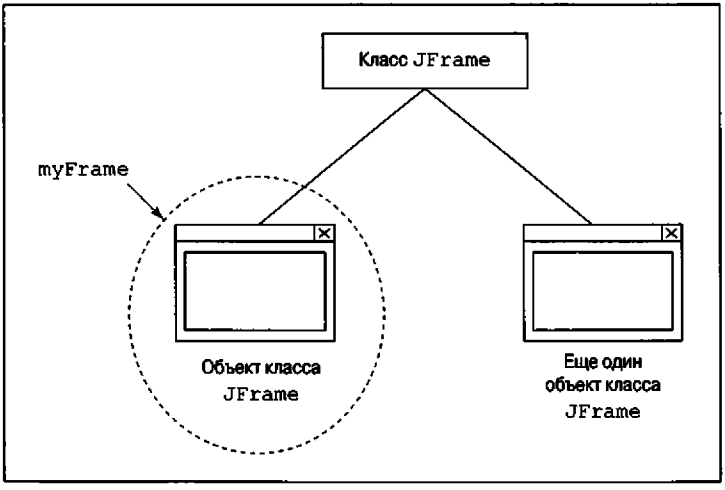


Рис. 4.10. Переменная `myFrame` ссылается на экземпляр класса `JFrame`



Объявляя переменную с помощью имени класса, вы сообщаете компилятору о том, что в этой переменной может храниться ссылка на экземпляр (объект) данного класса.



В листинге 4.6 фраза `JFrame myFrame` резервирует имя `myFrame` за объектом типа `JFrame`. В этой же инструкции фраза `new JFrame()` создает объект (т.е. экземпляр класса) типа `JFrame`. Не забывайте, что нужно не только объявить переменную, но и создать объект с помощью оператора `new`, который более подробно рассматривается в главе 7.

Все состоит из примитивных типов

"А как же ссылочные типы? Ведь вы только что сказали, что типы делятся на примитивные и ссылочные..." Действительно, это так. Но в определении любого ссылочного типа используются другие примитивные и ссылочные типы, причем в определении последних также используются примитивные и ссылочные типы. Если расписать эту древовидную структуру до конца, то в конечном итоге обязательно окажется, что все состоит только из примитивных типов.

Рассмотрим, например, фрейм, т.е. класс `JFrame`, отображающий на экране окно. При создании экземпляра окна нужно задать его параметры, такие как ширина, высота и расположение на экране. Поэтому в определении класса `JFrame` можно увидеть объявление этих параметров как переменных примитивных типов.

```
int width;
int height;
int x;
int y;
```

Класс — это всего лишь набор объявлений переменных и методов. Некоторые объявленные переменные имеют примитивные типы, а некоторые — ссылочные. Например, в классе `JFrame` в переменных (их называют полями) хранятся параметры фрейма (такие, как размеры окна, цвет фона, шрифт, текст заголовка и т.п.), а методы выполняют все операции, необходимые для отображения фрейма на экране.

Объявление импорта

Откуда компилятор знает, где нужно искать определение класса `JFrame`? Об этом ему сообщает объявление `import`. Ниже приведен фрагмент листинга 4.6.

```
import javax.swing.JFrame;

class ShowAFrame {
  public static void main(String args[]) {
    JFrame myFrame = new JFrame();
```

В данном случае объявление `import` сообщает компилятору о том, что, во-первых, `JFrame` является именем класса и, во-вторых, определение этого класса находится в библиотеке `javax.swing`. После этого имя класса `JFrame` можно использовать в коде.

Объявление импорта можно не добавлять, но тогда придется при каждом упоминании класса `JFrame` писать его полное имя `javax.swing.JFrame`. Тогда код листинга 4.6 будет выглядеть следующим образом.

```
class ShowAFrame {
  public static void main(String args[]) {
    javax.swing.JFrame myFrame = new javax.swing.JFrame();
    String myTitle = "Пустой фрейм";
    myFrame.setTitle(myTitle);
    myFrame.setSize(300, 200);
    myFrame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_
CLOSE);
    myFrame.setVisible(true);
  }
}
```

На первый взгляд, судя по сказанному выше, объявление `import` довольно простое, однако на самом деле оно весьма коварное. Дело в том, что в стандартном дистрибутиве Java поставляется огромное количество классов и библиотек, причем многие имена классов дублируются (естественно, они определены в разных библиотеках). К счастью, современные среды разработки (включая Eclipse) предоставляют команды упорядочения объявлений `import` и удобные инструменты работы с библиотеками. Например, во всплывающей подсказке приводятся имена библиотек, в которых можно найти нужный класс.



Более подробно объявление `import` рассматривается в главах 5, 9 и 10.

Создание новых значений с помощью операторов

Что может быть приятнее, чем иметь дело со старым добрым другом — знаком "плюс"? Впервые вы встретились с ним еще в школе, когда выводили неуверенным детским почерком классическое 2+2. Фактически выражение "2+2" стало метафорой, обозначающей нечто очень простое. Каждый раз, когда человек видит знак "плюс", он думает: "Слава Богу, это не так уж и сложно!" Но это не всегда так уж просто.

Есть знак "плюс" и в Java, но здесь он используется для нескольких целей. Конечно, его можно использовать для суммирования чисел.

```
int яблоки, апельсины, фрукты;
яблоки = 5;
апельсины = 16;
фрукты = яблоки + апельсины;
```

Кроме того, знак "плюс" можно использовать для объединения двух строк.

```
String началоГлавы =
  "Ранним солнечным утром Штирлиц шел по " +
  "улице Фридрихштрассе на встречу со своим агентом.";

System.out.println(началоГлавы);
```

Это часто бывает удобно, потому что в Java нельзя размещать объект `String` в нескольких строках. Иными словами, следующий код неправильный.

```
String началоГлавы =
  "Ранним солнечным утром Штирлиц шел по
   улице Фридрихштрассе на встречу со своим агентом.";
System.out.println(началоГлавы);
```



Во всех языках программирования операция объединения строковых значений типа `String` называется *конкатенацией*.

Знак "плюс" можно использовать даже для неявного преобразования типа `int` в тип `String`.

```
int яблоки, апельсины, фрукты;
яблоки = 5;
апельсины = 16;
фрукты = яблоки + апельсины;
System.out.println("В корзине есть " +яблоки+ "яблок," +
      апельсины+ " + " апельсин, всего - " +фрукты+ " фруктов");
```

Конечно, есть в Java и вечный спутник плюса — знак "минус" (но его нельзя использовать со значениями типа `String`). Ниже показано, как узнать количество яблок.

```
яблоки = фрукты - апельсины;
```

Аналогично этому знак `*` используется для умножения, а `/` — для деления двух значений.

```
double тариф, кОплате, отработаноЧасов;
тариф = 6.25;
отработаноЧасов = 35;
кОплате = тариф * отработаноЧасов;
System.out.println(кОплате);
```

Пример деления можно увидеть в листинге 4.3.



При делении значения `int` на значение `int` результатом также будет значение типа `int`. Учитывайте, что компьютер не округляет, а отбрасывает остаток деления. Например, инструкция `System.out.println(11/4)` выведет на консоль значение 2, а не 3. Если же один из операндов имеет тип `double`, результат также будет иметь тип `double`. Например, инструкция `System.out.println(11.0/4)` выведет на консоль число `2.75`.

Еще один полезный арифметический оператор Java — символ процента (`%`), вычисляющий остаток от деления целых чисел. Например, инструкция `System.out.println(11%4)` выведет на консоль число 3. Оператор вычисления остатка часто бывает очень полезным. Пример его использования приведен в листинге 4.7. Результат показан на рис. 4.11.

Листинг 4.7. Вычисление времени в часах и минутах

```
import static java.lang.System.out;
class MakeChange {
  public static void main(String args[]) {
  int общееВремя = 200;
  int часов, минут;
  часов = общееВремя / 60;
  минут = общееВремя % 60;
  out.println("Общее время = " + часов + " часа и "+ минут +
              " минут.");
  }
}
```

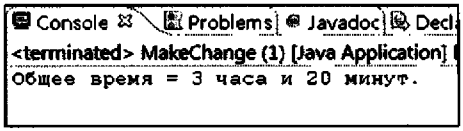


Рис. 4.11. Результат работы программы `MakeChange`

Инициализировать можно раз, а присваивать — много раз

Инициализация переменной выполняется при ее объявлении. Следовательно, инициализировать переменную можно только один раз. Если бы в листинге 4.7 нужно было вычислять время в часах и минутах многократно, то инициализация переменной потеряла бы смысл, потому что объявить переменную можно только один раз. Если в классе встретится повторное объявление одной и той же переменной, компилятор сообщит об ошибке. Изменять значение переменной много раз можно путем присваивания ей разных значений, например в цикле.

Статический импорт

Обратите внимание на объявление импорта в листинге 4.7.

```
import static java.lang.
System.out;
```

Сравните его с объявлением импорта в листинге 4.6.

```
import javax.swing.JFrame;
```

Добавив объявление `import static java.lang.System.out` в листинг 4.7, я упростил код класса. Теперь вместо `System.out.println()` можно писать `out.println()`, в результате чего код становится короче и понятнее. В некоторых случаях это позволяет сделать инструкцию короче и не переносить ее на следующую строку, в результате чего ее легче читать.

Что же означает ключевое слово `static` в данном случае? Мы подробно рассмотрим это в главе 10, а пока что запомните лишь, что оно позволяет удалить слово `System` из имени класса.

Операторы инкремента и декремента

В Java есть четыре оператора, которые предназначены исключительно для того, чтобы облегчить программистам их работу: два оператора инкремента и два оператора декремента. Оператор инкремента (++) добавляет к переменной единицу, а оператор декремента (--) вычитает единицу. Пример работы оператора инкремента приведен на рис. 4.12.

На рис. 4.13 показан результат, выведенный на консоль программой, приведенной на рис. 4.12.

```
import static java.lang.System.out;
class PreincrementDemo {
  public static void main(String args[]) {
    int n = 27;

    ++n;
    out.println(n);
    out.println(++n);
    out.println(n);
  }
}
```

Здесь n=28.

Вывод n=28.

Здесь n=29
и выводится 29.

Вывод n=29.

Рис. 4.12. Использование оператора префиксного инкремента

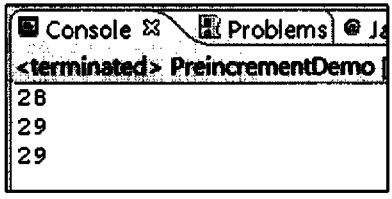


Рис. 4.13. Результат работы программы, показанной на рис. 4.12

В зависимости от того, где находится оператор ++, он называется по-разному и может играть разную роль. Если он находится перед переменной, он называется *префиксным инкрементом*, а после переменной — *постфиксным инкрементом*. В обоих случаях компьютер добавляет к переменной единицу, но в случае префиксного инкремента единица добавляется перед выполнением любой части инструкции, а в случае постфиксного инкремента — после выполнения любой части инструкции, в которой используется оператор инкремента.

Чтобы понять, как это работает, посмотрите на инструкцию, выделенную полужирным шрифтом, на рис. 4.12. Компьютер сначала прибавляет к переменной n единицу, а после этого выводит значение выражения на консоль. Поэтому после выполнения инструкции переменная n имеет то же значение, которое выведено на консоль, т.е. 29.

При использовании постфиксного инкремента (рис. 4.14) компьютер делает наоборот: сначала выводит значение выражения на консоль, а затем прибавляет единицу. Результат показан на рис. 4.15. Сравните его с рис. 4.13, на котором показан результат префиксного инкремента. Как видите, инструкция, выделенная полужирным шрифтом, вывела на консоль значение n, которое было до выполнения оператора постфиксного инкремента.

```
import static java.lang.System.out;
class PostincrementDemo {
  public static void main(String args[]) {
    int n = 27;

    n++;
    out.println(n);
    out.println(n++);
    out.println(n);
  }
}
```

Здесь n=28.

Вывод n=28.

Здесь n=29,
но выведено 28.

Вывод n=29.

Рис. 4.14. Использование оператора постфиксного инкремента

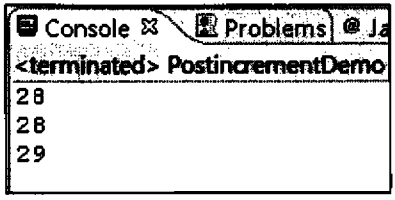


Рис. 4.15. Результат работы оператора постфиксного инкремента



Какой оператор следует использовать в конкретном случае — префиксный или постфиксный? Конечно, все зависит от ситуации. В каждом случае нужно внимательно проанализировать, что должен сделать компьютер: сначала выполнить внешнюю операцию, а затем увеличить переменную на единицу, или наоборот. Интересно отметить, что если нужно только увеличить значение переменной, оператор инкремента может использоваться без оператора присваивания.

```
n++;
```

В этом случае, конечно, тип инкремента (префиксный или постфиксный) безразличен.

Все сказанное выше справедливо и для оператора декремента за исключением того, что он не увеличивает, а уменьшает значение переменной на единицу. Соответственно, в Java существуют два типа оператора декремента — префиксный и постфиксный.

- ✓ Префиксный оператор декремента (`--n`) сначала вычитает из переменной единицу, а затем использует результат в выражении.
- ✓ Постфиксный оператор декремента (`n--`) сначала использует переменную в выражении, а затем вычитает из переменной единицу.

Описать поведение префиксных и постфиксных операторов инкремента и декремента можно одним из двух способов: так, как это почти все понимают, и правильно. Приведенное выше объяснение (в терминах "сначала" и "затем") принадлежит к первой категории (так, как это почти все понимают). К сожалению, эта концепция не совсем правильная. Увидев в коде `++` или `--`, вы можете думать в терминах времени — до того и после того. Но профессиональные программисты иногда применяют операторы `++` и `--` такими вычурными способами, что понятия "сначала" и "после" становятся бессмысленными. Поэтому, если вы хотите стать профессиональным программистом, напрягитесь и представьте себе эти операторы в терминах инструкций и выражений.

В первую очередь вспомните, что инструкция приказывает компьютеру сделать что-либо, а выражение возвращает значение (иными словами, выражение равно результирующему значению). К какой из этих двух категорий принадлежит фраза `n++`? Как ни странно, к обеим! Это инструкция и выражение одновременно.

В предыдущем примере перед тем, как компьютер выполняет код `out.println(n++)`, переменная `n` равна `28`.

- ✓ Как инструкция фраза `n++` приказывает компьютеру увеличить `n` на единицу.
- ✓ Как выражение значение `n++` равно `28`, а не `29`.

Таким образом, несмотря на то что компьютер добавляет единицу к переменной `n`, код `out.println(n++)` отображает `28`.

Почти все, что вы прочитали о фразе `n++`, справедливо и по отношению к фразе `++n`. Единственное отличие состоит в том, что `++n` ведет себя более "прилично", т.е. является "правильным выражением", а именно — оно равно результату суммирования переменной и единицы, в то время как `n++` не равно ему.

- ✓ Как инструкция фраза `++n` приказывает компьютеру увеличить значение `n` на единицу.
- ✓ Как выражение значение `++n` равно результату добавления единицы, т.е. числу `29`.

Следовательно, инструкция `out.println(++n)` приказывает компьютеру добавить к переменной единицу, а вызов метода `out.println(++n)` ведет себя так же, как `out.println(29)`.

Чтобы легче было удержать в памяти эти правила, запомните их так: "В префиксном операторе выражение равно выражению (как и положено), а в постфиксном выражение равно переменной (что весьма странно)".



Вместо `++n` можно написать `n=n+1`, результат будет тот же. Поэтому некоторые люди думают, что операторы `++` и `--` предназначены только для того, чтобы сэкономить несколько нажатий клавиш и позиций в коде. Но это в корне неправильно. Главная побудительная причина использования этих операторов состоит в стремлении устранить ошибки, вызываемые повторным вводом длинных имен. Короткие идентификаторы вроде `n` используются редко, потому что в профессиональном программировании принято закладывать в имя переменной ее описание, например `numberOfBunnies`. Возможность не вводить повторно длинный идентификатор, согласитесь, весьма полезная.

В простом выражении вроде `numberOfBunnies++` преимущества оператора инкремента все же менее ощутимы, чем в сложных выражениях вроде следующего.

```
quantityChecked =
   inventoryItems[(quantityReceived--*itemsPerBox+17)]++;
```

Операторы присваивания

Прочитав предыдущий раздел об операторах, которые добавляют или вычитают единицу, вы, наверное, думаете: "Можно ли использовать эти операторы для добавления двух, пяти или тысячи?" Можно ли написать `n++++`? Нет, нельзя. Если написать так, компилятор вернет сообщение об ошибке в коде.

Но, к счастью, в Java есть много операторов присваивания, с помощью которых можно добавлять, вычитать, делить и умножать любые числа, не вводя повторно имя переменной. С их помощью можно также выполнять другие сложные операции. В листинге 4.8 иллюстрируется использование операторов присваивания разных видов и в разных ипостасях. Выведенный на консоль результат показан на рис. 4.16.

Листинг 4.8. Использование операторов присваивания

```
class ОператорыПрисваивания {
  public static void main(String args[]) {
    int n = 27;
    int m = 53;
    n += 1;
    System.out.println(n);
    n += 5;
    System.out.println(n);
    n += m;
    System.out.println(n);
    n *= 2;
    System.out.println(n);
    System.out.println(n -= 7);
    System.out.println(n = 100);
  }
}
```

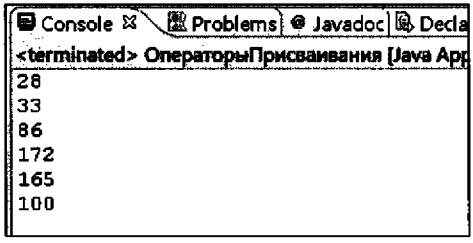


Рис. 4.16. Результат выполнения листинга 4.8

Как видите, операторов присваивания довольно много, и их можно использовать по-разному. С их помощью можно суммировать, вычитать, умножать и делить переменную на любое число. Обратите внимание на то, что фраза `+=5` добавляет `5`, а фраза `*=5` умножает на `5`. Можно также использовать справа от оператора другую переменную или любое выражение.

Последние две инструкции в листинге 4.8 демонстрируют специальные свойства операторов присваивания: их можно использовать внутри других инструкций. Предпоследняя инструкция вызова `println()` вычитает `7` из переменной `n`. До этого она была равна `172`, следовательно, инструкция выводит на консоль число `165`.



Последняя инструкция в листинге 4.8 одновременно присваивает переменной значение `100` и выводит его на консоль. Следовательно, инструкцию присваивания можно писать внутри любой другой инструкции. Не забывайте об этой полезной особенности оператора присваивания.



Каждый оператор присваивания играет в коде двойную роль: работает и как инструкция, и как выражение. Во всех случаях выражение равно присваиваемому значению. Например, перед выполнением инструкции `out.println(n-=7)` переменная `n` была равна `172`. В качестве инструкции фраза `n-=7` приказывает компьютеру вычесть из переменной число `7` (в результате чего переменная становится равной `165`). А в качестве выражения фраза `n-=7` должна быть чему-то равна, и равна она присваиваемому значению. Это означает, что фраза `out.println(n-=7)` выводит на консоль то же выражение, что и фраза `out.println(165)`.

Если вы забыли, какая разница между инструкциями и выражениями, прочитайте еще раз врезку "Инструкция и выражение", приведенную ранее.

Глава 5

Управляющие инструкции

В этой главе...

- Принятие решения с помощью инструкции if
- Условия с операторами сравнения и логическими операторами
- Вложение инструкций if
- Переключатель switch

Телевизионное шоу "Деннис-мучитель" продержалось на CBS с 1959 по 1963 год. Мне запомнился в нем эпизод, в котором мистер Вильсон должен был принять важное решение. Я уже забыл, в чем оно состояло (то ли смена работодателя, то ли переезд в другой город), но я до сих пор отчетливо помню, как он просидел в своем дворе весь день, прихлебывая лимонад и уставившись в одну точку. При этом зловредный Деннис постоянно пытался прервать это занятие, в чем и состоял весь юмор происходящего.

В этом эпизоде больше всего на меня произвело впечатление (вот почему он мне запомнился) непоколебимое намерение мистера Вильсона принять решение. Он оставил свои повседневные дела, перестал ходить по магазинам, не пошел в пивной бар с приятелями, а надолго погрузился в свои мысли. Он сидел во дворе, аккуратно делая мысленные отметки на воображаемом листе баланса доводов "за" и "против". Много ли людей способны принимать решения подобным способом?

В то время я был довольно молод, и мне еще не приходилось принимать решения, которые могли бы повлиять на мою судьбу или благополучие моей семьи. Но меня поразило и заинтересовало, что чувствует человек, принимающий решение? Нужно ли для этого сидеть часами неподвижно? Что лучше: взвесить тщательно все за и "против", сначала составив два списка доводов (преимуществ и недостатков), или положиться на случай и принять решение на основе интуитивного импульса? Сейчас я понимаю, что первая стратегия предпочтительнее почти всегда.

Принятие решения с помощью инструкции `if`

Когда вы пишете компьютерную программу, вы постоянно создаете "вилки решений". Например, правильно ли пользователь ввел пароль? Если да, нужно предоставить ему доступ к приложению, а если нет — предложить ему попытаться еще раз. В этом случае профессиональные программисты говорят, что программа *ветвится*. Вы как бы создаете развилку на дороге, как в древнерусской былине, в которой говорится: "Налево пойдешь — коня потеряешь, направо пойдешь..." Для реализации вилок в Java есть средство, позволяющее программе пойти по одному из двух путей. Это инструкция `if`.

Угадайте число

В листинге 5.1 приведена программа, иллюстрирующая использование инструкции `if`. На рис. 5.1 приведен результат, полученный путем выполнения программы два раза.

Листинг 5.1. Игра с угадыванием числа

```
import static java.lang.System.out;
import java.util.Scanner;
import java.util.Random;
class GuessingGame {
  public static void main(String args[]) {
    Scanner keyboard = new Scanner(System.in);
    out.print("Введите число от 1 до 10: ");
    int inputNumber = keyboard.nextInt();
    int randomNumber = new Random().nextInt(10) + 1;
    if (inputNumber == randomNumber) {
      out.println("**********");
      out.println("*Вы выиграли.*");
      out.println("**********");
    } else {
      out.println("Вы проиграли.");
      out.print("Случайное число равно ");
      out.println(randomNumber + ".");
    }
    out.println("Спасибо за игру.");
  }
}
```

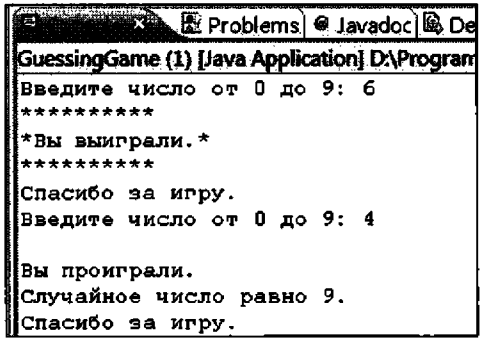


Рис. 5.1. Два сеанса выполнения программы

Программа, приведенная в листинге 5.1, играет с пользователем в игру на отгадывание числа. Если пользователь угадал случайно сгенерированное число, на консоль выводится сообщение `Вы выиграли`. В противном случае выводится сообщение `Вы проиграли`.

Ввод текста с клавиатуры

В листинге 5.1 есть следующие инструкции.

```
import java.util.Scanner;
    Scanner keyboard = new Scanner(System.in);
    int inputNumber = keyboard.nextInt();
```

Они принимают с клавиатуры введенное пользователем число и записывают его в переменную `inputNumber` типа `int`. Если сейчас эти инструкции вам не понятны, не беспокойтесь. Можете скопировать их слово в слово и применять в других программах. Отметим лишь, что метод `nextInt()` переменной `keyboard` принимает с клавиатуры введенную пользователем строку и преобразует ее в целое число.

В этих трех инструкциях я придумал только имена `inputNumber` и `keyboard` (следовательно, вместо них можно использовать любые другие имена). Все остальные слова — это имена библиотечных инструментов API и ключевые слова Java, поэтому их нельзя изменять.

Далее мы будем использовать данный шаблон, поэтому рассмотрим его немного подробнее.

- ✓ **В объявлении импорта `java.util.Scanner` ключевое слово `static` не используется.**

  Импорт класса `Scanner` отличается от импорта `System.out`. Во втором случае используется ключевое слово `static` (см. листинг 5.1). Различие между ними в том, что `Scanner` — это имя класса, а `System.out` не является именем класса.



  Использование ключевого слова `static` в объявлении импорта рассматривалось в главе 4. Более подробно ключевое слово `static` обсуждается в главе 10.

- ✓ **Имя `System.in` обозначает клавиатуру.**



  Чтобы получать символы с какого-либо другого устройства (не клавиатуры), нужно ввести между скобками другое имя, обозначающее это устройство.

  Что еще можно ввести между скобками и откуда еще можно получать символы? Об этом вы узнаете из главы 8.



  Имя `keyboard` (клавиатура) в листинге 5.1 напоминает программисту о том, что эта переменная обозначает ряд пластиковых клавиш перед его компьютером. В данном случае можно применить любое другое имя. Компилятор знает о том, что оно обозначает клавиатуру, из других источников. О том, что это клавиатура, компьютеру сообщает имя `System.in`, которое обозначает стандартное системное устройство ввода. Инструкция `Scanner keyboard=new(System.in)` связывает имя `keyboard` со стандартным устройством ввода, т.е. с клавиатурой.

- ✓ **Метод `nextInt()` принимает из заданного источника (в данном случае — из стандартного устройства ввода) строку и преобразует ее в целое значение.**

  Если пользователь должен ввести значение типа `double` (т.е. число, содержащее десятичную точку), используйте метод `nextDouble()`. Если ожидается ввод значения `true` или `false`, используйте метод `nextBoolean()`. Для ввода значения строкового типа `String` используйте метод `next()`.



В листинге 5.3 будет показан пример ввода строкового значения с клавиатуры. Пример ввода одного символа будет показан в листинге 6.4. Пример ввода целой строки за один раз есть в главе 8.

✓ **С клавиатуры можно прочитать несколько значений одно за другим.**

Для этого запишите инструкцию `keyboard.nextInt()` несколько раз подряд.



Пример ввода нескольких значений с клавиатуры показан в листинге 5.4.

Генерация случайных чисел

Для генерации *случайных чисел* в Java используется класс `Random`. Чтобы его можно было использовать в программе, нужно добавить в нее объявление импорта `import java.util.Random`. Кроме того, нужно создать объект типа `Random` с помощью оператора `new`. Метод `nextInt(10)` возвращает случайное целое число с равномерным распределением в диапазоне от 0 до 9 (аргументом является количество чисел в диапазоне). В листинге 5.1 есть две строки, связанные с генерацией случайных чисел.

```
import java.util.Random;
int randomNumber = new Random().nextInt(10) + 1;
```

Единица добавлена только для того, чтобы нижней границей диапазона был не нуль, а единица.

Важно отметить, что генерируемые таким образом числа являются, строго говоря, не случайными, а *псевдослучайными*. Вам, конечно, известно, что компьютер — это детерминированное устройство. Это означает, что любой расчет, повторенный еще раз, дает точно такой же результат. Компьютер не может отклониться от жестко заданного алгоритма ни на бит. Откуда же взяться случайности? И действительно, полученные таким образом числа не являются случайными, они только похожи на случайные. Если сгенерировать ряд псевдослучайных чисел еще раз, вы получите точно такой же ряд, как в первый раз. Каждый раз вы будете получать один и тот же ряд чисел. Это довольно полезное свойство псевдослучайных чисел, потому что оно позволяет точно повторять расчет в целях отладки. Чтобы получить другой ряд псевдослучайных чисел, нужно изменить *стартовое число* генератора с помощью метода `setSeed()`.

Обратите внимание на то, что в листинге 5.1 нет переменной, указывающей на объект генератора псевдослучайных чисел. Программа генерирует число только один раз, поэтому объект `new Random()` используется без имени.

Чтобы сгенерировать ряд истинно случайных чисел, нужно каким-либо образом получить что-то случайное. Как это сделать, если компьютер строго детерминированное устройство и ничего случайного в нем нет? Наиболее популярный способ состоит в считывании значения времени с часов, встроенных в компьютер. Момент считывания не связан с часами, поэтому, если отбросить часы, минуты и секунды, количество миллисекунд будет истинно случайным числом.

Инструкция `if`

В листинге 5.1 генерация случайного числа и прием текста с клавиатуры — вспомогательные процессы. Поэтому пока что не концентрируйте на них свое внимание.

Листинг 5.1 предназначен для демонстрации принятия решения программой с помощью инструкции `if`. Эта инструкция создает "развилку" на пути выполнения программы. Она приказывает компьютеру принять решение, по какому пути пойти: вывести на консоль фразу `Вы выиграли` или фразу `Вы проиграли`. Компьютер принимает решение на основе проверки *условия*, которое расположено в скобках после ключевого слова `if`. В листинге 5.1 проверяется следующее условие:

```
inputNumber == randomNumber;
```

В переводе на русский язык оно звучит так: "Равно ли значение `inputNumber` значению `randomNumber`?" Если условие равно `true` (т.е. значения равны), компьютер выполняет инструкции, расположенные между закрывающей круглой скобкой условия и ключевым словом `else`. Если же условие равно `false`, компьютер выполняет инструкции, расположенные после ключевого слова `else`. Таким образом, инструкция `if` создает развилку на пути выполнения программы (рис. 5.2). Инструкцию `println("Спасибо за игру!")` компьютер выполняет в любом случае, потому что она не входит ни в блок `if`, ни в блок `else`.

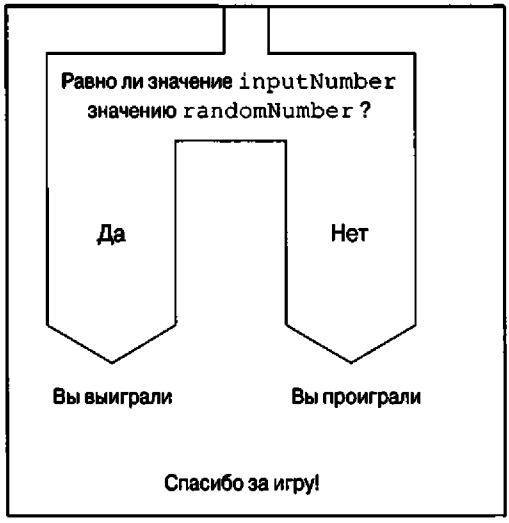


Рис. 5.2. Инструкция `if` создает развилку



Условие инструкции `if` должно быть заключено в скобки. Однако выражение `inputNumber==randomNumber` не является инструкцией, поэтому после него точка с запятой не ставится.



Наверное, вы обратили внимание на то, что фраза, заключенная в круглые скобки, называется иногда "условием", а иногда "выражением". И то, и другое правильно, потому что, согласно правилам Java, условие должно быть выражением. Выражение — это фрагмент кода, возвращающий значение. Условие может иметь только одно из двух значений: `true` или `false`. Следовательно, выражение должно возвращать значение булева типа (см. главу 4). Если выражение, используемое в `if`, возвращает значение другого типа, компилятор сообщит об ошибке.



В главе 4 отмечалось, что в Java инструкция заканчивается точкой с запятой. Однако инструкция `if` (и еще некоторые инструкции, такие как `while`, `for` и др.) является исключением из этого правила. Если есть блок `else` или `if`, она заканчивается закрывающейся фигурной скобкой последнего блока. Если же ни одного блока нет, она действительно заканчивается точкой с запятой.

Двойной знак равенства

В инструкции `if` условие всегда заключено в круглые скобки. В листинге 5.1 в условии применяется двойной знак равенства (==). Одинарный знак равенства (=) — это оператор присваивания. Сравнение и присваивание — разные операции, их нельзя путать друг с другом. Поэтому символ сравнения отличается от символа присваивания. Начинающие программисты часто пишут в условии одинарный знак равенства. Если присваивается не булево значение, компилятор сообщит об ошибке. Но будет хуже, если присваивается булево значение. Компилятор не заметит ошибку, программа будет работать неправильно, и вам придется искать, почему результат не тот, что нужно.

Эта ошибка настолько распространена среди начинающих программистов, что для меня она стала своеобразной игрой. На семинарах я рассказываю о ней студентам и предлагаю побиться об заклад, совершит ли кто-нибудь из них эту ошибку. У меня еще не было случая, чтобы я проиграл спор. Выполняя упражнение, кто-нибудь из студентов обязательно совершает эту ошибку.

Блоки

В листинге 5.1 инструкция `if` состоит из двух частей — верхней и нижней. Каждая из них имеет свое название — "часть `if`" и "часть `else`".

Часть `if` в листинге 5.1 содержит несколько инструкций. Они заключены в фигурные скобки, которые образуют *блок*. Блок — это набор инструкций, заключенный в фигурные скобки. Блоки встречаются не только в инструкциях `if`, но и в других случаях. Следует отметить, что часть `if` может состоять не из нескольких, а из одной инструкции. В этом случае заключать ее в фигурные скобки необязательно.

В предыдущем примере блок `if` объединяет три инструкции, а именно — три вызова метода `println()`, которые выводят на консоль сообщение о том, что пользователь угадал число. Таким образом, блок `if` выполняется, когда условие в круглых скобках равно `true`. В противном случае выполняется блок `else`.

Все, что сказано выше о блоке `if`, справедливо и для блока `else`. Он выполняется, когда переменные `inputNumber` и `randomNumber` не равны друг другу, т.е. когда условие равно `false`. Чтобы приказать компьютеру выполнить все инструкции, включенные в блок, заключите их в фигурные скобки.



Строго говоря, в листинге 5.1 часть `if` представляет собой одну инструкцию (как и часть `else`). Трюк состоит в том, что при заключении ряда инструкций в фигурные скобки создается блок, который ведет себя как одна инструкция. Фактически в официальной документации Java блок считается одним из видов инструкции. Таким образом, в листинге 5.1 все инструкции, заключенные в фигурные скобки, являются блоком, который выполняется как одно целое.

Отступы в инструкции `if`

Обратите внимание на отступы при вызове методов `print()` и `println()` в листинге 5.1. Как видите, в блоках `if` и `else` отступы одинаковые. Строго говоря, вы не обязаны делать отступы ни в блоках `if`, ни в каких-либо других местах. Можете даже написать всю программу в единственной строке или разместить ее на экране причудливыми зигзагами. Компилятору все равно. Правильные отступы нужны только человеку для облегчения визуального анализа программы. Например, в листинге 5.1 благодаря отступам хорошо видно, что методы вывода на консоль (`print()` и `println()`) в блоках `if` и `else` расположены на одном уровне вложенности.

В небольшой программе отсутствие отступов или плохие отступы всего лишь немного затрудняют понимание кода, однако в сложной программе плохие отступы легко приводят к невозможности понять код вообще.

Многие среды разработки (в частности Eclipse) создают отступы автоматически. Когда вы нажимаете клавишу <Enter> и создаете новую строку, курсор ввода автоматически устанавливается в позиции, необходимой согласно правилам создания отступов. Программа показывает вам, какие нужно делать отступы. И даже если вы по какой-либо причине написали код без отступов, вы легко можете их создать. Для этого нажмите комбинацию клавиш <Ctrl+A>, чтобы выделить весь код, и выберите в меню Eclipse команду Source⇨Correct Indentation (Исходный код⇨Исправить отступы).

Однако учитывайте, что отступы могут сыграть с вами злую шутку. Довольно часто начинающий программист правильно расставляет отступы, но забывает заключить блок в фигурные скобки. Когда несколько инструкций выделены отступами, они выглядят как блок, даже когда они не заключены в пару фигурных скобок. Из-за того что они похожи на блок, программист думает, что это блок. В результате в часть `else` войдет не три инструкции, а только одна. Эта ошибка тем более коварная, что компилятор ее не замечает.

Сломанная вилка (`if` без `else`)

В инструкции `if` может присутствовать только часть `if`, а часть `else` в нем не обязательная. Получается как бы вилка, но не с двумя зубцами, а с одним (развилку с одной дорогой "Налево пойдешь..." представить себе тяжелее, но это дела не меняет). Рассмотрим код в листинге 5.1. Пользователю неприятно получать сообщения о том, что он проиграл, поэтому давайте сделаем так, чтобы программа отображала только сообщение о выигрыше (листинг 5.2). На рис. 5.3 показан результат игры в двух случаях: выигрыша и проигрыша.

Листинг 5.2. Более деликатная игра в отгадывание числа

```
import static java.lang.System.in;
import static java.lang.System.out;
import java.util.Scanner;
import java.util.Random;
class DontTellThemTheyLost {
  public static void main(String args[]) {
    Scanner keyboard = new Scanner(in);
    out.print("Введите число от 1 до 10: ");
```

```
    int inputNumber = keyboard.nextInt();
    int randomNumber = new Random().nextInt(10) + 1;
    if (inputNumber == randomNumber) {
      out.println("*Вы выиграли.*");
    }
    out.println("Прекрасная попытка :-)");
    out.print("Правильное число равно ");
    out.println(randomNumber + ".");
    out.println("Спасибо за игру.");
  }
}
```

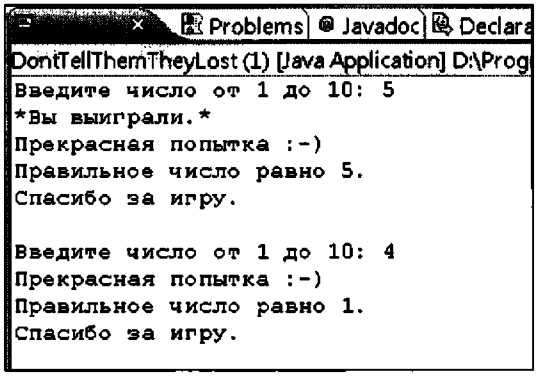


Рис. 5.3. Два сеанса программы, показанной в листинге 5.2

В листинге 5.2 инструкция `if` не имеет части `else`. Когда значения `inputNumber` и `randomNumber` равны, программа выводит на консоль фразу `Вы выиграли`. В противном случае программа не выводит эту фразу. В обоих случаях программа продолжает выполнять инструкции, расположенные после `if`. Можете представлять себе это так, будто часть `else` есть, но она пустая: ее нет в коде, и она ничего не делает.

В листинге 5.2 продемонстрирован еще один трюк. В операторе `import` объявлено стандартное устройство ввода `System.in`. В результате вместо `Scanner(System.in)` можно писать `Scanner(in)`. В данном случае мы не облегчили себе жизнь, а даже усложнили, потому что всего лишь перенесли слово `System` с одного места в другое, к тому же добавился оператор импорта. Как видите, клавишной работы прибавилось. Однако если ссылка на устройство ввода встречается в коде много раз, данный трюк будет очень полезным.

Условия с операторами сравнения и логическими операторами

В Java есть много средств, предназначенных для принятия решения в условном операторе. Их рассмотрению посвящен данный раздел. В коде они применяются в условии оператора `if`.

Сравнение чисел и символов

В табл. 5.1 перечислены операторы, используемые для сравнения двух значений.

Таблица 5.1. Операторы сравнения

Символ	Что делает	Пример
==	Равно	`numberOfCows == 5`
!=	Не равно	`buttonClicked != panicButton`
<	Меньше	`numberOfCows < 5`
>	Больше	`myInitial > 'B'`
<=	Меньше или равно	`numberOfCows <= 5`
>=	Больше или равно	`myInitial >= 'B'`

Все определенные в Java операторы сравнения можно использовать для сравнения чисел и символов. При сравнении чисел оператор работает точно так, как вы интуитивно предполагаете. Например, если слева от оператора < находится меньшее число, выражение возвращает значение `true`. Однако при сравнении символов все не так просто. Когда вы сравниваете символы одного алфавита (например, символ кириллицы с символом кириллицы) в одном и том же регистре, меньшей считается буква, расположенная в алфавите раньше. Например, выражение `'a'<'z'` возвращает `true`. Если же сравниваемые символы приведены в разном регистре, то любая буква в верхнем регистре считается меньшей, чем любая буква в нижнем регистре, например выражение `'Z'<'a'` возвращает `true`.



Это объясняется тем, что при сравнении символов фактически сравниваются их кодовые номера в Unicode. В кодировке Unicode сначала расположены все буквы от `A` до `Z`, а затем все буквы от `a` до `z`. Поэтому любая буква в верхнем регистре меньше любой буквы в нижнем регистре. По этой же причине можно сравнивать любые символы с любыми другими символами — буквы с цифрами, знаками препинания и т.п.



Будьте осторожны, проверяя на равенство (с помощью оператора ==) два числа в формате `float` или `double`. В принципе, я рекомендую никогда не делать этого. Если числа целые, то, например, `7` всегда равно `7`, сомнений в этом не возникает. Но два числа с плавающей точкой почти никогда не равны друг другу. Дело в том, что в памяти числа хранятся не в десятичном, а в двоичном формате, поэтому точного представления, казалось бы, простейшего числа `7` в формате `float` или `double` нет вообще. Числа с плавающей точкой, которое точно равно `7`, в компьютере не существует! В ячейке памяти может быть записано `7.00000001` или `6.99999999`, и оператор сравнения на равенство, скорее всего, вернет значение `false`, хотя вы считаете эти значения равными.

Сравнение объектов

Объекты можно сравнивать друг с другом с помощью операторов == и !=. Например, кнопка, которую вы видите на экране, — это объект. С помощью оператора сравнения можно выяснить, на этой ли кнопке только что щелкнул пользователь.

```
if(e.getSource() == bCopy) {
  clipboard.setText(which.getText());
```



Программирование реакции на щелчки на кнопках более подробно рассматривается в главе 16.

Однако в этом примере мы сравниваем фактически не объекты, а ссылки на объекты. В случае с кнопками это вполне допустимо, но сравнивать строковые значения типа `String` с помощью оператора == или != нельзя. Использование одного из этих операторов означает, что мы задаем компьютеру вопрос, хранятся ли эти две строки в одном и том же месте. Но ведь на самом деле нас интересует, совпадают ли все символы этих строк. Для ответа на этот вопрос в классе `String` определен метод `equals()`, который сравнивает две строки посимвольно. Пример использования метода `equals()` приведен в листинге 5.3. Метод возвращает значение `true`, если все символы строки `password` (пароль) совпадают с символами строки `swordfish`. (Учитывайте, что это всего лишь учебный пример. В реальных приложениях пароль нельзя хранить в коде или на диске, потому что хакер сможет легко его оттуда извлечь. В правильно защищенной системе должен храниться не пароль, а хеш пароля.)

Листинг 5.3. Проверка пароля, введенного пользователем

```
import static java.lang.System.*;
import java.util.Scanner;
class CheckPassword {
  public static void main(String args[]) {
    out.print("Введите пароль: ");
    Scanner keyboard = new Scanner(in);
    String password = keyboard.next();
    out.println("Вы ввели >>" + password + "<<");
    out.println();
    if (password == "swordfish") {
      out.println("Этого не должно быть!");
      out.println("Введенный и хранящийся пароли");
      out.println("находятся в разных местах.");
     } else {
      out.println("Что и ожидалось!");
      out.println("Это две разные строки, хотя");
      out.println("их символы могут совпадать.");
     }
    out.println();
    if (password.equals("swordfish")) {
      out.println("Введенный пароль правильный!");
      out.println("Можете войти в систему.");
    } else {
      out.println("Введенный пароль неправильный!");
      out.println("Попытайтесь еще раз...");
    }
  }
}
```

В листинге 5.3 метод `keyboard.next()` возвращает строку, которую пользователь ввел с клавиатуры. Код помещает ее в переменную `password`. Правильный пароль находится в строковом литерале `swordfish`. Затем в листинге 5.3 выполняются два

оператора `if`. В первом сравниваются адреса двух строк, а во втором — их содержимое. Адреса, естественно, не совпадают, но нас интересует, совпадает ли содержимое. Результат показан на рис. 5.4.

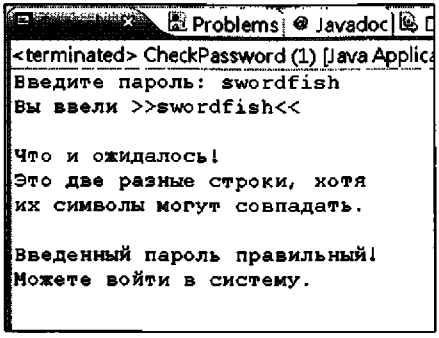


Рис. 5.4. Использование оператора == и метода `equals()`

В реальных задачах практически всегда нужно сравнивать не адреса, а содержимое строк. Это делается с помощью метода `equals()`. Он выглядит немного странно (объект одной строки вызывает другую), но так уж запрограммирован класс `String`.

При использовании метода `equals()` не играет роли, какая строка служит объектом (приведена перед точкой), а какая передается методу в качестве параметра. Например, в листинге 5.3 можно было написать так.

```
if ("swordfish".equals(password))
```

Результат будет тот же.



На первый взгляд, метод `equals()` кажется несимметричным, несбалансированным. Но это не так. Есть следующая веская причина поместить две равноправные строки в два разных места (перед точкой и в скобках). У нас есть два объекта: `password` и `"swordfish"`. Каждый имеет тип `String`. Конечно, `password` — это переменная, а `"swordfish"` — литерал, но это не важно. Главная причина состоит в том, что каждый метод должен принадлежать определенному объекту или классу. Можно было сделать метод статическим и передавать две строки посредством двух параметров, но тогда при каждом вызове пришлось бы дополнительно вводить имя класса `String`.



Методы классов рассматриваются в главе 7.



Для сравнения строк используйте метод `equals()`, а не оператор ==.

Импортируем все за один раз

Конечно, "втиснуть" все библиотеки Java в одно объявление `import` не удастся, но писать это объявление для каждого класса все же необязательно. В листинге 5.3 продемонстрирован способ импорта "для ленивых" — входной поток `System.in` и

выходной поток `System.out` (а заодно и все члены класса `System`, но они нам пока что не нужны) импортируются одной инструкцией. Чтобы за один раз импортировать все содержимое класса `System`, нужно добавить в инструкцию импорта точку и групповой символ `*`, означающий "все, что там есть". Таким образом, инструкция импорта `java.lang.System.*` эквивалентна приблизительно тридцати отдельным инструкциям `import`, включая `System.in`, `System.out`, `System.err`, `System.nanoTime` и т.п.

Использование звездочки в объявлении `import` считается плохим способом импорта, поэтому в примерах данной книги оно встречается редко. Но в некоторых программах, когда в одном файле члены какого-либо класса встречаются десятки раз, звездочка может быть довольно удобной. Но учитывайте, что имена статических членов класса при этом резервируются, в результате чего их нельзя использовать для других переменных. В результате вам придется следить за тем, чтобы случайно не назвать что-либо зарезервированным именем.



Символ звездочки нельзя поставить в любом месте объявления импорта. Например, нельзя импортировать все библиотеки Java, написав `import java.*`. Звездочку можно поставить только вместо имени класса или вместо статических членов класса. Использование звездочки в объявлении `import` более подробно рассматривается в главе 9. Статические члены класса рассматриваются в главе 10.

Логические операторы

Джон Буль был бы доволен: в Java есть все операторы, необходимые для выполнения булевых (логических) операций (табл. 5.2).

Таблица 5.2. Логические операторы

Символ оператора	Операция	Пример
`&&`	И (конъюнкция)	`5 < x && x < 10`
`||`	ИЛИ (дизъюнкция)	`x < 5 || 10 < x`
`!`	НЕТ (отрицание)	`!password.equals("swordfish")`

С их помощью можно проверять любые условия. Пример использования логических операторов приведен в листинге 5.4.

Листинг 5.4. Проверка имени пользователя (username) и пароля (password)

```
import javax.swing.JOptionPane;
class Authenticator {
  public static void main(String args[]) {
    String username =
      JOptionPane.showInputDialog("Username:");
    String password =
      JOptionPane.showInputDialog("Password:");
    if (
      username != null &&
```

```
      password != null &&
      (
        (username.equals("bburd") &&
         password.equals("swordfish")) ||
        (username.equals("hritter") &&
         password.equals("preakston"))
      )
    )
    {
      JOptionPane.showMessageDialog(null, "Вы допущены в систему!");
    } else {
      JOptionPane.showMessageDialog(null, "Попытайтесь еще раз...");
    }
  }
}
```

К системе имеют доступ только два пользователя: `bburd` с паролем `swordfish` и `hritter` с паролем `preakston`. Кроме того, нужно проверить, заполнены ли поля в диалоговых окнах. Если что-либо не заполнено, переменная `username` или `password` равна `null`. Каждый пользователь должен ввести свой пароль. Все эти требования сосредоточены в условии одного оператора `if`. Если пользователь введет неправильный пароль, программа должна выполнить блок `else`. Если пользователь прошел проверку, выполняется блок `if` и в окне отображается сообщение `Вы допущены в систему!`. Результат показан на рис. 5.5.

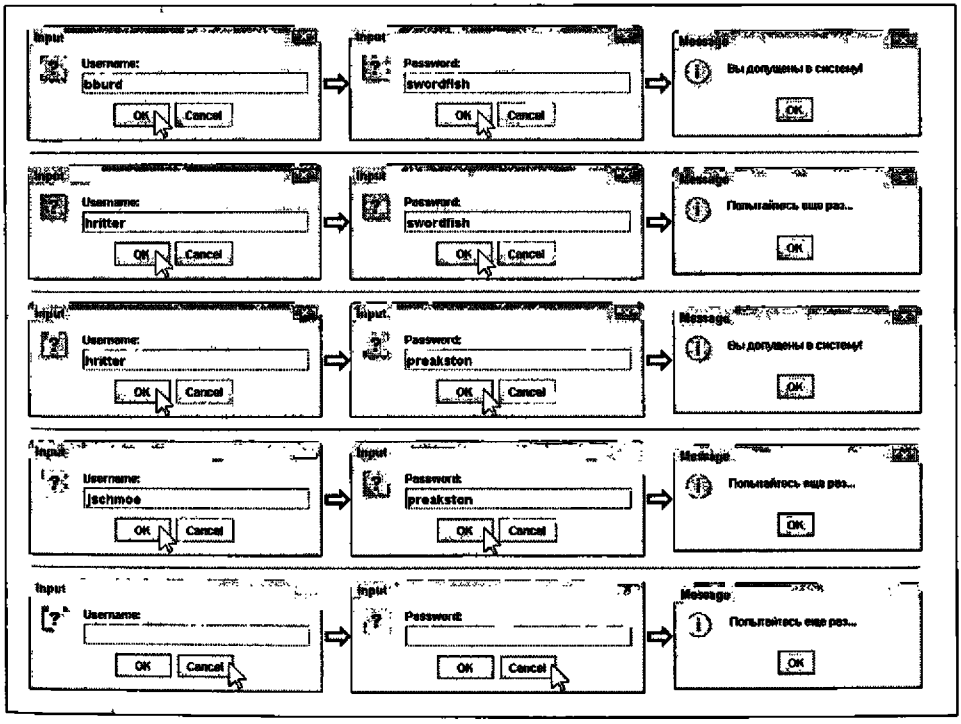


Рис. 5.5. Пять сеансов программы, показанной в листинге 5.4

В листинге 5.5 демонстрируется новый для вас способ приема строки, введенной пользователем с клавиатуры. Для этого программа отображает на экране диалоговое окно ввода с помощью следующей инструкции.

```
String password = JOptionPane.showInputDialog("Password:");
```

Эта инструкция выполняет ту же задачу, что и применявшийся в предыдущих примерах (см. листинг 5.3) метод `next()`.

```
String password = keyboard.next();
```

Существенное различие между ними заключается в том, что метод `next()` принимает текст с консоли, а метод `showInputDialog()` создает диалоговое окно, отображает в нем текстовое поле и принимает с него текст. Кроме того, этот метод создает кнопки, позволяющие отменить или подтвердить ответ. Сравните рис. 5.4 и 5.5. Как видите, программа в листинге 5.5 создает эстетически более привлекательный интерфейс. Метод `showInputDialog()` вызывается дважды: чтобы принять сначала имя пользователя, а затем пароль.

В конце листинга 5.4 приведен еще один метод класса `JOptionPane`.

```
JOptionPane.showMessageDialog(null, "Вы допущены в систему!");
```

Этот метод отображает простое диалоговое окно с надписью (см. рис. 5.5).

Как и тысячи других имен, имя класса `JOptionPane` определено в Java API (если точнее, то в каталоге `javax.swing`, расположенном в Java API). Чтобы имя `JOptionPane` можно было использовать в коде, в верхней части листинга находится объявление `import javax.swing.JOptionPane`.



Метод `showInputDialog()` возвращает значение типа `String`. Для задачи, решаемой в листинге 5.4, это подходит, но иногда необходимо, чтобы пользователь ввел число типа `int` или `double`. В этом случае нужно принять от метода строку `String` и после этого извлечь из нее число. Например, если пользователь должен ввести число типа `int`, нужно передать полученную строку методу `parseInt()`.

```
int numberOfBunnies = Integer.parseInt
  (JOptionPane.showInputDialog("Сколько кроликов ты купил?"));
```

Метод `parseInt()` преобразует полученную строку в значение типа `int`. Преобразовать полученную строку в значение типа `double` можно с помощью метода `parseDouble()`, который работает аналогично.

Чем "нуль" отличается от "ничто"

Ключевое слово `null` в Java означает отсутствие значения. Переменная числового типа (`int` или `double`) может быть равна нулю, но нуль — это не `null`, а вполне определенное значение, ничуть не хуже других значений (конечно, не в платежной ведомости). К тому же переменная примитивного типа не может принимать значение `null`. Если вы забудете присвоить переменной `int` или `double` какое-либо значение, компилятор сообщит, что она не инициализирована, и откажется запустить программу.

Переменная ссылочного типа имеет значение `null`, если объекта, на который она указывает, не существует. Это часто бывает, если переменная объявлена, но объект для нее не создан. В листинге 5.4 переменная `username` получает значение, возвращаемое методом `showInputDialog()`. Если пользователь передумал вводить имя и щелкнул на кнопке Cancel (Отмена), переменной `username` будет присвоено значение `null`. Приведенная ниже инструкция проверяет, присвоено ли переменной `username` какое-либо значение.

```
if (username != null ...
```

Посмотрите последнюю строку на рис. 5.5. Пользователь два раза щелкнул на кнопке Cancel, следовательно, обе переменные (`username` и `password`) имеют значения `null`. Выражение `username!=null` равно `false`, если пользователь щелкнул на кнопке Cancel. То же самое справедливо и для выражения `password!=null`.



В листинге 5.4 оба сравнения (`username!=null` и `password!=null`) обязательны. Если в программе одного из них нет и пользователь щелкнет на кнопке Cancel, будет сгенерировано исключение `NullPointerException` (исключение указателя `null`), и программа будет аварийно завершена так быстро, что пользователь не поймет, что произошло. Такое поведение программы считается недружелюбным, и вы должны его предотвратить. Значение `null` означает "ничто", а Java не может сравнивать "ничто" со строкой `bburd` или `swordfish`. Единственное назначение проверки `username!=null` состоит в том, чтобы предотвратить сравнение `username.equals("bburd")` после щелчка на кнопке Cancel.



Последние два значения `null` в листинге 5.4 отличаются от предыдущих. В качестве первого параметра метод `showMessageDialog(null,"Вы допущены в систему!")` принимает имя родительского диалогового окна. Передавая значение `null`, вы сообщаете методу о том, что родительского окна нет. Следовательно, окно сообщения не обязано появляться в центре родительского окна, и потому оно появится в центре экрана.

Условия в скобках и скобки в условиях

Скобки часто помогают визуально выделить в коде некоторую сущность, но для этого нужно уметь ими пользоваться. Если в условии несколько логических операторов объединено таким образом, что скобки не обязательны, лучше все же их поставить, чтобы выражение было понятным с первого взгляда. Рассмотрим следующее выражение.

```
2 < 5 || 100 < 6 && 27 < 1
```

Может показаться, что оно равно `false`. Визуально оно читается как (*некоторое_вложенное_выражение*) `&&27<1`. Поскольку `27` всегда больше единицы, можно сделать вывод, что все выражение ложное. Однако это не так, потому что в Java оператор `&&` имеет более высокий приоритет, чем `||`. Следовательно, оператор `&&` вычисляется в первую очередь. Это означает, что приведенное выше выражение нужно читать так:

2<5||(*некоторое_вложенное_выражение*). Поскольку два всегда меньше пяти, все выражение всегда равно `true`.

Чтобы изменить значение всего выражения с `true` на `false`, нужно заключить в скобки первые два оператора сравнения.

```
(2 < 5 || 100 < 6) && 27 < 1
```



В Java оператор `||` реализует *включающую дизъюнкцию*. Это означает, что результат равен `true` в следующих трех случаях: когда левый операнд равен `true`, правый операнд равен `true` или оба операнда равны `true`. Например, выражение `2<10||20<30` равно `true`.



В Java нельзя объединять операторы сравнения, как в естественном языке. Например, по-русски можно сказать: "За столом будут сидеть от 5 до 10 человек". Если же написать `5<=n<=10`, компилятор сгенерирует сообщение об ошибке. Чтобы выполнить указанную операцию, нужно написать `5<=n&&n<=10`.

В листинге 5.4 в условии оператора `if` есть много скобок. Что будет, если опустить одну пару?

```
if (
  username != null &&
  password != null &&
  // открывающая скобка опущена
     (username.equals("bburd") &&
      password.equals("swordfish")) ||
     (username.equals("hritter") &&
      password.equals("preakston"))
  // закрывающая скобка опущена
)
```

Компилятор Java пытается интерпретировать наши желания, группируя с помощью скобок все сравнения перед оператором `||`. Таким образом, сначала вычисляются три первых оператора `&&`.

```
if (
    username != null &&
    password != null &&
    (username.equals("bburd") &&
     password.equals("swordfish"))

    ||

    (username.equals("hritter") &&
     password.equals("preakston"))
)
```

Предположим, пользователь щелкнул на кнопке Cancel и переменная `username` стала равной `null`. Программа думает так: "Ну, ладно, значение перед `||` равно `false`, но, может, значение после `||` не равно `false`? Надо проверить..."

Надо, так надо. Программа вызывает метод `username.equals("hritter")` и терпит крах, потому что `username` равно `null`. Окно программы мгновенно закрывается, и пользователь остается в полном недоумении относительно того, что произошло. А все потому, что вы попытались вызвать метод `equals()` несуществующего объекта. Следовательно, внимательно анализируйте варианты и не ленитесь расставлять побольше скобок, потому что они не только делают код более наглядным, но и заставляют программу выполнять операции в той последовательности, которая нужна вам, а не компилятору.

Вложение инструкций `if`

Вы, конечно, знаете, как устроены куклы-матрешки. Откройте одну, а в ней находится вторая. Откройте вторую, а в ней находится третья и т.д. То же самое можно сделать с операторами `if` (листинг 5.5). Результат показан на рис. 5.6.

Листинг 5.5. Вложенные операторы `if`

```
import static java.lang.System.out;
import java.util.Scanner;
class Authenticator2 {
  public static void main(String args[]) {
    Scanner keyboard = new Scanner(System.in);
    out.print("Имя пользователя: ");
    String username = keyboard.next();
    if (username.equals("bburd")) {
      out.print("Пароль: ");
      String password = keyboard.next();
      if (password.equals("swordfish")) {
        out.println("Вы в системе!");
      } else {
        out.println("Неправильный пароль.");
      }
    } else {
      out.println("Неизвестный пользователь.");
    }
  }
}
```

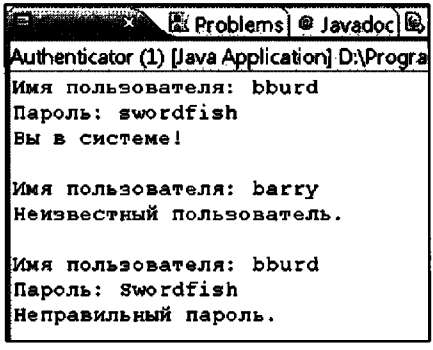


Рис. 5.6. Три сеанса программы, показанной в листинге 5.5

Главная идея состоит в том, что при попытке зарегистрироваться в системе пользователь должен пройти два теста (иными словами, истинными должны быть два условия). Первый тест — на правильность имени пользователя, второй — на правильность пароля. Если первый тест пройден, пользователь попадает на второй. В противном случае программа выводит сообщение о неправильном имени пользователя, не вынуждая его вводить пароль (зачем тратить время, если уже известно, что пользователь не будет допущен в систему?). Общая схема принятия решения показана на рис. 5.7.

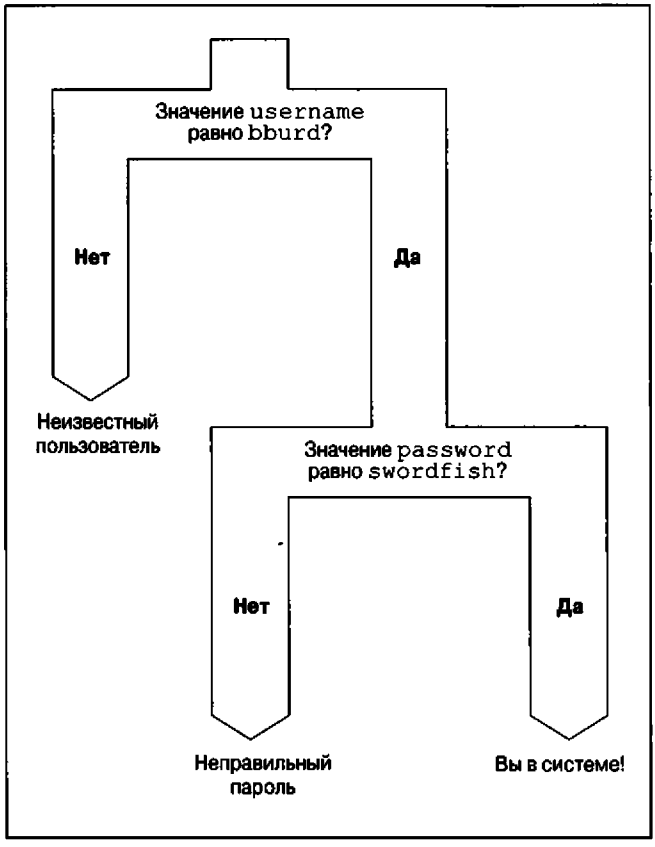


Рис. 5.7. Обедать такой вилкой неудобно, однако регистрироваться в системе удобнее, чем с помощью обычной вилки (см. рис. 5.2)



Не применяйте код листинга 5.5 в реальных программах! Это всего лишь учебный пример. Аутентифицировать пользователя таким образом нельзя ни в коем случае. Во-первых, никогда не отображайте пароль обычными символами. Вместо символов следует отображать звездочки, чтобы никто не мог подсмотреть пароль через плечо в момент его ввода. Во-вторых, никогда не храните незашифрованный пароль в коде или на диске. Хакер легко найдет его. В реальных программах обычно хранят не пароль, а хеш пароля, поэтому, если хакер найдет его, он ему все равно бесполезен, потому что проникнуть в систему с его помощью невозможно. В-третьих,

не сообщайте злоумышленнику, что именно введено неправильно — имя пользователя или пароль. В-четвертых... Можно продолжать еще долго, но не нужно, потому что листинг 5.5 иллюстрирует использование вложенных операторов `if`, а не способы защиты системы.

Переключатель `switch`

Должен сознаться, что я ненавижу принимать решения. Если дела пойдут неправильно, пусть это будет в результате ошибки кого-нибудь другого. В предыдущем разделе мы обсуждали оператор `if`, позволяющий выбрать один из двух вариантов. В этом разделе все значительно хуже: нам предстоит познакомиться с оператором `switch`, который выбирает один из многих вариантов.

Выбор варианта

Рассмотрим ситуацию, в которой пользователь должен выбрать один из вариантов. В качестве примера возьмем популярную песенку "Эл под дождем". Код должен вывести на консоль одну из причин того, почему Эл стоит под дождем.

Эл под дождем

Эл весь мокрый. Почему Эл весь мокрый? Эл весь мокрый, потому что он стоит под дождем. Почему Эл стоит под дождем? Потому что дождь идет очень долго, дождь идет очень долго, дождь идет, идет, идет.

О-ооооооо...

Эл весь мокрый. Почему Эл весь мокрый? Эл весь мокрый, потому что он стоит под дождем. Почему Эл стоит под дождем? Потому что он очень расстроен. Очень, очень, очень расстроен Эл под дождем, под дождем, под дождем.

О-ооооооо...

Эл весь мокрый. Почему Эл весь мокрый? Эл весь мокрый, потому что он стоит под дождем. Почему Эл стоит под дождем? Потому что он все равно уже мокрый, весь мокрый он под дождем, под дождем.

О-ооооооо...

— Гарриет Риттер, Барри Берд

Полнофункциональную версию программы "Эл под дождем" мы рассмотрим в главе 6, а пока что предположим, что пользователь вводит номер куплета (переменная `verse`), и программа должна вывести на консоль сообщение о том, почему Эл стоит под дождем. Если бы мы делали это с помощью инструкций `if`, нам пришлось бы написать довольно неуклюжий код.

```
if (verse == 1) {
  out.println("Потому что дождь идет очень долго.");
}
if (verse == 2) {
  out.println("Потому что он очень расстроен.");
}
if (verse == 3) {
  out.println("Потому что он все равно уже мокрый.");
}
```

Главный недостаток этого кода состоит в том, что проверка значения `verse` выполняется много раз. Например, если переменная `verse` равна 1, выполняется первый блок, и после этого `verse` проверяется еще два раза, хотя это уже не имеет смысла. Нельзя ли проверить значение `verse` один раз? Можно. Для этого существует инструкция `switch`, пример использования которой приведен в листинге 5.6.

Листинг 5.6. Инструкция `switch`

```
import static java.lang.System.out;
import java.util.Scanner;
class JustSwitchIt {
  public static void main(String args[]) {
    Scanner keyboard = new Scanner(System.in);
    out.print("Почему Эл стоит под дождем? Введите номер: ");
    int verse = keyboard.nextInt();
    switch (verse) {
      case 1:
        out.println("Потому что дождь идет очень долго.");
        break;
      case 2:
        out.println("Потому что он очень расстроен.");
        break;
      case 3:
        out.println("Потому что он все равно уже мокрый.");
        break;
      default:
        out.println("Такой причины нет. Попытайтесь еще раз.");
        break;
    }
    out.println("O-ooooooo...");
  }
}
```

На рис. 5.8 показаны два сеанса взаимодействия с программой, приведенной в листинге 5.6. На рис. 5.9 представлена общая схема работы программы. Сначала пользователь вводит число, например 2. Затем управление передается оператору `switch`, который находит нужную метку `case`. Значение в найденной метке `case` совпадает со значением аргумента `switch`, в данном примере — со значением переменной `verse`. После этого компьютер выполняет инструкции, расположенные после метки `case`.

```
case 2:
   out.println("Потому что он очень расстроен.");
   break;
```

Обратите внимание на инструкцию `break`. Она приводит к выходу из блока `switch`, в результате чего управление передается инструкции, выводящей фразу `O-ooooooo`. Если бы инструкции `break` не было, выполнение блока `switch` продолжалось бы и были бы отображены фразы, приведенные во всех следующих ветках `case`. Таким образом, `case` — это метка, по которой передается управление, а не заголовок выполняемого блока.

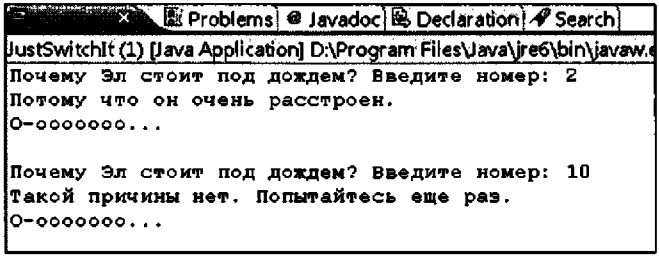


Рис. 5.8. Два запуска программы, приведенной в листинге 5.6



Рис. 5.9. Большая вилка (оператор `switch`)

По метке `default` управление передается, если аргумент `verse` не совпадает ни с одним значением `case`. В этом случае программа выводит на консоль фразу `Такой причины нет` и выходит из блока `switch`. После выхода из блока `switch` выполняется инструкция, расположенная после него, т.е. метод `println()` выводит на консоль фразу `O-ooooooo`.



Вставлять инструкцию `break` после метки `default` необязательно, потому что все равно больше нет ни меток, ни инструкций. Обычно инструкцию `break` добавляют в конец блока из эстетических соображений: чтобы метка `default` не отличалась от других.

Не забывайте вставлять `break`!

Отсутствие операторов `break` — очень распространенная ошибка. Наверное, нет ни одного программиста, который хотя бы раз в жизни ее не совершил. Для нее даже придумали специальный термин — *выпадение*. Управление как бы выпадает из группы инструкций, которые нужно выполнить по данной метке `case`, и движется дальше, продолжая "падать" до первого встретившегося слова `break` или, если не встретится, до конца блока `switch`.

Обычно выпадение — нежелательное явление, и чтобы его предотвратить, нужно расставить операторы `break`. Но иногда выпадение — это именно то, что должна сделать программа. Рассмотрим, например, программу в листинге 5.6, отображающую на консоли текст песни. Инструкции после каждой метки добавляют на консоль новые куплеты песни. Строки как бы накапливаются на консоли, причем все куплеты будут выведены, если начать с первого. Вариант программы без операторов `break` приведен в листинге 5.7. Результат показан на рис. 5.10.

Листинг 5.7. Инструкция `switch` без операторов `break`

```
import static java.lang.System.out;
import java.util.Scanner;
class JustSwitchIt {
  public static void main(String args[]) {
    Scanner keyboard = new Scanner(System.in);
    out.print("Почему Эл стоит под дождем? Введите номер: ");
    int verse = keyboard.nextInt();
    switch (verse) {
      case 1:
        out.println("Потому что дождь идет очень долго.");
      case 2:
        out.println("Потому что он очень расстроен.");
      case 3:
        out.println("Потому что он все равно уже мокрый.");
      default:
        out.println("Такой причины нет. Попытайтесь еще раз.");
    }
    out.println("О-ооооооо...");
  }
}
```

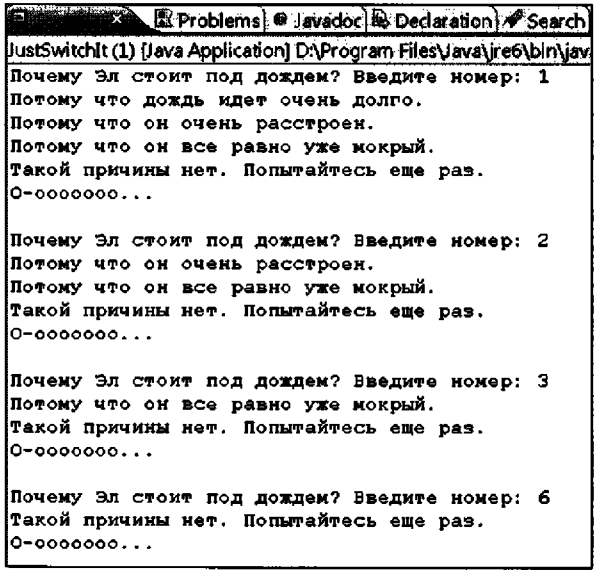


Рис. 5.10. Код, приведенный в листинге 5.7, выполняется четыре раза

Что происходит, когда пользователь вводит число, которого нет в метках case? Оператор switch передает управление метке default, и программа отображает на консоли только фразу Такой причины нет. После этого программа отображает общую для всех случаев фразу О-ооооооо, потому что ее вывод находится после блока switch.

Строковый аргумент — новинка в Java 7

В листингах 5.6 и 5.7 аргумент оператора switch — переменная verse — имеет тип int. С этим типом оператор switch работает во всех версиях Java, как старых, так и новых. Во всех версиях Java работает также тип char.

Однако в Java 7 список допустимых типов был дополнен. Теперь в качестве аргумента оператора switch можно использовать переменную типа String (листинг 5.8). Результат показан на рис. 5.11.

Листинг 5.8. Использование типа String в аргументе оператора switch

```
import static java.lang.System.out;
import java.util.Scanner;
class JustSwitchIt {
  public static void main(String args[]) {
    Scanner keyboard = new Scanner(System.in);
    out.print("Почему Эл стоит под дождем? Введите номер: ");
    String verse = keyboard.next();
    switch (verse) {
      case 1:
        out.println("Потому что дождь идет очень долго.");
        break;
      case 2:
        out.println("Потому что он очень расстроен.");
        break;
      case 3:
        out.println("Потому что он все равно уже мокрый.");
        break;
      default:
        out.println("Такой причины нет. Попытайтесь еще раз.");
        break;
    }
    out.println("О-ооооооо...");
  }
}
```

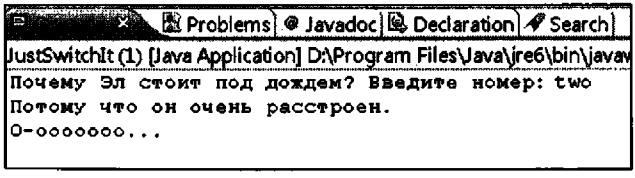


Рис. 5.11. Строковый аргумент оператора switch



При использовании Java 6 и более старых версий JRE аргумент switch не может иметь тип String.

Глава 6

Циклы

В этой главе...

- Цикл while
- Цикл for
- Цикл do

В 1966 году компания, выпускающая шампунь Head & Shoulders, поставила рекорд краткости. На тыльной стороне флакона они написали инструкцию по использованию: "Вспенить, смыть, повторить". Никто до них не сумел составить из трех слов полную инструкцию по использованию продукта. Специалисты по написанию инструкций воодушевленно приветствовали это монументальное достижение. Какой контраст с другими инструкциями! Возьмем, например, инструкцию по использованию велосипедного зеркальца заднего вида, один из пунктов которой гласит: "Не забывайте о том, что движущиеся объекты, которые вы видите в зеркальце, находятся позади вас, а не перед вами". Кто бы мог подумать?

Кроме краткости, инструкция к шампуню Head & Shoulders примечательна тем, что ее создателям удалось в трех словах выразить концепцию повторения набора операций, чрезвычайно важную не только в потребительском сегменте рынка, но и в программировании. Последнее слово, "повторить", превратило инструкцию с простой цепочки действий в изощренный процесс.

Фундаментальная идея состоит в том, что компьютер необязательно должен выполнять ваши инструкции, записанные в программе, одну за другой. Вы можете заставить его выполнить ряд инструкций многократно, заданное количество раз или до тех пор, пока не будет достигнута некоторая цель. В программировании многократное повторение набора инструкций называется *циклом*. Набор многократно выполняемых инструкций называется *телом цикла*. В данной главе рассматриваются циклы, применяемые в Java.

Цикл `while`

Давайте поиграем в угадывание чисел. Компьютер генерирует случайное число от 0 до 10 и предлагает вам его угадать. Если вы не угадали, игра продолжается: компьютер генерирует следующее случайное число и опять предлагает вам его угадать. Данный процесс повторяется, пока вы не угадаете число. В этом случае компьютер поздравляет вас с победой и сообщает количество попыток. Исходный код программы, реализующей данную игру, показан в листинге 6.1. На рис. 6.1 показан сеанс игры.

Листинг 6.1. Игра с отгадыванием чисел

```
import static java.lang.System.out;
import java.util.Scanner;
import java.util.Random;
class GuessAgain {
  public static void main(String args[]) {
    Scanner keyboard = new Scanner(System.in);
    int numGuesses = 0;
    int randomNumber = new Random().nextInt(10) + 1;
    out.println(" ************ ");
    out.println("Добро пожаловать в игру!");
    out.println(" ************ ");
    out.println();
    out.print("Введите число от 1 до 10: ");
    int inputNumber = keyboard.nextInt();
    numGuesses++;
    while (inputNumber != randomNumber) {
      out.println();
      out.println("Попытайтесь еще раз...");
      out.print("Введите число от 1 до 10: ");
      inputNumber = keyboard.nextInt();
      numGuesses++;
    }
    out.print("Вы угадали после ");
    out.println(numGuesses + " попыток.");
  }
}
```

На рис. 6.1 пользователь сделал три попытки угадать число. После каждой попытки компьютер проверяет, равно ли число, введенное пользователем, числу, сгенерированному компьютером. Если эти два числа не равны (т.е. попытка неудачная), компьютер повторно переходит к первой инструкции цикла, генерирует новое число и предлагает пользователю его угадать. Если два числа равны, компьютер выходит из цикла и отображает сообщение о количестве попыток. Таким образом, компьютер выполняет ряд инструкций повторно, пока соблюдается условие, приведенное в круглых скобках. При каждой попытке компьютер добавляет единицу к значению переменной numGuesses. В результате после выхода из цикла переменная numGuesses будет содержать количество попыток. Схема выполнения программы показана на рис. 6.2.

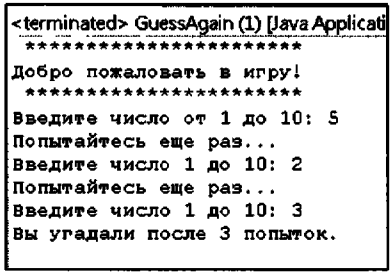


Рис. 6.1. Повторение попыток угадать число

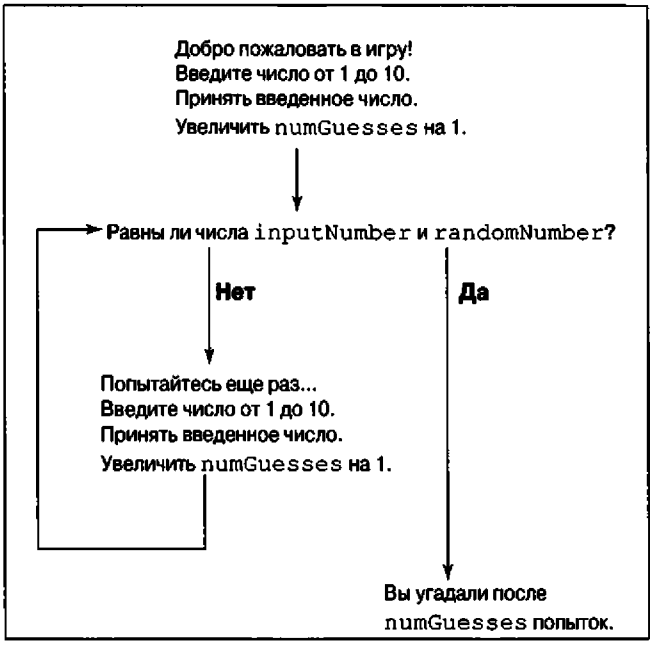


Рис. 6.2. Цикл — это хождение по кругу

Код игры приведен в листинге 6.1. Нас в нем сейчас интересует инструкция `while` (пока), которую часто называют циклом `while`. В переводе на естественный язык инструкция `while` гласит: "Выполнять этот блок инструкций, пока числа `inputNumber` и `randomNumber` не равны".

Блок повторяемых инструкций заключен в фигурные скобки. В этом блоке компьютер выводит приглашение ввести очередное число, принимает его с клавиатуры и увеличивает на единицу переменную `numGuesses`, в которой хранится количество попыток.



Переменная `numGuesses` называется *счетчиком цикла*. При использовании счетчика очень легко ошибиться на единицу в одну или другую сторону. Чтобы облегчить анализ кода и уменьшить вероятность ошибки, располагайте оператор ++ рядом с инструкциями, которые имеют непосредственное отношение к подсчитываемым событиям. Кроме того, следите за тем, с чего начинается подсчет и где расположен оператор ++. В листинге 6.1 подсчет начинается с нуля, потому что в начале программы пользователь еще не сделал ни одной попытки. Далее в коде переменная `numGuesses` увеличивается на единицу после каждого считывания числа с клавиатуры.

Обратите внимание на то, что блок инструкций в листинге 6.1 фактически записан два раза: один раз — перед циклом и один раз — в теле цикла. Это обусловлено тем, что к началу цикла переменные `inputNumber` и `randomNumber` должны иметь определенное значение, поскольку условие цикла `while` проверяется перед вхождением в цикл. Естественно, дублирование кода — это лишняя работа, и программисты

ее избегают. Далее будет рассмотрен цикл do, в котором условие проверяется после выполнения тела цикла. Это позволит не дублировать код и сделать его короче.

Инструкции, заключенные в фигурные скобки (*тело* цикла), выполняются многократно, пока условие `inputNumber!=randomNumber` равно `true`. Каждое выполнение тела цикла называется *итерацией*. На рис. 6.1 показаны три итерации. Наличие перед циклом продублированного кода может привести к ошибке при подсчете итераций вручную. Чтобы не ошибиться, воспользуйтесь тем, что количество итераций точно равно количеству отображенных фраз `Попытайтесь еще раз`.

Когда после нескольких попыток пользователь угадывает число, компьютер, как обычно, переходит в начало цикла и проверяет условие, но на этот раз оно оказывается равным `false`. Результат операции сравнения `!=` равен `false`, когда числа равны. Когда условие равно `false`, программа выходит из цикла, т.е. компьютер выполняет инструкцию, расположенную непосредственно после цикла.



В листинге 6.1 программа никогда не "выскакивает" из итерации, находясь в середине тела цикла. Выйти из данного цикла можно только в момент, когда проверяется условие `while`. Если в этот момент программа обнаруживает, что числа не равны, она выполняет все инструкции в теле цикла. Если же программа обнаруживает, что числа равны, она выходит из цикла и выполняет инструкцию, расположенную непосредственно после тела цикла. Далее будут рассмотрены операторы `break` и `continue`, позволяющие прервать выполнение тела цикла.

Цикл for

Учитель говорит ученику: "Напиши на доске фразу `Я не буду разговаривать в классе` десять раз".

На языке программирования это выглядит так.

```
Присвой переменной счетчика значение 0.
Проверь значение счетчика; если оно меньше десяти,
  напиши на доске фразу
    "Я не буду разговаривать в классе".
Увеличь переменную счетчика на 1.
Перейди в начало цикла
```

К счастью, в те времена вы еще ничего не знали о циклах и счетчиках. Если бы вы предложили учителю такую интерпретацию его слов, у вас было бы больше неприятностей, чем от болтовни во время урока.

Если внимательно присмотреться, наша жизнь полна примерами циклов, большинство из которых — не что иное, как циклы `for`. Компьютерная программа — это как бы формализованное отображение процессов, происходящих в реальной жизни (или наоборот). Когда вы приказываете компьютеру что-либо сделать, вы чаще всего приказываете сделать это многократно: напечатать три строки, обработать десять банковских счетов, проверить миллион телефонных номеров и т.п. Приказ сделать нечто один раз чаще всего не имеет смысла, потому что легче сделать это самому, чем написать программу. Например, умножить на калькуляторе два числа легче, чем написать программу, которая сделает это. Цикл `for` специально создан

для того, чтобы выполнить набор инструкций заданное количество раз. Сравните: рассмотренный выше цикл `while` устроен так, что заранее неизвестно, сколько раз он будет выполнен.

В листинге 6.2 приведен простейший пример использования цикла `for`. Более сложный пример мы рассмотрим в листинге 6.3.

Листинг 6.2. Очень простой и скучный пример использования цикла `for`

```
import static java.lang.System.out;
class Yawn {
  public static void main(String args[]) {
    for (int count = 1; count <= 10; count++) {
      out.print("Значение счетчика равно ");
      out.print(count);
      out.println(".");
    }
    out.println("Достаточно!");
  }
}
```

Результат показан на рис. 6.3 (получили в точности то, что хотели). Цикл `for` в листинге 6.2 начинается с присвоения счетчику `count` значения 1. После этого программа проверяет значение счетчика. Если оно меньше или равно 10 (в данный момент так оно и есть, потому что 1 меньше 10), компьютер выполняет тело цикла (т.е. блок инструкций, заключенных в фигурные скобки). В первой итерации счетчик равен 1. Это значение выводится на консоль методом `println()`. Закончив выполнять тело цикла, компьютер возвращается к заголовку цикла (*заголовок цикла* — это выражение в круглых скобках после ключевого слова `for`). В заголовке компьютер выполняет следующие операции: увеличивает значение счетчика на единицу и сравнивает значение счетчика с числом 10.

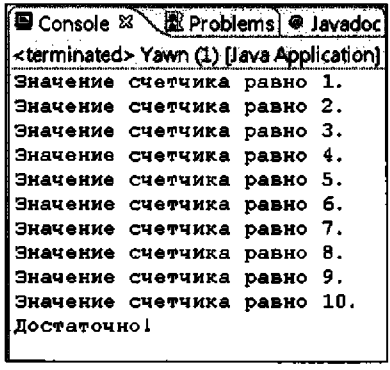


Рис. 6.3. Компьютер умеет считать до десяти

Теперь значение счетчика равно 2. Конечно, это меньше 10, поэтому опять выполняется тело цикла и метод `println()` выводит на консоль значение счетчика. И так далее. В конце концов значение счетчика становится равным 11, программа выходит из цикла и печатает слово `Достаточно`. Схема данного процесса изображена на рис. 6.4.



Рис. 6.4. Принцип работы цикла `for`

Структура цикла `for`

После ключевого слова `for` в скобках приведены три выражения. Первое называется *инициализацией цикла*, второе — *условием цикла* и третье — *инкрементом счетчика*.

```
for( инициализация; условие; инкремент )
```

Каждое из этих выражений играет свою роль.

- ✓ **Инициализация** — это присвоение счетчику начального значения. Инициализация всегда выполняется только один раз, когда программа начинает выполнять инструкцию `for`.
- ✓ **Условие** оценивается многократно **перед** каждой итерацией цикла.
- ✓ **Инкремент** счетчика выполняется многократно **после** каждой итерации цикла.

То же самое можно сделать с помощью цикла `while` следующим образом.

```
count = 1;
while ( count <= 10) {
      out.print("Значение счетчика равно ");
      out.print(count);
      out.println(".");
      count++;
    }
```

Результат будет тот же, но в этом случае код получается менее наглядным и более подверженным ошибкам. В инструкции `for` все, что относится к циклу, сосредоточено в заголовке, а в цикле `while` инструкции, управляющие циклом, "спрятаны" в его теле и за его пределами.



Если объявить переменную счетчика в круглых скобках (как в листинге 6.2), ее нельзя будет использовать за пределами цикла, иначе компилятор

сообщит об ошибке. Часто так и делают. За пределами цикла счетчик не нужен, и если он оказался там в результате ошибки, компилятор сообщит об этом.

Премьера хита "Эл под дождем"

В листинге 6.2 хорошо проиллюстрировано использование цикла `for`, но в нем не происходит ничего интересного. В листинге 6.3 приведен более увлекательный пример цикла `for`, который выводит на консоль знаменитый сингл "Эл под дождем", печатный текст которого можно найти в главе 5.

Листинг 6.3. Программа, объясняющая, почему Эл стоит под дождем

```
import static java.lang.System.out;
class AlsAllWet {
  public static void main(String args[]) {
    for (int verse = 1; verse <= 3; verse++) {
      out.print("Эл весь мокрый.");
      out.println("Почему Эл весь мокрый?");
      out.println("Потому, что он стоит под дождем");
      out.println("Почему Эл стоит под дождем?");
      switch (verse) {
      case 1:
        out.println("Потому что дождь идет очень долго");
        break;
      case 2:
        out.println("Потому что он очень несчастлив");
        break;
      case 3:
       out.println("Потому что он все равно уже мокрый");
        break;
      }
      switch (verse) {
      case 3:
        out.println("Очень долго, очень долго...");
      case 2:
        out.println("Очень, очень несчастлив.");
      case 1:
        out.println("Все равно, все равно уже мокрый.");
      }
      out.println("Под дождем, под дождем");
      out.println("О-оооооо");
      out.println();
    }
    out.print("Эл весь мокрый");
    out.println("Почему Эл весь мокрый?");
    out.print("Потому что он стоит под дождем. ");
    out.println("Под дождем, под дождем.");
  }
}
```

В листинге 6.3 две инструкции `switch` вложены в цикл `for`. В первой инструкции `switch` используются операторы `break`, а во второй не используются, в результате чего программа "выпадает" из инструкции `switch`. В цикле `for` используется счетчик `verse`, содержащий номер куплета и увеличивающийся от `1` до `3`. В результате куплеты и припевы песни повторяются в правильной последовательности и столько раз, сколько нужно для создания драматического эффекта. К сожалению, ритмическая структура стиха не позволяет раскрыть вопрос, будет ли Эл счастлив от того, что стоит под дождем. Когда программа выходит из цикла `for`, методы `print()` выводят на консоль заключительные слоги песни.



Чаще всего цикл `for` используется для выполнения заранее заданного количества итераций. Но его можно использовать и во многих других случаях. Например, можно опустить выражение инкремента, записав заголовок так: `for(i=0;i<10;)`. В результате счетчик не будет увеличиваться в заголовке. Счетчик можно изменять в теле цикла, и в результате программа выйдет из цикла, когда значение счетчика окажется больше десяти. В этом случае счетчик необязательно должен увеличиваться в каждой итерации на единицу. Можно также опустить все выражения заголовка, записав `for(;;)`. Такой цикл будет выполняться вечно. Иногда это нужно, если программа управляет непрерывным технологическим процессом, который длится годами и не прекращается ни на минуту ни днем, ни ночью. Однако на практике чаще всего цикл `for` используется, когда нужно повторить некоторый набор инструкций заданное количество раз.



В листинге 6.3 операторы `break` приводят к выходу из блока `switch`. Но их можно использовать и для завершения цикла. В теле цикла они могут играть ту же роль, что и в блоке `switch`. Встретив оператор `break`, программа немедленно выходит из цикла и выполняет инструкцию, находящуюся непосредственно после тела цикла.

Цикл do

Хорошенько подумай,
прежде чем сделать что-либо.
Конфуций

В этом разделе мы нарушим важное правило, которому учит Конфуций. Вернитесь к листингу 6.2 и обратите внимание на то, как работает цикл `while`. Когда программа подходит к циклу, она в первую очередь проверяет, истинно ли условие. Если условие равно `false`, программа не входит в цикл. Таким образом, тело цикла в некоторых случаях может не выполниться ни разу. Но во многих случаях оно должно быть выполнено хотя бы один раз, например при игре в угадывание чисел (см. листинг 6.1) пользователь должен совершить хотя бы одну попытку. Не может же он угадать число за `0` попыток.

Цикл `while` используется, когда программе нужно хорошенько подумать, прежде чем выполнить тело цикла хотя бы один раз. Это мудрое правило применяется чаще всего. Предположим, например, что программа перечисляет деньги с текущего счета

на корреспондентские счета в некотором цикле. Прежде чем начать транзакцию, программа должна проверить, есть ли деньги на текущем счете, чтобы не начинать операцию, которая все равно не будет успешно завершена.

Но в некоторых ситуациях необходимо выполнить тело цикла хотя бы один раз, независимо от любых условий. Например, такая ситуация возникает, когда пользователь должен каким-либо образом отреагировать на запрос программы. Возможно, ответ пользователя имеет смысл, но иногда пользователь совершает ошибку или попросту "отмахивается", введя первые попавшиеся несколько букв. Возможно, пользователь не попал по нужной клавише или не понял вопрос. Если ответ "не лезет ни в какие ворота", программа должна повторить запрос.

На рис. 6.5 показаны два сеанса программы, предлагающей удалить файл. В первую очередь программа должна спросить пользователя, не возражает ли он. Пользователь должен ввести `y` или `n`, в результате чего программа удаляет файл или отменяет данную операцию. Но если пользователь введет любой другой символ, программа должна спросить еще раз, и так до тех пор, пока пользователь не введет `y` или `n`.

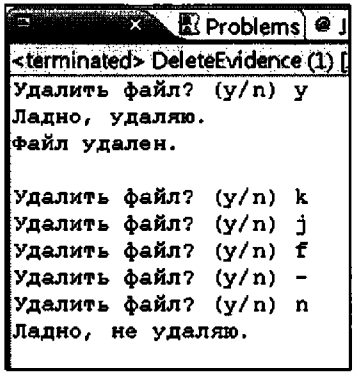


Рис. 6.5. Два сеанса программы, приведенной в листинге 6.4

Очевидно, что для создания этой программы необходим цикл, который будет повторять запрос о том, нужно ли удалить файл. Запрос будет повторяться, пока пользователь не введет осмысленный ответ. Обратите внимание на то, что первая итерация цикла должна быть выполнена в любом случае независимо от каких-либо условий. Пока пользователь не ввел что-либо, проверять нечего. Цикл начинает выполняться не с проверки условия, а с первой инструкции тела цикла. Сначала должна пройти первая итерация, чтобы пользователь мог ввести что-нибудь, и только после этого можно проверить условие.

Это означает, что в данной программе нужно использовать цикл `do`, а не `while`. Впрочем, иногда его называют циклом `do...while`, потому что условие `while` находится в конце цикла (листинг 6.4). При выполнении цикла `do` программа сначала выполняет тело цикла и только после этого проверяет условие. Если условие равно `true`, программа опять выполняет тело цикла. Если же условие равно `false`, программа выходит из цикла и выполняет первую инструкцию, расположенную непосредственно после цикла.

Листинг 6.4. Удалить файл? Принимаются только ответы y и n

```
import java.io.File;
import static java.lang.System.out;
import java.util.Scanner;
class DeleteEvidence {
  public static void main(String args[]) {
    File evidence = new File("cookedBooks.txt");
    Scanner keyboard = new Scanner(System.in);
    char reply;
    do {
      out.print("Удалить файл? (y/n) ");
      reply = keyboard.findWithinHorizon(".",0).charAt(0);
    } while (reply != 'y' && reply != 'n');
    if (reply == 'y') {
      out.println("Ладно, удаляю.");
      evidence.delete();
      out.println("Файл удален.");
    } else {
      out.println("Ладно, не удаляю.");
    }
  }
}
```

На рис. 6.5 показаны два сеанса работы этой программы. Программа принимает буквы y и n в нижнем регистре, но, если пользователь введет их в верхнем регистре, программа сочтет такой ответ бессмысленным и повторит вопрос. Чтобы программа принимала буквы Y и N, измените условие следующим образом.

```
do {
  out.print("Удалить файл? (y/n) ");
  reply = keyboard.findWithinHorizon(".", 0).charAt(0);
} while (reply != 'y' && reply != 'Y' &&
         reply != "n" && reply!="N");
if (reply == 'y' || reply == 'Y') {
```

Схема работы программы, приведенной в листинге 6.4, показана на рис. 6.6. Поскольку тело цикла выполняется в первую очередь, без проверки условия, данный цикл гарантирует, что произойдет как минимум одна итерация цикла.



Местоположение файла `cookedBooks.txt` на жестком диске вашего компьютера может быть разным в зависимости от определенных обстоятельств. Если файл находится не в той папке, которую ожидает программа, то она не сможет удалить файл. Однако, вероятнее всего, вы тестируете листинг 6.4 в рабочей среде Eclipse, поэтому новый проект находится в определенной папке жесткого диска. Эта папка становится текущей, и файл `cookedBooks.txt` попадает точно в нее. Например, в среде Eclipse проект примера имеет имя `Listing06-04`. Проект находится в папке `C:\Eclipse_Workspace\Listing06-04`. В эту папку я специально для вас поместил файл `cookedBooks.txt`. Возможно, экспериментируя с

листингом 6.4, вы уже удалили его. Кроме того, в этой же папке есть вложенная папка `src`, в которой находится файл `DeleteEvidence.java` с кодом листинга 6.4.



Рис. 6.6. Схема выполнения цикла `do`



Манипулирование файлами и папками подробно рассматривается в главе 8.

Считывание одного символа с клавиатуры

В листинге 5.3 пользователь вводил с клавиатуры слово. Метод `keyboard.next()` считывал введенное слово и помещал его в переменную `password` типа `String`. Все работало безукоризненно, потому что переменная типа `String` может хранить много символов одновременно, а метод `next()` считывает много символов.

Однако в листинге 6.4 не нужно считывать много символов. Пользователь должен ввести одну букву — `y` или `n`. Поэтому программа создает не переменную типа `String`, в которой можно хранить много символов, а переменную типа `char`, хранящую один символ.

В Java API нет метода `nextChar()`, который считывал бы с клавиатуры один символ, поэтому нам пришлось импровизировать. Ниже приведен результат импровизации.

```
keyboard.findWithinHorizon(".",0).charAt(0);
```

Метод `findWithinHorizon(".",0)` принимает с клавиатуры строку до первой точки длиной один символ (это звучит неуклюже, но программная реализация оказалась простой и эффективной), а метод `charAt(0)` возвращает первый символ полученной строки. Когда вам понадобится принимать один символ с клавиатуры, можете использовать этот код, даже если не вполне понимаете, как он работает.



Переменная типа `String` может содержать произвольное количество символов, в том числе один символ или нуль символов (в этом случае строка называется *пустой*, но не путайте ее со значением `null`, которое означает, что переменной не присвоено никакого значения). Обратите внимание на то, что переменная типа `String`, содержащая один символ, и переменная

типа char, которая также содержит один символ, — это совершенно разные вещи. В первую очередь, важно то, что это переменные разных типов, поэтому в коде они не взаимозаменяемые.

Манипулирование файлами

В листинге 6.4 заслуживают внимания инструкции, работающие с файлами. В этих инструкциях используются классы, объекты и методы, более подробное описание которых приводится в главах 7–9. А сейчас я лишь вкратце коснусь того, как работает очень полезный и популярный класс File.

Приведенная ниже инструкция создает объект evidence типа File.

```
File evidence = new File("cookedBooks.txt");
```

Объект evidence содержит всю информацию о том, как файл cookedBooks.txt хранится на жестком диске компьютера. После создания объекта evidence он ссылается на файл cookedBooks.txt.

Посмотрите в справочник по классам API Java, и вы увидите, что объект типа File содержит ряд методов для манипулирования файлами, в частности метод delete(), который удаляет данный файл с диска.

Объявление переменной в блоке

Блок — это набор инструкций, заключенных в фигурные скобки. Переменная, объявленная внутри блока, за его пределами невидима. Ее нельзя использовать за пределами блока. Например, если изменить часть листинга 6.4, как показано ниже, компилятор вернет сообщение об ошибке.

```
do {
  out.print("Удалить файл? (y/n) ");
  char reply =
    keyboard.findWithinHorizon(".", 0).charAt(0);
} while (reply != 'y' && reply != 'n');
if (reply == 'y')
```

Переменную reply нельзя упоминать за пределами фигурных скобок даже в операторе while. Если вы попытаетесь скомпилировать этот код, то получите три сообщения об ошибке — два сообщения о присутствии переменной reply в операторе while и одно сообщение о ее присутствии в операторе if.

Поэтому в листинге 6.4 переменная reply должна быть объявлена во внешнем блоке, т.е. в методе main(). Чтобы она была доступной после цикла, нужно объявить ее перед циклом. Обратите внимание на то, что в этой ситуации переменную reply можно не инициализировать.



Инициализация рассматривается в главе 4, а блоки — в главе 5.

Три рассмотренных выше цикла (while, for и do) доступны во всех версиях Java. Кроме того, в последних версиях (Java 5 и выше) доступны расширенные циклы for, которые рассматриваются в главе 11.

Часть III

Объектно-ориентированное программирование

The 5th Wave

Рич Теннант



В этой части...

Вы слышали что-нибудь об объектно-ориентированном программировании? Наверняка слышали, потому что в настоящее время этот термин часто упоминается. Однако людям, не связанным с объектно-ориентированным программированием, оно кажется чем-то вроде закрытого клуба со своими секретными условными сигналами, непонятными для непосвященных. Прочитайте эту часть, и секретов для вас больше не будет. Я не только сниму покров тайны с этой концепции, но и научу вас использовать ее для создания программ на Java.

Глава 7

Классы и объекты

В этой главе...

- Определение класса
- Определение метода в классе
- Передача параметров методу и получение значения от метода
- Как сделать, чтобы числа выглядели красиво
- Модификаторы доступа

Как автору книг о компьютерах мне неоднократно говорили редакторы, что я не должен рассчитывать на то, что читатели будут читать мои книги строго последовательно, от корки до корки. Читатель практически всегда ищет в книге то, что ему необходимо, открывает нужную страницу и пропускает то, что ему не хочется читать. Поэтому я вынужден предполагать, что вы (или, по крайней мере, многие читатели) пропустили главу 1. Если это так, можете не чувствовать себя виноватым. Вам необязательно возвращаться к главе 1. Вы в значительной степени компенсируете нехватку знаний, прочитав следующий абзац, который представляет собой "настойку", или "выжимку", из главы 1.

> *Java — это объектно-ориентированный язык программирования, поэтому ваша главная задача состоит в создании классов и объектов. Класс представляет собой определение некой сущности, а объект — это конкретный экземпляр данной сущности. Программист сначала определяет класс, а затем создает его экземпляры.*

Конечно, вы можете пропустить даже этот абзац. И в таком случае вы легко наверстаете упущенное, прочитав еще более короткую "выжимку" из главы 1.

> *Сначала создайте класс, а затем — экземпляры класса.*

Определение класса

Чем один банковский счет отличается от другого? Если вы спросите об этом у банкира, то услышите длинное объяснение, в котором будут упомянуты процентные ставки, комиссии, платежи, штрафные санкции, тарифы, условия транзакций и многое другое. К счастью, все это нам сейчас не нужно. Все, что сейчас нужно, — это знать, чем мой счет отличается от вашего. Ведь мы, конечно же, пользуемся разными счетами. Мой счет называется *Барри Берд, автор книги*, а ваш — *Джейн Ридер, начинающий джабер*. На моем счете сейчас 24 доллара и 2 цента, а на вашем?

Конечно, счета отличаются не только названиями, но и многими другими параметрами, такими как дата открытия, сумма на счету, тип счета, процентная ставка,

паспортные данные клиента и т.п. Все эти параметры можно представить в программе как переменные, которым присвоены определенные значения. Предположим, одна из переменных называется `balance` и содержит текущую сумму на счете. Мое значение `balance` равно `24.02`, а ваше — `55.63`. При написании программы вопрос состоит в том, как мы будем отличать мою переменную `balance` от вашей?

Ответ состоит в создании двух отдельных объектов. Мое значение `balance` мы поместим в мой объект, а ваше — в ваш. Кроме того, определим в классе переменные `name` (имя) и `address` (адрес), в результате чего они будут присутствовать в каждом объекте. Следовательно, у нас есть два объекта, каждый из которых является экземпляром класса `Account` (рис. 7.1) и представляет банковский счет.

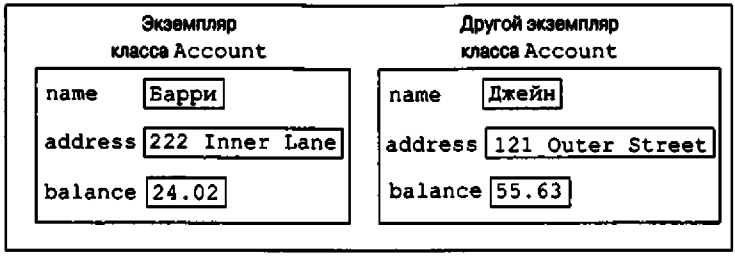


Рис. 7.1. Два объекта — это два банковских счета

Пока все прекрасно: у нас есть два набора переменных. Однако исходная проблема все еще не решена. Как мы будем в коде программы ссылаться на мою переменную `balance` таким образом, чтобы она отличалась от вашей с тем же именем? Для этого объявим две разные переменные: `myAccount` и `yourAccount`. Переменная `myAccount` будет обозначать мой объект (т.е. мой экземпляр класса `Account`), а сумма на моем счете будет обозначаться следующим образом:

```
myAccount.balance
```

Мое имя будет храниться в такой переменной:

```
myAccount.name
```

Аналогично переменная `yourAccount.balance` обозначает ваше значение `balance`, а переменная `yourAccount.name` — ваше имя. Теперь можно присвоить значение `24.02` моей переменной `balance`.

```
myAccount.balance = 24.02;
```

Отобразить ваше имя на консоли можно с помощью следующей инструкции:

```
out.println(yourAccount.name);
```

Определение класса `Account` в коде программы приведено в листинге 7.1.

Листинг 7.1. Класс `Account`

```
public class Account {
  String name;
  String address;
  double balance;
}
```

В определении класса находится вся информация о нем. В частности, определение класса Account, приведенное в листинге 7.1, сообщает о том, что каждый экземпляр класса содержит три переменные — name, address и balance, которые в Java называются полями.



Поле — это переменная, объявленная внутри класса, но не внутри метода. В листинге 7.1 полями являются переменные name, address и balance. Другое название поля — *переменная экземпляра*.

Если вы читали главы 4–6, то определение класса Account может вас удивить. Можно ли определить целый класс всего лишь в четырех строках кода? Конечно, можно. Фактически определение класса Account, приведенное в листинге 7.1, прекрасно иллюстрирует концепцию класса. Класс группирует некоторые данные. В листинге 7.1 класс Account группирует три переменные. Две из них имеют тип String, а третья — тип double. Теперь, когда есть определение класса Account, его можно считать типом и говорить, что экземпляры этого класса — это объекты типа Account.

Открытый класс

Класс называется *открытым*, если он объявлен с модификатором public. В листинге 7.1 в заголовке класса Account приведено ключевое слово public, следовательно, это открытый класс. Наиболее важное свойство открытого класса состоит в том, что его можно использовать в любом другом классе. Например, если вы пишете программу ATMController, работающую в отдаленном конце киберпространства, она может содержать ссылку на класс Account. В ней можно написать myAccount.balance=24.02. Конечно, код программы ATMController должен знать, в какой точке киберпространства находится определение класса Account.

В в листинге 7.2 мы применим ссылку на класс Account в другом классе. Поэтому, казалось бы, класс Account должен быть открытым, поскольку он используется в другом классе. К сожалению, не все так просто. Фактически, чтобы использовать первый класс в коде второго класса, первый класс не обязательно должен быть открытым. Поэтому в данной главе ключевое слово public в листинге 7.1 не обязательное. Программа, приведенная в листинге 7.2, правильно работает независимо от того, есть ли в определении класса Account ключевое слово public.



Более подробная информация о модификаторе public и других модификаторах приведена в главе 13.

Объявление переменных и создание объектов

Я иду по улице. Ко мне подходит молодой человек и просит написать "Я обожаю Java!" Я пишу эти слова мелом на асфальте. Как и зачем я это делаю, сейчас не важно. Важно лишь то, что некто произнес вслух инструкцию, а я ее выполнил.

В тот же день вечером я сижу на скамейке в парке. Рядом садится пожилая женщина и говорит: "Счет содержит имя, адрес и сумму". Я отвечаю: "Прекрасно! Но что я должен сделать в связи с этим?" В ответ она смотрит на меня и ничего больше не говорит, поэтому я ничего не делаю. Я всего лишь принял к сведению то, что она сказала.

Листинг 7.1 похож на пожилую женщину в парке. Он сообщает о том, что такое Account, но ничего не говорит о том, что нужно что-либо сделать с моим или любым другим счетом. Чтобы сделать что-нибудь, нужен дополнительный код. Нужен еще один класс, содержащий метод, в котором что-то делается с классом Account, поскольку все, что делается в Java, делается только внутри методов (за пределами методов можно только инициализировать поля класса, но вряд ли это можно считать серьезной работой). В листинге 7.2 приведен класс, в котором используется объект типа Account. Учитывайте, что в этой фразе слово "тип" обычно опускают. Следовательно, выражение "объект Account" чаще всего означает "объект типа Account".

Листинг 7.2. Использование объектов типа Account

```
import static java.lang.System.out;
class UseAccount {
  public static void main(String args[]) {
    Account myAccount;
    Account yourAccount;

    myAccount = new Account();
    yourAccount = new Account();

    myAccount.name = "Барри";
    myAccount.address = "222 Inner Lane";
    myAccount.balance = 24.02;
    yourAccount.name = "Джейн";
    yourAccount.address = "121 Outer Street";
    yourAccount.balance = 55.63;

    out.print(myAccount.name);
    out.print(" (");
    out.print(myAccount.address);
    out.print(") имеет $");
    out.print(myAccount.balance);
    out.println();
    out.print(yourAccount.name);
    out.print(" (");
    out.print(yourAccount.address);
    out.print(") имеет $");
    out.print(yourAccount.balance);
  }
}
```

Взятые вместе, два этих класса — Account и UseAccount — составляют полнофункциональную программу Java. В коде листинга 7.2 определен класс UseAccount, в котором есть метод main(). В методе main() определены две переменные: youtAccount и myAccount.

Первые две строки метода main() могут ввести начинающего программиста в заблуждение из-за ложной аналогии с объявлением примитивных типов. Например, объявление int k означает, что переменная k типа int создана и ее можно использовать в программе. Многие начинающие программисты аналогично поступают и с

типом Account. Однако объявление Account yourAccount не означает, что переменную yourAccount можно использовать в программе. Пока что она всего лишь объявлена, но в ней ничего нет. Сейчас она имеет значение null.

Непосредственно после объявления Account yourAccount переменная еще не ссылается ни на какой объект. Объявление всего лишь резервирует имя yourAccount, которое впоследствии можно будет использовать для ссылки на объект типа Account (или, что то же самое, экземпляр класса Account). Объект будет создан позже в коде, когда компьютер выполнит оператор new Account().



Технически, когда компьютер выполняет оператор new Account(), он создает объект путем вызова так называемого "конструктора" класса Account. Более подробно конструкторы рассматриваются в главе 9.

Инструкция yourAccount=new Account() приказывает компьютеру создать объект типа Account и присвоить его переменной yourAccount. После этого *объект хранится в переменной* (или, что то же самое, *переменная ссылается на объект*). Данная операция проиллюстрирована на рис. 7.2. До выполнения оператора new мы имеем пустую переменную yourAccount, а после его выполнения — переменную yourAccount, ссылающуюся на объект (правда, объект пока что пустой, потому что поля не инициализированы).

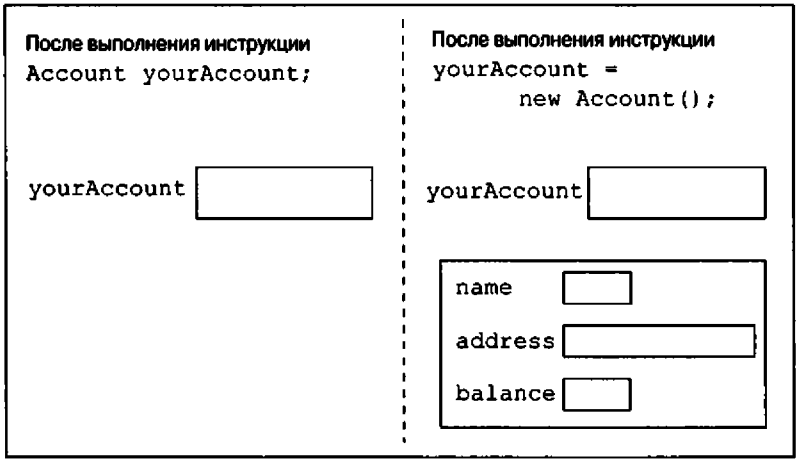


Рис. 7.2. До и после вызова конструктора

Чтобы проверить вышесказанное практически, добавьте несколько строк в листинг 7.2 и попытайтесь вывести на консоль значение yourAccount.name после объявления переменной yourAccount, но перед вызовом оператора new Account().

```
Account myAccount;
Account yourAccount;
out.println(yourAccount.name);
myAccount = new Account();
yourAccount = new Account();
```

При попытке скомпилировать этот код вы получите сообщение о том, что переменная `yourAccount` не инициализирована. Это означает, что до выполнения оператора `new Account()` вывести на консоль значение переменной `name` нельзя, потому что объекта не существует.



Для переменной ссылочного типа ее объявления недостаточно. Нужно, кроме этого, создать объект путем вызова конструктора с помощью ключевого слова `new`.



Ссылочные типы рассматриваются в главе 4.

Инициализация переменной

В главе 4 рассматривалась инициализация переменной примитивного типа непосредственно в объявлении.

```
int weightOfAPerson = 150;
```

То же самое можно делать с переменными ссылочного типа, например с переменными `myAccount` и `yourAccount` в листинге 7.2. Вы можете заменить первые четыре строки кода в методе `main()` двумя следующими строками.

```
Account myAccount = new Account();
Account yourAccount = new Account();
```

Даже более того: это можно сделать за пределами любого метода. Это единственное исключение из того правила, что любая операция может выполняться только в теле метода, а за пределами метода можно только объявлять переменные.



В некоторых ситуациях сообщение компилятора о том, что переменная не инициализирована, появляется, даже когда она на самом деле инициализирована. Это происходит, если блок, в котором выполняется инициализация, по некоторым причинам может быть не выполнен (например, если он находится в инструкции `if` или `while`). Компилятор чрезмерно осторожничает: если есть хотя бы малейшая возможность ошибки, он не пропускает программу на выполнение.

Как видите, код листинга 7.2 довольно скучный. Нам нашлось что сказать лишь о первых четырех строках метода `main()`. Дальше все просто: программа присваивает значения полям объектов и выводит их на консоль. Результат показан на рис. 7.3.

```
<terminated> UseAccount (5) [Java Application] D:\Progr
Барри (222 Inner Lane) имеет $24.02
Джейн (121 Outer Street) имеет $55.63
```

Рис. 7.3. Выполнение кода листингов 7.1 и 7.2

Одна программа — это много классов

Каждая программа в главах 3–6 состояла из одного класса. Это приемлемо для начальных глав учебника по программированию, но в реальной жизни программы состоят из сотен, а иногда даже тысяч классов. В двух листингах (7.1 и 7.2) приведены два класса, из которых состоит одна программа. Конечно, двум очень далеко до нескольких тысяч, но, по крайней мере, это шаг в правильном направлении.

Среди программистов Java принято помещать каждый класс в отдельный файл с расширением `.java`, причем имя файла должно совпадать с именем класса. Компилятор не требует этого, тем не менее рекомендуется соблюдать данное правило, потому что оно поможет вам избежать ошибок. К тому же рабочая среда Eclipse не позволит вам нарушить это правило. Таким образом, для выполнения предыдущего примера нужно создать два файла с именами `Account.java` и `UseAccount.java`.

Определение метода в классе

Представьте себе таблицу, содержащую информацию о двух счетах (табл. 7.1).

Таблица 7.1. Счета без объектно-ориентированного программирования

Имя	Адрес	Баланс
Барри	222 Inner Lane	24.02
Джейн	121 Outer Street	55.63

В табл. 7.1 у каждого счета есть три компонента: имя, адрес и баланс. Так обстояли дела до появления объектно-ориентированного программирования. Казалось бы, все прекрасно. Каждый счет содержит всю необходимую информацию о нем. Однако концепция объектно-ориентированного программирования радикально изменила парадигму мышления. Согласно ей, с каждым счетом должна быть ассоциирована не только информация о нем, но и методы обработки этой информации. Например, со счетами, приведенными в табл. 7.1, можно ассоциировать методы вывода имени, адреса и баланса на консоль.

В объектно-ориентированной программе в каждый объект встроена собственная функциональность. Счет "знает", как вывести себя на консоль. Переменная типа `String` может сообщить вам о том, есть ли в строке указанные вами символы. Объект класса `System.out` "знает", как отобразить заданную строку. С каждым объектом ассоциирован набор методов. Эти методы можно вызывать в коде для выполнения нужных операций над данными, находящимися в этом же или в других объектах.

Какие преимущества это предоставляет по сравнению с хранением только информации? Главное преимущество — возможность собрать в одном месте информацию и методы ее обработки. Всю функциональность, ассоциированную со счетами, можно поместить в код класса `Account`. Аналогично все, что имеет отношение к типу `String`, находится в файле `String.java`. Каждый программист, у которого возникнут проблемы с классом `Account`, знает, что все, что ему может понадобиться, он увидит в файле `Account.java`. Со временем вы убедитесь в том, что это огромное преимущество.

Представим себе расширенную таблицу, содержащую счета и построенную на основе принципов объектно-ориентированного программирования. С каждым объектом ассоциирована встроенная функциональность. Каждый счет "знает", как отобразить себя на консоли. Каждая строка таблицы имеет собственную копию метода `display()`, который выводит информацию о счете на консоль (рис. 7.2).

Таблица 7.2. Счета в объектно-ориентированной программе

Имя	Адрес	Баланс	Функциональность
Барри	222 Inner Lane	24.02	`display()`
Джейн	121 Outer Street	55.63	`display()`

Счет, отображающий сам себя

В табл. 7.2 каждый счет содержит четыре компонента: имя, адрес, баланс и способ вывода этих данных на консоль. Когда у вас включится "объектно-ориентированный способ мышления", вы уже никогда не вернетесь обратно. Вы будете думать: "Без функциональности в классе пришлось бы либо искать ее где-нибудь в других файлах, либо создавать каждый раз заново. Но ведь это собачья жизнь!" И действительно, без концепции объектно-ориентированного программирования современные программы были бы значительно более длинными и запутанными. В листингах 7.3 и 7.4 приведена программа, реализующая идеи, представленные в табл. 7.2.

Листинг 7.3. Усовершенствованный счет, выводящий себя на консоль

```
import static java.lang.System.out;
public class Account {
  String name;
  String address;
  double balance;
  public void display() {
    out.print(name);
    out.print(" (");
    out.print(address);
    out.print(") имеет на счету $");
    out.print(balance);
  }

  }
```

Листинг 7.4. Использование усовершенствованного класса `Account`

```
class UseAccount {
  public static void main(String args[]) {
    Account myAccount = new Account();
    Account yourAccount = new Account();
    myAccount.name = "Барри";
    myAccount.address = "222 Inner Lane";
    myAccount.balance = 24.02;
    yourAccount.name = "Джейн";
```

```
    yourAccount.address = "121 Outer Street";
    yourAccount.balance = 55.63;
    myAccount.display();
    System.out.println();
    yourAccount.display();
  }
}
```

Результат выполнения кода листингов 7.3 и 7.4 тот же, что и листингов 7.1 и 7.2 (см. рис. 7.3). Разница в том, что инструкции вывода на консоль теперь записаны в коде один раз, а не два. Если же программе нужно обработать тысячи счетов, можете себе представить, какая будет экономия рабочего времени программиста.

В новом классе `Account` (см. листинг 7.3) есть три поля (`name`, `address` и `balance`) и один метод `display()`. Эти четыре сущности соответствуют четырем столбцам табл. 7.2. Следовательно, с каждым экземпляром класса `Account` ассоциированы имя, адрес и баланс, а также способ их вывода на консоль. В объектно-ориентированном программировании они называются *членами класса*. Таким образом, поля и методы являются членами класса. Ссылка на любой член класса выглядит кратко. Чтобы сослаться на поле `name` объекта `myAccount`, нужно написать `myAccount.name`. Чтобы вызвать метод `display()` объекта `myAccount`, нужно написать `myAccount.display()`. Разница только в том, что при вызове метода нужно добавлять скобки.



При вызове метода нужно после его имени добавить скобки. В скобках приводится список параметров, передаваемых методу.

Заголовок метода

Просмотрите еще раз листинги 7.3 и 7.4. Инструкция вызова метода `display()` находится в методе `main()` класса `UseAccount`, а определение метода `display()` находится в классе `Account`. Определение метода состоит из тела и заголовка (см. главу 3). Заголовок состоит из трех частей и пары скобок.

- ✓ **Модификатор `public` играет ту же роль, что и в листинге 7.1.** Слово `public` означает, что метод открытый (в отличие от закрытого). Открытый метод можно вызвать в любом коде, в том числе в коде другого класса. В данном примере (листинги 7.3 и 7.4) метод `display()` можно было не объявлять открытым, потому что он вызывается своим же объектом.
- ✓ **Спецификатор `void` означает, что метод `display()` не возвращает в точку вызова никакого значения.** В следующем разделе рассматривается метод, возвращающий значение.
- ✓ **Слово `display` — это имя метода.** Каждый метод имеет имя. В противном случае как же его можно было бы вызвать?
- ✓ **В скобках приводится список параметров, передаваемых методу.** При вызове метода ему можно передать определенную информацию. Однако метод `display()` в листинге 7.3 выглядит немного странно, потому что в скобках ничего нет. Это значит, что методу не передается никакая информация. Ему достаточно информации, хранящейся в объекте. В данном случае список параметров пустой.



Определение метода `display()` находится в листинге 7.3, а его вызов — в листинге 7.4. В этих листингах находятся разные классы, расположенные в разных файлах, тем не менее в листинге 7.3 оба вхождения ключевого слова `public` не обязательные. Почему это так, рассматривается в главе 13.

Передача параметров методу и получение значения от метода

Представьте себе, что вы посылаете кого-то в магазин за хлебом. Вы говорите ему: "Пойди в магазин и купи хлеб". В другой раз вы посылаете этого же человека в магазин за бананами и говорите ему: "Пойди в магазин и купи бананы". Но какое отношение это имеет к программированию? Самое непосредственное. Вызов метода — это инструкция, приказывающая компьютеру сделать что-либо. Что именно нужно сделать, написано в определении метода. Имя — это не инструкция, а всего лишь название метода. Метод называется `пойдиВМагазинИКупи`. При вызове метода ему можно передать некоторую информацию в зависимости от того, что вам сейчас нужно. В Java вызовы указанного выше метода могут выглядеть следующим образом.

```
пойдиВМагазинИКупи(хлеб);
пойдиВМагазинИКупи(бананы);
```

То, что находится в скобках, называется *списком параметров*. Использование параметров позволяет сделать методы более гибкими и разнообразными. Вместо того чтобы делать каждый раз одно и то же, вы можете дать инструкцию купить разные вещи: хлеб, бананы или что-нибудь еще. Что именно нужно купить, можно задать в момент вызова метода.

Что происходит после вызова метода? Человек возвращается из магазина и говорит: "Вот хлеб, который ты просил купить". Выполняя вашу инструкцию, человек возвращает вам нечто такое, что называется результатом вызова метода.

В программировании результат работы метода называется *возвращаемым значением*. Важно помнить, что каждое возвращаемое значение должно иметь некоторый тип, заданный в определении метода. Если метод ничего не возвращает, перед его именем нужно ввести ключевое слово `void` (пусто). Эти концепции демонстрируются в листингах 7.5 и 7.6. Переменная `percentageRate` содержит процентную ставку, а метод `getInterest()` возвращает годовой доход.

Листинг 7.5. Счет, вычисляющий собственный доход

```
import static java.lang.System.out;
public class Account {
  String name;
  String address;
  double balance;
  public void display() {
    out.print(name);
    out.print(" (");
    out.print(address);
    out.print(") имеет на счету $");
```

```
    out.print(balance);
  }
  public double getInterest(double percentageRate) {
    return balance * percentageRate / 100.00;
  }
}
```

Листинг 7.6. Вычисление дохода

```
import static java.lang.System.out;
class UseAccount {
  public static void main(String args[]) {
    Account myAccount = new Account();
    Account yourAccount = new Account();
    myAccount.name = "Барри";
    myAccount.address = "222 Inner Lane";
    myAccount.balance = 24.02;
    yourAccount.name = "Джейн";
    yourAccount.address = "121 Outer Street";
    yourAccount.balance = 55.63;
    myAccount.display();
    out.print(" плюс $");
    out.print(myAccount.getInterest(5.00));
    out.println(" дохода ");
    yourAccount.display();
    double yourInterestRate = 7.00;
    out.print(" плюс $");
    double yourInterestAmount =
         yourAccount.getInterest(yourInterestRate);
    out.print(yourInterestAmount);
    out.println(" дохода ");
  }
}
```

Результат выполнения листингов 7.5 и 7.6 показан на рис. 7.4. В листинге 7.5 класс `Account` содержит метод `getInterest()`. Этот метод в листинге 7.6 дважды вызывается в методе `main()`. Как видите, для разных счетов баланс и доход разные.

```
<terminated> UseAccount (5) [Java Application] D:\Program Files\Java\jre6\bin\javaw.exe (19.11.2011 16:58:42)
Барри (222 Inner Lane) имеет на счету $24.02 плюс $1.2009999999999998 дохода
Джейн (121 Outer Street) имеет на счету $55.63 плюс $3.8941000000000003 дохода
```

Рис. 7.4. Результат выполнения кода, приведенного в листингах 7.5 и 7.6

- ✓ **В первом вызове баланс равен `24.02`, а процентная ставка — `5.00`.** При этом используется объект `myAccount` и методу `getInterest()` передается значение `5.00`. В таком случае говорят, что метод `getInterest()` вызывается **через** объект `myAccount`. Используемые при этом значения локальных переменных (`name`, `address` и `balance`) находятся в объекте `Account` (рис. 7.5). Следовательно, выражение `balance*percentageRate/100.00` вычисляется как `24.02*5.00/100.00`.

- ✓ **При втором вызове баланс равен 55.63, а процентная ставка — 7.00.** В методе `main()` непосредственно перед вторым вызовом переменной `yourInterestRate` присвоено значение `7.00`. Вызов метода `getInterest()` выполняется через объект `yourAccount`, поэтому при вычислении дохода в методе `getInterest()` используются поля объекта `yourAccount`. Следовательно, выражение `balance*percentageRate/100.00` вычисляется как `55.63*7.00/100.00`.

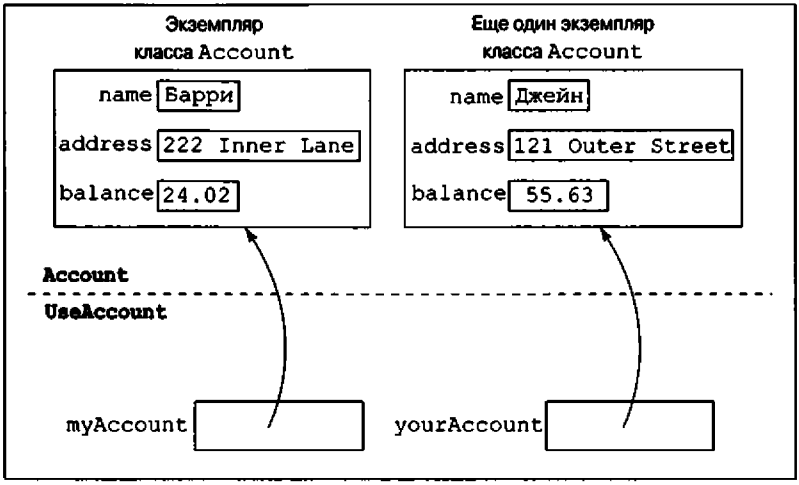


Рис. 7.5. Два счета

В листинге 7.6 метод `main()` содержит два вызова метода `getInterest()`. При первом вызове ему в списке параметров передается литерал `5.00`, а при втором вызове — переменная `yourInterestRate`. Почему в первом случае передается литерал, а во втором — переменная? Нет никакой причины. Просто я хотел показать, что методу можно передать либо одно, либо другое.

Передача значения в метод

Посмотрите на заголовок метода `getInterest()`. На рис. 7.6 показано, как значение `5.00` проходит от вызова метода к заголовку и от заголовка к месту его использования.

- ✓ **Слово `double` перед именем `getInterest` сообщает компьютеру о том, что в результате вызова метода `getInterest()` он должен вернуть значение типа `double`.** Инструкция в теле метода `getInterest()` полностью согласуется с этим требованием, потому что выражение `balance*percentageRate/100.00` имеет тип `double` (поскольку все его компоненты имеют тип `double`).

  При вызове метода `getInterest()` инструкция `return` вычисляет выражение `balance*percentageRate/100.00` и передает его значение

обратно в вызывающий метод. В вызывающем методе выражение `getInterest()` равно возвращаемому значению.

- ✓ **Слово `getInterest` в заголовке метода является его именем.** Имя метода используется для его вызова в классе `UseAccount`.
- ✓ **В скобках приведен список параметров, передаваемых методу при вызове.** В данном примере список параметров метода `getInterest()` состоит из одного параметра. Ключевое слово `double`, приведенное в списке параметров в заголовке метода, сообщает компьютеру о том, что метод ожидает значение типа `double`.



Посмотрите на первый вызов метода `getInterest()`. Как видите, ему передается литерал `5.00`, имеющий тип `double`.

Литералы рассматриваются в главе 4.

То же самое происходит и при втором вызове метода `getInterest()` в листинге 7.6. Методу передается переменная `yourInterestRate`, которая имеет тип `double`.

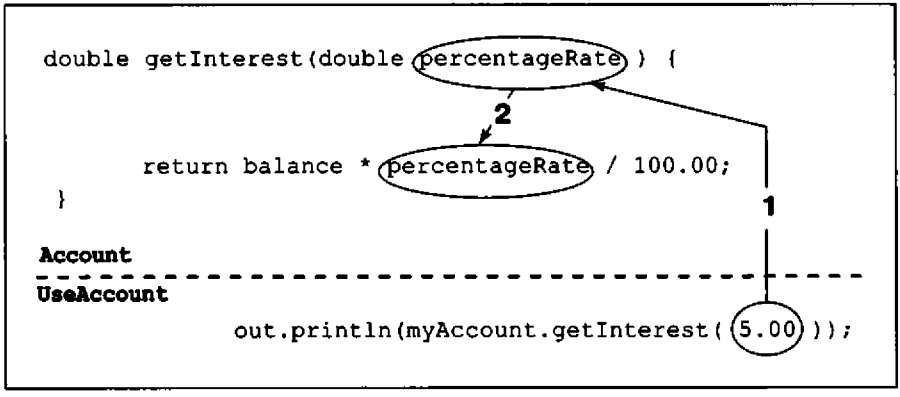


Рис. 7.6. Путь, проходимый значением

Код, приведенный в листингах 7.5 и 7.6, выполняется не сверху вниз, а как бы зигзагами. Сначала выполняется метод `main()`, затем `getInterest()`, затем опять `main()`, опять `getInterest()` и, наконец, все завершается выполнением последних инструкций метода `main()`. Путь выполнения кода показан на рис. 7.7.

Значение, возвращаемое методом

При вызове метод `getInterest()` выполняет единственную инструкцию, записанную в теле метода, а именно — инструкцию `return`, которая вычисляет значение выражения `balance*percentageRate/100.00`. Если значение `balance` равно `24.02`, а `percentageRate` — `5.00`, то значение выражения приблизительно равно `1.201`. Но поскольку в памяти компьютера числа хранятся не в десятичном, а двоичном виде, выражение возвращает немного другой результат. Точного представления числа `1.201` в двоичном виде не существует, поэтому компьютер возвращает число `1.200999999999998`, очень близкое к ожидаемому результату.

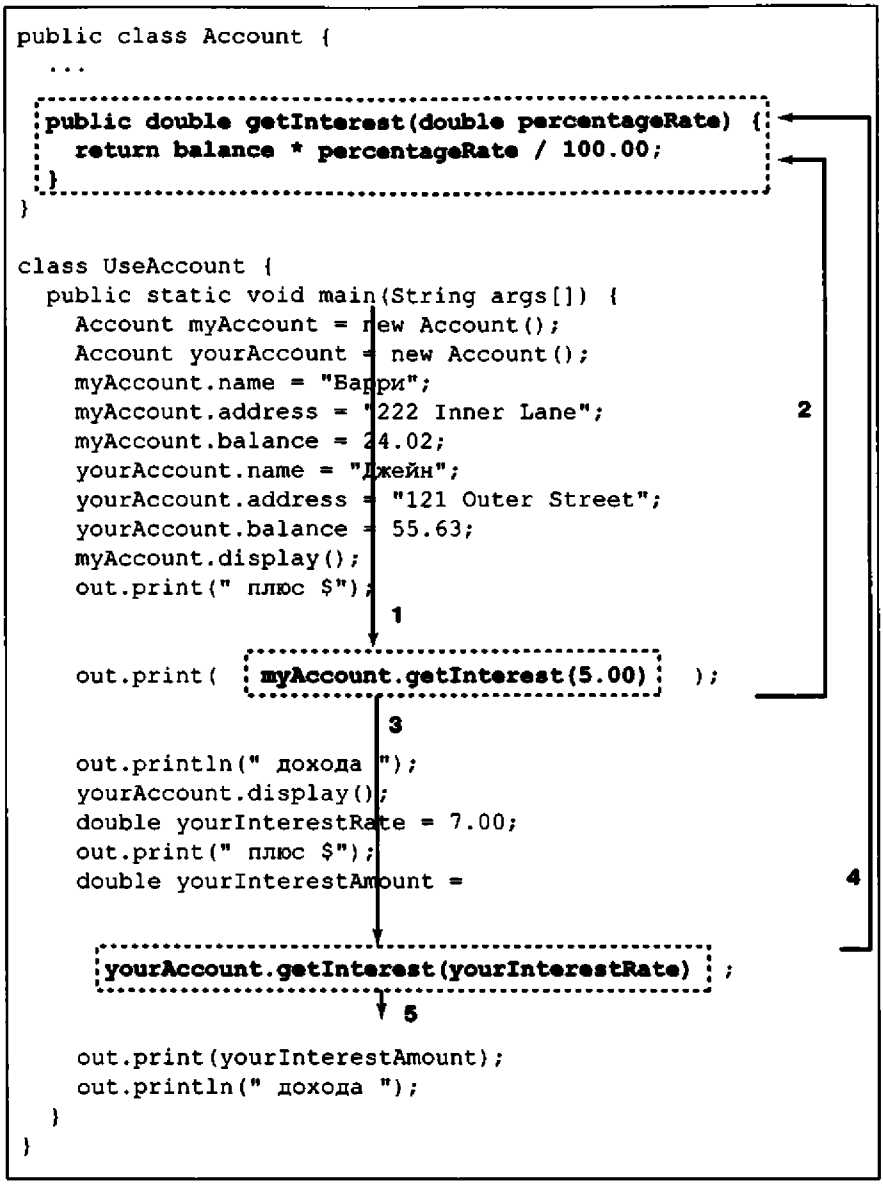


Рис. 7.7. Путь выполнения программы, показанной в листингах 7.5 и 7.6

Вычислив значение выражения, компьютер выполняет оператор `return`, который возвращает значение обратно в вызывающий метод и завершает выполнение текущего метода. В этот момент в методе `main()` выражение `myAccount.getInterest(5.00)` получает значение `1.200999999999998`. Инструкция вызова расположена в списке параметров метода `println()`.

```
out.println(myAccount.getInterest(5.00));
```

Поэтому приведенная выше инструкция эквивалентна следующей:

```
out.println(1.200999999999998);
```

Процесс возвращения результирующего значения метода проиллюстрирован на рис. 7.8.

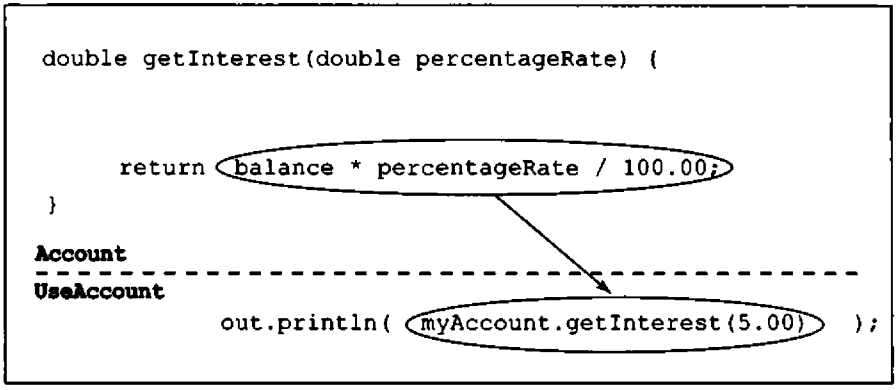


Рис. 7.8. Инструкция вызова метода интерпретируется как возвращаемое значение



Если метод что-либо возвращает, вызов метода интерпретируется как возвращаемое значение. Его можно вывести на консоль, присвоить переменной, использовать в любом другом выражении, т.е. можно делать с ним все, что можно делать со значением данного типа. Если же метод ничего не возвращает (объявлен со спецификатором `void`), то вызов — это просто инструкция сделать нечто.



Класс `Account`, приведенный в листинге 7.5, в принципе после некоторых изменений можно использовать в реальной программе. Метод `display()` позволяет отображать параметры счета на консоли, а метод `getInterest()` вычисляет доход. Однако класс `UseAccount`, приведенный в листинге 7.6, нигде нельзя применить. Это сугубо искусственный класс, созданный мною лишь для того, чтобы продемонстрировать работу класса `Account` (не думаете же вы, что в некотором банке есть клиенты Джейн и Барри). Класс `UseAccount` — это *тестовый модуль*, т.е. "короткоживущий" класс, нужный только для отладки других классов. В данной книге каждый тестовый класс написан в произвольной форме, как в листинге 7.6, и содержит метод `main()`, чтобы можно было запустить проверку. Программисты часто пишут тестовые модули в произвольной форме, однако для этого существует специальный программный инструмент JUnit, автоматизирующий создание тестовых модулей и отладку программы. Профессиональные программисты часто пользуются им при разработке программ.

Как сделать, чтобы числа выглядели красиво

На рис. 7.4 видно, что доходность моего счета равна `1.200999999999998` долларов. Кажется, банк украл у меня `0.000000000002` цента. Я пойду в банк прямо

сейчас и потребую вернуть причитающиеся мне деньги. А еще лучше — давайте пойдем вместе и покажем им, где раки зимуют. Если мои предположения верны, нехватка вызвана тем, что кто-то из служащих банка вставил в программу жучок, который "откусывает" с каждого счета крошечную сумму и переводит ее на свой счет. Люди не замечают крошечной недостачи, а на счету служащего может накопиться огромная сумма.

Однако почему у вас на счету оказался избыток `0.000000000003` цента (см. рис. 7.4)? Может, вы соучастник преступной схемы?

Впрочем, все это объясняется довольно прозаически. Число хранится в ячейке, состоящей из конечного количества битов. Если бы ячейка была бесконечно длинной, точность числа тоже была бы бесконечно большой. Рассмотрим, например, число `1.200999999999998`. Его погрешность очень маленькая, разница с точным значением только в 16-м разряде, однако девятки появились уже в 5-м разряде. Эстетически это некрасиво, и визуально создается впечатление, будто погрешность большая. Как избавиться от этого эффекта? Наиболее простое решение состоит в округлении числа при выводе на консоль. Для этого достаточно задать в методе `printf()` формат вывода, ограничивающий количество разрядов после точки. Как это делается, показано в листинге 7.7. Результат показан на рис. 7.9.

Листинг 7.7. Округление чисел при выводе на консоль

```
import static java.lang.System.out;
class UseAccount {
  public static void main(String args[]) {
    Account myAccount = new Account();
    Account yourAccount = new Account();
    myAccount.balance = 24.02;
    yourAccount.balance = 55.63;
    double myInterest = myAccount.getInterest(5.00);
    double yourInterest = yourAccount.getInterest(7.00);
    out.printf("$%4.2f\n", myInterest);
    out.printf("$%5.2f\n", myInterest);
    out.printf("$%.2f\n", myInterest);
    out.printf("$%3.2f\n", myInterest);
    out.printf("$%.2f $%.2f",myInterest,yourInterest);
  }
}
```

```
<terminated> UseAccount (5
$1,20
$ 1,20
$1,20
$1,20
$1,20 $3,89
```

Рис. 7.9. Теперь числа выглядят красиво

Обратите внимание на то, что метод `println()` в качестве десятичного разделителя выводит точку (см. рис. 7.4), а метод `printf()` — запятую, т.е. символ, установленный в качестве десятичного разделителя в региональных настройках Windows.



Важно отметить, что при работе с целыми числами подобные эффекты отсутствуют. Дважды два всегда равно четыре. Компьютер никогда не покажет, что результат равен `3.9999999999999`. Это объясняется не только тем, что при отображении целого числа компьютер округляет результат до ближайшего целого. Главная причина в том, что целое десятичное число можно точно представить целым двоичным, чего нельзя сказать о числах с плавающей точкой. В компьютере числа хранятся в двоичном виде, а для человека нужно отображать их в десятичном виде. Точность теряется при преобразовании из двоичного формата в десятичный. Например, даже простейшее, казалось бы, число `7` нельзя точно представить значением типа `double`. Однако в некоторых случаях погрешность недопустима даже в последнем разряде (например, в финансовых расчетах). Для решения этой проблемы в Java существует класс `BigDecimal`, в котором числа с плавающей точкой хранятся не в двоичном, а в десятичном виде. При использовании типа `BigDecimal` упомянутые выше эффекты никогда не возникают.

В листинге 7.7 используется удобный метод `printf()`, выводящий информацию на консоль. При вызове метода `printf()` ему нужно передать как минимум два параметра.

- ✓ **Первый параметр — *строка форматирования*.**

  Это значение типа `String`, в котором специальным образом записано, как должны выводиться значения, передаваемые через другие параметры.
- ✓ **Остальные параметры — отображаемые значения.**

Посмотрите на последний вызов `printf()` в листинге 7.7. В строке форматирования вы видите два заполнителя, которые при выводе будут заменены числами. Структура заполнителя определяет формат вывода числа. Первый заполнитель `%.2f` означает, что число должно быть выведено с двумя разрядами после десятичной точки (или запятой, если вам так больше нравится). Вместо этого заполнителя будет подставлено первое в списке число, т.е. значение переменной `myInterest`. Второй заполнитель выглядит так же, но определяет формат вывода другого числа — значения переменной `yourInterest`. Примеры форматирующих строк приведены на рис. 7.10 и 7.11. Учитывайте, что в заполнителе число перед точкой задает общее количество позиций, включая позицию для десятичной точки.



Дополнительные примеры использования строк форматирования можно найти в главах 8 и 9. Полный список параметров форматирования метода `printf()` можно найти на странице `java.util.Formatter` справочника по классам Java API.



Форматирующая строка в инструкции вызова метода `printf()` не влияет на число, хранящееся в компьютере. Она влияет только на отображение числа на экране.

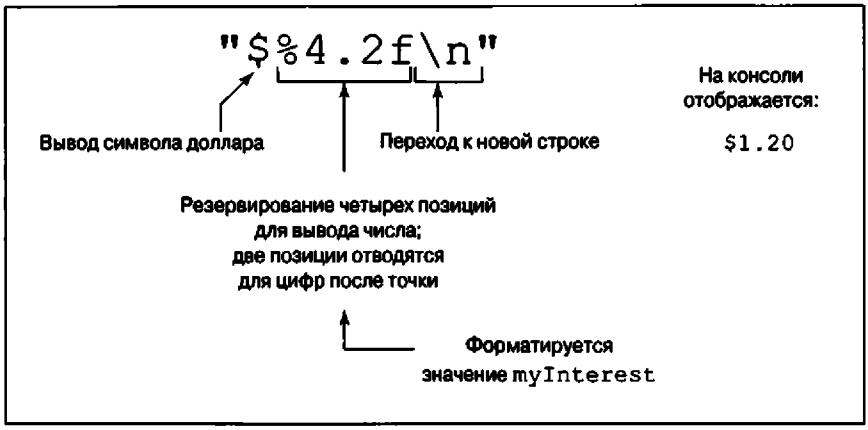


Рис. 7.10. Использование строки форматирования

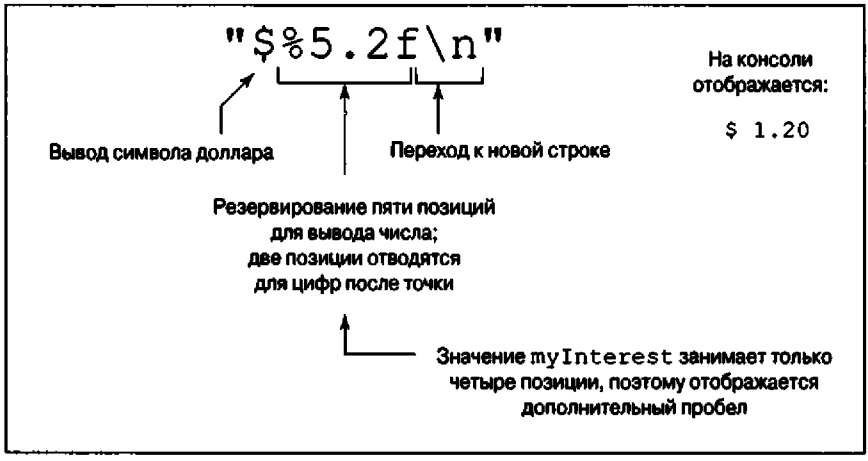


Рис. 7.11. Добавление позиций для вывода значения

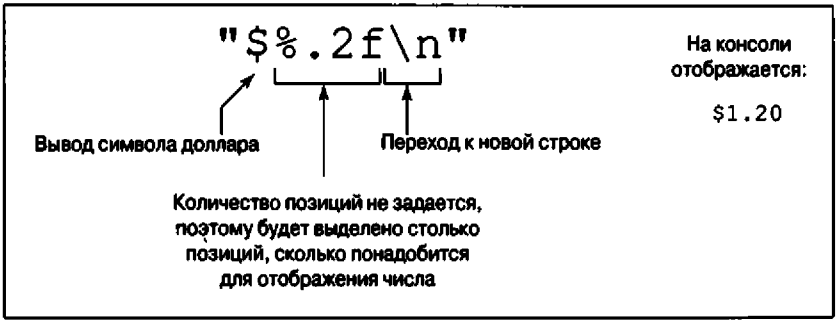


Рис. 7.12. Вывод значения без задания точного количества позиций

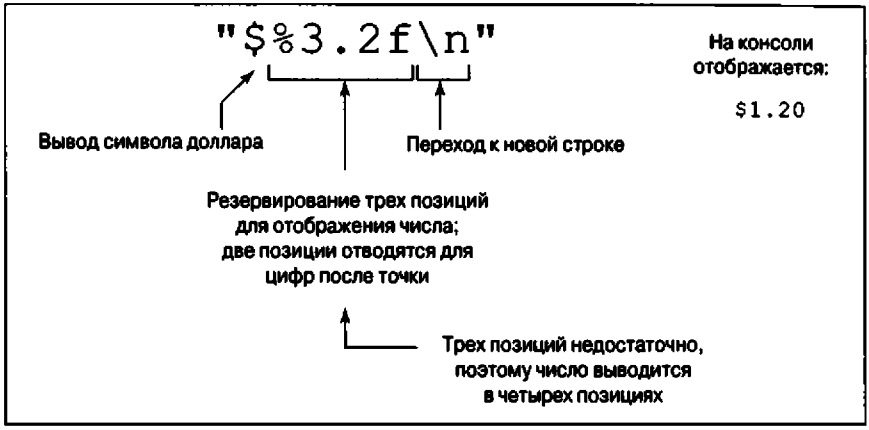


Рис. 7.13. Что будет, когда задано меньше позиций, чем нужно?

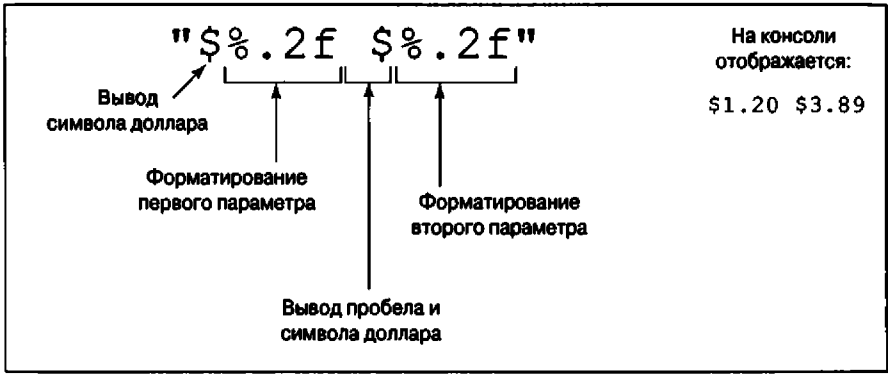


Рис. 7.14. Вывод нескольких значений

Модификаторы доступа

Положите эту книгу на стол и наденьте шляпу. Поскольку вы — читатель моей книги, я приглашаю вас на ужин.

Но есть одна проблема. В данный момент в моем кошельке недостаточно наличных денег. Вы не будете возражать, если по дороге в кафе мы остановимся возле банкомата и возьмем несколько долларов? Нам придется воспользоваться вашим счетом, потому что мой счет сейчас пустой.

К счастью, пользоваться банкоматом совсем несложно. Нужно лишь подойти к нему и ввести свой PIN-код. После ввода PIN-кода вам на выбор предлагается несколько имен переменных, которые вы можете использовать для просмотра суммы на счету (представим себе такой фантастический сценарий). Вам предложены варианты `balance324`, `myBal`, `currentBalance`, `b$`, `BALANCE`, `asj999` и `conStanTinople`. После выбора имени переменной вы должны указать ячейку памяти, в которой она хранится. Вы должны задать произвольное число от `022AAA` до `0555AA` (в шестнадцатеричном формате). Сконфигурировав программное обеспечение банкомата, вы

получаете свои деньги. Как вам нравится такая услуга? Вы не забыли захватить из дома отвертку на случай, если и она понадобится?

Правильный стиль программирования

Когда дело касается практики программирования, одно слово ценится выше всех остальных, вместе взятых. Это слово — **простота**. Когда вы работаете со сложным кодом, созданным кем-либо другим, вам очень не понравится иметь дело с плохими именами переменных, которые не намекают на их назначение, с запутанными связями между классами, которые не отражают логическую структуру задачи, с наспех вставленными "заплатами", написанными в последний момент перед сдачей программного продукта. Вы хотите иметь чистый интерфейс, помогающий вам решить свою проблему и не нагружающий вас чужими проблемами.

В приведенном выше сценарии с банкоматом главная проблема состоит в том, что его конструкция вынуждает вас заниматься чужими проблемами. Вам нужно думать о том, сколько денег вам понадобится, чтобы оплатить счет в кафе, но вам придется думать также об именах переменных и адресах ячеек памяти. Наверняка кто-то другой должен был решить эти проблемы. Клиент не должен даже подозревать об их существовании.

Все аспекты компьютерной программы должны быть простыми, верно? Не совсем. Иногда, чтобы сделать нечто простым, нужно выполнить огромную подготовительную работу. Чаще всего за простотой скрывается большая сложность. Люди, которые создавали банкомат (не описанный выше, а реально работающий), приложили много усилий для того, чтобы клиенту нужно было всего лишь ввести PIN-код и требуемую сумму денег. Простым должен быть интерфейс программы или класса, обращенный к пользователю или к другому программисту, работающему с данным классом. Однако для реализации этой простоты может понадобиться весьма сложный код.

Чтобы сделать нечто простым, нужно заранее спланировать простоту. В случае объектно-ориентированного программирования один из способов сделать использование класса простым состоит в том, чтобы запретить коду, расположенному **вне** класса, обращаться непосредственно к полям, определенным **внутри** класса. Посмотрите на код листинга 7.1. Предположим, вы работаете в компании, которая заплатила 10 млн. долларов за код класса `Account`. Компания поставила перед вами задачу создать класс `UseAccount`, в котором используется класс `Account`. Вы написали следующую инструкцию:

```
myAccount.name = "Барри Берд";
```

Но этим вы влезли слишком глубоко во "внутренности" класса `Account`. В конце концов, людям, которые пользуются банкоматом, нельзя позволять программировать переменные внутри банкомата. Они не должны использовать виртуальную клавиатуру банкомата, чтобы ввести следующую инструкцию.

```
balanceOnAccount29872865457 = balanceOnAccount29872865457 +
                              1000000.0;
```

Вместо этого они должны нажимать кнопки, чтобы решить задачу путем выполнения определенной последовательности операций. Только так можно обеспечить безопасность и простоту устройства.

Следовательно, чтобы программа была изящной и упорядоченной, нужно изменить класс `Account` (см. листинг 7.1) таким образом, чтобы устранить инструкции, обращающиеся непосредственно к полям класса.

```
myAccount.name = "Барри Берд";
out.print(yourAccount.balance);
```

Конечно, при этом возникает ряд проблем. Вы создаете код класса `UseAccount`. Если вам запрещено ссылаться на поля `myAccount.name` и `yourAccount.balance`, как вы вообще можете что-нибудь сделать? Ответ состоит в применении *методов доступа*. Их использование продемонстрировано в листингах 7.8 и 7.9.

Листинг 7.8. Сокрытие полей класса

```
public class Account {
  private String name;
  private String address;
  private double balance;

  public void setName(String n) {
    name = n;
  }

  public String getName() {
    return name;
  }

  public void setAddress(String a) {
    address = a;
  }

  public String getAddress() {
    return address;
  }

  public void setBalance(double b) {
    balance = b;
  }

  public double getBalance() {
    return balance;
  }
}
```

Листинг 7.9. Вызов методов доступа

```
import static java.lang.System.out;
class UseAccount {
  public static void main(String args[]) {
    Account myAccount = new Account();
    Account yourAccount = new Account();
    myAccount.setName("Барри");
```

```
        myAccount.setAddress("222 Inner Lane");
        myAccount.setBalance(24.02);
        yourAccount.setName("Джейн");
        yourAccount.setAddress("121 Outer Street");
        yourAccount.setBalance(55.63);
        out.print(myAccount.getName());
        out.print(" (");
        out.print(myAccount.getAddress());
        out.print(") имеет на счету $");
        out.print(myAccount.getBalance());
        out.println();
        out.print(yourAccount.getName());
        out.print(" (");
        out.print(yourAccount.getAddress());
        out.print(") имеет на счету $");
        out.print(yourAccount.getBalance());
    }
}
```

Результат выполнения кода, приведенного в листингах 7.8 и 7.9, ничем не отличается от результата листингов 7.1 и 7.2. Его можно увидеть на рис. 7.3. Тем не менее между этими программами есть большая разница. Вариант класса `Account`, приведенный в листинге 7.8, вынуждает правильно использовать поля `name`, `address` и `balance`.

Как сделать поле недоступным

Обратите внимание на слово `private` перед объявлением каждого поля в классе `Account`. Это ключевое слово языка Java, которое называется *модификатором доступа*. Его присутствие в объявлении означает, что поле объявлено как *закрытое*. Модификатор `public` создает *открытое* поле. Закрытое поле отличается от открытого тем, что никакой код, расположенный за пределами класса, не имеет к нему непосредственного доступа. Если вставить в класс `UseAccount` (см. листинг 7.9) инструкцию `myAccount.name="Барри"`, компилятор сообщит о том, что поле `name` имеет доступ `private` и к нему нельзя обращаться.

Вместо обращения к полю `myAccount.name` программист, создающий метод `UseAccount`, должен вызвать метод `myAccount.getName()` или `myAccount.setName()`. Они называются методами доступа, потому что с их помощью класс `Account` предоставляет доступ к закрытому полю `name`. Формально термин "метод доступа" в синтаксисе Java не определен. Это лишь соглашение, соблюдение которого требуется для правильного стиля программирования.



Методы доступа часто называют get-методами, или "геттерами", и set-методами, или "сеттерами". Еще одно популярное название — "бин-методы". Оно появилось в среде JavaBeans, позволяющей встраивать Java-программы в существующие графические пользовательские интерфейсы. Методы доступа используются в JavaBeans очень часто, поэтому многие программисты ассоциируют их со спецификацией JavaBeans.



Во многих интегрированных средах разработки можно не вводить код методов доступа вручную, потому что в них есть средства автоматизации этой задачи. Сначала введите объявление поля, например `private String name`. Затем выберите в строке меню команду Source⇨Generate Getter or Setter (Исходный код⇨Сгенерировать геттер или сеттер). В некоторых средах команда называется Code⇨Insert Code⇨Setter (Код⇨Вставить код⇨Сеттер). После запуска соответствующей команды среда разработки автоматически создает методы доступа и добавляет их в код программы.



Ключевые слова `public` и `private` более подробно рассматриваются в главе 13.

Вернемся еще раз к нашему банкомату. Пользователь не может ввести команду, которая непосредственно изменяет значение поля `balance`. Вместо этого он должен пройти процедуру снятия денег со счета. Эта процедура должна быть простой. Каждый разработчик подобных устройств знает, что, если сделать процедуру сложной, огромное количество клиентов не смогут пройти ее и будут очень недовольны, в результате чего у банка возникнут большие проблемы. Отсюда следуют два важных правила: сделайте невозможным то, что пользователь не должен делать, и максимально упростите то, что он должен делать.



Использование методов доступа — это всего лишь соглашение, а не догма. Можете не писать те методы доступа, которые не собираетесь использовать. Например, в листинге 7.8 можно опустить определение метода `getAddress()`. Программа останется работоспособной и приведет к тому же результату. Единственная проблема состоит в том, что опущенный метод доступа может понадобиться другому человеку, работающему с классом `Account`.



Синтаксис Java не требует, чтобы метод, присваивающий значение полю `balance`, назывался `setBalance`. Компилятор не будет возражать, если вы назовете его `tunaFish` или как-нибудь еще. Возражать будут программисты, которые привыкли к стилю кодирования, определяемому данным соглашением об именовании. Кроме того, возражения могут появиться у некоторых интегрированных сред разработки, которые позволяют создавать элементы интерфейса путем перетаскивания компонентов с панели инструментов на фрейм. Вы потеряете эту возможность, если не будете соблюдать соглашения об именовании. Интегрированные среды разработки кратко рассматриваются в главе 2.



При вызове присваивающего метода доступа (сеттера) ему нужно предоставить значение того типа, который он ожидает. Например, в листинге 7.9 при вызове `yourAccount.setBalance(55.63)` методу передается литерал типа `double`. Это полностью согласуется с тем, что в определении метода `setBalance()` в листинге 7.8 получаемый параметр имеет тип `double`. В то же время извлекающий метод доступа (геттер) обычно ничего не получает,

и его список параметров пустой. Обратите внимание на то, что получить и присвоить значение можно в одной инструкции. Например, добавить один доллар на счет можно с помощью следующей инструкции:

```
yourAccount.setBalance(yourAccount.getBalance()+1.00);
```

Проверка правил с помощью методов доступа

Одно из главных преимуществ, предоставляемых методами доступа по сравнению с непосредственным обращением к полям, состоит в том, что они позволяют задать в момент доступа выполнение дополнительных операций. Например, в методе доступа можно закодировать проверку присваиваемого значения. Вернемся к листингу 7.8 и посмотрим на определение метода `setName()`. Предположим, нужно запретить присвоение полю `name` пустой строки. Мы легко можем добавить инструкцию проверки в тело метода доступа.

```
public void setName(String n) {
  if (!n.equals("")) {
    name = n;
  }
  else printf("Ошибка! Вы забыли задать имя!");
}
```

Теперь, если программист напишет в классе `UseAccount` инструкцию `myAccount.setName("")`, на консоль будет выведено сообщение об ошибке. Более того, поскольку поле `name` закрытое, компилятор сгенерирует сообщение об ошибке в ответ на следующую инструкцию:

```
myAccount.name = "";
```

Это означает, что теперь полю `name` невозможно присвоить пустую строку никаким способом! В то же время полю `name` легко присвоить любую непустую строку, выполнив, например, инструкцию `myAccount.setName("Барри Берд")`.

Данный пример — всего лишь иллюстрация проверки значения. На практике в методы доступа часто добавляют проверку намного более сложных правил. Например, в методе доступа к полю `name` можно запрограммировать проверку, есть ли в базе данных банка клиент с таким именем.

Глава 8

Повторное использование кода

В этой главе...

- Определение класса
- Работа с файлами (небольшое отступление)
- Наследование
- Использование производных классов
- Переопределение существующих методов

В некотором царстве, в некотором государстве жила-была прекрасная принцесса. Когда ей исполнилось 25 лет (оптимальный возраст для исполнения желаний), ее отец, добрый король, подарил ей хрустальную шкатулку в позолоченной обертке. Желая побыстрее увидеть, что находится в шкатулке, принцесса разорвала позолоченную обертку (ей не хватило терпения развязать бантик) — и ахнула! В хрустальной шкатулке было то, о чем она мечтала всю свою жизнь, — компьютерная программа, которая всегда работает правильно. Программа может сделать все, чего пожелает принцесса, причем точно так, как ей нужно. Принцесса была счастлива, и поэтому счастлив был ее отец, добрый король.

Шли годы, а компьютерная программа все не совершала ни единой ошибки. Прошло много лет, изменились желания принцессы, она все больше ожидала от жизни, ее требования непрерывно повышались. Она пожелала сделать успешную карьеру и достичь ошеломляющего профессионального успеха. Программа выполняла желания ее мужа и детей, успешно управляла семейным бюджетом и оберегала мир и покой в ее душе. На протяжении всей жизни программа оставалась ее верным и надежным компаньоном.

Время шло, принцесса постарела, а вместе с ней постарела и программа. Однажды зимним вечером, сидя у камина, принцесса задала программе давно мучивший ее коварный вопрос: "Как тебе удается делать все это? Почему твои ответы всегда правильные?"

"Причина в том, — ответила программа — что я постоянно поддерживаю себя в форме и веду здоровый образ жизни. Я запускаю 20 приложений ежедневно, применяю Java для отпугивания вирусов, избегаю рекурсивных алгоритмов, не курю в хранилищах резервных копий и никогда не записываю в пул больше задач, чем могу обработать".

К сожалению, жизнь сильно отличается от сказки. Чудесной программы не существует, и вам придется самостоятельно удовлетворять все эти требования. Начнем с правильного стиля кодирования при определении классов.

Определение класса

Как хорошо было бы, если бы каждая часть программного обеспечения делала то, что нам нужно! В идеальном мире вы покупаете программу, устанавливаете ее, адаптируете к любой ситуации, легко обновляете при изменении ваших желаний и конфигурируете ее пользовательский интерфейс, как вам хочется. К сожалению, такой идеальной программы не существует. Суровая правда состоит в том, что каждая программа делает то, на что она закодирована, а не то, что вам понадобилось в данный момент.

Это одна из причин ошеломляющего успеха концепции объектно-ориентированного программирования. На протяжении многих лет компании покупали дорогостоящие программные продукты и обнаруживали, что программы делают не то, что им нужно. Что компания делает в таком случае? Иногда она пытается изменить программу. Нанятые компанией программисты регенерируют исходный код, изменяют имена переменных, перемещают подпрограммы, переделывают формулы и в конце концов делают код еще хуже. Реальность состояла в том, что, если программа не делала то, что нужно (даже если она делала нечто очень близкое к тому, что нужно), улучшить ее поведение было практически невозможно. Практически всегда наилучшим решением было выбросить дорогостоящую программу и купить новую или создать нужную программу самому.

С появлением инструментов объектно-ориентированного программирования ситуация радикально изменилась. Объектно-ориентированная программа сконструирована таким образом, чтобы ее легко было изменять. Когда код правильно структурирован, можно использовать преимущества уже встроенных средств, добавлять новые средства и переопределять средства, которые не во всем удовлетворяют ваши нужды. Лучше всего то, что изменения могут быть чистыми. Вам не придется взламывать чужие модули. Вместо этого вы сможете добавлять аккуратные компоненты, не изменяя существующую структуру кода и его внутреннюю логику. Но все это возможно, только если при создании кода он с самого начала был правильно структурирован.

Класс, определяющий служащего

Начиная писать объектно-ориентированную программу, вы сначала думаете о данных. Если вы пишете о счете, вы думаете о том, что такое счет. Кодируя реакцию на щелчок на кнопке, вы думаете о том, что такое кнопка. Когда вы пишете программу, создающую платежные чеки для служащих, вы думаете о том, что такое служащий.

Для компьютерной программы служащий — это некоторый объект, у которого есть два параметра: имя (`name`) и название занимаемой им должности (`jobTitle`). Конечно, в реальной жизни у служащего есть не только эти параметры, но пока что нам этих двух вполне достаточно. Код листинга 8.1 дает исчерпывающее определение того, что такое служащий (`Employee`) в данном смысле.

Листинг 8.1. Класс `Employee`

```
import static java.lang.System.out;
public class Employee {
  private String name;
```

```
  private String jobTitle;

  public void setName(String nameIn) {
    name = nameIn;
  }

  public String getName() {
    return name;
  }

  public void setJobTitle(String jobTitleIn) {
    jobTitle = jobTitleIn;
  }

  public String getJobTitle() {
    return jobTitle;
  }

  public void cutCheck(double amountPaid) {
    out.printf("Выплатить служащему %s ", name);
    out.printf("(%s) ***$", jobTitle);
    out.printf("%,.2f\n", amountPaid);
  }
}
```

Согласно листингу 8.1 у каждого служащего (класс `Employee`) есть семь компонентов. Два из них весьма просты — у каждого служащего есть имя (`name`) и название его должности (`jobTitle`).

Что еще есть у служащего? У каждого служащего есть четыре метода доступа, позволяющих обращаться к его имени и названию должности. Это методы `setName()`, `getName()`, `setJobTitle()` и `getJobTitle()`. Методы доступа рассматривались в главе 7.

И наконец, у каждого служащего есть метод `cutCheck()`. Идея состоит в том, что метод, выписывающий платежный чек, должен принадлежать одному из классов. Куда его лучше поместить? Большая часть информации о платежном чеке имеет отношение к конкретному служащему, поэтому лучше всего поместить метод `cutCheck()` в класс `Employee`.

Правильное использование класса

В классе `Employee` (см. листинг 8.1) нет метода `main()`, поэтому он не может служить стартовой точкой программы. Чтобы программу можно было запустить на выполнение, нужно написать класс, содержащий метод `main()` и манипулирующий экземплярами класса `Employee`. Такой класс показан в листинге 8.2. Фактически он является тестовым модулем для класса `Employee`.

Листинг 8.2. Класс, выписывающий платежные чеки для служащих

```
import java.util.Scanner;
import java.io.File;
import java.io.IOException;
```

```
class DoPayroll {
  public static void main(String args[]) throws IOException {
    Scanner diskScanner = new Scanner(new File("EmployeeInfo.txt"));
    for (int empNum = 1; empNum <= 3; empNum++) {
          payOneEmployee(diskScanner);
    }
  }

  static void payOneEmployee(Scanner aScanner) {
    Employee anEmployee = new Employee();
    anEmployee.setName(aScanner.nextLine());
    anEmployee.setJobTitle(aScanner.nextLine());
    anEmployee.cutCheck(aScanner.nextDouble());
    aScanner.nextLine();
  }
}
```



Чтобы можно было выполнить код, приведенный в листинге 8.2, на жестком диске должен быть файл `EmployeeInfo.txt`. В архиве примеров для интегрированной рабочей среды Eclipse я подготовил для вас папку `Listing08-02`, в которой находится файл `EmployeeInfo.txt`[1].

В классе `DoPayroll` определены два метода — `main()` и `payOneEmployee()`. В методе `main()` происходит три вызова метода `payOneEmployee()`. Переменная `empNum` обозначает номер служащего в списке. На каждой итерации цикла `for` метод `payOneEmployee()` извлекает данные о служащем из файла `EmployeeInfo.txt` и записывает их в поля объекта `Employee`, который представляет текущего служащего.

Метод `payOneEmployee()` вызывается три раза. При каждом вызове оператор `new` создает объект `anEmployee`, представляющий очередного служащего. Затем метод `payOneEmployee()` присваивает значения полям класса `Employee` и вызывает метод `cutCheck()`. Весь этот процесс показан на рис. 8.1.

---

[1] В качестве десятичного разделителя в файле может применяться точка или запятая (аналогично с клавиатуры можно ввести точку или запятую). Правила локализации разделителя в Windows 7 определяются в двух местах. Во-первых, в раскрывающемся списке Язык и региональные стандарты⇨Форматы⇨Формат. Можно выбрать значение *Английский* или *Русский*, задав таким образом точку или запятую. Во-вторых, если в этой же вкладке щелкнуть на кнопке Дополнительные параметры, будет открыто диалоговое окно Настройка формата. На его вкладке Числа есть текстовое поле Разделитель целой и дробной части. В нем можно задать точку или запятую. Символ разделителя (точка или запятая), заданный в одном месте, может не совпадать с заданным в другом месте. В этом случае некоторые классы Java ожидают точку, а некоторые — запятую. Аналогично некоторые методы вывода отображают точку, а некоторые — запятую. Например, метод `aScanner.nextDouble()` игнорирует текстовое поле Разделитель целой и дробной части и руководствуется значением в раскрывающемся списке Формат, хотя, казалось бы, это текстовое поле как более специфичное должно переопределять значение в раскрывающемся списке Формат. Если метод считывания (из файла или с клавиатуры) получает не тот символ десятичного разделителя, который ожидает, рабочая среда Eclipse во время выполнения генерирует ошибку Type mismatch (Несовпадение типов) и программа аварийно завершается. Чтобы программа на Java не зависела от региональных настроек Windows и других операционных систем, рекомендуется отформатировать ввод и вывод с помощью класса `Formatter` и провайдера форматов. — *Примеч. ред.*

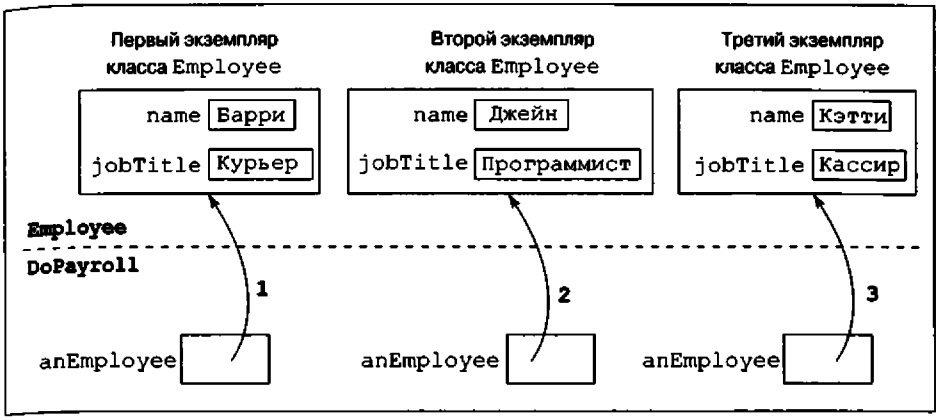


Рис. 8.1. Три вызова метода `payOneEmployee`

Создание платежного чека

В листинге 8.1 есть три вызова `printf()`. В каждом вызове используются строка форматирования (например, `"(%s) ***$"`) и отображаемое значение (например, `jobTitle`). В каждой строке форматирования есть заполнитель (например, `%s`), который определяет, в каком месте и как выводится значение.

Во втором вызове `printf()` строка форматирования содержит заполнитель `%s`, который при выводе заменяется значением переменной `jobTitle`. Согласно правилам Java фраза `%s` всегда служит заполнителем для значения типа `String`. Это означает, что при вызове метода `printf()` фраза `%s` заменяется строкой, хранящейся в переменной `jobTitle`, которая должна быть объявлена с типом `String`. Скобки, окружающие фразу `%s`, выводятся без изменений, поэтому на экране значение `jobTitle` заключено в скобки (рис. 8.2).

```
<terminated> DoPayroll_ [Java Application] D:\Program Files\Java\jre6\bin\ja
Выплатить служащему Барри (Курьер) ***$5 000,00
Выплатить служащему Джейн (Программист) ***$7 000,00
Выплатить служащему Кэтти (Кассир) ***$10 000,00
```

Рис. 8.2. Каждый получает свою зарплату

В листинге 8.1 обратите внимание на запятую в заполнителе `%,.2f`. Она сообщает компьютеру о том, что при выводе значения нужно применить *группирующий разделитель* — символ, разделяющий в длинном числе группы из трех цифр. В качестве группирующего разделителя в странах СНГ и России принято применять пробел, а в Соединенных Штатах — запятую. На рис. 8.2 в качестве группирующего разделителя служит пробел, потому что в региональных параметрах Windows на моем компьютере в качестве страны пребывания установлена Россия.



Если нужно применить группирующий разделитель, отличный от указанного в региональных параметрах Windows, воспользуйтесь классами `Formatter` и `Locale`. О них можно почитать в документации по классам Java API.

Работа с файлами (небольшое отступление)

В предыдущих главах программы считывали символы и строки с клавиатуры. Но код листинга 8.2 считывает строки из существующего файла. Файл `EmployeeInfo.txt` находится на жестком диске компьютера.

Файл `EmployeeInfo.txt` сохранен в текстовом формате и может содержать буквы, цифры и любые другие символы. В отличие от документа Word, файл в текстовом формате не может содержать элементы форматирования, такие как курсивный шрифт, маркированные и нумерованные списки и т.п. Кроме того, он не может содержать изображения, таблицы, гиперссылки и пр.

В файле в текстовом формате могут храниться только символы. В главах 5 и 6 была рассмотрена программа игры в угадывание, которая считывала символы с клавиатуры. Важное различие между считыванием символов из файла и с клавиатуры состоит в том, что во втором случае выполняется диалог пользователя с компьютером. Сначала компьютер отображает на экране приглашение ввести символ, и только после этого пользователь вводит его. При считывании из файла никакого диалога нет. Программа `DoPayroll` (см. листинг 8.2) читает символы с жесткого диска, не предлагая пользователю что-либо сделать и даже не сообщая ему об этом.

Данная глава посвящена повторному использованию кода. Однако листинг 8.2 посвящен другой важной идее, не связанной непосредственно с повторным использованием кода, — считыванию информации из файла. Поэтому в нескольких следующих разделах я немного отвлекусь от главной темы главы, чтобы совершить короткую экскурсию по файлам на жестком диске.

Хранение данных в файле

Код листинга 8.2 не может быть выполнен, если на диске нет файла `EmployeeInfo.txt`. Поэтому перед выполнением листинга 8.2 я создал небольшой текстовый файл, показанный на рис. 8.3. Результат считывания данных из него можно увидеть на рис. 8.2.



Рис. 8.3. Содержимое файла `EmployeeInfo.txt`



Чтобы листинг 8.2 был простым, необходимо, чтобы файл `EmployeeInfo.txt` заканчивался пустой строкой. Поэтому, введя последнее число `10000.00`, не забудьте нажать клавишу <Enter>, чтобы создать строку. Обратите внимание на символ курсора после числа `10000.00` на рис. 8.3. Он

свидетельствует о том, что после этого числа есть пустая строка. Если ее не будет, выполнение кода листинга 8.2 завершится аварийно, потому что объект сканера натолкнется на символ конца файла.

Копирование и вставка кода

В любом языке программирования чтение данных и манипулирование ими в файлах — довольно коварная задача. Вы добавляете несколько строк кода, чтобы сообщить компьютеру, что он должен сделать. Иногда вы можете облегчить себе жизнь, скопировав эти несколько строк из другой программы, написанной вами же или другим человеком. Возможно, эта другая программа — не что иное, как старый вариант текущей программы. Рассмотрим следующий шаблон чтения данных из файла.

```
/*
* Шаблон листинга 8.2
*/
import java.util.Scanner;
import java.io.File;
import java.io.IOException;
class SomeClassName {
  public static void main(String args[]) throws IOException {
    Scanner scannerName = new Scanner(new File("имя_файла"));
    //Сюда нужно вставить код
    scannerName.nextLine();
    scannerName.nextLine();
    scannerName.next();
    scannerName.nextLine();
    //Некоторый код, обрабатывающий данные файла
  }
}
```

Предположим, вам нужно читать данные из файла. Для начала представьте себе, что вы читаете данные с клавиатуры. Поместите, как обычно, коды `Scanner` и `next` в программу. Затем добавьте несколько дополнительных элементов из шаблона листинга 8.2.

- ✓ Добавьте два дополнительных оператора импорта — `java.io.File` и `java.io.IOException`.
- ✓ Введите в заголовок метода фразу **`throws IOException`**.
- ✓ Введите фразу **`new File("имя_файла")`** в оператор `new Scanner`.
- ✓ Создайте на жестком диске текстовый файл. Присвойте ему имя, указанное в предыдущем пункте между двойными кавычками.
- ✓ Выберите имя для сканера (`scannerName`) и применяйте его повторно при вызовах метода `nextLine()`.

Иногда копирование и вставка кода могут привести к проблемам. Возможно, вы решаете задачу, которая не вписывается в рассмотренный выше шаблон. В таком случае нужно немного изменить шаблон. Но для этого следует понимать идеи, используемые в шаблоне. Этому посвящено несколько следующих разделов.

Чтение из файла

В предыдущих главах программы считывали данные с клавиатуры с помощью компонентов `Scanner`, `System.in` и `nextDouble()`, определенных в библиотеке Java API. Программа `DoPayroll` (см. листинг 8.2) привнесла кое-что новенькое. Она читает символы не с клавиатуры, а из файла `EmployeeInfo.txt`, который находится на жестком диске компьютера.

Чтение с клавиатуры и чтение из файла — похожие задачи, поэтому при их решении применяются некоторые общие компоненты, такие как метод `Scanner.nextDouble()`. Но это все же разные задачи, поэтому для чтения данных из файла вам нужно ознакомиться с рядом дополнительных средств.

- ✓ **Объект типа `File`.** Другими словами, это экземпляр класса `File`, определенного в Java API. Для его создания используется следующий оператор:

  ```
  new File("EmployeeInfo.txt");
  ```

  Имя файла, на который ссылается объект типа `File`, заключено в двойные кавычки. Файл содержит символы (см. рис. 8.3).

  В коде программы объект типа `File` может не обладать именем, потому что его нужно всего лишь передать конструктору объекта типа `Scanner`.

  ```
  Scanner diskScanner = new Scanner(new File("EmployeeInfo.txt"));
  ```

  После этого можете забыть об объекте типа `File`, упоминаться в коде он больше не будет. Теперь файл `EmployeeInfo.txt` представлен в коде сканером `diskScanner`.



  Инструкция `new File()` создает объект типа `File`, а инструкция `new Employee()` — объект типа `Employee`. Разница между ними только в том, что класс `File` определен в Java API, а класс `Employee` — в коде разрабатываемой программы.



  Не забудьте включить в инструкцию `new Scanner()` оператор `new File()`. Если напишете `new Scanner ("EmployeeInfo.txt")` без `new File()`, компилятор не будет возражать. Вы не получите сообщения об ошибке или предупреждения вплоть до выполнения кода. Однако при выполнении программы компьютер не будет считывать данные из файла.

- ✓ **На класс `File` можно сослаться с помощью его полного имени `java.io.File`.** Тогда в листинге 8.2 можно будет опустить инструкцию импорта `java.io.File`. Но код программы будет более громоздким.

  ```
  Scanner distScanner = new Scanner(new
                                    java.io.File("EmployeeInfo.txt");
  ```

- ✓ **При использовании класса `File` необходима директива `throws IOException`.** Когда программа обращается к файлу на диске, могут возникнуть проблемы. Например, на диске может не быть файла `EmployeeInfo.txt`. Или он может быть не в той папке, в которой его ожидает увидеть программа. Или в нем может храниться совсем другая

информация, которую программа не может обработать. Чтобы обезопасить программу от всех этих потенциальных бедствий, компилятор настаивает на принятии ряда мер предосторожности.

Самое простое, что вы можете сделать — написать директиву `throws IOException`. Это означает, что метод может сгенерировать исключение (см. главу 12), которое будет обработано вызывающим методом (в случае метода `main()` — операционной системой). Однако этим вы не решаете проблему, а всего лишь приказываете компилятору замолчать. Это допустимо в простом учебном примере, но при разработке реальной программы вы должны будете закодировать перехват и обработку исключения, чтобы ваша программа была более дружественной к пользователям.



Исключения рассматриваются в главе 12.

- ✓ **На класс исключения можно сослаться с помощью его полного имени `java.io.IOException`.** Тогда в код программы можно будет не включать инструкцию импорта `java.io.IOException`. Но директива `throws` станет более длинной.

  ```
  public static void main(String args[]) throws java.io.IOException {
  ```

- ✓ **Методу `payOneEmployee()` нужно передать имя объекта типа `Scanner`.** В листинге 7.5 метод `getInterest()` получал параметр `percentageRate`. Это позволяло задавать новое значение `percentageRate` при каждом вызове метода `getInterest()`.

  То же самое происходит в листинге 8.2. Метод `payOneEmployee()` получает параметр `aScanner`. При каждом вызове метода `payOneEmployee()` ему можно передавать обновленное значение. Однако в данном случае при каждом вызове метод получает один и тот же объект типа `Scanner`.

Вам может показаться странным то, что методу `payOneEmployee()` нужен параметр, ведь в листинге 8.2 метод `payOneEmployee()` читает данные всегда из одного и того же файла. Зачем сообщать ему при каждом вызове, что это файл `EmployeeInfo.txt`?

Собственно говоря, существует много способов сообщить ему об этом. Некоторые из них не требуют передачи параметра. В примере используются два отдельных метода — `main()` и `payOneEmployee()`. Сканер создается в методе `main()` один раз, после этого он три раза используется в методе `payOneEmployee()`.

Все, что определено внутри метода, доступно только внутри метода. Переменная `diskScanner` определена внутри метода `main()`, поэтому в методе `payOneEmployee()` она невидима. Чтобы сообщить методу `payOneEmployee()` о файле на диске, переменная `diskScanner` передается ему в качестве параметра.



Области видимости переменных рассматриваются в главе 10.

Куда подевался мой файл?

Загрузив коды примеров с веб-страницы данной книги на сайте издательства, вы найдете файлы `Employee.java` и `DoPayroll.java`. В них находятся коды, приведенные в листингах 8.1 и 8.2. Кроме того, вы найдете файл `EmployeeInfo.txt`. Позаботьтесь о том, чтобы программа могла его найти, потому что в противном случае будет сгенерировано исключение `FileNotFoundException` (файл не найден).

Сделать такую ошибку довольно легко. Особенно она сбивает с толку, когда в проводнике вы видите файл, а программа сообщает, что не видит его. Не забывайте, что компьютер очень глуп. Он точно выполняет только то, что написано в инструкциях программы, и не может догадаться, что файл совсем рядом. Компьютер не читает ваши мысли. Если файл `EmployeeInfo.txt` находится не в той папке, где программа его ожидает увидеть, или в имени файла есть опечатка, программа завершается аварийно.

Если вы видите файл, а программа его не видит, то, скорее всего, он находится не в той папке жесткого диска, что нужно. Конечно, это зависит от точки зрения. Может, файл находится там, где нужно, но программа ищет его не в той папке. Увидев исключение `FileNotFoundException`, скопируйте файл в другую папку и выполните программу повторно. Скорее всего, программа ожидает увидеть файл в папке, определенной в диалоговом окне Run Configurations (Конфигурация выполнения) рабочей среды Eclipse. Папка, в которой программа запущена рабочей средой на выполнение, считается текущей. Если в коде программы указано только имя файла, программа ищет его в текущей папке. Кроме того, внимательно проверьте по буквам имя файла.

Добавление имен папок в имя файла

В коде Java можно задать точный маршрут файла на жестком диске. Выражение `new File("C:\\Users\\bburd\\workspace\\Listing08-01-02\\EmployeeInfo.txt)` выглядит ужасно, но работает безупречно.

Обратите внимание на двойные вхождения обратной косой черты (`\\`) в маршруте файла. Если вы работаете в Windows, у вас может возникнуть искушение написать `C:\Users\...` Но в Java одиночная обратная косая имеет специальное назначение. Например, в листинге 7.7 выражение `\n` означает переход к новой строке. Поэтому в строковом литерале вместо одинарной обратной косой нужно вводить двойную.



Пользователям Mac и Linux повезло: у них в качестве разделителя имен применяется не обратная, а простая косая черта (`/`), которая в Java не имеет специального назначения. Применив косую черту вместо обратной, можно сделать программу универсальной. Код `new File("/Users/bburd/workspace/Listing08-01-02/EmployeeInfo.txt")` нормально работает во всех операционных системах, включая Mac, Linux и Windows. Правда, в Windows корневой папкой может быть только текущий диск, например `C:`. Поэтому лучше всего задавать не абсолютные, а относительные маршруты с косой чертой.



Если вы знаете, в какой папке находится файл, нужный программе на Java, можете "пробраться" в нее из текущей папки. Например, код листинга 8.2 ищет файл `EmployeeInfo.txt` в папке `Listing08-01-02`. В качестве

эксперимента создайте в ней вложенную папку dataFiles и переместите в нее файл EmployeeInfo.txt. Измените листинг 8.2, записав относительный маршрут new File("dataFiles/EmployeeInfo.txt"). Программа должна найти файл во вложенной папке. Можете двигаться вверх по дереву маршрутов. Например, запись ./ означает переход вверх на один уровень вложенности, запись ../ — переход вверх на два уровня вложенности и т.д.

Построчное чтение

В листинге 8.2 метод payOneEmployee() иллюстрирует несколько полезных приемов считывания данных. У каждого сканера есть метод nextLine(). Вы можете его не использовать, но он все же присутствует. При вызове метода nextLine() он захватывает все символы до конца строки. В листинге 8.2 метод nextLine() за один вызов читает одну строку файла EmployeeInfo.txt. В другой программе метод nextLine() может считывать все символы, введенные с клавиатуры вплоть до нажатия пользователем клавиши <Enter>.

При чтении с файла метод nextLine() считывает всю строку, включая символ новой строки. Поэтому смешивание метода nextLine() с методами nextDouble() и nextInt() может привести к путанице. В листинге 8.2 они смешиваются только для иллюстрации, на практике так не делайте. Если все же их придется смешивать, тщательно анализируйте порядок работы программы и получаемый результат.

Для понимания принципа работы сканера необходимо учитывать наличие в текстовом файле невидимого символа конца строки (иногда его называют символом новой строки или символом разрыва строк). Представляйте его себе как невидимый символ между строками текста. Метод nextLine() считывает символы один за другим, пока не встретит символ конца строки. Метод считывает и его, а после этого немедленно возвращает управление вызывающему методу.

Принципы считывания конца строки представлены на рис. 8.4.

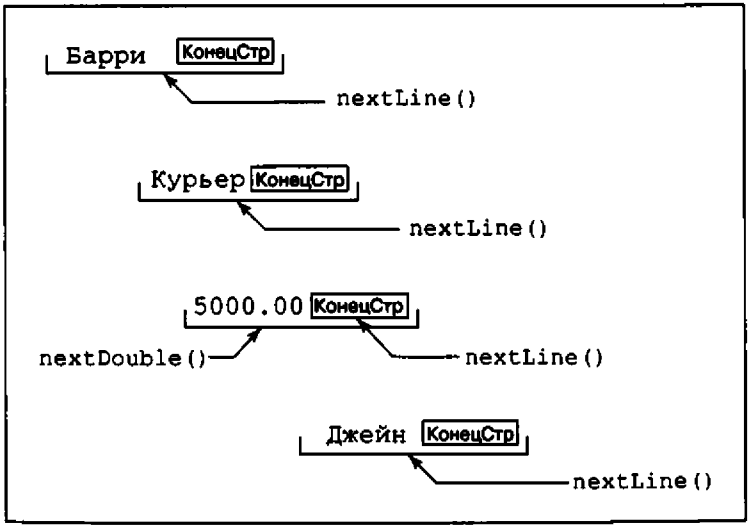


Рис. 8.4. Вызовы методов nextLine() и nextDouble()

- ✓ При первом вызове метод `nextLine()` считывает символы `Барри[Конец строки]`. Следующий вызов считывает `Курьер[Конец строки]`.
- ✓ Первый вызов метода `nextDouble()` считывает символы `5000.00[Конец строки]`. Символ конца строки вынуждает метод прекратить считывание, однако этот символ остается в буфере чтения. Окончательно его считывает следующий вызов метода `nextLine()`.
- ✓ Еще один вызов метода `nextLine()` считывает символы `Джейн[Конец строки]`.

Следовательно, после считывания числа `5000.00` метод `nextLine()` нужно вызвать два раза. Первый вызов "убирает грязь" после метода `nextDouble()`, а второй считывает строку `Джейн`. Начинающие программисты часто забывают "убрать после `nextDouble()`".



Посмотрите еще раз на содержимое файла `EmployeeInfo.txt` на рис. 8.3. Чтобы код листинга 8.2 работал правильно, после числа `10000.00` должен быть символ новой строки. На рис. 8.3 о его наличии свидетельствует курсор ввода в новой строке. Если его нет, метод `nextLine()` "упрется" в конец файла и программа завершится крахом.



Меня всегда удивляло то, что во всех языках программирования со считыванием текстовых файлов связано огромное количество разнообразных причуд. Например, первый вызов метода `nextLine()` считывает символы `Барри` и символ конца строки, однако программе он возвращает только символы `Барри`. Куда же подевался символ конца строки? Другие методы чтения из файла могут возвращать символ конца строки программе, что позволяет копировать содержимое файла в другой файл. Однако если вы попытаетесь сделать это с помощью метода `nextLine()`, то строки окажутся "слипшимися". Не забывайте об этой странности метода `nextLine()`.



Если вас смущает "грязь" после метода `nextDouble()` и необходимость "пачкать руки", чтобы убрать ее, не обвиняйте в этом разработчиков Java. Чтение и вывод текстовых строк — коварная задача в любом языке программирования. В других ситуациях возникают еще более "грязные" проблемы. Это обусловлено самой природой текстового файла. Казалось бы, что может быть проще, чем последовательность символов? Однако многообразие стандартов и форматов привело к невообразимой путанице даже в этом простейшем вопросе. В Windows конец строки обозначается двумя символами — символом конца строки и символом создания новой строки. В Mac и Linux конец старой и начало новой строки обозначаются одним символом. Но главная проблема состоит в том, откуда метод считывания может узнать о формате кодирования символов? Даже внедрение универсального стандарта Unicode не решает эту проблему. Метод считывания может проанализировать маркер BOM (Byte Order Mark — маркер последовательности байтов), но по непонятной причине получили широкое распространение файлы в формате Unicode, но без BOM!

Наследование

Предположим, в прошлом году ваша компания заплатила 10 млн. долларов компании "Берд Брейн Консалтинг" за разработку программного обеспечения для вашего отдела кадров. Купленное вами программное обеспечение — это класс `Employee.class`. Компания "Берд Брейн Консалтинг" не предоставила вам исходный код программы, чтобы вы не украли их идеи. Следовательно, у вас нет файла `Employee.java`, содержащего исходный код на Java. У вас есть только скомпилированный файл `Employee.class`, содержащий байтовый код. Вы можете читать документацию программы на странице `Employee.html`, но вы не видите инструкции на Java и не можете изменять код программы.

Прошел год. Ваша компания разрослась. Теперь в ней работают служащие двух типов: штатные (полный рабочий день) и внештатные (неполный рабочий день и работа на дому). Каждый штатный служащий еженедельно получает фиксированную зарплату. Если штатный служащий работает по ночам и в выходные, он получает благодарность администрации. Каждый внештатный служащий оплачивается по тарифу за час работы. Ваша компания вычитает некоторый процент зарплаты каждого штатного служащего на социальное страхование и для компенсации пакета льгот и преференций. Внештатные служащие не имеют льгот и преференций, поэтому с их зарплаты ничего не вычитается.

Проблема состоит в следующем. Как приспособить к новому порядку программное обеспечение, купленное в прошлом году за десять миллионов долларов? Вы инвестировали большие средства в программу, которая выписывает платежные чеки, но она не учитывает различия между штатными и внештатными служащими. У вас есть несколько вариантов решения проблемы.

- ✓ Поговорите со своим соседом, 15-летний сын которого, школьник, знает о программировании больше, чем кто-либо в вашей компании. Попросите юного гения переписать программное обеспечение так, как вам нужно.

  Однако, немного поразмыслив, вы откажетесь от этой идеи. Независимо от того, насколько "продвинут" этот парень в программировании, сложности программного обеспечения, помноженные на сложности бюрократических процедур в вашей компании, окажутся для него непосильной задачей. Вы получите программное обеспечение, полное ошибок и нестыковок. К тому же у вас нет файла `Employee.java`, который вы могли бы дать парню. Все, что у вас есть, — файл `Employee.class`, который невозможно читать и редактировать в текстовом редакторе. Кроме того, в случае успешного завершения предприятия ваш сосед будет иметь повод задирать нос. Особенно будет обижен ваш сын, который учится с сыном соседа в одном классе.
- ✓ Выбросьте файл `Employee.class` в корзину. Закажите новое программное обеспечение у другой фирмы или наймите профессионального программиста. Иными словами, скажите "гудбай" десяти миллионам долларов.
- ✓ Создайте два специальных программных интерфейса для класса `Employee`. Иными словами, создайте код, который предварительно

обрабатывает данные по зарплате по-разному для штатных и внештатных служащих. Затем интерфейсы передают результат своей работы программе за десять миллионов долларов.

Эта идея может либо оказаться плодотворной, либо привести к катастрофе. Уверены ли вы в том, что в классе `Employee` есть необходимые "зацепки", т.е. точки входа, позволяющие вашим интерфейсам встроить данные в существующий код? Не забывайте, что в этом случае существующее программное обеспечение для вас выглядит как большой черный ящик, внутренности которого вам не видны, и, кроме того, изменить что-либо в нем вы не можете. Распределить обязанности между новым кодом и старым черным ящиком будет непросто. Создав новый слой программного обеспечения, вы рискуете сделать его запутанным и неэффективным.

- ✓ Позвоните д-ру Берду и попросите его создать новую версию программного обеспечения, которая учитывала бы разницу между штатными и внештатными работниками. Д-р Берд скажет: "Мы с удовольствием поможем вам решить вашу проблему в кратчайшее время за пять миллионов долларов". Он-то хорошо знает, что у вас нет исходного кода класса `Employee`, поэтому с вас можно будет содрать завышенную цену. Конечно, сейчас существуют декомпиляторы Java, преобразующие байтовый код в исходный. Но результат преобразования мало пригоден для работы над ним вручную. Это больше похоже не на исходный код на Java, а на машинный код, написанный словами Java.
- ✓ Создайте два новых класса: `FullTimeEmployee` — для штатных служащих и `PartTimeEmployee` — для внештатных. Каждый новый класс будет *производным* от класса `Employee` и унаследует от него все его поля и методы. Даже если д-р Берд сделал поля и методы закрытыми, вы можете в байтовом коде заменить `private` на `public` и сделать их открытыми. Таким образом, при минимальной затрате усилий у вас будет программное обеспечение, учитывающее разницу между штатными и внештатными работниками.

Идеальное решение! Полученная в результате структура классов показана на рис. 8.5.

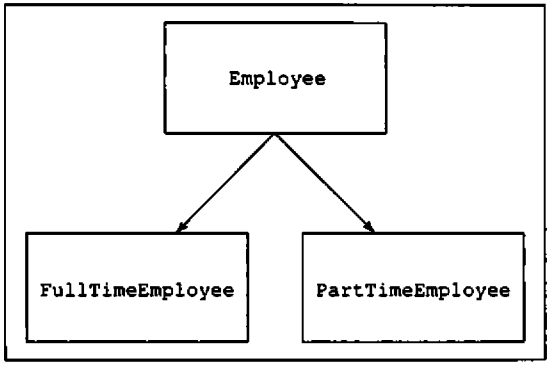


Рис. 8.5. Дерево производных классов

Создание производного класса

Определение класса Employee приведено в листинге 8.1. Мы можем использовать то, что определено в листинге 8.1, и, расширив определение, создать более специализированный класс, в котором учтены особенности штатного сотрудника. Определение такого класса, наследующего класс Employee, приведено в листинге 8.3. Фразы "наследует класс Employee" и "является производным от класса Employee" означают одно и то же. Класс Employee в этом случае называется *базовым* по отношению к *производному* классу.

Часто используются также термины "суперкласс" и "подкласс", "родительский" и "дочерний", "класс-предок" и "класс-потомок". Иногда вместо "производный" пишут "порожденный". Все это синонимы терминов "базовый" и "производный", которые обычно употребляются в теории объектно-ориентированного программирования.

Листинг 8.3. Класс FullTimeEmployee, представляющий штатного работника

```
public class FullTimeEmployee extends Employee {
  private double weeklySalary;
  private double benefitDeduction;

  public void setWeeklySalary(double weeklySalaryIn) {
    weeklySalary = weeklySalaryIn;
  }
  public double getWeeklySalary() {
    return weeklySalary;
  }
  public void setBenefitDeduction(double benefitDedIn) {
    benefitDeduction = benefitDedIn;
  }
  public double getBenefitDeduction() {
    return benefitDeduction;
  }
  public double findPaymentAmount() {
    return weeklySalary - benefitDeduction;
  }
}
```

Каждый экземпляр класса FullTimeEmployee имеет два закрытых поля: weeklySalary (оплата за неделю) и benefitDeduction (вычет за льготы). Но это не все поля, доступные в классе FullTimeEmployee. Первая строка листинга 8.3 содержит "волшебное слово" extends, которое сообщает о том, что класс FullTimeEmployee наследует существующий класс Employee. Это означает, что, кроме полей weeklySalary и benefitDeduction, каждый экземпляр класса FullTimeEmployee имеет еще и поля класса Employee, а именно — name и jobTitle. Эти два поля определены в классе Employee (см. листинг 8.1) и, значит, доступны в производном классе.

Производный класс автоматически наследует функциональность, определенную в базовом классе. Следовательно, класс FullTimeEmployee наследует поля

name и jobTitle. Кроме того, он наследует все методы, определенные в базовом классе — setName(), getName(), setJobTitle(), getJobTitle() и cutCheck().

Производный класс можно представлять себе как класс, в который перенесены все поля и методы базового класса (листинг 8.4). Но все же между листингами 8.3 и 8.4 есть небольшое отличие, которое мы рассмотрим далее.

Листинг 8.4. В класс FullTimeEmployee_Fake перенесены все поля и методы класса Employee

```
import static java.lang.System.out;

public class FullTimeEmployee_Fake {
  private String name;
  private String jobTitle;
  private double weeklySalary;
  private double benefitDeduction;

  public void setName(String nameIn) {
    name = nameIn;
  }

  public String getName() {
    return name;
  }

  public void setJobTitle(String jobTitleIn) {
    jobTitle = jobTitleIn;
  }

  public String getJobTitle() {
    return jobTitle;
  }

  public void setWeeklySalary(double weeklySalaryIn) {
    weeklySalary = weeklySalaryIn;
  }

  public double getWeeklySalary() {
    return weeklySalary;
  }

  public void setBenefitDeduction(double benefitDedIn) {
    benefitDeduction = benefitDedIn;
  }

  public double getBenefitDeduction() {
    return benefitDeduction;
  }

  public double findPaymentAmount() {
    return weeklySalary - benefitDeduction;
  }
```

```
  public void cutCheck(double amountPaid) {
    out.printf("Выплатить служащему %s ", name);
    out.printf("(%s) ***$", jobTitle);
    out.printf("%,.2f\n", amountPaid);
  }
}
```



В листинге 8.4 приведен класс `FullTimeEmployee_Fake`, содержащий все поля и методы классов `Employee` и `FullTimeEmployee`. В листинге 8.3 приведен класс `FullTimeEmployee`, наследующий класс `Employee`. Казалось бы, классы `FullTimeEmployee` и `FullTimeEmployee_Fake` должны быть эквивалентными. Однако между ними есть существенная разница. В производном классе `FullTimeEmployee` нельзя непосредственно ссылаться на закрытые поля и методы базового класса. Сослаться на них можно только посредством методов доступа, определенных в базовом классе. В листинге 8.3 инструкция `setName("Джейн")` допустима, а инструкция `name="Джейн"` — нет. В то же время в листинге 8.4 инструкция `name="Джейн"` допустима.



Чтобы создать класс, производный от класса `Employee`, иметь его исходный код `Employee.java` не обязательно. Достаточно иметь скомпилированный файл `Employee.class`.

Точно так же можно создать другой производный класс (`PartTimeEmployee`), обрабатывающий данные внештатных работников (листинг 8.5).

Листинг 8.5. Класс `PartTimeEmployee`

```
public class PartTimeEmployee extends Employee {
  private double hourlyRate;

  public void setHourlyRate(double rateIn) {
    hourlyRate = rateIn;
  }

  public double getHourlyRate() {
    return hourlyRate;
  }

  public double findPaymentAmount(int hours) {
    return hourlyRate * hours;
  }
}
```

В отличие от класса `FullTimeEmployee`, обрабатывающего данные штатных сотрудников, в нем не вычисляются еженедельная оплата и отчисления на преференции, но зато есть поле `hourlyRate`, содержащее почасовой тариф оплаты. Можно также добавить поле `numberOfHoursWorked` (количество отработанных часов) и другие параметры для каждого внештатного работника. Чтобы не усложнять пример, я решил ничего этого не добавлять.

Использование производных классов

В предыдущем разделе рассматривается создание производных классов. Но производный класс бесполезен, если не написать код, в котором он используется.

Давайте классифицируем всех окружающих нас людей (в том числе себя) на три типа: Ф, П и Т. Поскольку я — автор книги, я могу строить сюжет, как мне заблагорассудится. Я могу даже сказать громко и в присутствии других людей: "Смотрите! Это человек типа Т!"

- ✓ **Тип Ф — фундаменталист.** Он говорит: "Покажите мне программу, которая иллюстрирует рассматриваемый фундаментальный принцип программирования в наиболее чистом виде". Фундаменталиста не интересуют украшательства, они появятся позже или вообще никогда не появятся. Если вы принадлежите к типу Ф, вы захотите увидеть программу, в которой используются производные классы `FullTimeEmployer` и `PartTimeEmployee`. Затем я должен буду убрать эту программу, чтобы она больше никогда не попадалась вам на глаза, но чтобы у вас осталось впечатление, будто работа сделана.
- ✓ **Тип П — практик.** Хочет увидеть практическое применение рассматриваемой концепции. Ему нужно видеть идею в определенном контексте, иначе она быстро улетучится из головы. Он говорит: "Покажите мне программу, которая демонстрирует полезность классов `FullTimeEmployee` и `PartTimeEmployee`. Мне не нужны надуманные абстракции, я хочу видеть реальные, практически полезные примеры".
- ✓ **Тип Т — тестировщик.** Ему в первую очередь не терпится запустить коды производных классов `FullTimeEmployee` и `PartTimeEmployee`. Работающую программу он понимает лучше, чем разговоры о ней. Тестирование означает прохождение кодом определенных этапов: создание входных данных, запуск тестирующей программы в определенной среде, анализ полученных результатов работы программы. Программа должна быть протестирована в разных ситуациях, в том числе при неправильных или нереалистичных входных данных. Более того, человек типа Т хочет использовать стандартные, хорошо знакомые ему тестирующие коды на основе инструментов JUnit, чтобы он мог легко и быстро интерпретировать поведение производных классов `FullTimeEmployee` и `PartTimeEmployee`.

Листинг 8.6 предназначен для людей типа Ф. Код листинга простой и чистый, благодаря чему его можно рекомендовать для чтения на ночь. В нем показан голый каркас, состоящий всего лишь из нескольких инструкций, в которых используются классы `FullTimeEmployee` и `PartTimeEmployee`. Результат показан на рис. 8.6.

Листинг 8.6. Использование производных классов `FullTimeEmployee` и `PartTimeEmployee`

```
class DoPayrollTypeF {
  public static void main(String args[]) {
    FullTimeEmployee ftEmployee = new FullTimeEmployee();
```

```
    ftEmployee.setName("Барри");
    ftEmployee.setJobTitle("Директор");
    ftEmployee.setWeeklySalary(5000.00);
    ftEmployee.setBenefitDeduction(500.00);
    ftEmployee.cutCheck(ftEmployee.findPaymentAmount());
    System.out.println();

    PartTimeEmployee ptEmployee = new PartTimeEmployee();
    ptEmployee.setName("Стив");
    ptEmployee.setJobTitle("Водитель");
    ptEmployee.setHourlyRate(7.53);
    ptEmployee.cutCheck(ptEmployee.findPaymentAmount(10));
  }
}
```

```
<terminated> DoPayrollTypeF [Java Application] D:\Program Files\Java\jre7\b
Выплатить служащему Барри (Директор) ***$4 500,00

Выплатить служащему Стив (Водитель) ***$75,30
```

Рис. 8.6. Результат выполнения листинга 8.6

Чтобы понять листинг 8.6, нужно краем глаза посматривать на коды трех классов — `Employee`, `FullTimeEmployee` и `PartTimeEmployee`, приведенные в листингах 8.1, 8.3 и 8.5.

Первая часть листинга 8.6 имеет дело со штатным служащим. Обратите внимание на то, как много методов доступно через переменную `ftEmployee`. Например, можно вызвать метод `ftEmployee.setWeeklySalary()`, определенный в классе `FullTimeEmployee`. Можно также вызвать метод `ft.Employee.setName()`, определенный в классе `Employee`.

Метод `cutCheck()` определен в классе `Employee`, поэтому его можно вызвать через объект `ftEmployee`. Через этот же объект можно вызвать метод `findPaymentAmount()`, определенный в классе `FullTimeEmployee`.

Соответствие типов

Посмотрите еще раз на первую половину листинга 8.6. Обратите внимание на предпоследнюю инструкцию, в которой вызываются методы `cutCheck()` и `findPaymentAmount()`.

- ✓ Метод `ftEmployee.findPaymentAmount()` вызывается с пустым списком параметров (листинг 8.6). Это правильно, поскольку в определении метода `findPaymentAmount()`, приведенном в листинге 8.3, список параметров также пуст.
- ✓ Согласно определению метода `findPaymentAmount()` он возвращает значение типа `double` (см. листинг 8.3).
- ✓ Значение типа `double`, возвращенное методом `ftEmployee.findPaymentAmount()`, передается методу `ftEmployee.cutCheck()`, как показано в листинге 8.6. Это правильно, потому что согласно определению метода `cutCheck()`, приведенному в листинге 8.1, он принимает один параметр типа `double`.

Типы передаваемых методу и ожидаемых методом значений должны соответствовать друг другу, как детали пазла (рис. 8.7).

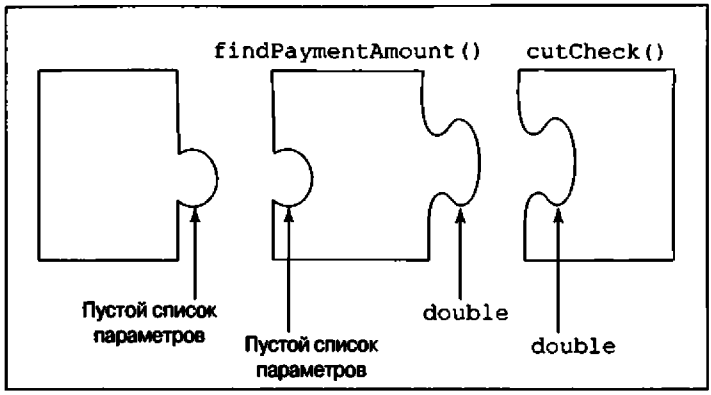


Рис. 8.7. Соответствие типов передаваемых и ожидаемых параметров



В списке параметров всегда передавайте методу значения тех типов, которые он ожидает.

Использование класса `PartTimeEmployee`

Во второй половине листинга 8.6 компьютер создает объект типа `PartTimeEmployee`, который может делать часть того, что делает объект типа `FullTimeEmployee`. Но в классе `PartTimeEmployee` нет методов `setWeeklySalary()` и `setBenefitDeduction()`. Тем не менее класс `PartTimeEmployee` может делать кое-что такое, чего не может делать класс `FullTimeEmployee`, потому что он имеет метод `setHourlyRate()`, определение которого приведено в листинге 8.5. Вызов метода `setHourlyRate()` происходит в предпоследней инструкции листинга 8.6.

Но наиболее интересна последняя инструкция листинга 8.6. В ней код передает число `10` (количество отработанных часов) методу `findPaymentAmount()`. Сравните его вызов через объект `ptEmployee` с предыдущим вызовом через объект `ftEmployee` в первой половине листинга 8.6. В классах `FullTimeEmployee` и `PartTimeEmployee` есть два разных метода с одинаковым именем `findPaymentAmount()`. Отличаются они только списком параметров.

- ✓ В классе `FullTimeEmployee` метод `findPaymentAmount()` не принимает ни одного параметра (см. листинг 8.3).
- ✓ В классе `PartTimeEmployee` метод `findPaymentAmount()` принимает один параметр типа `int` (см. листинг 8.5).

Чего и следовало ожидать. Метод `findPaymentAmount()` вычисляет сумму оплаты, а для штатного и внештатного работников она вычисляется по-разному. Для внештатного работника оплата изменяется каждую неделю в зависимости от количества отработанных часов, которое нужно передать методу. Для штатного работника оплата не зависит от количества отработанных часов. Поэтому в первом случае методу `findPaymentAmount()` передается число, а во втором случае ничего не передается.

Переопределение существующих методов

Как вам это нравится?! Конкурирующая фирма предложила своим внештатным работникам двойной тариф за работу во внеурочное время! Часть наших внештатных работников узнала об этом и потребовала того же. Мы попытаемся объяснить им, что, если так будет продолжаться, фирма разорится и они вообще не получат оплаты.

Но пока что у нас есть две категории внештатных работников: получающие за внеурочное время по двойному и обычному тарифам. Соответственно, нужно изменить программное обеспечение, начисляющее выплаты. У нас есть два варианта.

- ✓ Мы можем изменить код класса `PartTimeEmployee` и надеяться на лучшее. Но это не очень хорошая идея, потому что данная ситуация должна чем-то закончиться. Или все будут получать по двойному тарифу, или все по обычному, или фирма разорится. Чтобы не сглазить, не будем рассматривать последний вариант. А первые два означают, что, когда ситуация стабилизируется, нам опять придется менять код класса `PartTimeEmployee`. Естественно, при каждом изменении в код могут прокрасться ошибки. Как бы сделать так, чтобы решить задачу, не изменяя код стабильного класса?
- ✓ Это можно сделать, создав еще один производный класс, для которого базовым будет класс `PartTimeEmployee`. Однако существующий класс `PartTimeEmployee` уже имеет метод `findPaymentAmount()`. Означает ли это, что нам нужен некий хитроумный способ обойти существующий метод `findPaymentAmount()` для каждого работника с двойным тарифом?

  В этот момент можете считать, что вам повезло, что вы программируете на объектно-ориентированном языке Java. С его помощью можно создать производный класс, который переопределяет функциональность базового класса (листинг 8.7).

Листинг 8.7. Класс, производный от производного

```
public class PartTimeWithOver extends PartTimeEmployee {
  @Override
  public double findPaymentAmount(int hours) {
    if(hours <= 40) {
      return getHourlyRate() * hours;
    } else {
      return getHourlyRate() * 40 + getHourlyRate() * 2 *
            (hours - 40);
    }
  }
}
```

Взаимоотношения классов показаны на рис. 8.8. Класс `PartTimeEmployee`, производный по отношению к `Employee`, является базовым по отношению к `PartTimeWithOver`. На практике в объектно-ориентированных программах такие цепочки — обычное дело. Часто они намного длиннее, чем данная цепочка из двух звеньев.

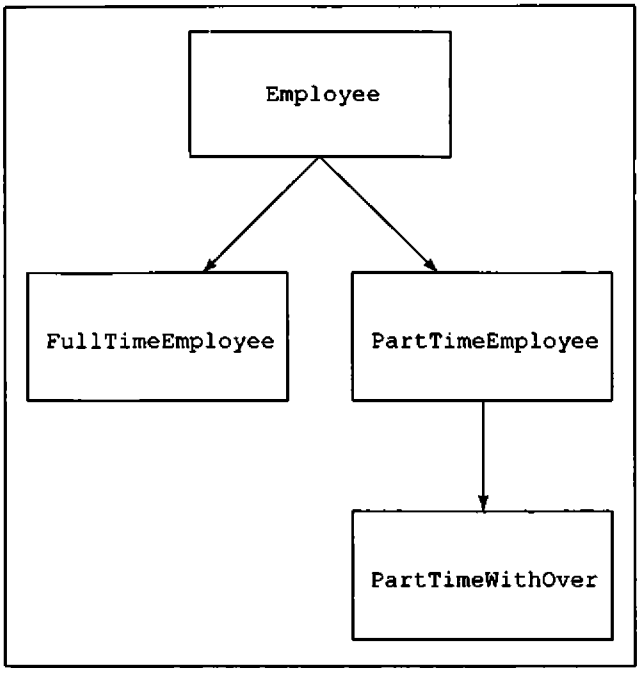


Рис. 8.8. Дерево наследования классов

Класс `PartTimeWithOver` наследует класс `PartTimeEmployee`, но не совсем так, как в предыдущих примерах. В данном случае класс `PartTimeWithOver` сам выбирает, что он хочет или не хочет унаследовать от базового класса. В классе `PartTimeWithOver` есть собственное определение метода `findPaymentAmount()`, поэтому он не наследует метод `findPaymentAmount()` базового класса (рис. 8.9).

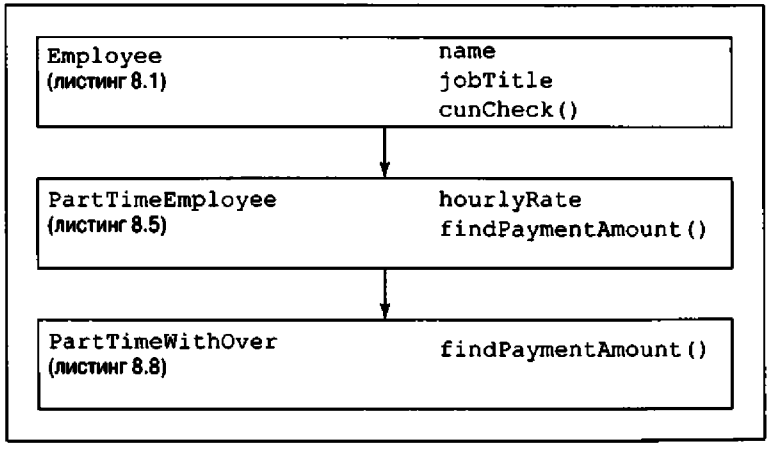


Рис. 8.9. Метод `findPaymentAmount()` не наследуется, потому что еще раз определяется в производном классе

Если создать объект класса `PartTimeWithOver`, в нем будут члены `name`, `jobTitle` и `cutCheck()`, определенные в классе `Employee`, поле `hourlyRate`, определенное в классе `PartTimeEmployee`, и метод `findPaymentAmount()`, определенный в классе `PartTimeWithOver`. С миру по нитке, классу — функциональность.

Аннотации

Инструкция `@Override` (переопределить) в листинге 8.7 — это *аннотация*. Аннотация сообщает компилятору Java нечто о коде. В частности, аннотация `@Override` в листинге 8.7 говорит компилятору о том, что он должен проверить, есть ли в базовом классе метод с такими же именем и параметрами. Если такового нет, компилятор должен вернуть сообщение об ошибке.

Следовательно, если ошибочно ввести

```
public double findPaymentAmount(double hours) {
```

вместо `int hours`, как в листингах 8.5 и 8.7, компилятор напомнит, что метод `findPaymentAmount()` ничего не переопределяет.

В Java есть много других аннотаций, например `@Deprecated` (устаревший) или `@SuppressWarning` (подавить предупреждения). Аннотация `@SuppressWarning` рассматривается в главе 9.



Добавлять аннотации в код Java необязательно. Если удалить слово `@Override` из листинга 8.7, код будет выполняться так же, как и с ним. Но при использовании аннотации уменьшается вероятность ошибки, потому что компилятор проверит, делаете ли вы то, что намеревались сделать (т.е. действительно ли вы переопределяете метод базового класса). Однако некоторые аннотации не такие уж необязательные. Удалить некоторые типы аннотаций можно, только заменив их соответствующими кодами на Java.

Вызов методов базовых и производных классов

Использование переопределенного метода иллюстрируется в листинге 8.8. Результат его выполнения показан на рис. 8.10.

Листинг 8.8. Тестирование кода, приведенного в листинге 8.7

```
class DoPayrollTypeF {

  public static void main(String args[]) {

    FullTimeEmployee ftEmployee = new FullTimeEmployee();

    ftEmployee.setName("Барри");
    ftEmployee.setJobTitle("Директор");
    ftEmployee.setWeeklySalary(5000.00);
    ftEmployee.setBenefitDeduction(500.00);
    ftEmployee.cutCheck(ftEmployee.findPaymentAmount());

    PartTimeEmployee ptEmployee = new PartTimeEmployee();
```

```
        ptEmployee.setName("Крис");
        ptEmployee.setJobTitle("Программист");
        ptEmployee.setHourlyRate(7.53);
        ptEmployee.cutCheck(ptEmployee.findPaymentAmount(50));

        PartTimeWithOver ptoEmployee = new PartTimeWithOver();

        ptoEmployee.setName("Стив");
        ptoEmployee.setJobTitle("Водитель");
        ptoEmployee.setHourlyRate(7.53);
        ptoEmployee.cutCheck(ptoEmployee.findPaymentAmount(50));
    }
}
```

```
<terminated> DoPayrollTypeF (1) [Java Application] D:\Program Files\Java\jre7\b
Выплатить служащему Барри (Директор) ***$4 500,00
Выплатить служащему Крис (Программист) ***$376,50
Выплатить служащему Стив (Водитель) ***$451,80
```

Рис. 8.10. Выполнение листинга 8.8

Код в листинге 8.8 выписывает платежные чеки трем служащим. Первый служащий штатный, второй — внештатный. Третий — также внештатный, но, в отличие от второго, по отношению к нему применяется измененное правило оплаты сверхурочных работ.

Благодаря системе производных классов все три типа служащих сосуществуют в листинге 8.8. Производный класс `PartTimeEmployee` сам является базовым по отношению к `PartTimeWithOver`, но это не значит, что нельзя создать его экземпляр. В листинге 8.8 есть три вызова метода `findPaymentAmount()`, причем каждый раз выполняется другая версия метода, определенная в другом классе.

- ✓ При первом вызове `ftEmployee.findPaymentAmount()` переменная `ftEmployee` содержит объект типа `FullTimeEmployee`, поэтому вызывается метод, приведенный в листинге 8.3.
- ✓ При втором вызове `ptEmployee.findPaymentAmount()` переменная `ptEmployee` содержит объект типа `PartTimeEmployee`. Поэтому вызывается метод, приведенный в листинге 8.5.
- ✓ При третьем вызове `ptoEmployee.findPaymentAmount()` переменная `ptoEmployee` содержит объект типа `PartTimeWithOver`, поэтому вызывается метод, приведенный в листинге 8.7.

Код получился чистый и элегантный. Нам не пришлось изменять код класса `PartTimeEmployee`. Поэтому, когда ситуация с оплатой сверхурочных утрясется и все внештатники станут одинаковыми, нам достаточно будет отказаться от класса `PartTimeWithOver`, а все остальное останется неизменным.

Глава 9

Конструкторы

В этой главе...

- Определение конструктора
- Конструктор базового класса в производном классе
- Конструктор может не только заполнять поля

Кому: мисс Дженни Берд,
121001, Нью-Джерси, ул. Школьная, 15.

Дорогая мисс Берд!

В ответ на Ваше письмо от 21 июня сообщаю Вам, что объекты не возникают спонтанно из ничего. Должен признаться, что мне никогда не приходилось видеть, как создаются объекты. И более того, никто в нашем офисе не присутствовал при этом. Однако я все же твердо убежден в том, что эти чрезвычайно интересные и полезные вещи создаются в некотором методе, который мы договорились называть конструктором. Мы, весь коллектив ClassesAndObjects.com, единодушно поддерживаем мнение об искусственном происхождении объектов, превалирующее в научных кругах и частном секторе. Нет никакого сомнения в том, что спонтанное создание объектов пагубно сказалось бы на стабильности текущей ситуации в экономике.

Позвольте заверить Вас в том, что мы примем все необходимые меры для обеспечения Вашей безопасности и благополучия Вашего бизнеса, поскольку Вы являетесь нашим постоянным клиентом. Если у Вас появятся любые вопросы, пожалуйста, немедленно обращайтесь к нам — в службу поддержки. Связаться с главным менеджером службы можно, посетив веб-сайт компании.

Еще раз благодарю за то, что Вы обратились к нам, и надеюсь на дальнейшее плодотворное сотрудничество.

Искренне Ваш,
м-р Брикенчикен

Определение конструктора

Приведенная ниже инструкция создает объект.

```
Account myAccount = new Account();
```

Я знаю, что она работает, потому что я взял ее из примера главы 7. В той же главе 7 сказано: "Когда компьютер выполняет оператор `new Account()`, он создает объект путем вызова конструктора класса `Account`". Теперь пришло время объяснить, что это означает.

Когда вы просите компьютер создать объект, компьютер выполняет некоторые действия. В первую очередь он находит в памяти место для хранения информации о новом объекте. Если у объекта есть поля, им нужно место для хранения значений, которые им, смеем надеяться, будут присвоены (иначе зачем они нужны?).



Поля рассматриваются в главе 7.

Отсюда следует интересный вопрос: "Когда вы просите компьютер создать объект, можете ли вы управлять тем, что помещается в поля объекта?" Кроме того, нельзя ли сделать в этот момент еще что-нибудь, кроме заполнения полей объекта? Создание объекта — такой важный и торжественный момент, что при этом компьютер обычно выполняет много операций, не связанных непосредственно с созданием объекта. Например, при создании кнопки компьютер должен переупорядочить или изменить размеры других элементов пользовательского интерфейса, чтобы кнопка не "налезла" на существующие элементы.

Создание объекта и операции, выполняемые при этом, сконцентрированы в методе, который называется *конструктором*. Кроме создания объекта, конструктор обычно выполняет ряд задач по его инициализации, конфигурированию и т.п.

Что такое температура

"Доброе утро! Вас приветствует Служба новостей объекта! Сейчас у нас довольно приятная температура — 73° по Фаренгейту".

Каждая температура состоит из двух частей: числа и температурной шкалы. Число — это значение типа `double`, например `32.0` или `70.52`. Но что такое "температурная шкала"? Это строка символов, такая как `Fahrenheit` или `Celsius`? Не совсем. Например, температурной шкалы `Quelploof` не существует, и программа, которая отображает "73 градуса по Куэлплуфу", — это плохая программа. Как же ограничить доступные нам температурные шкалы небольшим количеством шкал, которые можно использовать? Очевидное решение состоит в том, чтобы привести их список. В Java такой список называется *перечислением* и создается с помощью специального ключевого слова `enum`.

Что такое температурная шкала

В Java есть много средств группирования различных сущностей. В главе 11 рассматривается группирование объектов с помощью массивов и коллекций. В данной главе мы рассмотрим группирование с помощью типов `enum` (по-русски языке читается как "энум" с ударением на втором слоге).

Создать сложный тип `enum` непросто, но для создания простого `enum` достаточно написать ряд идентификаторов в фигурных скобках. Пример определения типа `enum` (говорят также "типа перечисления") приведен в листинге 9.1. Имя типа — `TempScale`. Тип `TempScale` содержит перечисление доступных температурных шкал.

Листинг 9.1. Тип перечисления TempScale

```
public enum TempScale {
   CELSIUS, FAHRENHEIT, KELVIN, RANKINE,
   NEWTON, DELISLE, REAUMUR, ROMER, LEIDEN
};
```

При определении типа перечисления происходят два важных события.

- ✓ **Вы создаете значения.** Точно так же, как 13 или 151 являются значениями типа int, слова CELSIUS и FAHRENHEIT являются значениями типа TempScale.
- ✓ **Вы можете объявить переменную, ссылающуюся на значение перечисления.** В листинге 9.2 объявлены поле number типа double и поле scale типа TempScale.

  Переменная типа TempScale может иметь значения CELSIUS, FAHRENHEIT, KELVIN и т.д. Поэтому переменной scale можно присвоить значение TempScale.FAHRENHEIT.



Тип перечисления — это "замаскированный" класс Java. Поэтому объявление типа перечисления следует поместить в отдельный файл. Например, код листинга 9.1 должен находиться в файле TempScale.java.

Так что же такое температура?

Итак, температура состоит из числа и температурной шкалы. Если привести только число, то будет непонятно: это температура по Цельсию, Кельвину, Фаренгейту или Реомюру? Теперь, зная, что такое тип перечисления, мы можем определить температуру как класс, содержащий число и температурную шкалу.

Листинг 9.2. Класс Temperature

```
public class Temperature {
    private double number;
    private TempScale scale;

    public Temperature() {
        number = 0.0;
        scale = TempScale.FAHRENHEIT;
    }

    public Temperature(double number) {
        this.number = number;
        scale = TempScale.FAHRENHEIT;
    }

    public Temperature(TempScale scale) {
        number = 0.0;
        this.scale = scale;
    }
```

```
    public Temperature(double number, TempScale scale) {
        this.number = number;
        this.scale = scale;
    }

    public void setNumber(double number) {
        this.number = number;
    }

    public double getNumber() {
        return number;
    }

    public void setScale(TempScale scale) {
        this.scale = scale;
    }

    public TempScale getScale() {
        return scale;
    }
}
```

Код в листинге 9.2 содержит обычные методы доступа к полям `number` (число) и `scale` (шкала).



Методы доступа рассматриваются в главе 7.

Класс `Temperature` (листинг 9.2) содержит четыре члена, очень похожих на методы. Каждый из них носит имя `Temperature`, совпадающее с именем класса. Ни один из них не имеет типа возвращаемого значения, даже типа `void`. Вам не показалось это странным? Тип `void` означает, что метод не возвращает ничего. А что означает его отсутствие? Что не возвращает совсем ничего? Чем же тогда отличается "ничего" от "совсем ничего"?

Каждый из этих членов класса, похожих на методы, называется *конструктором*. В сущности, конструктор можно считать методом, но специального вида. Как и в методе, в конструкторе выполняются присутствующие в нем инструкции, но в отличие от метода конструктор нельзя вызвать в произвольной точке программы. Конструктор имеет специальное назначение: он вызывается для создания объекта.



При создании каждого нового объекта компьютер выполняет инструкции, расположенные внутри конструктора.



В первых строках листингов 9.1 и 9.2 можно опустить модификаторы `public`. Но тогда другие программы на Java могут не получить доступа к средствам классов `TempScale` и `Temperature`. Впрочем, о примерах данной главы можете не беспокоиться. Все они работоспособны в обоих случаях: как при наличии, так и при отсутствии модификатора `public`. Из главы 13 вы узнаете, на какие программы может повлиять его отсутствие.

В частности, если модификатор `public` присутствует в первой строке листинга 9.1, значит, код листинга 9.1 **обязательно** должен находиться в отдельном файле `TempScale.java`. Аналогично, если слово `public` есть в первой строке листинга 9.2, код должен находиться в отдельном файле `Temperature.java`.

Что можно сделать с температурой

В листинге 9.3 иллюстрируются идеи, высказанные в предыдущих разделах. В нем вызываются конструкторы, объявленные в листинге 9.2. Результат работы листинга 9.3 показан на рис. 9.1.

Листинг 9.3. Использование класса `Temperature`

```
import static java.lang.System.out;

class UseTemperature {

   public static void main(String args[]) {
       final String format = "%5.2f градусов по %s\n";

       Temperature temp = new Temperature();
       temp.setNumber(70.0);
       temp.setScale(TempScale.FAHRENHEIT);
       out.printf(format, temp.getNumber(), temp.getScale());

       temp = new Temperature(32.0);
       out.printf(format, temp.getNumber(), temp.getScale());

       temp = new Temperature(TempScale.CELSIUS);
       out.printf(format, temp.getNumber(), temp.getScale());

       temp = new Temperature(2.73, TempScale.KELVIN);
       out.printf(format, temp.getNumber(), temp.getScale());
   }
}
```

```
<terminated> UseTemperature [Java Applicatior
70,00 градусов по FAHRENHEIT
32,00 градусов по FAHRENHEIT
 0,00 градусов по CELSIUS
 2,73 градусов по KELVIN
```

Рис. 9.1. Результат выполнения листинга 9.3

В листинге 9.3 четыре раза встречается инструкция следующего вида.

```
temp = new Temperature(...);
```

Это вызовы конструкторов, определенных в листинге 9.2. Код листинга 9.3 создает четыре объекта класса `Temperature`, причем каждый раз вызывается другой конструктор, а старый объект, хранящийся в ячейке памяти `temp`, уничтожается.

Конструктор, как и метод, получает параметры от вызывающего кода посредством списка параметров, разделенных запятыми.

Как и при вызове методов, количество и типы параметров должны быть одни и те же в определении конструктора и в операторе вызова конструктора. При последнем вызове конструктору передаются два параметра: `2.73` и `TempScale.KELVIN`. Оператор вызова конструктора имеет следующий вид:

```
new Temperature(2.73, TempScale.KELVIN)
```

Вот почему компьютер ищет конструктор, имеющий два параметра указанных типов. В данном случае компьютер находит конструктор, объявленный следующим образом:

```
public Temperature(double number, TempScale scale)
```

В нем, как и положено согласно инструкции вызова, есть два параметра: первый — типа `double` и второй — типа `TempScale`. Если компьютер не найдет конструктор, у которого количество и типы параметров совпадают с указанными при вызове, программа будет завершена аварийно. В большинстве случаев об ошибке сообщает компилятор.

В предыдущих главах мы уже встречали методы с одинаковыми именами, но с разным количеством параметров. Например, метод `printf()` в первой позиции принимает форматирующую строку, а в остальных позициях — произвольное количество параметров. Однако при разном количестве параметров вызывается один и тот же метод `printf()`, а при создании объекта вызываются разные конструкторы. Эта особенность конструкторов присуща и обычным методам, и ее можно использовать в любом классе. Не забывайте о том, что можно создавать методы с одинаковыми именами, но с разными количеством и типами передаваемых параметров. Компилятор всегда найдет нужный метод.

Поиск нужного конструктора

Когда компьютер выполняет один из операторов `new Temperature()` в листинге 9.3, он должен решить, какой конструктор выбрать в листинге 9.2. Компьютер решает это, просмотрев список параметров в инструкции вызова (т.е. в операторе `new`). Предположим, объект создается следующей инструкцией (см. листинг 9.3).

```
temp = new Temperature(32.0);
```

Компьютер рассуждает так: "В скобках приведен один параметр — литерал `32.0`, имеющий тип `double`. Следовательно, нужно найти в листинге 9.2 конструктор, принимающий один параметр типа `double`". Найти этот конструктор несложно. Это второй конструктор в коде класса `Temperature`. Его заголовок имеет следующий вид.

```
public Temperature(double number)
```

После этого компьютер выполняет инструкции, приведенные в теле найденного конструктора.

```
this.number = number;
scale = TempScale.FAHRENHEIT;
```

В результате компьютер создает объект типа `Temperature` и присваивает его полям определенные значения. Поле `number` получает значение `32.0`, а поле `scale` — значение `TempScale.FAHRENHEIT`.

Типы перечисления и оператор switch

В листингах 9.2 и 9.3 используются длинные имена `TempScale.FAHRENHEIT` и `TempScale.CELSIUS`. Короткие имена `FAHRENHEIT` и `CELSIUS` принадлежат типу перечисления `TempScale`, определенному в листинге 9.1. За пределами контекста `TempScale` они не имеют смысла. Попробуйте удалить префикс `TempScale` из выражения `TempScale.FAHRENHEIT`, и вы увидите, что компилятор вернет сообщение об ошибке.

Компилятор Java очень придирчив к именам типов и точкам. Однако в некоторых случаях вы можете создать нужный контекст, например, в операторе `switch`. Если в его условии приведена переменная типа перечисления, компилятор считает тело оператора `switch` контекстом данного типа и не требует специфицировать имя типа. Например, приведенный ниже код компилятор обрабатывает без сообщения об ошибке.

```
TempScale scale;
scale = TempScale.RANKINE;
char letter;
switch (scale) {
case CELSIUS:
  letter = 'C';
  break;
case KELVIN:
  letter = 'K';
  break;
case RANKINE:
case REAUMUR:
case ROMER:
  letter = 'R';
  break;
default:
  letter = 'X';
  break;
}
```

В первой строке этого кода объявлена переменная `scale` типа `TempScale`. Эта же переменная вставлена в условие оператора `switch`. Следовательно, тело оператора `switch` является контекстом типа `TempScale`. Кстати, попробуйте теперь написать, например, `TempScale.KELVIN` вместо `KELVIN`, и вы увидите, что компилятор вернет сообщение об ошибке.

Тело второго конструктора содержит две инструкции, присваивающие значения полям `number` и `scale`. Посмотрите на вторую инструкцию, которую немного легче понять. Она присваивает полю `scale` значение `TempScale.FAHRENHEIT`. В списке параметров данного конструктора нет значения для `scale`, поэтому конструктор всегда присваивает полю `scale` значение `TempScale.FAHRENHEIT`. Так решил программист, который живет в Нью-Джерси и которому больше нравится температурная шкала Фаренгейта, потому что она используется в Соединенных Штатах.

Первая инструкция присваивает значение полю `number` создаваемого объекта. Вы видите здесь часто применяемый трюк: две переменные с одним и тем же именем — `number` и `this.number`. Поскольку его нередко можно увидеть в реальных конструкторах, уделим ему немного внимания. Для понимания этого трюка взгляните на листинг 9.4. В нем приведены два полностью эквивалентных варианта конструктора.

Листинг 9.4. Два разных способа сделать одно и то же

```
// Можно использовать этот конструктор...
public Temperature(double whatever) {
  number = whatever;
  scale = TempScale.FAHRENHEIT;
}
```

```
// ...или этот...
public Temperature(double number) {
  this.number = number;
  scale = TempScale.FAHRENHEIT;
}
// ...но никогда не используйте оба этих
// конструктора одновременно!
```

В листинге 9.4 приведены два конструктора класса `Temperature`. В первом используются разные числа (`number` и `whatever`). Во втором используется только одно имя, два имени не нужны. Вместо объявления нового имени для параметра конструктора повторно используется существующее имя путем добавления префикса `this`.

Вот что происходит во втором конструкторе в листинге 9.4.

- ✓ В инструкции `this.number=number` имя `this.number` ссылается на поле `number` создаваемого объекта. Это поле объявлено в верхней части листинга 9.2 (рис. 9.2).
- ✓ В этой же инструкции `this.number=number` слово `number` (без префикса `this`) ссылается на параметр конструктора, передаваемый посредством списка параметров. Эта переменная объявлена в списке параметров.

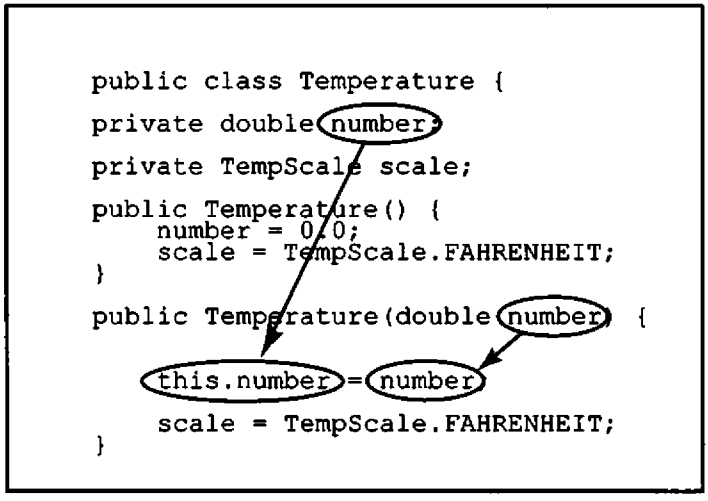


Рис. 9.2. Значения `this.number` и `number` — это разные числа

В общем случае выражение `this.`*`некоторое_имя`* ссылается на поле объекта. В противоположность этому *`некоторое_имя`* ссылается на переменную, объявленную на самом низком уровне вложенности. Если переменная *`некоторое_имя`* объявлена в методе, значит, она принадлежит методу, а `this.`*`некоторое_имя`* принадлежит классу.

Зачем такие сложности?

Не проще ли объявить еще одно имя?

Предположим, код содержит конструктор, похожий на рассматриваемый выше (первый конструктор в листинге 9.4). Посредством параметра `whatever` конструктор получает число `32.0`. Затем значение параметра присваивается полю `number`. Код работает безупречно. Но для этого нам пришлось придумать для параметра новое имя (`whatever`). Единственная цель этого параметра — передать свое значение полю `number`. Данный параметр — всего лишь временное хранилище значения, предназначенного для поля `number`.

Присвоение имен — сложная задача, хотя на первый взгляд она не кажется такой. Нужно, чтобы имя было, с одной стороны, коротким, а с другой — оно должно достаточно полно информировать о назначении переменной и не быть похожим на другие имена. Понаблюдайте за работой профессионального программиста, и вы увидите, что он часто подолгу задумывается над именем какой-либо малозначимой переменной.

Как назвать параметр, передающий значение полю `number`? Его имя должно как-то напоминать имя поля, например `aNumber` или `tmp_number`. В главе 8 всем параметрам присвоен суффикс `In`. Например, для `jobTitle` параметром служит `jobTitleIn`. Но любая подобная идея не очень хороша. Когда вы посмотрите на свой код через год, вы вряд ли вспомните, что означает префикс `a` или суффикс `In`. Между тем даже придумывать ничего не нужно! Можно использовать это же имя, добавив к нему префикс `this`, назначение которого понятно любому программисту, знающему синтаксис Java.

Некоторые вещи никогда не изменяются

В главе 7 мы кратко рассмотрели метод `printf()`, получающий строку форматирования, которая определяет способ отображения параметров. В предыдущих примерах строка форматирования была литералом, заключенным в двойные кавычки. Например, в листинге 7.7 использовалась следующая инструкция.

```
out.printf("$%4.2f\n", myInterest);
```

В листинге 9.3 я нарушил эту традицию и поместил в список параметров `printf()` переменную `format`.

```
out.printf(format, temp.getNumber(), temp.getScale());
```

Переменная `format` имеет тип `String` и содержит строку форматирования, определенную с помощью следующей инструкции.

```
final String format = "%5.2f degrees %s\n";
```

Обратите внимание на ключевое слово `final`. В Java оно означает, что переменная `format` не может изменяться в коде, т.е. она не будет изменена нигде и никогда. Если в каком-либо месте кода встретится инструкция, изменяющая значение `format`, компилятор сообщит об ошибке и не допустит программу на выполнение.

В листинге 9.3 ключевое слово `final` не обязательное. Оно добавлено лишь для защиты от неумышленного изменения переменной, которая не должна изменяться. Во многих случаях такая защита очень полезна. Когда программа содержит десятки тысяч строк кода и разрабатывается много месяцев, очень легко забыть, что данная переменная не должна изменяться. Когда же по неосторожности вы измените ее,

а потом обнаружите, что она изменилась, вам будет тяжело найти, какая инструкция изменила ее. Поэтому рекомендуется всегда добавлять ключевое слово `final`, если значение переменной не должно изменяться. Переменные такого вида часто называют *константами*.

Конструктор базового класса в производном классе

В главе 8 мы довольно подробно рассмотрели производные классы. Они весьма полезны тем, что позволяют повторно использовать коды базовых классов. В этом разделе мы создадим производный класс `TemperatureNice` на основе базового класса `Temperature`.

Усовершенствованный класс температуры

Проанализировав код листинга 9.3, нетрудно прийти к выводу, что инструкции, выводящие температуру на консоль, находятся не там, где им положено быть. В листинге 9.3 они повторяются несколько раз, чего в хорошей программе не должно быть. В 1970-е годы профессиональный программист посоветовал бы вам включить эти инструкции в метод (впрочем, тогда это называлось не методом, а функцией) и повторить несколько раз вызов метода, а не набор инструкций. Однако в те времена не было объектно-ориентированного программирования. Теперь оно есть, и мы, соответственно, должны мыслить шире. Нужно сделать так, чтобы каждый объект температуры сам выводил свои данные. Если создать метод `display()` и включить его в класс температуры, им смогут воспользоваться другие программисты, работающие с классом температуры.

Ну, что ж. Добавить метод в класс несложно. Однако, как и в главе 8, усложним себе задачу. Предположим, класс температуры разрабатывался другими людьми, у нас нет его исходного кода, а если и есть, то мы не можем его изменять. В общем, все, как в главе 8. Мы должны добавить метод `display()` в класс температуры, не изменив код класса. Если вы не читали главу 8, эта задача покажется вам фантастической. Как можно добавить что-либо в класс, не изменив его код? Очень просто. Нужно создать производный класс и добавить в него все, что нам нужно. При этом новый класс унаследует всю функциональность старого.

Создадим производный класс на основе класса `Temperature` как базового. Код класса `Temperature` приведен в листинге 9.2. Назовем производный класс `TemperatureNice` (листинг 9.5) и добавим в него метод `display()`, отображающий значения температуры и температурной шкалы унифицированным образом.

Листинг 9.5. Класс `TemperatureNice`

```
import static java.lang.System.out;

public class TemperatureNice extends Temperature {

    public TemperatureNice() {
        super();
    }
```

```
    public TemperatureNice(double number) {
        super(number);
    }

    public TemperatureNice(TempScale scale) {
        super(scale);
    }

    public TemperatureNice(double number, TempScale scale)
    {
        super(number, scale);
    }

    public void display() {
        out.printf("%5.2f градусов по %s\n", getNumber(),
                   getScale());
    }
}
```

В определении метода `display()` в листинге 9.5 обратите внимание на вызовы методов `getNumber()` и `getScale()`, которые определены в классе `Temperature`. Зачем они здесь нужны? Дело в том, что при непосредственном обращении кода класса `TemperatureNice` к закрытым полям базового класса `Temperature` компилятор сгенерирует ошибку. В каждом объекте класса `TemperatureNice` есть поля `number` и `scale`, однако обращаться к ним нельзя. В базовом классе эти поля объявлены как закрытые (с помощью модификатора `private`), поэтому непосредственно обращаться к ним можно только в коде класса `Temperature`. Из всех остальных классов к ним можно обращаться посредством методов доступа, определенных в классе `Temperature`.



Не объявляйте поля `number` и `scale` в коде класса `TemperatureNice`. Если вы сделаете это, вы неумышленно создадите четыре переменные (две `number` и две `scale`). Конструктор присвоит значения одной паре переменных, а пользоваться вы будете другой парой, думая, что это те же переменные. Когда коды классов длинные, заметить такую ошибку очень тяжело.



Если код объекта содержит вызов метода этого же объекта, добавлять к имени метода имя объекта необязательно. Например, в листинге 9.5 методы `getNumber()` и `getScale()` вызываются без имени объекта, потому что эти методы существуют в объекте типа `TemperatureNice`. Если вам все же хочется применить точку, можете написать `this.getNumber()` и `this.getScale()`. Этим вы указываете на то, что методы принадлежат данному объекту.

Конструкторы производных классов

Класс `TemperatureNice` содержит четыре собственных конструктора. Если вы уже освоили концепцию наследования, вы можете удивиться — зачем они нужны? Разве класс `TemperatureNice` не наследует их от базового класса? Оказывается, не наследует.



В отличие от методов производные классы не наследуют конструкторы базовых классов.

В листинге 9.5 каждый конструктор носит имя `TemperatureNice` и содержит собственный, уникальный список параметров. Кроме того, в каждом конструкторе присутствует нечто новенькое для вас — вызов метода `super()`.

Ключевое слово `super` означает в Java конструктор непосредственного базового класса.

- ✓ Инструкция `super()` в листинге 9.5 вызывает конструктор `Temperature()` без параметров, приведенный в листинге 9.2. Этот конструктор присваивает полю `number` значение `0.0`, а полю `scale` — значение `TempScale.FAHRENHEIT`.
- ✓ Инструкция `super(number,scale)` вызывает конструктор `Temperature(double number, TempScale scale)`, который присваивает полям значения, задаваемые при вызове конструктора.
- ✓ Аналогично этому инструкции `super(number)` и `super(scale)` вызывают соответствующие конструкторы, определенные в листинге 9.2.

Чтобы решить, какой конструктор класса `Temperature` должен быть вызван, компьютер анализирует список параметров в скобках после ключевого слова `super`. Предположим, компьютер встретил следующую инструкцию:

```
super(number, scale);
```

Переменные `number` и `scale` имеют типы `double` и `TempScale` соответственно. Однако только один конструктор `Temperature` в листинге 9.2 имеет два параметра с типами `double` и `TempScale`. Заголовок этого конструктора имеет такой вид:

```
public Temperature(double number, TempScale scale)
```

Следовательно, компьютер вызывает этот конструктор.

Использование усовершенствованного класса температуры

Класс `TemperatureNice` определен в листинге 9.5. Его использование демонстрируется в листинге 9.6.

Листинг 9.6. Проверка работы класса `TemperatureNice`

```
class UseTemperatureNice {

    public static void main(String args[]) {

        TemperatureNice temp = new TemperatureNice();
        temp.setNumber(70.0);
        temp.setScale(TempScale.FAHRENHEIT);
        temp.display();

        temp = new TemperatureNice(32.0);
        temp.display();
```

```
        temp = new TemperatureNice(TempScale.CELSIUS);
        temp.display();

        temp = new TemperatureNice(2.73, TempScale.KELVIN);
        temp.display();
    }
}
```

Листинг 9.6 довольно похож на листинг 9.3. Различия между ними состоят только в следующем.

- ✓ В листинге 9.6 создаются экземпляры класса `TemperatureNice`, а не `Temperature`. Соответственно, вызываются конструкторы `TemperatureNice`.
- ✓ В листинге 9.6 для отображения значений температуры используется метод `display()` класса `TemperatureNice`. Соответственно, код листинга 9.6 намного более аккуратный и компактный, чем код листинга 9.3.

Результат выполнения листинга 9.6 точно такой же, как листинга 9.3. Его можно увидеть на рис. 9.1.

Конструктор, выполняемый по умолчанию

Главный тезис предыдущего раздела состоит в том, что производный класс не наследует конструкторы базового класса. Однако вернемся к главе 8. В листинге 8.6 есть следующая инструкция:

```
FullTimeEmployee ftEmployee = new FullTimeEmployee();
```

Код производного класса `FullTimeEmployee`, определенный в листинге 8.3, не имеет конструктора. Унаследовать конструктор от базового класса он также не может. Откуда же взялся конструктор, вызываемый в приведенной выше инструкции?

Ответ довольно прост. Если в коде производного класса не определен его конструктор, компилятор по умолчанию применяет встроенный конструктор, который иногда называют *неявным*. При создании открытого производного класса `public FullTimeEmployee` компилятор автоматически применяет неявный конструктор, показанный в листинге 9.7.

Листинг 9.7. Неявный конструктор производного класса

```
public FullTimeEmployee() {
    super();
}
```

Неявный конструктор не принимает параметров, и его тело состоит из единственной инструкции — вызова конструктора базового класса. Но, если в базовом классе нет конструктора без параметров, компилятор сгенерирует сообщение об ошибке.

Неявный конструктор генерируется не всегда. В частности, если вы создали производный класс и определили в нем конструкторы, компилятор не добавит неявный конструктор (и, как вы помните, производный класс не унаследует конструктор).

В листинге 9.8 приведен код листинга 8.3 с одним добавлением: в классе FullTimeEmployee дополнительно определен конструктор с одним параметром типа double.

Листинг 9.8. В класс FullTimeEmployee добавлен конструктор

```
public class FullTimeEmployee extends Employee {
  private double weeklySalary;
  private double benefitDeduction;

  public FullTimeEmployee(double weeklySalary) {
    this.weeklySalary = weeklySalary;
  }

  public void setWeeklySalary(double weeklySalaryIn) {
    weeklySalary = weeklySalaryIn;
  }
  public double getWeeklySalary() {
    return weeklySalary;
  }
  public void setBenefitDeduction(double benefitDedIn) {
    benefitDeduction = benefitDedIn;
  }
  public double getBenefitDeduction() {
    return benefitDeduction;
  }
  public double findPaymentAmount() {
    return weeklySalary - benefitDeduction;
  }
}
```

Теперь приведенная ниже инструкция приведет к аварийному завершению программы.

```
FullTimeEmployee ftEmployee = new FullTimeEmployee();
```

Это объясняется тем, что, когда в классе определен конструктор с одним параметром, компилятор не создает для него неявный конструктор без параметров.

Что же делать? Если вы определяете в классе конструкторы, позаботьтесь о том, чтобы определить все конструкторы, которые могут понадобиться. Возьмите конструктор из листинга 9.7 и добавьте его в листинг 9.8. Тогда оператор `new FullTimeEmployee()` опять будет работать.

Конструктор может не только заполнять поля

Иногда при создании объекта нужно выполнить ряд операций, таких как конфигурирование объекта, его подготовка к работе, создание образа объекта в базе данных, установка сетевого соединения для связи объекта с сервером и т.а. Все это обычно делается в конструкторе. В данном разделе приведен пример (листинги 9.9 и 9.10), в

котором конструктор объекта фрейма (класса JFrame) настраивает его внешний вид на экране. Результат показан на рис. 9.3.

Листинг 9.9. Определение фрейма

```
import java.awt.FlowLayout;
import javax.swing.JFrame;
import javax.swing.JButton;
@SuppressWarnings("serial")
public class SimpleFrame extends JFrame {

  public SimpleFrame() {
    setTitle("Не щелкайте на кнопке!");
    setLayout(new FlowLayout());
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    add(new JButton("Щелкните на мне!"));
    setSize(300, 100);
    setVisible(true);
  }
}
```

Листинг 9.10. Вывод фрейма на экран

```
class ShowAFrame {
  public static void main(String args[]) {
    new SimpleFrame();
  }
}
```

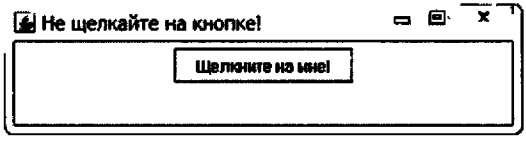


Рис. 9.3. Простой фрейм, отображенный объектом типа SimpleFrame

Код листинга 9.9 состоит главным образом из вызовов методов библиотечного класса JFrame, который отображает на экране стандартный фрейм. Код содержит много незнакомых вам имен. Некоторые начинающие программисты думают, что изучить Java означает запомнить все имена. Как раз наоборот! Реальное знание Java — это умение применять на практике концепции объектно-ориентированного программирования. А имена всегда можно найти в справочнике по стандартным библиотечным классам Java API.

В листинге 9.10 главный метод содержит единственную инструкцию: вызов конструктора класса SimpleFrame. Обратите внимание на то, что объект типа SimpleFrame даже не сохраняется в переменной. Это допустимо, потому что код нигде не ссылается на этот объект.

Класс SimpleFrame состоит всего лишь из определения конструктора. В конструкторе вызываются методы Java API, конфигурирующие фрейм и отображающие его на экране.

Все методы, вызываемые в конструкторе класса `SimpleFrame`, определены в базовом классе `JFrame`, который находится в пакете `javax.swing`. Этот пакет, а также пакет `java.awt`, содержат классы, позволяющие отображать на экране фреймы, изображения и элементы интерфейса, такие как кнопки, флажки, переключатели, раскрывающиеся списки. Например, в листинге 9.9 используется класс `JButton`, создающий на экране кнопку.



То, что в мире компьютеров обычно называется "окном", в Java называется *фреймом*. Фрейм создается с помощью класса `JFrame`.

Объявления импорта и пакеты

В Java можно группировать классы в *пакеты*. Если приложение состоит из многих пакетов и классов и разрабатывается многими людьми, имена пакетов и классов могут конфликтовать. Если вы работаете в команде, состоящей из тысячи человек, и назвали свой класс, например, `Calculate`, можете ли вы быть уверены в том, что никто больше не назвал так какой-нибудь другой класс? Проблема усугубляется тем, что созданный вами класс может использоваться в других приложениях, разрабатываемых в других компаниях.

Чтобы исключить конфликты имен, в Java принято группировать классы в пакетах и присваивать пакетам уникальные имена. Что в организации может быть уникальным? Адрес ее веб-сайта. Поэтому вам часто придется встречать имена пакетов, похожие на веб-адреса, например `com.allmycode.utils.text`. Стандартные библиотечные классы находятся в пакетах, имена которых начинаются с префикса `java` или `javax`. Например, популярны пакеты `java.lang`, `java.util`, `java.awt`, `javax.swing`.

Любое объявление `import`, которое не начинается с ключевого слова `static`, должно начинаться с имени пакета и заканчиваться одним из двух:

- ✓ именем класса в пакете;
- ✓ символом звездочки, которая обозначает все классы в пакете.

Например, объявление `import java.util.Scanner` правильное, потому что `java.util` — это имя одного из пакетов Java API, а `Scanner` — имя класса в этом пакете. Фраза `java.util.Scanner` называется *полностью квалифицированным именем* класса `Scanner`. Полностью квалифицированное имя класса должно содержать имя пакета, в котором определен класс. Рассмотрим еще один пример объявления импорта.

```
import javax.swing.*;
```

Фраза `javax.swing` — это имя одного из пакетов Java API. Звездочка обозначает импорт всех классов пакета `javax.swing`. Когда это объявление импорта присутствует в верхней части кода класса, можно использовать не полностью квалифицированные, а сокращенные имена классов, например `JFrame` вместо `javax.swing.JFrame`.

Обратите внимание на то, что объявление `import javax.*` неправильное, потому что в Java API не существует пакета `javax`. Это означает, что заменять звездочкой имена пакетов нельзя. Если сделать так, компилятор сгенерирует сообщение об ошибке.

Классы и методы Java API

Глядя на рис. 9.3, вы скажете, что класс `SimpleFrame` делает не так уж много. Фрейм содержит всего лишь одну кнопку, после щелчка на которой ничего не происходит. Однако учитывайте, что я сделал фрейм таким только для того, чтобы не усложнять

пример. Но даже при этом в листинге 9.9 используется восемь методов, и каждый из них что-либо делает. Ниже приведен список имен, используемых в листинге 9.9.

- ✓ **`setTitle()`**. Этот метод позволяет задать текст, который будет отображаться в строке заголовка фрейма.
- ✓ **`FlowLayout()`**. Вызов конструктора класса `FlowLayout`. Экземпляр этого класса позиционирует объекты в рамке фрейма. Поскольку фрейм содержит единственный элемент интерфейса (кнопку), она центрируется по горизонтали и вертикали. Если бы в фрейме было 10 кнопок, они были бы автоматически размещены равномерно по рядам, причем последний ряд был бы центрирован по горизонтали.
- ✓ **`setLayout()`**. Присвоение фрейму объекта `FlowLayout`, который размещает элементы интерфейса в фрейме.
- ✓ **`setDefaultCloseOperation()`**. Этот метод сообщает виртуальной машине Java, что она должна сделать, когда пользователь щелкнет на кнопке × в правом конце строки заголовка фрейма. Когда вызова этого метода нет, после щелчка будет закрыт фрейм, но приложение продолжит выполнять другие свои компоненты. Вы попадете в неловкую ситуацию: фрейм исчез, но приложение, уже невидимое, продолжает выполняться. Если приложение выполняется в среде Eclipse, его можно закрыть, щелкнув на красной квадратной кнопке справа от корешка вкладки консоли (рис. 9.4).

  Инструкция `setDefaultCloseOperation(EXIT_ON_CLOSE)` приказывает виртуальной машине Java закрыть приложение после щелчка на кнопке ×. Существуют и другие варианты, например `HIDE_ON_CLOSE` (скрыть вместо закрытия), `DISPOSE_ON_CLOSE` (уничтожить объект вместо закрытия) и `DO_NOTHING_ON_CLOSE` (ничего не делать вместо закрытия).
- ✓ **`JButton`**. Класс `JButton` рисует в фрейме кнопку и находится в пакете `javax.swing`. Один из конструкторов класса принимает строку, отображаемую на кнопке. Как видите, на рис. 9.3 на кнопке отображается строка, переданная конструктору `JButton`.
- ✓ **`add()`**. Метод, добавляющий кнопку на поверхность фрейма.
- ✓ **`setSize()`**. Метод, задающий размеры фрейма 300×100 пикселей. В пакете `javax.swing` при каждом задании двух размеров чего-либо первое число всегда означает ширину, а второе — высоту.
- ✓ **`setVisible()`**. Метод, управляющий видимостью фрейма на экране. После создания фрейм невидим. Чтобы он появился на экране, нужно вызвать метод `setVisible(true)`.

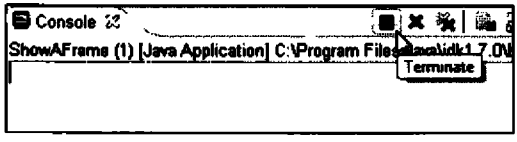


Рис. 9.4. Кнопка в Eclipse, принудительно завершающая программу Java

Аннотация @SuppressWarnings

В главе 8 были введены аннотации — ключевые слова Java, сообщающие компилятору дополнительную информацию о коде, помеченном аннотацией. В частности, в главе 8 была рассмотрена аннотация `@Override`, которая сообщает компилятору о том, что данный метод производного класса переопределяет метод базового класса.

В данной главе рассмотрим еще одну аннотацию — `@SuppressWarnings`. Она приказывает компилятору не сообщать о том, что программа содержит подозрительный код. В листинге 9.9 строка `@SuppressWarnings("serial")` приказывает отменить вывод предупреждений о том, что пропущено поле `serialVersionUID`. На рис. 9.5 показано, что будет, если закомментировать эту аннотацию.

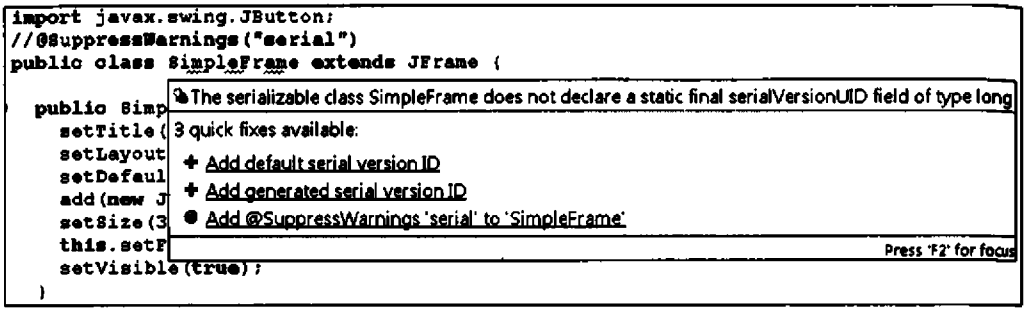


Рис. 9.5. Предупреждение о том, что опущено поле `serialVersionUID`

Вы спросите: "Ну и что нам до того, что оно пропущено?" В общем-то, ничего. Вряд ли вам когда-либо придется иметь дело с сериализацией объектов, т.е. их представлением в строковом виде для передачи в другие контексты. Поэтому спите спокойно, рискуйте разумно и не бойтесь пропустить предупреждение, которое при отсутствии данной аннотации будет постоянно лезть вам в глаза и закрывать нужный код в окне редактора.

Часть IV

Передовые методики программирования

The 5th Wave — Рич Теннант



"Мы пришли почистить код".

В этой части...

В этой части мы наконец-то приступим к настоящему программированию. Я расскажу вам о хитроумных трюках, специальных правилах и не очень специальных исключениях (не из правил, а из процессов). В данной части есть много такого, что может напугать вас, однако, как и в других частях, не пугайтесь. Все, что написано, вы в состоянии понять, ведь я постарался сделать это для вас интересным и простым.

Глава 10

Правильное применение переменных и методов

В этой главе...

- Определение класса
- Статические поля и методы
- Поэкспериментируем с переменными
- Передача параметров

Привет! Это снова я, Сэм Берд, и вы слушаете радио WWW. Начался большой сезон бейсбола, и сегодня наша станция WWW предлагает вам прослушать репортаж с очередной игры Ханков против Соксов. В данный момент я ожидаю новостей о счете.

Как вы помните, еще полчаса назад Соксы выглядели так, будто собираются положить Ханков на обе лопатки, но сейчас Ханки отбивают одно очко за другим, а Соксы совсем скисли. Ох, уж эти Соксы! Не хотел бы я быть на их месте.

Однако ситуация быстро меняется. Соксы взяли себя в руки, и теперь Ханки и Соксы идут "ноздря в ноздрю". До финала осталось несколько минут, и мы спешим напомнить вам: оставайтесь на волне после этой передачи, чтобы прослушать важную информацию. И не забудьте, что на следующей неделе Кливлендские Клоуны играют с Бермудскими Акулами.

А вот и окончательный счет! Победитель... О нет, это невероятно!

Определение класса

Что означает быть игроком в бейсбол? Для нас это сейчас означает обладать именем (`name`) и иметь средний бэттинг-показатель (`average`). Такой упрощенный взгляд на бейсбол выражен на языке Java в листинге 10.1.

Листинг 10.1. Класс `Player` (Игрок)

```
import java.text.DecimalFormat;

Public class Player {
    private String name;
    private double average;

    public Player(String name, double average) {
        this.name=name;
        this.average=average;
    }
```

```
    public String getName() {
        return name;
    }

    public double getAverage() {
        return average;
    }

    public String getAverageString() {
        DecimalFormat decFormat = new DecimalFormat();
        decFormat.setMaximumIntegerDigits(0);
        decFormat.setMaximumFractionDigits(3);
        decFormat.setMinimumFractionDigits(3);
        return decFormat.format(average);
    }
}
```

Итак, приступим к анализу листинга 10.1. К счастью, в предыдущих главах мы уже многое узнали о классах. В данном коде приведено определение класса `Player`, которое состоит из следующих компонентов.

- ✓ **Объявление полей `name` и `average`, содержащих имя и средний показатель игрока.** Правила объявления полей рассматриваются в главе 7.
- ✓ **Конструктор объекта типа `Player`.** Конструкторы рассматриваются в главе 9.
- ✓ **Методы доступа к полям `name` и `average`.** Использование методов доступа рассматривается в главе 7.
- ✓ **Метод, возвращающий средний показатель игрока в формате `String`.** Методы и возвращаемые значения рассматриваются в главе 7.

Еще один способ сделать числа красивыми

Метод `getAverageString()` считывает число из поля `average` (средний показатель игрока) типа `double`, преобразует его в объект типа `String` и возвращает этот объект вызывающей программе. Тип `DecimalFormat` определен в Java API и предназначен для форматирования чисел в строке `String`. Благодаря методам `decFormat.setMaximum...()` и `decFormat.setMinimum...()` результирующее значение типа `String` имеет точно три цифры справа от десятичного разделителя и не имеет цифр слева от десятичного разделителя.

Библиотечный класс `DecimalFormat` очень удобен во многих случаях. Например, для отображения значений `345` и `-345` в формате, к которому привыкли бухгалтеры, можно использовать следующий код.

```
DecimalFormat decFormat = new DecimalFormat();
decFormat.setMinimumFractionDigits(2);
decFormat.setNegativePrefix("(");
decFormat.setNegativeSuffix(")");
System.out.println(decFormat.format(345));
System.out.println(decFormat.format(-345));
```

Первая инструкция, выделенная полужирным шрифтом, определяет вывод положительных чисел. Вторая и третья инструкции, выделенные полужирным шрифтом, определяют вывод отрицательных чисел. Числа `345` и `-345` будут отображены на консоли следующим образом.

```
345.00
(345.00)
```

Богатые возможности, предоставляемые классом `DecimalFormat`, описаны в справочнике по библиотечным классам Java API.

Использование класса `Player`

В листингах 10.2 и 10.3 приведены классы, в которых используется класс `Player`, определенный в листинге 10.1.

Листинг 10.2. Использование класса `Player`

```
import java.util.Scanner;
import java.io.File;
import java.io.IOException;
import javax.swing.JFrame;
import javax.swing.JLabel;
import java.awt.GridLayout;

@SuppressWarnings("serial")
public class TeamFrame extends JFrame {

    public TeamFrame() throws IOException {
        Player player;
        Scanner keyboard = new Scanner(new File("Hankees.txt"));

        for (int num = 1; num <= 9; num++) {
            player = new Player(keyboard.nextLine(),
                                keyboard.nextDouble());
            keyboard.nextLine();

            addPlayerInfo(player);
        }

        setTitle("Средние бэттинги Ханков");
        setLayout(new GridLayout(9, 2, 20, 3));
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }

    void addPlayerInfo(Player player) {
        add(new JLabel("  " + player.getName()));
        add(new JLabel(player.getAverageString()));
    }
}
```

Листинг 10.3. Вывод фрейма на экран

```
import java.io.IOException;

class ShowTeamFrame {

    public static void main(String args[]) throws IOException {
        new TeamFrame();
    }
}
```

Результат выполнения листингов 10.1–10.3 показан на рис. 10.1.

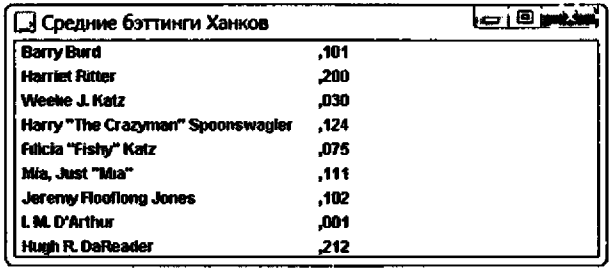


Рис. 10.1. Вы поставили бы на них деньги?

Данные для программы находятся в файле `Hankees.txt` (рис. 10.2). Чтобы программа нормально выполнилась, она должна найти этот файл и прочитать из него данные.

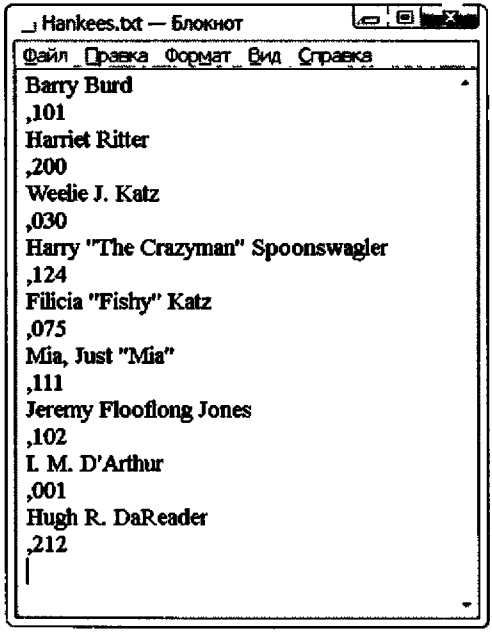


Рис. 10.2. Файл `Hankees.txt`

Файл `Hankees.txt` должен находиться в определенном месте жесткого диска. Если вы работаете в Eclipse, он должен находиться в папке проекта в рабочем пространстве Eclipse. Если же вы запускаете Java из командной строки, файл `Hankees.txt` должен находиться в одной папке с листингом 10.3. В любом случае выполнить программу без этого файла невозможно. Если программа не найдет файл `Hankees.txt`, она сгенерирует исключение `FileNotFoundException` (файл не найден, исключение).



Чтобы код данной программы работал правильно, после строки `,212` (это последняя строка в файле) должен быть символ новой строки. На рис. 10.2 о его наличии свидетельствует курсор ввода.

Девять вызовов конструктора

В листинге 10.2 конструктор класса `Player` вызывается девять раз. Это означает, что код создает девять экземпляров класса `Player`. На первой итерации цикла код создает экземпляр с именем `Barry Burd`. На второй итерации код уничтожает экземпляр `Barry Burd` и создает экземпляр с именем `Harriet Ritter`, и т.д. В каждый момент времени существует только один экземпляр, но всего код создает девять экземпляров.

Каждый экземпляр `Player` имеет собственные поля `name` и `average`. Кроме того, у каждого экземпляра есть собственный конструктор `Player()` и методы `getNave()`, `getAverage()` и `getAverageString()`. Посмотрите на рис. 10.3 и представьте себе класс `Player` в девяти его ипостасях.

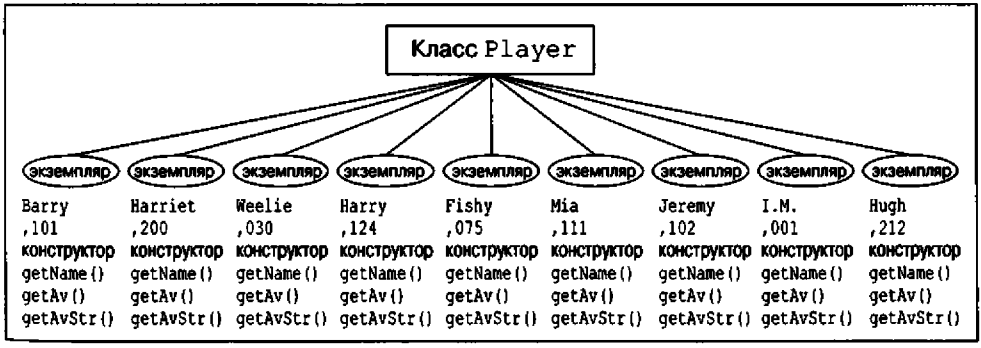


Рис. 10.3. Класс и его объекты

Графический интерфейс пользователя

В листинге 10.2 используется ряд имен, влияющих на внешний вид фрейма и определенных в Java API. Некоторые из этих имен описаны в главе 9. Оставшиеся описываются ниже.

- ✓ **`JLabel`.** Объект надписи отображает на поверхности фрейма заданный текст. Необходимо с помощью библиотечного метода `add()` добавить экземпляр класса надписи в фрейм.

  В листинге 10.2 метод `addPlayerInfo()` вызывается девять раз, по одному разу для каждого игрока. При каждом вызове метод

addPlayerInfo() добавляет на поверхность фрейма два объекта типа JLabel (т.е. две надписи). Текст каждой надписи извлекается с помощью метода доступа из закрытого поля класса Player.

- ✓ **GridLayout()**. Класс GridLayout равномерно размещает элементы интерфейса на поверхности фрейма по строкам и столбцам.

  В листинге 10.2 конструктор GridLayout() принимает четыре параметра (9,2,20,3). Первые два из них — количество строк и столбцов. Как видите, на рис. 10.1 выведены девять строк (по одной для каждого игрока) и два столбца (для имени и среднего показателя игрока). Третий параметр задает горизонтальный зазор между столбцами равным 20 пикселям. Четвертый параметр задает вертикальный зазор между строками равным 3 пикселям.

- ✓ **pack()**. При упаковке фрейма вы устанавливаете его размеры. Эти размеры фрейм имеет при отображении на экране. Операция упаковки сокращает фрейм до размеров, необходимых для отображения расположенных в нем объектов.

  В листинге 10.2 к моменту вызова метода pack() компьютер уже вызвал метод addPlayerInfo() девять раз и добавил во фрейм 18 надписей. При выполнении метода pack() компьютер подбирает размеры для каждой надписи с учетом длины и высоты выводимого текста. Затем компьютер вычисляет размеры всего фрейма таким образом, чтобы в нем поместились 18 надписей.

Метод setLayout() задает способ компоновки элементов интерфейса на поверхности фрейма. Задать компоновку можно как до, так и после добавления надписей и других элементов на поверхность фрейма. Если сначала вызвать метод setLayout(), а затем добавить надписи, они появятся во фрейме аккуратно размещенными. Если же сначала добавить надписи, а затем вызвать метод setLayout(), этот метод упорядочит уже размещенные надписи. Результат будет один и тот же.

При создании фрейма необходимо соблюдать следующую последовательность операций.

1. Добавление элементов и компоновка фрейма.
2. Упаковка с помощью метода pack().
3. Отображение с помощью метода setVisible(true).

Если сначала вызвать pack(), а затем добавить во фрейм еще несколько элементов, они не будут учтены при упаковке и могут "выпасть" за пределы фрейма. Если вызвать метод setVisible(true) перед добавлением элементов и упаковкой, пользователь увидит на экране процесс конструирования фрейма. Это выглядит, как мелькание элементов, поэтому делать так не рекомендуется. И наконец, если вы забудете установить размеры фрейма (путем вызова метода pack() или явного задания размеров), фрейм будет выглядеть, как на рис. 10.4. Я бы не показывал вам ошибочный рисунок, но студенты так часто делают эту ошибку, что я решил предостеречь вас от нее.

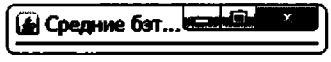


Рис. 10.4. Забыли упаковать...

Отфутболим исключение другому методу

В главе 8 рассматривается считывание данных из файла на диске, в связи с чем в данной главе было введено понятие исключения. Вы можете отладить программу, и она не будет содержать ошибок. Но вы не можете гарантировать, что во время ее выполнения на диске будет нужный ей файл. Поэтому всегда есть возможность ошибки. Пользователю очень не понравится, если программа вдруг исчезнет с экрана по неизвестной причине. Поэтому операционная система не закрывает программу, а генерирует *исключение*, давая программе возможность исправить ошибку или хотя бы сообщить пользователю, в чем она состоит. При ошибке ввода-вывода генерируется исключение `IOException`. Директива `throws IOException` в листинге 10.2 сообщает компилятору о том, что данный метод может сгенерировать исключение и передать его вызывающему методу.

Но зачем директива `throws IOException` методу `main()` в листинге 10.3? В нем нет обращения к файлу, откуда же в нем возникнет исключение `IOException`? Дело в том, что исключение похоже на горячую картофелину, которую перебрасывают из рук в руки. Получив картофелину, вы должны или съесть ее, или быстренько перебросить другому, иначе обожжете руки. То же самое справедливо и для исключения. Метод, сгенерировавший или получивший исключение, должен либо обработать его в блоке `try` (как это делается, мы рассмотрим в главе 12), либо передать вызывающему методу. Однако без директивы `throws` у метода нет выбора. Он либо сам обработает исключение, либо программа будет аварийно завершена.

В листинге 10.2 конструктор `TeamFrame()` генерирует исключение `IOException`. Однако кто его получает? Несомненно, его получает метод, который вызвал конструктор, т.е. метод `main()`, код которого приведен в листинге 10.3. В свою очередь, метод `main()` также генерирует исключение.



Если в вызываемом методе установлена директива `throws`, вызывающий метод должен либо обработать исключение, либо передать его своему вызывающему методу. Но для этого в его заголовке также должна быть установлена директива `throws`. Обработка исключений рассматривается в главе 12.



Кто получает исключение, сгенерированное методом `main()`? Виртуальная машина Java. В частности, она обработает (вернее, попытается обработать) исключение, сгенерированное методом `main()` в листинге 10.3. Если у программы возникли трудности при чтении файла `Hankees.txt`, ответственность в конце концов ляжет на виртуальную машину. Она отобразит сообщение об ошибке и завершит выполнение программы.

Статические поля и методы

Средние показатели игроков важны, но иногда нужен показатель, усредненный по всей команде. Глядя на рис. 10.1 видно, что успехи Ханков не очень впечатляют, но лучше было бы выразить показатель успешности команды одним числом. Хотя и так видно, что команде нужно тренироваться. Пока ребята будут упражняться на поле, у нас есть время подойти к этому вопросу с точки зрения Java.

В листингах 10.1–10.3 приведены три класса. Класс `Player` содержит данные об игроках, а два других класса (`TeamFrame` и `ShowTeamFrame`) отображают эти данные на экране. Где в этой структуре должна находиться переменная, содержащая средний показатель команды?

- ✓ Размещать ее в классе `TeamFrame` или `ShowTeamFrame` нет смысла, потому что эти классы предназначены для отображения, а не хранения данных. Их задача — создать фрейм и отобразить в нем надписи.
- ✓ Размещать ее в экземпляре класса `Player` не очень удобно, потому что в экземпляре класса хранятся данные об одном игроке. Здесь не место для информации обо всей команде. Конечно, можно поместить переменную в класс `Player` и заставить код работать, но это не будет элегантным решением проблемы.

На помощь приходит ключевое слово `static`. Все, что объявлено с ключевым словом `static`, принадлежит классу, а не конкретному экземпляру класса. Когда вы объявляете статическое поле `totalOfAverages` (средний показатель команды), компилятор создает единственную копию поля, принадлежащую всему классу `Player`. Код другого класса может создать много экземпляров класса `Player`, но поле `totalOfAverages` всегда будет только одно. Но, раз уж мы так успешно начали, создадим и другие статические поля — `playerCount` (количество игроков) и `decFormat` (десятичный формат), а также статические методы `findTeamAverage()` и `findTeamAverageString()`. Заодно создадим для этого класс `PlayerPlus` (рис. 10.5), чтобы не путать его с предыдущим примером.

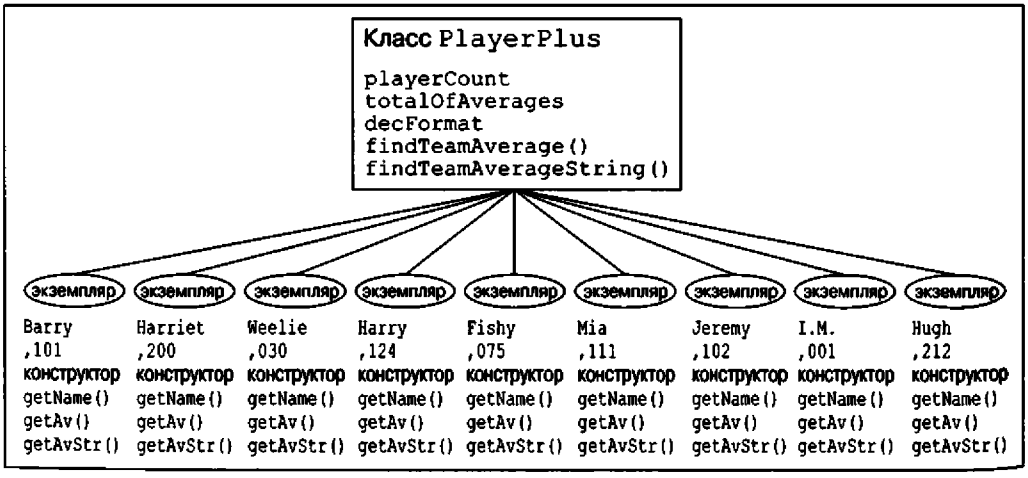


Рис. 10.5. Статические и нестатические поля и методы

Учитывая нашу страсть к наследованию, создадим класс `PlayerPlus` (листинг 10.4) как производный от класса `Player`, чтобы повторно использовать всю функциональность существующего класса.

Листинг 10.4. Класс, усредняющий показатель по команде

```
import java.text.DecimalFormat;

public class PlayerPlus extends Player {
    private static int playerCount = 0;
    private static double totalOfAverages = .000;
    private static DecimalFormat decFormat = new DecimalFormat();

    static {
        decFormat.setMaximumIntegerDigits(0);
        decFormat.setMaximumFractionDigits(3);
        decFormat.setMinimumFractionDigits(3);
    }

    public PlayerPlus(String name, double average) {
        super(name, average);
        playerCount++;
        totalOfAverages += average;
    }

    public static double findTeamAverage() {
        return totalOfAverages / playerCount;
    }

    public static String findTeamAverageString() {
        return decFormat.format(totalOfAverages / playerCount);
    }
}
```

Зачем столько статики

Наверняка вы заметили, что в листинге 10.4 почти все статическое. Это потому, что почти все в этом коде принадлежит классу `PlayerPlus` в целом, а не отдельным экземплярам класса. Например, поле `playerCount` (количество игроков) явно не имеет никакого отношения к отдельному игроку; оно имеет смысл только для всей команды. Если бы у нас было девять отдельных полей `playerCount`, в каждом поле была бы или единица (для одного игрока количество игроков равно единице), или девятка. Но тогда число 9 нужно было бы записать в девять разных полей. Что за бессмыслица!

То же самое справедливо и для поля `totalOfAverages` (средний показатель команды). В нем хранится сумма средних показателей всех игроков команды. Для девяти игроков сумма равна `0.956`. Чтобы найти ее, нужно вызвать методы `findTeamAverage()` и `findTeamAverageString()`. Эти методы также должны быть статическими, потому что нам не нужны восемнадцать методов (по два для каждого экземпляра класса). Каждый экземпляр вычислял бы собственное отношение `totalOfAverages/playerCount`, причем каждый раз получалось бы одно и то же значение.



В общем случае любую задачу, общую для всех экземпляров класса (или такую, которая возвращает один и тот же результат для каждого экземпляра), рекомендуется закодировать как статический метод.



Статические конструкторы недопустимы.

Статическая инициализация

В листинге 10.4 поле `decFormat` статическое. Для него имеет смысл быть статическим, потому что оно определяет формат вывода отношения статических величин `totalOfAverages/playerCount`. Иными словами, для всех экземпляров класса нужна только одна переменная, форматирующая числа. Один и тот же объект `decFormat` может форматировать как поля экземпляров, так и поля класса. Поэтому `decFormat` должен принадлежать классу. Создание отдельного объекта `decFormat` для каждого игрока было бы расточительством ресурсов и времени.

Однако при объявлении поля `decFormat` как статического возникает небольшая проблема. Чтобы сконфигурировать форматирование, нужно вызвать ряд методов объекта `decFormat`, таких как `decFormat.setMaximumIntegerDigits(0)`. Вызовы этих методов нельзя вставить в любое место класса `PlayerPlus`. Например, следующий код неправильный.

```
// Это неправильный код!
public class PlayerPlus extends Player {
private static DecimalFormat decFormat = new DecimalFormat();
  decFormat.setMaximumIntegerDigits(0); //  Неправильно!
  decFormat.setMaximumFractionDigits(3); // Неправильно!
  decFormat.setMinimumFractionDigits(3); // Неправильно!
```

Посмотрите на примеры из предыдущих глав. В них вызовы методов никогда не "висят" в классе. Вызов метода — это операция, а любую операцию можно выполнить только в теле какого-либо метода.

Куда же нам поместить вызовы методов `setMax...()` и `setMin...()`? Их можно поместить в тело метода `findTeamAverageString()`, как в листинге 10.1. Но тогда не имеет смысла делать поле `decFormat` статическим. В конце концов, программа может вызывать метод `findTeamAverageString()` многократно, и каждый раз будет вызываться метод `decFormat.setMaximumIntegerDigits(0)`. Но зачем делать много раз то, что можно сделать один раз? Класс `PlayerPlus` имеет только одно поле `decFormat`, его поле `MaximumIntegerDigits` всегда равно нулю, поэтому вычислять его многократно не имеет смысла.

Наилучшее решение состоит в конфигурировании объекта `decFormat` в *статическом инициализаторе* (см. листинг 10.4). Статический инициализатор — это блок, которому предшествует ключевое слово `static` и который выполняется один раз при запуске программы.

Отображение общей статистики команды

Возможно, вы уже заметили, что в примерах часто повторяется следующий шаблон. Пример разбивается на две части: класс примера, представляющий объект реального мира, и тестовый класс, используемый для запуска класса примера. Для запуска и тестирования класса примера тестирующий класс вызывает конструктор, ссылается на открытые поля, вызывает методы и т.п. Согласно этому шаблону определение класса примера приведено в листинге 10.4, а в листинге 10.5 содержится тестирующий класс.

Листинг 10.5. Тестирование класса `PlayerPlus`

```
import java.util.Scanner;
import java.io.File;
import java.io.IOException;
import javax.swing.JFrame;
import javax.swing.JLabel;
import java.awt.GridLayout;

@SuppressWarnings("serial")
public class TeamFrame extends JFrame {

    public TeamFrame() throws IOException {
        PlayerPlus player;
        Scanner keyboard = new Scanner(new File("Hankees.txt"));

        for (int num = 1; num <= 9; num++) {
            player = new PlayerPlus(keyboard.nextLine(),
                                    keyboard.nextDouble());
            keyboard.nextLine();
            addPlayerInfo(player);
        }

        add(new JLabel());
        add(new JLabel("  ------"));
        add(new JLabel("Средний бэттинг команды:"));
        add(new JLabel(PlayerPlus.findTeamAverageString()));

        setTitle("Бэттинги игроков и команды");
        setLayout(new GridLayout(11, 2, 20, 3));
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }

    void addPlayerInfo(PlayerPlus player) {
        add(new JLabel("  " + player.getName()));
        add(new JLabel(player.getAverageString()));
    }
}
```

Чтобы запустить код листинга 10.5, необходим класс, содержащий метод `main()`. В данном случае прекрасно подойдет класс `ShowTeamFrame`, приведенный в листинге 10.3.

На рис. 10.6 показан результат выполнения листинга 10.5. Работа программы зависит от наличия на диске файла `Hankees.txt` (см. рис. 10.2). Код листинга 10.5 — почти точная копия листинга 10.2. Новые места отмечены полужирным шрифтом. Как видите, класс `Player` везде заменен на `PlayerPlus`, и добавлен вывод среднего показателя команды.

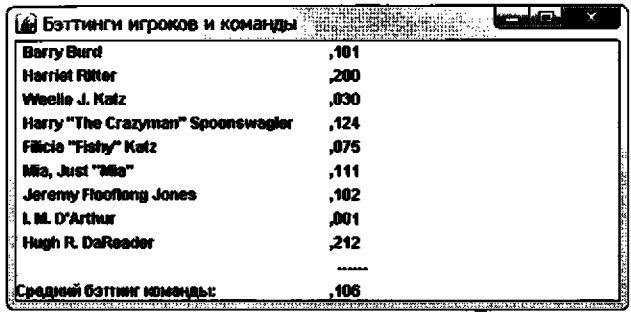


Рис. 10.6. Выполнение листинга 10.5

В листинге 10.5 объект `GridLayout` добавил две дополнительные строки: одну — для отображения черточек и вторую — для вывода среднего показателя команды. В каждой из этих двух строк размещены по два объекта `JLabel`.

- ✓ **Первая строка содержит пустую надпись и надпись с черточками.** Пустая надпись служит заполнителем. Компоненты добавляются в `GridLayout` построчно, начиная с левого конца строки и переходя к правому концу. Без пустой надписи черточки оказались бы слева, а справа появилась бы надпись `Средний бэттинг команды`.
- ✓ **Во второй строке отображаются надпись `Средний бэттинг команды` и значение, возвращенное методом `findTeamAverageString()`.** Рассмотрим подробнее вызов метода, получающего число `0.106`. Вызов выглядит так:

  ```
  PlayerPlus.findTeamAverageString()
  ```

  Как видите, вызов выполняется в необычной форме.

  `имя_класса.имя_метода``()`

  Для вас это нечто новенькое. До сих пор мы вставляли в качестве префикса имя объекта, а не класса. Почему же здесь вставлено имя класса? Потому, что вызывается статический метод, принадлежащий классу, а не конкретному объекту. То же самое справедливо и при обращении к статическим полям. Не забывайте, что при обращении к статическому полю или методу необходимо указывать имя класса, а не объекта.



В некоторых случаях компилятор можно обмануть, обратившись к статическому полю или методу через имя объекта, а не класса. Иногда такой трюк полезен при работе с экземплярами класса.

Статический импорт

Мы уже много говорили о статических полях и методах, но есть еще один статический компонент, который мы кратко упомянули в главе 3. В этой главе мы ввели метод `System.out.println()`. Имя `System` ссылается на класс, а имя `out` — на статическое поле этого класса. Ключевое слово `static` используется для импорта поля `out`.

```
import static java.lang.System.out;
```

После этого можно немного сократить инструкцию вызова до `out.println()`. Обратите внимание на то, что применять полностью квалифицированное имя `System.out.println()` теперь нельзя.

В программах на Java статические поля и методы встречаются довольно часто. Однако, прежде чем объявить поле или метод статическим, хорошенько подумайте. Они станут более удобными, в некоторых случаях к ним можно будет обращаться, не создавая объект, но на них будут наложены серьезные ограничения (см. далее).



Впервые в книге ключевое слово `static` упоминается в листинге 3.1, в определении метода `main()`. Все методы `main()` в данной книге статические. Зачем им быть статическими? Как вы помните, нестатические методы принадлежат объектам, а не классам. Если метод `main()` не статический, его нельзя вызвать, пока не будет создан объект, которому он принадлежит. Но выполнение любой программы должно начинаться с вызова метода `main()`. Возникает проблема яйца и курицы. Метод `main()` нельзя вызвать, потому что нет объекта, а объект нельзя создать, потому что нельзя вызвать метод `main()`. Проблема легко решается путем объявления метода `main()` статическим.

Осторожно, статика!

Когда я начал изучать Java, я так часто получал одно сообщение об ошибке, что оно стало сниться мне по ночам. Вот это сообщение: `non-static variable or method cannot be referenced from a static context` (на нестатическую переменную или метод нельзя ссылаться в статическом контексте).

Воспоминание об этом сообщении так глубоко укоренилось в моем подсознании, что я помню о нем до сих пор, хотя теперь я знаю, почему оно появлялось так часто. Сейчас я могу даже вызвать его умышленно. Но у меня до сих пор мурашки по телу, когда вижу его на экране.

Прежде чем я смогу объяснить вам, почему появляется это сообщение и как исправить ошибку, вы должны ознакомиться с некоторой терминологией. Если метод не является *статическим*, значит, он *нестатический* (ничего удивительного, не так ли?). Теперь вы знаете необходимые термины, и я могу приступить к объяснению ошибки. Чтобы вызвать ее умышленно, поместите в программу одну из следующих конструкций.

- ✓ Поместите в любом месте программы ссылку следующего вида: ***имя_класса.имя_нестатической_сущности***.
- ✓ Поместите в тело статического метода ссылку следующего вида: ***имя_нестатической_сущности***.

В любом из этих случаев вы получите упомянутое выше сообщение. В первом случае вы пытаетесь извлечь из класса то, что принадлежит объекту. Во втором случае вы берете нечто, принадлежащее объекту (нестатическую сущность), и помещаете его в место, в котором нет никаких объектов.

Рассмотрим для примера первую ситуацию. Посмотрите на листинг 10.5. В конце листинга замените `player.getName()` на `Player.getName()`. Вы получите

"желаемое" сообщение об ошибке. Что может означать `Player.getName()`? На русском оно означает "Вызвать метод `getName()`, принадлежащий классу `Player`". Однако посмотрите в листинг 10.1. В нем метод `getName()` не является статическим. Каждый экземпляр класса `Player` или `PlayerPlus` имеет метод `getName()`, но ни один метод `getName()` не принадлежит классу. Поэтому инструкция `Player.getName()` не имеет смысла.

Чтобы смоделировать вторую ситуацию (имя нестатической сущности в теле статического метода), вернемся к листингу 10.4. Незаметно для окружающих удалите слово `static` из объявления поля `decFormat` в верхней части листинга. Поле станет нестатическим. Каждый игрок команды получит персональный объект `decFormat`.

Все будет сносно до тех пор, пока компьютер не вызовет метод `findTeamAverageString()`. В этом статическом методе четыре раза упоминается ссылка `decFormat.`***`имя_сущности`***. Что может означать эта ссылка? Метод `findTeamAverageString()` не принадлежит ни одному экземпляру (он принадлежит классу в целом). Как вы помните, теперь у каждого игрока есть персональный объект `decFormat`, но общего объекта `decFormat` нет нигде. Следовательно, вы ссылаетесь на объект, которого не существует. И так будет каждый раз, когда вы сошлетесь на нестатическое поле в статическом методе.

Поэкспериментируем с переменными

Однажды летом, когда я учился в колледже, я сидел на скамейке в парке колледжа и разговаривал с девушкой, с которой только что познакомился. Ее звали Джанин. Я спросил: "Откуда вы?" Она ответила: "С Марса".

Я подумал, что она шутит, но потом оказалось, что она действительно выросла в небольшом городке Марс, штат Пенсильвания, в двадцати милях к северу от Питтсбурга. Однако при чем здесь Java? Оказывается, к Java это имеет самое непосредственное отношение. Значение имени зависит от контекста, в котором оно находится. Если севернее Питтсбурга спросить: "Как добраться до Марса?", никто не удивится, и вы получите вразумительный ответ. Но если этот же вопрос задать прохожему на улице Манхэттена, он подумает, что вы сумасшедший (впрочем, насколько я знаю Манхэттен, вас и ваш вопрос, скорее всего, проигнорируют).

Конечно, люди, живущие в Марсе, штат Пенсильвания, хорошо осознают, что название их города очень необычное. Но часто они забывают о существовании красной планеты, и тогда может произойти следующий разговор:

Вы: Как мне добраться до Марса?

Местный житель: Ты в Марсе, приятель. Какая улица тебе нужна?

Вы: Нет, я имею в виду не город Марс, штат Пенсильвания, а планету Марс.

Местный житель: А-а, планету... Ну, тогда садитесь в поезд 8:19, идущий на мыс Канаверал... Впрочем, минутку. Это местный поезд. Он идет в Западную Виргинию.

Итак, значение имени зависит от того, в каком контексте оно употребляется. Большинство людей считают Марс местом с атмосферой из диоксида углерода, но некоторые жители штата Пенсильвания считают, что Марс — прекрасное место для шопинга. Для ребят из Пенсильвании понятие "Марс" имеет два значения, но оба они выражаются одним словом. А вот в Java, где не должно быть двусмысленности, они выражаются разными словами: `Mars` и `planets.Mars`.

Переменная на своем месте

Первый эксперимент будет с листингами 10.6 и 10.7. Полужирным шрифтом отмечены переменные `mars`, имеющие одинаковые имена, но объявленные внутри и вне метода `visitPennsilvania()`.

Листинг 10.6. Два значения слова "Марс"

```
import static java.lang.System.out;

class EnglishSpeakingWorld {
    String mars = "   красная планета";

    void visitPennsylvania() {
        out.println("Выполняется visitPennsylvania():");

        String mars = "   родной город Джанин";

        out.println(mars);
        out.println(this.mars);
    }
}
```

Листинг 10.7. Запуск кода листинга 10.6.

```
import static java.lang.System.out;

class GetGoing {

    public static void main(String args[]) {

        out.println("Выполняется main():");

        EnglishSpeakingWorld e = new EnglishSpeakingWorld();

        //out.println(mars);    // Неправильно.
                                // Неразрешимая ссылка.

        out.println(e.mars);
        e.visitPennsylvania();
    }
}
```

На рис. 10.7 показан результат выполнения листингов 10.6 и 10.7. На рис. 10.8 показана структура кода. Метод `main()` класса `GetGoing` создает экземпляр класса `EnglishSpeakingWorld`. Переменная `e` ссылается на этот экземпляр. В объекте типа `EnglishSpeakingWorld` объявлено поле `mars`. Ему присвоено значение `красная планета`. Запомните: это поле объекта.

```
<terminated> GetGoing [Java Application] D:\Pro
Выполняется main():
   красная планета
Выполняется visitPennsylvania():
   родной город Джанин
   красная планета
```

Рис. 10.7. Выполнение листингов 10.6 и 10.7

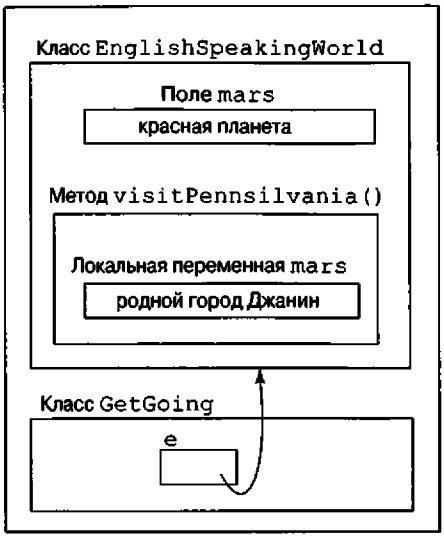


Рис. 10.8. Структура кода листингов 10.6 и 10.7



Нестатическое поле можно считать *переменной экземпляра*, потому что оно принадлежит экземпляру класса. Например, переменная `mars`, имеющая значение `красная планета`, принадлежит экземпляру класса `EnglishSpeakingWorld`. В противоположность этому статические поля, такие как `playerCount`, `totalOfAverages` и `decFormat` в листинге 10.4, можно считать *переменными класса*, потому что они принадлежат классу. Например, `playerCount` в листинге 10.4 является переменной класса, потому что существует только одна копия `playerCount`, принадлежащая классу.

Рассмотрим метод `main()` в листинге 10.7. В теле метода `main()` класса `GetGoing` нельзя написать `out.println(mars)`. Иными словами, любая ссылка без квалификаторов на переменную `mars` запрещена. Эта переменная принадлежит объекту типа `EnglishSpeakingWorld`, а не классу `GetGoing`.

Однако в теле метода `main()` класса `GetGoing` можно написать `e.mars`, потому что переменная `e` ссылается на объект `EnglishSpeakingWorld`.

В нижней части кода вызывается метод `visitPennsylvania()`. В теле этого метода объявлена еще одна переменная `mars`, имеющая значение `родной город Джанин`. Эта переменная называется *локальной*, потому что она принадлежит только методу `visitPennsylvania()`. Обращаться к ней можно только в теле этого метода. Ее область видимости — тело данного метода.

Итак, у нас есть две разные переменные с одним именем `mars`. Одна — нестатическое поле класса `EnglishSpeakingWorld`, имеющее значение `красная планета`. Вторая — локальная переменная метода `visitPennsylvania()`, имеющая значение `родной город Джанин`. Если в коде вы напишете `mars`, на какую из этих двух переменных вы ссылаетесь?

Когда вы в Пенсильвании (т.е. внутри метода `visitPennsilvania()`), вы ссылаетесь на локальную переменную, имеющую значение `родной город Джанин`. Имена одинаковые, но конфликта имен не существует, потому что в теле метода приоритет имеет переменная, объявленная здесь же — в теле метода.

Однако как быть, если в теле метода `visitPennsilvania()` нужно сослаться на красную планету — поле `mars` класса `EnglishSpeakingWord1`? Ответ прост: напишите `this.mars`. Ключевое слово `this` означает объект, которому принадлежит код в данной точке. Этим объектом является текущий экземпляр класса `EnglishSpeakingWord1`. В нем есть поле `mars`, имеющее значение `красная планета`. На это поле ссылается фраза `this.mars`. Обратите внимание на то, что `this.mars` можно написать за пределами метода `visitPennsilvania()`. Ошибки не будет, но префикс `this` здесь будет лишним.



Дополнительную информацию о ключевом слове `this` можно найти в главе 9.

Переменные в разных местах

Итак, переменная, полностью работоспособная в одном месте, в другом месте может оказаться полностью неработоспособной. Еще раз проиллюстрируем этот тезис с помощью немного измененного примера (листинги 10.8 и 10.9), в котором локальная переменная называется не `mars`, а `atomicCity`.

Листинг 10.8. Посетите Атомик-Сити, штат Айдахо. Что там знают о Марсе?

```
import static java.lang.System.out;

class EnglishSpeakingWorld2 {
    String mars;

    void visitIdaho() {
        out.println("Выполняется visitIdaho():");

        mars = "   красная планета";
        String atomicCity = "   население: 25 человек";

        out.println(mars);
        out.println(atomicCity);
    }

    void visitNewJersey() {
        out.println("Выполняется visitNewJersey():");

        out.println(mars);
        //out.println(atomicCity);
        //   Переменная atomicCity недоступна
    }
}
```

Листинг 10.9. Запуск листинга 10.8

```
class GetGoing2 {

    public static void main(String args[]) {
       EnglishSpeakingWorld2 e = new EnglishSpeakingWorld2();

       e.visitIdaho();
       e.visitNewJersey();
    }
}
```

На рис. 10.9 показан результат выполнения листингов 10.8 и 10.9. На рис. 10.10 показана структура кода. В классе `EnglishSpeakingWorld2` есть две переменные: поле `mars`, принадлежащее объекту, и локальная переменная `atomicCity`, принадлежащая методу `visitIdaho()`.

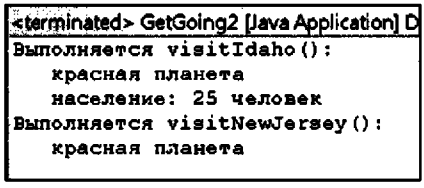


Рис. 10.9. Результат выполнения листингов 10.8 и 10.9

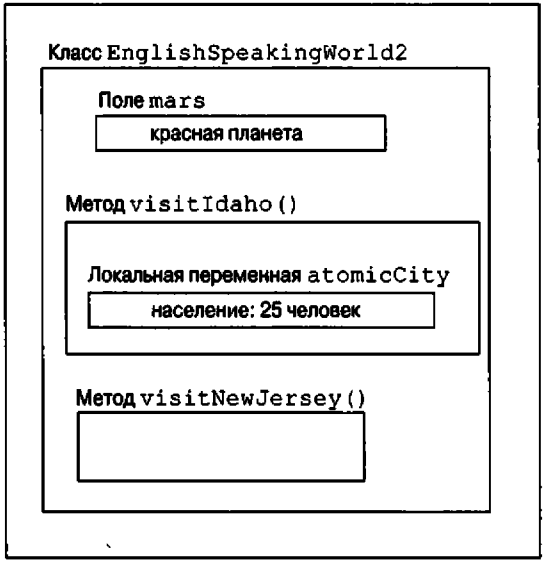


Рис. 10.10. Структура кода листингов 10.8 и 10.9

В листинге 10.8 обратите внимание на то, где можно, а где нельзя использовать каждую переменную. При попытке обратиться к переменной `atomicCity` в теле метода `visitNewJersey()` компилятор сообщит об ошибке. Выражаясь образно, Атомик-Сити находится в Айдахо, поэтому в Нью-Джерси о нем ничего

неизвестно. Выражаясь технически, переменная `atomicCity` видима только в теле метода `visitIdaho()`, потому что она является локальной переменной, объявленной в этом методе.

В теле метода `visitIdaho()` можно использовать локальную переменную `atomicCity`, потому что она объявлена в этом методе.

А как насчет Марса? Все в порядке. Поле `mars` видимо и может использоваться в обоих методах. Оно объявлено в классе как нестатическое, методы тоже нестатические, следовательно, для использования поля `mars` нет никаких препятствий. Выражаясь образно, и в Айдахо, и в Нью-Джерси знают о планете Марс, поэтому, находясь в этих штатах, можете упоминать о ней, не опасаясь, что вас не поймут.

В данном примере жизненный цикл поля `mars` состоит из трех этапов.

1. При создании экземпляра класса `EnglishSpeakingWorld2` компьютер видит объявление `String mars` и резервирует имя `mars` и ячейку памяти для хранения содержимого поля `mars`.
2. При выполнении метода `visitIdaho()` полю `mars` присваивается значение `красная планета`. Затем в теле этого же метода значение поля выводится на консоль.
3. При выполнении метода `visitNewJersey()` значение поля `mars` еще раз выводится на консоль.

Как видите, поле получило значение в одном методе, а используется полученное значение в другом методе. Если бы любой из методов был статическим, это было бы невозможно. Однако обратное не верно. Если в данном примере сделать поле `mars` статическим, то ничего не изменится.

Передача параметров

Метод может сообщаться с другими частями программы разными способами. Один из них — посредством полей класса — рассмотрен в предыдущем разделе. Сейчас рассмотрим еще один — передачу методу данных посредством списка параметров. Этот способ позволяет определить, какие значения должен обработать метод в момент его вызова.

Предположим, что информация, которую нужно передать методу в момент вызова, находится в некоторой переменной. Что метод делает с этой переменной? Рассмотрим несколько вариантов.

Передача по значению

Согласно муниципальному сайту городка Смаковер, штат Арканзас, в нем проживают 2232 человека. Но эти данные устарели. Как раз сегодня Дора Керменгос в коммунальной клинике Смаковера родила прелестную голубоглазую дочурку (вес — 7 фунтов и 4 унции, высота — 21 дюйм), и теперь население городка составляет 2233 человека.

В листинге 10.10 мы попытались увеличить значение переменной `smackoverARpop`, в которой хранится количество жителей Смаковера, но оно почему-то не увеличилось. Рассмотрим, почему так получилось.

Листинг 10.10. Эта программа не работает

```
class TrackPopulation {

    public static void main(String args[]) {
        int smackoverARpop = 2232;

        birth(smackoverARpop);
        System.out.println(smackoverARpop);
    }

    static void birth(int cityPop) {
        cityPop++;
    }
}
```

Запустите эту программу, и вы убедитесь в том, что она выводит на консоль число 2232, хотя, казалось бы, оно должно было увеличиться на единицу в теле метода `birth()`. Придется сообщить Доре, что нам не удалось зарегистрировать рождение дочурки. Боюсь, она будет разгневана. Но, может, еще не поздно исправить ошибку?

Проблема вызвана неправильным использованием передаваемого параметра. В Java при передаче методу в качестве параметра переменной одного из восьми примитивных типов параметр передается *по значению*.



Список примитивных типов Java приведен в главе 4.

На естественном языке это означает следующее. Когда параметр передается по значению, метод получает значение переменной, а не переменную. Переменной `cityPop` в момент вызова присваивается значение 2232. В теле метода `birth()` оператор ++ увеличивает переменную `cityPop` на единицу, но это никак не влияет на переменную `smackoverARpop`.

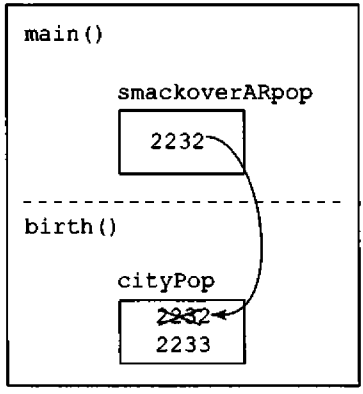


Рис. 10.11. Передача параметра по значению

Говоря техническим языком, копируется значение (рис. 10.11). Когда в методе `main()` код вызывает метод `birth()`, значение, хранящееся в ячейке памяти, на

которую указывает переменная smackoverARpop, копируется в другую ячейку памяти, на которую указывает переменная cityPop. При выполнении метода birth() увеличивается значение в другой ячейке, а не в первой. Значение в первой ячейке остается нетронутым.



При передаче переменной одного из восьми примитивных типов компьютер применяет передачу по значению. Значение, хранящееся в вызывающем методе, всегда остается незатронутым. Это происходит даже тогда, когда имена переменных в инструкции вызова и в определении вызывающего метода совпадают.

Возвращение результата

Проблему, возникшую в листинге 10.10, нужно как-то решать. В конце концов, юная Керменгос не может появиться на свет незамеченной. Для регистрации ее существования нужно добавить единицу к значению переменной smackoverARpop. Это можно сделать многими способами, и способ, представленный в листинге 10.11, не самый простой. Я применил его, только чтобы проиллюстрировать получение результата работы вызываемого метода.

Листинг 10.11. Эта программа работает

```
class TrackPopulation2 {

    public static void main(String args[]) {
        int smackoverARpop = 2232;

        smackoverARpop = birth(smackoverARpop);
        System.out.println(smackoverARpop);
    }

    static int birth(int cityPop) {
        return cityPop + 1;
    }
}
```

После запуска этой программы вы увидите на консоли правильное число — 2233.

Код листинга 10.11 не содержит ничего нового для нас (если не считать новинкой то, что он работает). Наиболее важная концепция, иллюстрируемая листингом 10.11, — ключевое слово return. Впрочем, с ним мы уже встречались в главе 7, и вы уже знаете, что после него расположено выражение, значение которого возвращается вызывающему методу. Выполнение метода завершается, как только компьютер доходит до инструкции return, где бы она ни была расположена. В теле метода может быть много инструкций return. Если метод объявлен как void, инструкция return, если она есть, не содержит никакого выражения, и после слова return ставится точка с запятой.

Передача по ссылке

Выше я уже несколько раз упомянул, что, если методу передается параметр одного из восьми примитивных типов, он всегда передается по значению. Обратите внимание

на то, что это касается только восьми примитивных типов, потому что при передаче объекта (т.е. переменной ссылочного типа) это утверждение неверное.

Объект передается методу *по ссылке*. Это означает, что инструкция в теле вызываемого метода **может** изменять поля объекта. Эти изменения сохраняются после завершения работы вызываемого метода, поэтому их можно видеть в теле вызывающего метода. Данная концепция иллюстрируется листингами 10.12 и 10.13. Объект типа `City` (Город) имеет поле `population` (население).

Листинг 10.12. Что такое город

```
class City {
    int population;
}
```

Листинг 10.13. Передача объекта методу

```
class TrackPopulation3 {

    public static void main(String args[]) {
        City smackoverAR = new City();
        smackoverAR.population = 2232;
        birth(smackoverAR);
        System.out.println(smackoverAR.population);
    }

    static void birth(City aCity) {
        aCity.population++;
    }
}
```

В результате выполнения листингов 10.12 и 10.13 на консоль будет выведено число 2233. Следовательно, мы увеличили количество жителей Смаковера на единицу. Увеличение поля `smackoverAR.population` произошло (причем под другим именем) в теле метода `birth()`, тем не менее оно видно в теле метода `main()`.

Чтобы увидеть, как метод `birth()` изменил значение `smackoverAR.population`, посмотрите на рис. 10.12. Когда вы передаете объект методу (естественно, по ссылке), компьютер не создает копию всего объекта. Он создает копию лишь ссылки на данный объект (иными словами, копию адреса данного объекта). Следовательно, вызываемый метод, имея весь объект в своем полном распоряжении, может делать с ним все, что ему заблагорассудится, несмотря даже на то, что внутри метода данный объект имеет другое имя.

На рис. 10.12 вы видите только один экземпляр класса `City` (так оно и есть на самом деле) и принадлежащее ему поле `population`. Жизненный цикл этого объекта включает два переименования и состоит из следующих этапов.

1. Перед вызовом метода `birth()` переменная `smackoverAR` ссылается на экземпляр класса `City`.
2. В момент вызова метода `birth()` ему передается переменная `smackoverAR`, содержащая адрес объекта. Адрес присваивается переменной `aCity`. С этого

момента объект называется aCity. Компьютер создает копию адреса. Копию объекта он не создает, и обе переменные ссылаются на один и тот же объект.

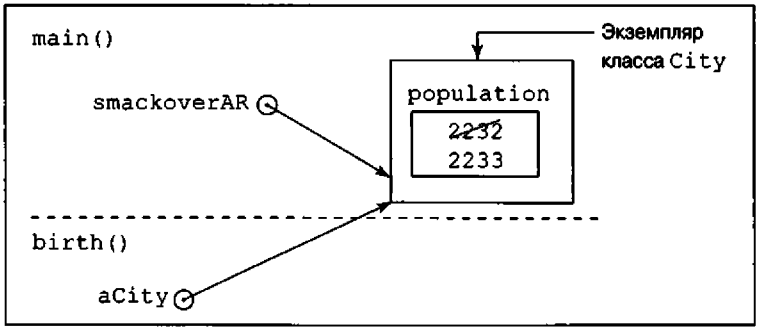


Рис. 10.12. Передача по ссылке; в методе `main()` объект называется `smackoverAR`, а в методе `birth()` — `aCity`

3. В теле метода `birth()` выполняется инструкция `aCity.population++`, которая увеличивает значение поля `population` на единицу. Теперь поле `population` объекта типа `City` имеет значение 2233. Заметим, что это единственный объект типа `City`.
4. Выполнение метода `birth()` завершается, и управление возвращается в метод `main()`. Значение `population` остается тем же, каким оно было в теле метода `birth()`. Это значение выводится на консоль. Оно равно 2233. Керменгосы могут быть довольными.

Возвращение объекта из метода

Вызываемый метод может вернуть что-нибудь в вызывающий метод. В предыдущих главах мы возвращали примитивные типы (такие, как `int`) или ничего не возвращали (`void`). В данном разделе мы вернем целый объект типа `City` (листинг 10.14).

Листинг 10.14. Метод `doBirth()` возвращает объект типа `City`

```
class TrackPopulation4 {

    public static void main(String args[]) {
        City smackoverAR = new City();
        smackoverAR.population = 2232;
        smackoverAR = doBirth(smackoverAR);
        System.out.println(smackoverAR.population);
    }

    static City doBirth(City aCity) {
        City myCity = new City();
        myCity.population = aCity.population + 1;
        return myCity;
    }
}
```

Выполнив листинг 10.14, вы получите на консоли число 2233, как и должно быть. Метод doBirth() создает еще один объект типа City. В нем значение поля population равно 2233 (рис. 10.13).

После выполнения метода doBirth() он возвращает новый экземпляр класса City методу main(). Этот новый объект присваивается переменной smackoverAR, которая прежде ссылалась на старый объект типа City (рис. 10.14). Таким образом, на старый объект теперь ничто не ссылается, он как бы "зависает" в воздухе, и обратиться к нему уже невозможно. А объект smackoverAR теперь содержит поле population с новым значением 2233. О дальнейшей судьбе старого объекта можете не беспокоиться: сборщик мусора автоматически уничтожит его, чтобы он не занимал зря память.

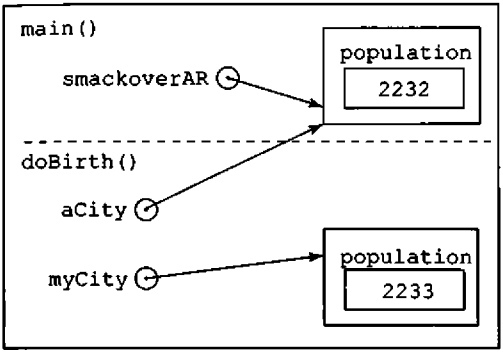


Рис. 10.13. Метод doBirth() создает объект типа City

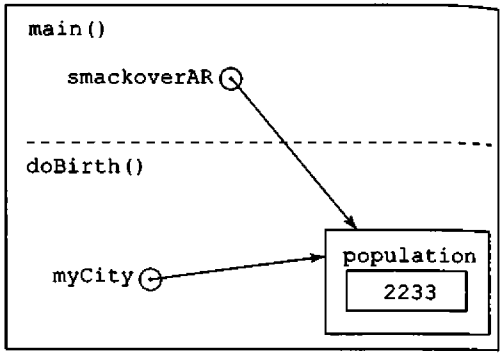


Рис. 10.14. Новый объект присваивается переменной smackoverAR

В листинге 10.14 обратите внимание на соответствие типов возвращаемого значения в вызываемом и вызывающем методах.

- ✓ В вызывающем методе переменная smackoverAR имеет тип City. Она передается параметру aCity, который также имеет тип City.
- ✓ В вызываемом методе переменная myCity также имеет тип City. Она же возвращается вызывающему методу с помощью инструкции return. Это правильно, поскольку метод doBirth() объявлен с типом City (в заголовке определения метода перед его именем находится слово City).
- ✓ Метод doBirth() возвращает объект типа City. В методе main() этот объект присваивается переменной smackoverAR, которая, как вы уже догадались, также имеет тип City.

Компилятор проверяет совпадение типов везде, где они встречаются. Если типы где-либо не совпадут, компилятор не допустит программу на выполнение и вернет сообщение об ошибке incompatible types (несовместимые типы).

Эпилог

Дора и ее новорожденная дочурка благополучно вернулись из клиники и живут счастливо в своем доме в Смаковере, штат Арканзас.

Глава 11

Массивы и коллекции

В этой главе...

- Как расположить предметы в ряд
- Массивы объектов
- Аргументы командной строки
- Коллекции

Добро пожаловать в мотель Java! У нас вы найдете умеренные цены, чистые двухместные номера и полный комплекс услуг для путешественников и отдыхающих.

Как расположить предметы в ряд

Мотель Java состоит из десяти комфортабельных комнат и расположен в уютном местечке на достаточном расстоянии от оживленной трассы. Кроме небольшого отдельного офиса, мотель включает в себя ряд одноэтажных строений. Каждая комната легко доступна с просторной парковки.

От обычных мотелей мотель Java отличается тем, что комнаты нумеруются не с единицы, а с нуля. В обычной жизни принято считать предметы, начиная с единицы, однако в Java принято нумеровать все, начиная с нуля. Вам будет нелегко привыкнуть к этому, может, вы даже будете возмущаться этой странностью, однако вам ничего не остается, кроме как привыкнуть, и через месяц вы уже не будете замечать в этом что-либо необычное. Таким образом, номера отеля Java пронумерованы от 0 до 9.

Необходимо как-то отслеживать количество проживающих в каждой комнате. Поскольку есть десять номеров, первое, что приходит в голову, — объявить десять переменных и хранить в каждой количество постояльцев в конкретной комнате.

```
int guestsInRoomNum0, guestsInRoomNum1, guestsInRoomNum2,
    guestsInRoomNum3, guestsInRoomNum4, guestsInRoomNum5,
    guestsInRoomNum6, guestsInRoomNum7, guestsInRoomNum8,
    guestsInRoomNum9;
```

Как видите, получилось не очень хорошо. К тому же что произойдет, если в мотеле будет не десять, а сто комнат? Но главный недостаток данного подхода состоит в том, что он не позволяет обрабатывать переменные в цикле. Например, чтобы прочитать их значения из файла, необходимо десять раз вызвать метод `nextInt()`.

```
guestsInRoomNum0 = distScanner.nextInt();
guestsInRoomNum1 = distScanner.nextInt();
guestsInRoomNum2 = distScanner.nextInt();
...
```

Наверняка должен существовать лучший способ. И он существует. Для хранения ряда значений в Java используется *массив*. Представляйте его как ряд комнат в мотеле.

- ✓ Комнаты в мотеле расположены в ряд. Изобразим их на карте в виде ряда квадратиков.
- ✓ В каждый квадратик впишем количество постояльцев, а рядом с каждым квадратиком напишем номер комнаты.
- ✓ Эти два ряда чисел (рис. 11.1) — это текущая информация о мотеле, т.е. его состояние на данный момент. Конечно, реальная жизнь значительно сложнее. В комнате 6 испортился телевизор, в комнате 9 постояльцы режут барашка и собираются делать шашлык, постояльцы комнаты 2 не хотят платить и т.п. Все эти проблемы нужно как-то решать, однако компьютеру это пока что неинтересно, потому что ему поставлена более простая задача: отслеживание количества постояльцев.

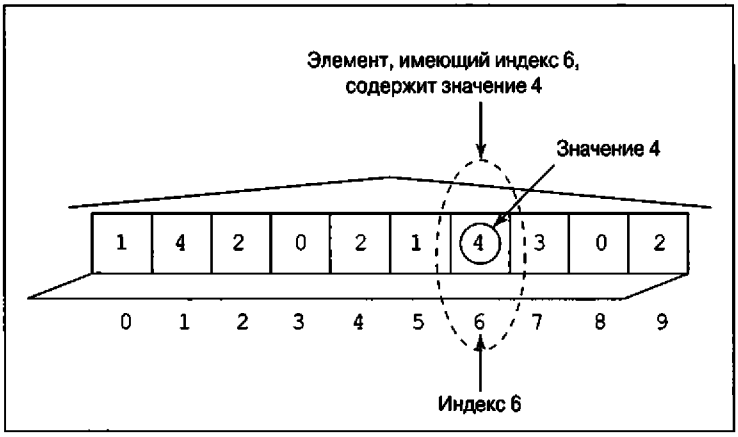


Рис. 11.1. Абстрактное представление мотеля

Согласно принятой в Java терминологии ряд комнат нашего воображаемого мотеля называется *массивом*. Каждая комната называется *элементом массива*. С каждым элементом массива ассоциированы два числа.

- ✓ Номер комнаты — это *индекс* данного элемента массива
- ✓ Количество постояльцев, проживающих в данной комнате, — это *значение* элемента массива

Использование массивов избавляет от скучной необходимости повторять код. Например, чтобы объявить массив, состоящий из десяти элементов (и, соответственно, содержащий десять значений), достаточно одной короткой строки.

```
int guests[] = new int[10];
```

В приведенной выше инструкции совмещены операции объявления и создания массива. В некоторых случаях лучше выполнять эти операции в двух разных инструкциях.

```
int guests[];
guests = new int[10];
```

В обоих примерах обратите внимание на использование целочисленного литерала `10`. Он сообщает компилятору о том, что массив `guests` должен содержать десять элементов. Каждый элемент массива занимает собственную ячейку памяти. Кроме того, каждый элемент имеет собственное имя. Первый элемент имеет имя `guests[0]`, второй — `guests[1]` и т.д.



При создании массива нужно задать количество элементов. Элементы массива нумеруются с нуля. Следовательно, последний элемент имеет номер, на единицу меньший, чем количество элементов.



Выше приведены два способа создания массива. В первом способе используется одна инструкция, а во втором — две. При создании массива с помощью одной инструкции она может быть расположена внутри или за пределами метода. Однако при использовании второго способа вторая инструкция `guests=new int[10]` обязательно должна быть расположена внутри метода.



В объявлении массива квадратные скобки можно ввести после имени типа или после имени переменной. Например, можно написать `int guests[]` или `int[] guests`. Результат будет один и тот же: компилятор создаст переменную массива `guests`.

Создание массива в два этапа

Посмотрите еще раз на две инструкции, создающие массив.

```
int guests[];
guests = new int[10];
```

Каждая инструкция решает свою задачу.

- ✓ **`int guests[]`**. Объявление переменной массива. Имя `guests` резервируется компилятором для использования в программе. В примере с мотелем Java эта инструкция как бы говорит: "Я планирую построить здесь мотель и разместить в каждой комнате некоторое количество постояльцев, количество комнат пока что неопределенное". Эта ситуация проиллюстрирована на рис. 11.2, *вверху*.

  Пока что можете не беспокоиться о том, что фактически делает объявление `int guests[]`. Намного важнее понять, чего оно **не делает**. Оно не резервирует ячейки памяти для десяти переменных. Оно не создает массив. Оно всего лишь резервирует имя переменной `guests` и сообщает о том, что это будет переменная целочисленного массива. В данный момент (непосредственно после объявления) переменная `guests` не ссылается на массив. Иными словами, мотель имеет название, но самого мотеля еще нет.

- ✓ **`guests=new int[10]`**. Эта инструкция выполняется в два этапа. Сначала оператор `new` создает объект массива из десяти элементов и резервирует для них десять ячеек памяти, а затем объект массива присваивается переменной `guests`. Иными словами, мотель уже построен, но в комнатах еще нет постояльцев (рис. 11.2, *внизу*).

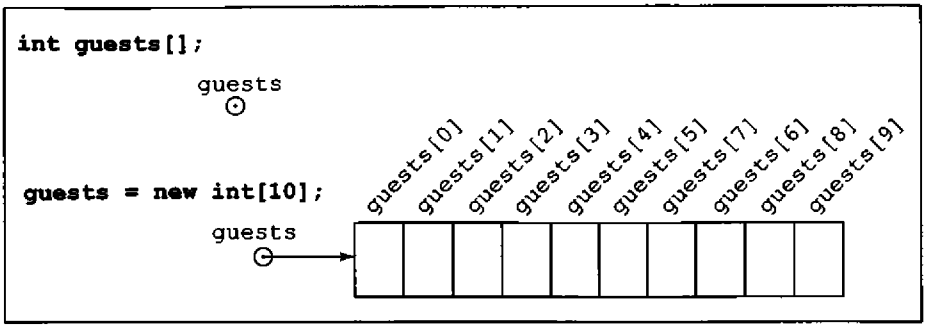


Рис. 11.2. Создание массива с помощью двух инструкций

Сохранение значений

После создания массива в его элементах можно сохранять значения. Это делается с помощью оператора присваивания. Например, чтобы сохранить в элементе 6 значение 4 (т.е. отобразить тот факт, что в шестой комнате проживают четыре постояльца), напишите инструкцию `guests[6]=4`.

Предположим, бизнес успешно развивается. На стоянку мотеля въезжает большой автобус туристической фирмы. Из него вываливает шумная толпа туристов, 25 пар, которых нужно где-то разместить. В мотеле Java есть только десять комнат, но эта проблема легко решается. В нескольких милях от мотеля Java находятся еще несколько небольших мотелей, куда можно направить автобус и не поместившиеся 15 пар.

Чтобы зарегистрировать 10 пар в мотеле Java, нужно записать число 2 в каждый элемент массива `guests`. Для этого нам не придется писать десять инструкций. Всю работу можно проделать в одном цикле.

```
for (int roomNum = 0; roomNum < 10; roomNum++) {
  guests[roomNum] = 2;
}
```

Этот цикл заменяет десять инструкций. Обратите внимание на то, что значение счетчика цикла `roomNum` изменяется от 0 до 9. Взгляните еще раз на рис. 11.2. Как вы помните, нумерация массива начинается с нуля, поэтому индекс последнего элемента массива на единицу меньше количества элементов.

Однако в реальности туристы не прибывают исключительно по двое. В каждой комнате может проживать произвольное количество человек. Обычно информация о каждой комнате и количестве проживающих в ней постояльцев находится в базе данных мотеля. Ниже приведен код, заполняющий массив `guests` информацией из базы данных. В каждый элемент массива заносится количество постояльцев.

```
resultset = statement.executeQuery("select GUESTS from RoomData");
for (int roomNum = 0; roomNum < 10; roomNum++) {
  resultset.next();
  guests[roomNum] = resultset.getInt("GUESTS");
}
```

В данной книге базы данных рассматриваются, только начиная с главы 16, поэтому пока что будем извлекать количество посетителей из текстового файла `GuestList`.

txt. Его содержимое показано на рис. 11.3. Когда файл GuestList.txt находится на диске, хранящиеся в нем числа можно извлечь в программу с помощью класса Scanner. Создать этот файл можно с помощью любого текстового редактора, например блокнота Windows. Код программы показан в листинге 11.1, а результат выполнения — на рис. 11.4.

```
1 4 2 0 2 1 4 3 0 2
```

Рис. 11.3. Содержимое файла GuestList.txt

Листинг 11.1. Заполнение массива значениями

```
import static java.lang.System.out;
import java.util.Scanner;
import java.io.File;
import java.io.IOException;

class ShowGuests {

    public static void main(String args[]) throws IOException {
        int guests[] = new int[10];
        Scanner diskScanner = new Scanner(new File("GuestList.txt"));

        for(int roomNum = 0; roomNum < 10; roomNum++) {
            guests[roomNum] = diskScanner.nextInt();
        }

        out.println("Комната\tКоличество");

        for(int roomNum = 0; roomNum < 10; roomNum++) {
            out.print(roomNum);
            out.print("\t");
            out.println(guests[roomNum]);
        }
    }
}
```

```
<terminated> ShowGuests [Ja
Комната Количество
0       1
1       4
2       2
3       0
4       2
5       1
6       4
7       3
8       0
9       2
```

Рис. 11.4. Результат выполнения листинга 11.1

В листинге 11.1 есть два цикла. В первом цикле программа считывает из файла количество постояльцев в каждой комнате, а во втором выводит содержимое массива `guests` на консоль.



Массив — это объект. В каждом объекте массива есть встроенное поле `length` (длина), содержащее количество элементов массива. Следовательно, в листинге 11.1 можно пользоваться выражением `guests.length`, которое в данном случае равно десяти.

Табулостопы и другие специальные символы

В листинге 11.1 используются *управляющие последовательности символов* `\t` (другое название — *Esc-последовательность*). В строковых литералах Java они применяются для обозначения специальных символов, которые либо не видны на экране, либо отсутствуют на клавиатуре, либо не могут быть включены в строковый литерал по некоторым причинам. Например, в строковый литерал нельзя включить символы одинарной или двойной кавычки, потому что они обозначают конец строки. Управляющая последовательность `\t` обозначает табулостоп — символ, невидимый на экране и задающий отступ следующего символа. Табулостопы были очень популярны в первые годы компьютерной эпохи. Тогда они имитировали средства табуляции, встроенные в механические пишущие машинки. В настоящее время уже никто не помнит, как работали механические пишущие машинки, поэтому табулостопы сейчас почти не применяются. В табл. 11.1 приведены наиболее популярные управляющие последовательности.

Таблица 11.1. Управляющие последовательности

Последовательность	Назначение
`\b`	Отмена последнего символа
`\t`	Горизонтальная табуляция
`\n`	Подача строки
`\f`	Подача формы
`\r`	Возврат каретки
`\"`	Двойная кавычка
`\'`	Одинарная кавычка
`\\`	Обратная косая черта

Инициализация массива

Кроме способа, представленного в листинге 11.1, в Java есть еще один способ заполнения массива числами. Это можно сделать путем его инициализации. При этом можно не сообщать компьютеру, сколько элементов должен иметь массив, потому что компилятор сам подсчитает количество элементов.

В листинге 11.2 представлена альтернативная версия кода, заполняющего массив числами. Вывод содержимого массива на консоль выполняется так же, как и в листинге 11.1, а результат — тот же, что на рис. 11.4. Единственное отличие состоит в

том, что в листинге 11.2 массив инициализируется при объявлении. Код инициализации отмечен в листинге полужирным шрифтом.

Листинг 11.2. Инициализация массива guests

```
import static java.lang.System.out;

class ShowGuests {

    public static void main(String args[]) {
        int guests[] = {1, 4, 2, 0, 2, 1, 4, 3, 0, 2};

        out.println("Комната\tКоличество");

        for (int roomNum = 0; roomNum < 10; roomNum++) {
            out.print(roomNum);
            out.print("\t");
            out.println(guests[roomNum]);
        }
    }
}
```



В инициализаторе можно использовать не только литералы, но и любые выражения. Например, можно написать так: `{1+3, keyboard.nextInt(), 2, 0, 2, 1, 4, 3, 0, 2}`. Естественно, при этом нужно следить, чтобы все компоненты выражений в момент инициализации были определены.

Расширенный цикл for

В Java доступен вариант цикла `for`, в котором не обязательно использовать индексы или счетчики (листинг 11.3).



Рассматриваемый в данном разделе вариант цикла `for` доступен, только начиная с версии JRE 5. В более старых версиях JRE он не работает. Номера версий Java рассматривались в главе 2.

Листинг 11.3. Расширенный вариант цикла for

```
import static java.lang.System.out;

class ShowGuests {

    public static void main(String args[]) {
        int guests[] = {1, 4, 2, 0, 2, 1, 4, 3, 0, 2};
        int roomNum = 0;

        out.println("Комната\tКоличество");
        for (int numGuests : guests) {
            out.print(roomNum++);
            out.print("\t");
```

```
            out.println(numGuests);
        }
    }
}
```

Листинги 11.1 и 11.3 приводят к одному и тому же результату, показанному на рис. 11.4.

Заголовок цикла в листинге 11.3 состоит из трех частей.

```
for (тип_переменной имя_переменной : диапазон_значений)
```

В цикле, приведенном в листинге 11.3, определена переменная `numGuests` типа `int`. В каждой итерации цикла она принимает новое значение. На рис. 11.4 ее значения в разных итерациях приведены в столбце Количество.

Где цикл находит значения `numGuests`? В выражении *диапазон_значений*. В листинге 11.3 в качестве диапазона значений используется переменная массива `guests`. Цикл проходит по элементам диапазона значений. Индекс массива, заданного в качестве диапазона значений, служит неявным счетчиком цикла. В первой итерации компьютер неявно присваивает переменной `numGuests` значение `guests[0]`, которое равно 1, во второй итерации — значение `guests[1]`, которое равно 4, и т.д.



При использовании расширенного цикла `for` нужно быть осторожным. Учитывайте, что в каждой итерации в переменной цикла `numGuests` сохраняется копия одного из значений диапазона `guests`. Переменная `numGuests` не указывает ни на диапазон, ни на его текущее значение.

Что будет, если в теле цикла присвоить переменной `numGuests` какое-либо значение? Изменится только значение `numGuests`. Инструкция присваивания не повлияет на содержимое массива `guests`. Предположим, например, что дела идут плохо, комнаты мотеля не заполнены и каждый элемент массива `guests` равен нулю. Выполним следующий код.

```
for (int numGuests : guests) {
  numGuests += 1;
  out.print(numGuests + " ");
}
out.println();
  for (int numGuests : guests) {
  out.print(numGuests + " ");
}
```

Ниже приведен результат его выполнения.

```
1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0
```

На каждой итерации переменная `numGuests` сначала получает значение 0, но затем инструкция, отмеченная полужирным шрифтом, увеличивает его на единицу. Однако на массив `guests` это не повлияло, и в нем по-прежнему находятся только нули.

Поиск

Вы сидите за стойкой мотеля Java, когда прибывает новая партия туристов (пять человек). Они хотят снять комнату, и вам нужно проверить, есть ли свободная комната, т.е. существует ли элемент массива `guests`, значение которого равно нулю. Иными словами, если в массиве `GuestsList.txt` (см. рис. 11.3) есть число 0, его нужно найти и заменить числом, которое вы вводите с клавиатуры. Программа, выполняющая эту работу, приведена в листинге 11.4.

Листинг 11.4. Есть ли у вас свободная комната?

```
import static java.lang.System.out;
import java.util.Scanner;
import java.io.File;
import java.io.IOException;
import java.io.PrintStream;

class FindVacancy {

    public static void main(String args[]) throws IOException {
        Scanner keyboard = new Scanner(System.in);
        Scanner diskScanner = new Scanner(new File("GuestList.txt"));
        int guests[] = new int[10];
        int roomNum;

        for (roomNum = 0; roomNum < 10; roomNum++) {
            guests[roomNum] = diskScanner.nextInt();
        }

        roomNum = 0;
        while (roomNum < 10 && guests[roomNum] != 0) {
            roomNum++;
        }

        if (roomNum == 10) {
            out.println("Извините, свободных " + "комнат нет.");
        } else {
            out.print("Сколько человек " + "поселятся в комнате ");
            out.print(roomNum);
            out.print("? ");
            guests[roomNum] = keyboard.nextInt();

            PrintStream listOut = new PrintStream("GuestList.txt");

            for (roomNum = 0; roomNum < 10; roomNum++) {
                listOut.print(guests[roomNum]);
                listOut.print(" ");
            }
        }
    }
}
```

Результаты выполнения листинга 11.4 показаны на рис. 11.5–11.7. На каждом из этих рисунков слева показано состояние файла `GuestList.txt` перед запуском программы, а справа — консоль с результатами выполнения программы. Сначала свободны комнаты 3 и 8 (как вы помните, нумерация комнат начинается с нуля). При первом запуске листинга 11.4 компьютер сообщает, что свободна комната 3, и поселяет в нее 5 человек. На втором запуске программа сообщает, что свободна комната 8, и поселяет в нее 10 человек (почти футбольная команда!). На третьем запуске программа сообщает, что свободных комнат не осталось.

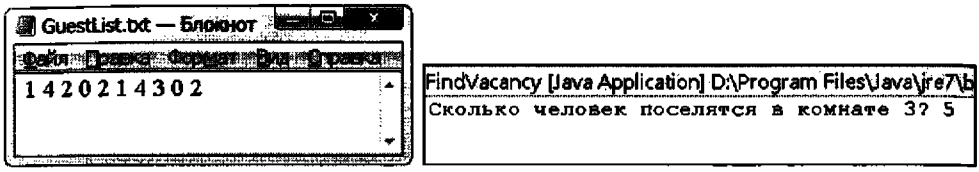


Рис. 11.5. Заполнение первой свободной комнаты

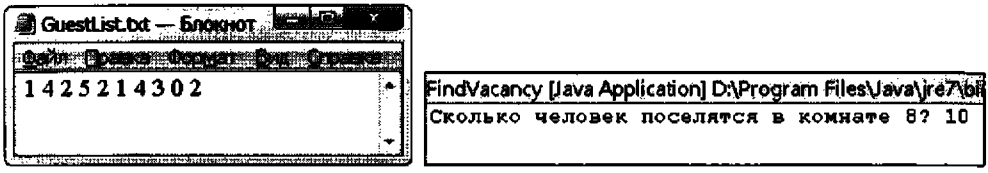


Рис. 11.6. Заполнение второй свободной комнаты

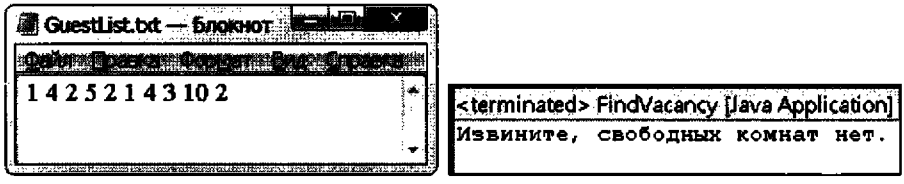


Рис. 11.7. Свободных комнат больше нет



При каждом своем запуске программа записывает заново файл `GuestList.txt`. Учитывайте, что программа Блокнот при изменении содержимого файла не обновляет изображение в своем окне. Нужно обновить его вручную. Для этого выберите в меню команду **Файл⇨Открыть**. Можете также открыть файл `GuestList.txt` в окне редактора Eclipse, но и в этом случае вам придется обновлять изображение вручную.

В листинге 11.4 используются приемы и трюки, описанные в других главах книги. Единственное в этом листинге новое для вас средство — класс `PrintStream`, предназначенный для записи данных в файл. Он работает так же, как знакомый вам класс `System.out`, записывающий данные на консоль.

При записи на консоль используется объект `System.out`, определенный в Java API. Фактически он является экземпляром класса `java.io.PrintStream` (для друзей — просто `PrintStream`). В каждом объекте класса `PrintStream` есть методы

print() и println(). В предыдущих главах мы не раз использовали метод System.out.println(), но до сих пор вы не знали, что он принадлежит экземпляру класса PrintStream.

Таким образом, System.out — это библиотечный объект типа PrintStream, выводящий текст на консоль. Если же вы сами создадите объект типа PrintStream, он будет ссылаться на заданный вами файл на диске. С помощью принадлежащего ему метода print() можно записывать текст в файл на диске.

Рассмотрим следующие инструкции, использовавшиеся в листинге 11.4.

```
PrintStream listOut = new PrintStream("GuestList.txt");
listOut.print(guests[roomNum]);
listOut.print(" ");
```

Первая инструкция сообщает компьютеру о том, что объект listOut — это файл GuestList.txt, вторая записывает в этот файл число guests[roomNum], а третья записывает в этот файл пробел.

С помощью объекта типа PrintStream программа обновляет количество постояльцев, хранящееся в файле GuestList.txt. При каждом запуске программы заново записывается весь файл, несмотря на то что изменилось только одно число. При первом запуске программа ищет нуль и находит его в третьей позиции. После этого программа спрашивает у пользователя, какое число записать в эту позицию, и записывает его. При втором запуске программа опять ищет нуль и на этот раз находит его уже в восьмой позиции, потому что в третьей позиции уже не нуль, а пятерка. И наконец, при третьем запуске программа не находит ни одного нуля.



Если конструктор объекта PrintStream не находит указанный файл, он его создает. Если же указанный файл существует, конструктор по умолчанию уничтожает его и подготавливает запись в файл с таким же именем. При необходимости можно задать запись нового содержимого в конец существующего файла.



Это, скорее, важное замечание, чем совет. Предположим, вам нужно **читать** данные из файла Employees.txt. Для этого вы создаете сканер с помощью конструктора new Scanner(new File("Employees.txt")). Если вы ошибочно упустите конструктор File, то конструктор Scanner не свяжет файл Employee.txt со сканером. Теперь для сравнения предположим, что нужно **записывать** данные в файл. Для этого вы создаете объект типа PrintStream с помощью конструктора new PrintStream("GuestList.txt"). Обратите внимание на то, что на этот раз конструктор файла new File() не нужен. Если вставить его, компилятор сообщит об ошибке и откажется запускать программу на выполнение.



В листинге 11.4 используется условие roomNum<10&&guests[roomNum]!=0. На первый взгляд, оно безобидное, но на самом деле оно весьма коварное. Если поменять выражения местами и написать guests[roomNum]!=0&&roomNum<10, программа будет завершена аварийно. Дело в том, что по правилам Java сначала вычисляется первое выражение. Если оно ложное, второе не вычисляется (зачем зря тратить время?). Рассмотрим, что происходит, когда переменная roomNum равна 10. В первом случае

выражение `roomNum<10` оказывается ложным, и второе выражение не вычисляется. Однако во втором случае сначала вычисляется выражение `guests[roomNum]!=0`. Элемента с номером 10 не существует, поэтому операционная система генерирует исключение и аварийно завершает выполнение программы.

Массивы объектов

Бизнес успешно развивается, и нам нужно подумать об усовершенствовании программного обеспечения для мотеля Java. В конце концов, программа должна уметь делать еще что-нибудь, кроме сохранения в файле количества постояльцев. Не забывайте, что мы работаем с объектно-ориентированным языком программирования и должны использовать объекты. Поэтому введем в рассмотрение объект типа `Room` (Комната).

Реальная комната мотеля имеет три свойства: количество постояльцев, тариф и допустимость курения (комната может быть предназначена для курящих или некурящих). Соответственно, определим три поля, которые могут иметь разные значения для каждого экземпляра класса: поле `guests` (количество постояльцев) типа `int`, поле `rate` (тариф) типа `double` и поле `smoking` (курение) типа `boolean`. Кроме того, определим статическое поле `currency` (валюта), которое будет общим для всех экземпляров класса `Room`. Наличие поля `currency` не означает, что одни постояльцы могут расплачиваться долларами, а другие — рублями. Статическое поле является общим для всех объектов, поэтому все постояльцы должны расплачиваться либо долларами, либо рублями. Описанная выше структура класса `Room` показана на рис. 11.8.

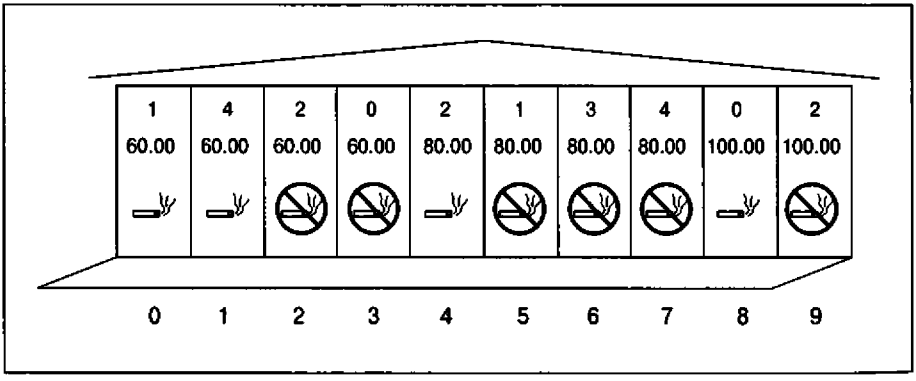


Рис. 11.8. С каждой комнатой ассоциированы три ее свойства

Код класса `Room` приведен в листинге 11.5. Как и обещано, каждый экземпляр класса имеет три поля: `guests`, `rate` и `smoking`. Значение `false` поля `smoking` означает, что данная комната предназначена для некурящих. Поле `currency`, имеющее тип `NumberFormat` и модификатор `static`, определяет форматирование чисел. В данном примере оно задает вывод символа доллара.



Статические поля рассматриваются в главе 10.

Листинг 11.5. Класс Room

```
import static java.lang.System.out;
import java.util.Locale;
import java.util.Scanner;
import java.text.NumberFormat;

public class Room {
    private int guests;
    private double rate;
    private boolean smoking;
    private static NumberFormat currency =
        NumberFormat.getCurrencyInstance(Locale.US);

    public void readRoom(Scanner diskScanner) {
        guests = diskScanner.nextInt();
        rate = diskScanner.nextDouble();
        smoking = diskScanner.nextBoolean();
    }

    public void writeRoom() {
        out.print(guests);
        out.print("\t");
        out.print(currency.format(rate));
        out.print("\t\t");
        out.println(smoking ? "да" : "нет");
    }
}
```

В листинге 11.5 применяется несколько интересных трюков, но мы рассмотрим их позже, когда увидим весь код в действии. Сейчас у нас есть класс Room, и мы можем создать массив объектов типа Room.



Это предупреждение уже неоднократно встречалось в главах 4, 7 и других, но ввиду его важности не лишним будет повторить его еще раз. Будьте очень осторожны при хранении денежных значений в переменных с плавающей точкой (типа double или float). При их использовании результат вычисления может отличаться от точного в десятом знаке, что может привести к неожиданным, тяжело предсказуемым последствиям.



Данный совет не имеет никакого отношения к Java. Если вы предпочитаете комнату для курящих (поле smoking в листинге 11.5 имеет значение true), найдите некурящего человека, который готов провести с вами три дня, и поселитесь с ним в комнате для некурящих. Через три дня вы узнаете, что такое счастье.

Использование класса Room

Код, создающий массив комнат, приведен в листинге 11.6. Программа считывает данные из файла RoomList.txt, содержимое которого показано на рис. 11.9. На рис. 11.10 показан результат выполнения листинга 11.6.

Листинг 11.6. Использование класса Room

```
import static java.lang.System.out;
import java.util.Scanner;
import java.io.File;
import java.io.IOException;

class ShowRooms {

    public static void main(String args[]) throws IOException {
        Room rooms[];
        rooms = new Room[10];

        Scanner diskScanner = new Scanner(new File("RoomList.txt"));

        for (int roomNum = 0; roomNum < 10; roomNum++) {
            rooms[roomNum] = new Room();
            rooms[roomNum].readRoom(diskScanner);
        }

        out.println("Комната\tКолич.\tТариф\t\t" + "Для курящих");
        for (int roomNum = 0; roomNum < 10; roomNum++) {
            out.print(roomNum);
            out.print("\t");
            rooms[roomNum].writeRoom();
        }
    }
}
```

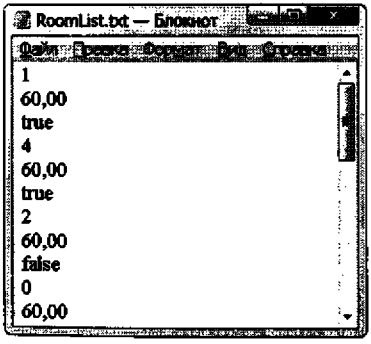


Рис. 11.9. Фрагмент файла RoomList.txt с данными о комнатах

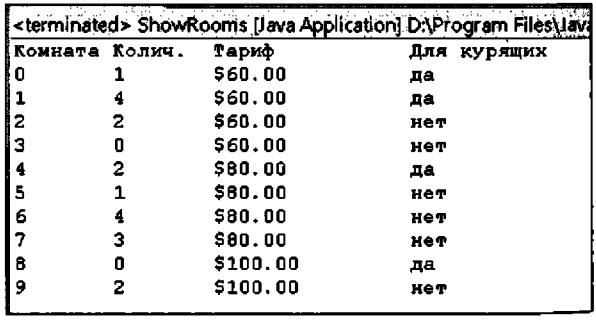


Рис. 11.10. Результат выполнения листинга 11.6

В листинге 11.6 нас пока что интересует только создание массива объектов типа Room. Мы уже создавали массивы переменных примитивного типа. Создать массив объектов немного сложнее. Для этого нужно выполнить три операции: объявить переменную массива, создать массив (пока что пустой) и, наконец, создать отдельный объект для каждого элемента массива. При создании массива значений примитивного типа (например, типа int) нужно выполнить только две первые операции.

Чтобы понять приведенное ниже объяснение, посматривайте на листинг 11.6 и рис. 11.11, иллюстрирующий процесс создания массива объектов.

- ✓ **`Room rooms[];`** Объявление переменной массива `rooms`. Эта переменная ссылается на массив (не на элемент массива и не на объект типа `Room`, а именно на массив как целое). Пока что массива нет, и переменная `rooms` имеет значение `null`. Фактически она пока что ни на что не ссылается.
- ✓ **`rooms = Room[10];`** Эта инструкция резервирует десять ячеек памяти, в которых будут храниться ссылки на объекты типа `Room`. Теперь переменная `rooms` ссылается на группу из десяти ячеек памяти, но они, в свою очередь, пока что содержат значения `null` и ни на что не ссылаются.
- ✓ **`rooms[roomNum] = new Room();`** Эта инструкция находится в теле цикла `for`. Она выполняется по одному разу для каждого элемента массива и создает для него объект типа `Room`. Например, в первой итерации она выглядит как `rooms[0]=new Room()` и присваивает нулевому элементу массива объект типа `Room`. Таким образом, в ячейке `room[0]` остается ссылка на экземпляр класса `Room`. Всего данная инструкция создает десять объектов `Room`.

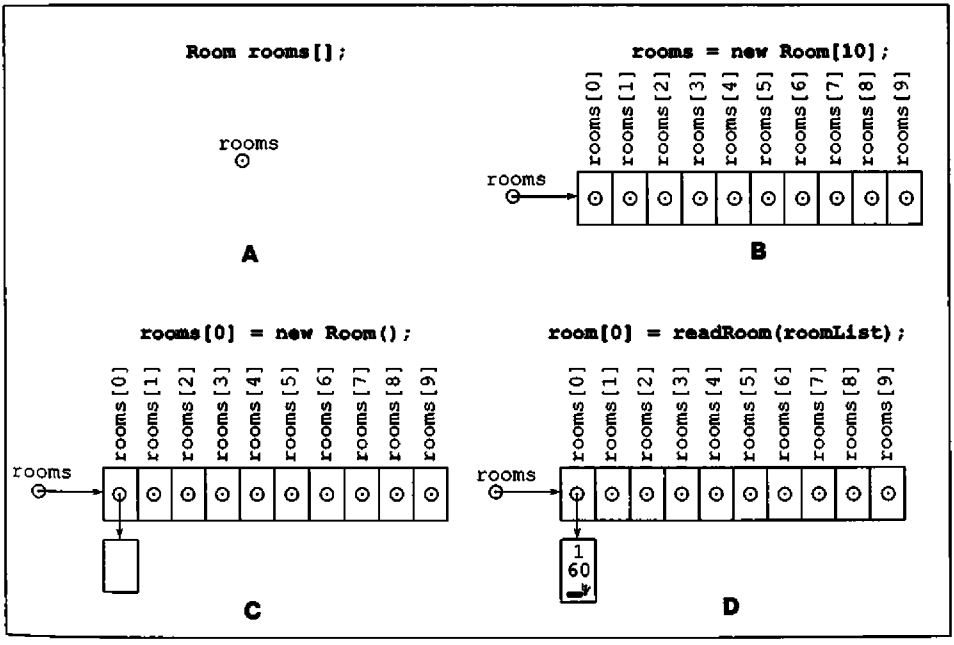


Рис. 11.11. Этапы создания массива объектов

Но мы упустили еще одну операцию, о которой следует упомянуть, хотя она и не считается этапом создания массива. Мы должны заполнить поля каждого объекта некоторыми значениями. Например, в первой итерации цикла вызывается метод `rooms[0].readRoom(diskScanner)`, который считывает первую порцию данных из

файла `RoomList` и записывает их в поля `guests`, `rate` и `smoking` первого объекта типа `Room`. В каждой итерации программа создает объект `Room` и записывает в его поля данные.

Как и в случае массива примитивных типов, первый и второй этапы можно совместить в одной инструкции.

```
Room rooms[] = new Room[10];
```

Можно также применить инициализатор массива (см. ранее).

Еще один способ украшения чисел

Существует много способов форматирования чисел с плавающей точкой. Некоторые из них уже рассматривались в предыдущих главах. В листинге 7.7 для форматирования чисел используется метод `printf()`, а в листинге 10.1 — класс `DecimalFormat`. Рассмотрим еще один способ форматирования. В листинге 11.5 для вывода денежного значения используются класс `NumberFormat` и его метод `getCurrencyInstance`.

Если сравнить инструкции форматирования в листингах 10.1 и 11.5, вы не найдете между ними много различий.

- ✓ **В первом листинге используется конструктор, а во втором — метод `getCurrencyInstance()`**, служащий типичным примером так называемых *фабричных методов* (factory methods). Фабричный метод — удобный инструмент создания часто используемых объектов. Во многих программах необходим код, отображающий сумму в долларах. Метод `getCurrencyInstance()` создает нужный для этого формат, позволяя избежать спецификации формата вида `new DecimalFormat("$###0.00;($###0.00)")`.

  Как и конструктор, фабричный метод возвращает новый объект. Однако, в отличие от конструктора, фабричный метод не имеет специального статуса. При вызове фабричного метода не нужно использовать ключевое слово `new`.
- ✓ **В первом листинге используется класс `DecimalFormat`, а во втором — класс `NumberFormat`.** Десятичные числа — это один из видов чисел (а именно числа с основанием счисления 10). Соответственно, класс `DecimalFormat` является производным от базового класса `NumberFormat`. Методы класса `DecimalFormat` более специфичны, чем методы класса `NumberFormat`, однако метод `getCurrencyInstance()` класса `DecimalFormat` сложно использовать. Поэтому, когда дело касается денег, я рекомендую использовать класс `NumberFormat`.
- ✓ **В обоих листингах используется метод `format()`.** Когда формат подготовлен, достаточно написать выражение вида `currency.format(rate)` или `decFormat.format(average)`, в результате чего компилятор Java отформатирует число автоматически.



Начиная с главы 4 я постоянно предупреждаю вас о нежелательности хранения денежных значений в переменных типа `double` и `float`. Для этого рекомендуется использовать класс `BigDecimal`. В данной главе тип `double` используется только для упрощения примера. Не делайте так в реальных задачах.



О типах `double` и `float`, а также о денежных значениях рассказывается в главе 5.

Тернарный условный оператор

В листинге 11.5 введено новое для вас средство: *тернарный условный оператор*, принимающий три выражения и возвращающий одно из них. Он похож на оператор `if` в миниатюре. Термин "тернарный" означает, что оператор имеет три операнда. Синтаксис тернарного условного оператора имеет следующий вид:

```
условие ? выражение_1 : выражение_2
```

В первую очередь компьютер вычисляет условие, которое должно быть выражением булева типа. Если условие равно `true`, оператор возвращает *выражение_1*. В противном случае оператор возвращает *выражение_2*.

Рассмотрим, как работает следующий код:

```
smoking ? "да" : "нет"
```

Сначала компьютер проверяет значение переменной `smoking`. Если оно равно `true`, оператор возвращает строковое значение да. В противном случае оператор возвращает строковое значение нет.

В листинге 11.5 тернарный условный оператор используется в качестве параметра при вызове метода `out.println()`. Фактически параметром служит значение, возвращаемое тернарным условным оператором. Метод `out.println()` выводит на консоль слово да или нет в зависимости от того, чему равна булева переменная `smoking`.

Аргументы командной строки

В былые времена, когда еще не было интегрированных сред разработки, таких как Eclipse, программисты вводили код программы в текстовом редакторе и запускали компиляцию и выполнение в командной строке. Чтобы выполнить программу `Displayer`, код которой приведен в главе 3, необязательно иметь среду разработки Eclipse. Вместо этого можно запустить скомпилированную программу в командной строке (рис. 11.12) и в этом же окне увидеть результат.

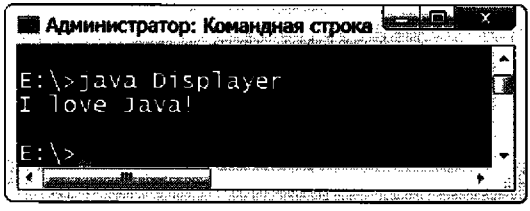


Рис. 11.12. Без Eclipse тоже можно что-то сделать!

Окно командной строки часто называют *консолью*. В рабочей среде Eclipse результат отображается во вкладке Console, которая также называется просто консолью.

На консоли можно не только запустить программу, но и передать ей аргументы. На рис. 11.13 мы запускаем программу `MakeRandomNumsFile` и передаем ей два аргумента: имя файла `MyNumberedFile.txt` и число 5. В результате программа создает файл `MyNumberedFile.txt`, содержимое которого мы выводим на консоль с помощью оператора `type`.

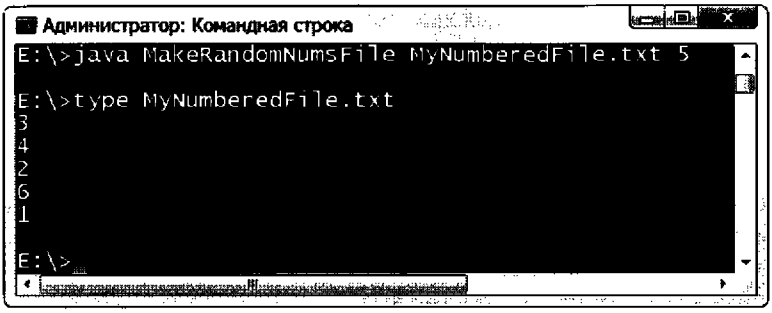


Рис. 11.13. Программа получает строку `MyNumberedFile.txt` и число 5

На рис. 11.13 программист вводит команду `java MakeRandomNumsFile`, чтобы запустить программу `MakeRandomNumsFile.class`. После этой команды в той же строке он вводит два аргумента: `MyNumberedFile.txt` и 5. Когда программа `MakeRandomNumsFile` выполняется, она считывает два этих аргумента с командной строки, сохраняет их в своих переменных и выполняет над ними все необходимые операции. На рис. 11.13 программа принимает аргументы `MyNumberFile.txt` и 5, однако можно было ввести другие строки, например `Набор_чисел.txt` и 28 или `Случайные_числа.txt` и 2000. При разных запусках программы аргументы командной строки могут быть разными.

В каком месте кода программа находит аргументы командной строки? Иными словами, где их взять при создании кода? Практически в каждом примере в этой и предыдущих главах вы видели в заголовке метода `main()` выражение `String args[]`, но до сих пор я не объяснял вам, что оно означает. Однако теперь вы будете знать, что это аргументы командной строки. Параметр `args[]` представляет собой массив переменных типа `String`, т.е. это набор строк. В командной строке аргументы разделены пробелами. Если аргумент должен содержать пробел, его нужно заключить в двойные кавычки, чтобы этот пробел не был воспринят как разделитель аргументов.

Использование аргументов командной строки в коде

В листинге 11.7 приведен пример использования аргументов командной строки в программе Java.

Листинг 11.7. Запись набора случайных чисел в файл

```
import java.util.Random;
import java.io.PrintStream;
import java.io.IOException;

class MakeRandomNumsFile {
```

```
    public static void main(String args[]) throws IOException {
        Random generator = new Random();

        if (args.length < 2) {
            System.out.println("Использование: MakeRandomNumsFile" + "
                               имя_файла число");
            System.exit(1);
        }

        PrintStream printOut = new PrintStream(args[0]);
        int numLines = Integer.parseInt(args[1]);

        for (int count = 1; count <= numLines; count++) {
            printOut.println(generator.nextInt(10) + 1);
        }
    }
}
```



Как задаются аргументы командной строки при запуске программы не с консоли, а в интегрированной среде разработки? Это зависит от типа среды разработки. Например, в Eclipse нужно выбрать в главном меню команду Run⇨Run Configuration⇨Arguments (Выполнить⇨Конфигурация выполнения⇨Аргументы) и ввести аргументы командной строки в текстовое поле Program Arguments (Аргументы программы).

Когда код листинга 11.7 начинает выполняться, операционная система записывает аргументы командной строки в массив `args`. Например, если в командной строке введены два аргумента — `MyNumberedFile.txt` и `5`, — элемент `args[0]` получает значение `MyNumberedFile.txt`, а элемент `args[1]` — значение `5`. Компилятор воспринимает инструкции следующим образом.

```
PrintStream printOut = new PrintStream("MyNumberedFile.txt");
int numLines = Integer.parseInt("5");
```

Согласно этим инструкциям программа создает файл `MyNumberedFile.txt` и присваивает переменной `numLines` значение `5`. Дальше в коде программа сгенерирует пять случайных чисел и запишет их в файл `MyNumbered.txt` (рис. 11.14).

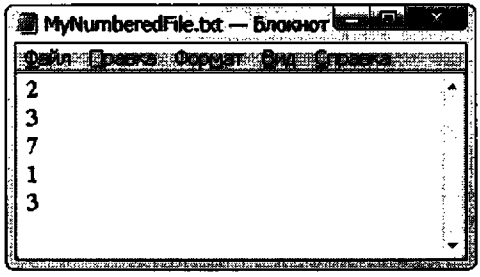


Рис. 11.14. Содержимое файла `MyNumberedFile.txt`



Код листинга 11.7 создает файл `MyNumberedFile.txt`. Где можно найти этот файл на жестком диске? Ответ зависит от многих факторов. Если программа выполняется в интегрированной среде разработки, созданный файл записывается в папку проекта. Если программа запускается с командной строки, файл записывается в текущую папку. Если нужно, чтобы файл всегда попадал в одно и то же место, задайте в листинге 11.7 абсолютный маршрут файла, например `"C:\\MyNumberedFile.txt"`.



В Windows в качестве разделителя имен в маршрутах служит обратная косая черта (\). Однако в строковых литералах Java обратная косая используется в управляющих последовательностях (см. ранее). Поэтому при задании маршрута в строковых литералах нужно вводить двойную обратную косую (\\), например `"C:\\MyNumberedFile.txt"`. В противоположность этому в Linux и Mac разделителем имен в маршрутах служит обычная косая черта (/). Ее можно использовать в строковых литералах Java, например `"/bin/Documents/MyNumberedFile.txt"`. Важно отметить следующее. Если обычная косая используется в маршруте, программа будет работоспособна и в Windows. Однако при этом доступен только текущий диск, причем букву диска писать не нужно.

Обратите внимание на то, что каждый аргумент командной строки является значением типа `String`. Когда вы смотрите на значение `args[1]` и видите число `5`, не забывайте, что на самом деле это не число, а строка `"5"`. Ее нельзя использовать в арифметических выражениях, например умножать ее на что-либо. Кроме того, она не может обозначать количество чего бы то ни было. Необходимо преобразовать строку `"5"` в число `5`. В листинге 11.7 это делается с помощью метода `parseInt()`.

Метод `parseInt()` является статическим и принадлежит классу `Integer`. Поэтому для его вызова не нужно создавать объект типа `Integer`, достаточно написать `Integer.parseInt()`. Кроме того, класс `Integer` содержит много других методов, полезных для обработки значений типа `int`.



В Java слово `Integer` — это имя класса, а `int` — это имя примитивного типа. Это не одно и то же. Класс `Integer` содержит инструменты и методы, а в `int` ничего нет.

Проверка количества аргументов командной строки

Что произойдет, если пользователь забудет ввести `5` в командной строке (см. рис. 11.13)? Компьютер присвоит элементу `args[0]` значение `MyNumberedFile.txt`, а элементу `args[1]` ничего не присвоит. Это очень плохо, потому что, когда компьютер дойдет до инструкции `numLimes=Integer.parseInt(args[1])`, он не найдет значения `args[1]`, и программа завершится крахом.

Чтобы этого не произошло, в листинге 11.7 проверяется длина массива `args`. Значение поля `args.length` сравнивается с числом `2`. Если массив `args` имеет менее двух элементов, программа выводит на консоль сообщение (рис. 11.15), напоминающее пользователю о том, что нужно ввести в командной строке.

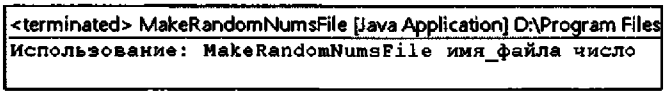


Рис. 11.15. Напоминание о правильном запуске программы



Несмотря на то что программа проверяет количество аргументов, код листинга 11.7 все же не защищен от краха полностью. Например, если пользователь вместо `5` введет `пять`, виртуальная машина Java сгенерирует исключение `NumberFormatException`. Второй аргумент командной строки обязательно должен быть числом, причем обязательно целым. Конечно, можно добавить в листинг 11.7 код, проверяющий аргументы командной строки более тщательно, однако предусмотреть все ошибочные варианты невозможно. Поэтому лучше добавить в код обработку исключений, как показано в главе 12.



Аргумент командной строки может содержать пробелы. Чтобы программа могла отличить пробелы, принадлежащие аргументу, от пробелов, служащих разделителями смежных аргументов, нужно заключить аргумент в двойные кавычки. Например, на рис. 11.13 можно ввести два аргумента `"Файл с числами.txt" 5`.

На этом мы завершаем обсуждение массивов. Следующий раздел посвящен немного другой теме. Однако, прежде чем мы расстанемся с массивами окончательно, позвольте обратить ваше внимание на следующее. Массив — это ряд объектов. Но не каждый набор объектов можно выстроить в ряд. Предположим, например, что наш мотель не одноэтажный, а многоэтажный. Как мы будем нумеровать комнаты в этом случае? В многоэтажных зданиях номера комнат довольно часто начинаются с номера этажа. Например, 301 — это комната 1 на третьем этаже. Если использовать такие номера в качестве индексов массива `rooms`, многие номера окажутся "пустыми", и в программу придется добавлять код, отличающий "пустые" номера от настоящих. Можно выполнить сквозную нумерацию комнат, но тогда номер комнаты не будет указывать на номер этажа, что очень неудобно не только для постояльцев, но и для обслуживающего персонала. Одно из решений данной проблемы — *многомерные массивы*. Например, элемент `rooms[3][15]` означает комнату 15 на третьем этаже. К сожалению, в данной книге многомерные массивы не рассматриваются в связи с ограничением на объем текста.

Коллекции

Массивы — очень полезное средство, но им все же присущ ряд ограничений. Предположим, например, что в массиве хранятся имена ста клиентов. Приведенный ниже код создает массив, состоящий из ста элементов типа `String`.

```
String name[] = new String[100];
for (int i = 0; i < 100; i++) {
    name[i] = new String();
}
```

Все идет хорошо, пока в один прекрасный день не появляется 101-й клиент. Программа продолжает выполняться, и вы пытаетесь ввести данные для нового клиента, надеясь на то, что ничего плохого не произойдет. Однако надежды эти напрасные. Программа генерирует исключение `ArrayIndexOutOfBoundsException` (индекс массива нарушил границы) и аварийно завершается.

В следующий раз вы решаете сделать массив достаточно большим. Например, вы точно знаете, что больше тысячи клиентов у фирмы никогда не будет, и создаете массив имен клиентов следующим образом.

```
String name[] = new String[1000];
for (int i = 0; i < 1000; i++) {
    name[i] = new String();
}
```

Однако экономическая конъюнктура изменилась в неблагоприятную сторону, и вместо ста одного у фирмы теперь есть только три клиента. Вы зарезервировали память для тысячи клиентов, а у вас их только три. Какое расточительство!

К тому же что будет, если конъюнктура существенно изменится в другую сторону? От клиентов нет отбоя, фирма бурно расширяется, и в один прекрасный день программа отказывается регистрировать очередного клиента и терпит крах.

Чтобы подобные проблемы не возникали, не только в Java, но и в программировании вообще принято резервировать для массива или коллекции ровно столько места, сколько нужно в данный момент. Тогда вам не придется ожидать краха программы в каждый момент времени и гадать, произойдет ли он.

Рассмотрим еще одну ситуацию. Предположим, клиенты расположены в массиве в определенной последовательности (в алфавитном порядке по фамилиям или по номерам карточек социального страхования). Когда появится новый клиент, его нужно будет каким-либо образом втиснуть в нужное место, растолкав остальных клиентов.

Предположим, клиент должен получить индекс 7, а всего у вас 824 элемента массива, т.е. последний индекс равен 823. Чтобы втиснуть нового клиента в элемент 7, вы должны сдвинуть вправо всех клиентов, расположенных справа. Сначала нужно переместить клиента 823 в элемент 824, затем — клиента 822 в элемент 823, и так до тех пор, пока не дойдете до клиента 7. Конечно, вы делаете это не вручную. Несложно написать код, выполняющий эту операцию автоматически, но отнимать зря время у процессора тоже не следует. Процессорное время часто является очень ценным ресурсом.

Есть еще одно решение проблемы: оставить между всеми соседними клиентами несколько пустых элементов. Однако, во-первых, это сильно усложнит программу, а во-вторых, между некоторыми двумя клиентами все равно может оказаться недостаточно места. Кроме того, существует ряд непростых проблем, связанных с фильтрацией, сортировкой и группированием наборов данных. Единственное рациональное решение указанных проблем состоит в использовании коллекции вместо массива.

Классы коллекций

Проблемы, представленные в предыдущем разделе, не новы. Ученые, работающие в области информационных технологий, ломают головы над ними уже не одно десятилетие. За все это время они не нашли универсального решения, пригодного в любом случае, но все же изобрели немало полезных трюков.

В библиотеках Java API есть ряд классов, предназначенных для решения указанных проблем. Они называются *классами коллекций*. Профессиональные программисты считают, что класс коллекций — это любой класс, реализующий интерфейс `Collection`. Что это такое, вам пока что не нужно знать. Сейчас вам нужно знать, что в каждом классе коллекций определены методы, предназначенные для хранения и обработки наборов значений или объектов, причем в каждом из этих методов используются хитроумные трюки. При программировании на Java знать подробности этих трюков необязательно. Запомните лишь, что каждый метод классов коллекций решает свою задачу оптимальным образом. Разработчики библиотек Java API приложили немало усилий и потратили много времени на то, чтобы увеличить быстродействие этих методов и уменьшить потребление ресурсов. Вам остается только использовать эти классы. Вам не нужно ломать голову над тем, как втиснуть клиента 7 в коллекцию. Для этого достаточно добавить его с помощью метода `add()`, который решит задачу оптимальным образом. Скорость выполнения данной операции (как и всех других) зависит от многих факторов. В лучшем случае эффективность операции будет очень высокой. В худшем случае можно быть уверенным в том, что класс сделал все, что можно было сделать для повышения эффективности операции.

Класс `ArrayList`

Один из наиболее популярных классов коллекций — класс `ArrayList`. Его использование демонстрируется в листинге 11.8.

Листинг 11.8. Использование коллекции `ArrayList`

```
import static java.lang.System.out;
import java.util.Scanner;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;

class ShowNames {

    public static void main(String args[]) throws IOException {
        ArrayList<String> people = new ArrayList<String>();
        Scanner diskScanner = new Scanner(new File("names.txt"));

        while (diskScanner.hasNext()) {
            people.add(diskScanner.nextLine());
        }

        people.remove(0);
        people.add(2, "Мезенцева Ольга");

        for (String name : people) {
            out.println(name);
        }
    }
}
```

На рис. 11.16 показано содержимое файла `names.txt`. Код листинга 11.8 считывает текст из файла `names.txt`, удаляет первую строку, добавляет новую строку в заданную позицию и выводит все строки на консоль. Результат показан на рис. 11.17.

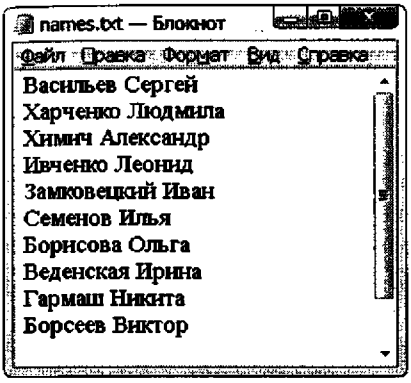


Рис. 11.16. Файл `names.txt`

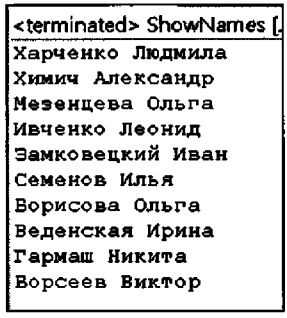


Рис. 11.17. Результат выполнения листинга 11.8

Сейчас нас интересуют методы `remove()` и `add()`. Переменная `people` ссылается на коллекцию — объект типа `ArrayList`. Метод `remove()` данной коллекции удаляет из нее элемент, имеющий указанный индекс.

```
people.remove(0);
```

В результате выполнения этой инструкции первая строка `Васильев Сергей` удаляется (в коллекциях, как и в массивах, элементы нумеруются с нуля), а ее место занимает вторая строка — `Харченко Людмила`.

Осталось только восемь элементов, однако следующая инструкция вставляет строку `Мезенцева Ольга` в позицию 2, и количество элементов опять становится равным 9.

```
people.add(2, "Мезенцева Ольга");
```

Обратите внимание на то, что объект `ArrayList` имеет два разных метода `add()`. Приведенный выше метод получает два параметра: номер позиции, в которую нужно вставить объект, и добавляемый объект. Приведенный ниже метод получает один параметр.

```
people.add(diskScanner.nextLine());
```

Этот метод получает следующую строку, найденную во входном файле, и добавляет ее в конец коллекции `ArrayList`.

В последних двух строках листинга 11.8 используется расширенный цикл `for`. Как и в листинге 11.3, он имеет следующий синтаксис:

```
for (тип_переменной имя_переменной : диапазон_значений)
```

В листинге 11.8 используется тип переменной `String` и имя переменной `name`, а в качестве диапазона значений применяется коллекция `people`. На каждой итерации цикла переменная `name` получает одно из значений типа `String`, хранящееся в коллекции `people`. В цикле выполняется столько итераций, сколько элементов есть в коллекции `people`. В каждой итерации выполняется инструкция, расположенная в теле цикла и отображающая значение `name` на консоли.

Одна из главных целей разработчиков языка Java — сделать его как можно более простым. За основу были взяты языки C и C++. Из них были удалены многие средства, которые разработчики Java сочли лишними, в результате чего получился простой и элегантный язык. Однако многие считали, что он получился слишком элегантным. После нескольких лет дискуссий и споров Java стал немного более сложным. К 2004 году в Java появились типы `enum`, расширенный цикл `for`, статический импорт и некоторые другие интересные средства. Больше всего споров было о том, нужно ли добавить в Java обобщенные типы.

```
ArrayList<String> people = new
ApparList<String>();
```

Написать такую инструкцию можно, начиная с Java 5.0. В более ранних версиях нужно было писать так.

```
ArrayList people = new
ArrayList();
```

Писать так можно и сейчас, но это считается устаревшим стилем программирования и приводит к ряду проблем. При таком стиле объявления коллекции в нее можно добавлять объекты любых типов, включая `String`, `Room`, `Account` и др. Класс `ArrayList` получается очень гибким, даже слишком гибким. Если вы можете поместить в коллекцию все что угодно, вы не сможете предвидеть, что вы извлечете из нее. В результате вы не сможете написать код, предполагающий, что коллекция содержит объекты определенного типа. К тому же компилятор не сможет оптимизировать быстродействие выполняемого кода, потому что он не знает, какие объекты будут использоваться в коллекции. Рассмотрим следующий пример.

```
//Это пример плохого кода, не
делайте так!
ArrayList things = new
ArrayList();
things.add(new Account());
Account myAccount = things.
get(0);
```

Метод `get()` извлекает из коллекции `things` самый первый объект. Вызов метода выполняется безупречно, но дальше компилятор пытается присвоить извлеченное значение переменной `myAccount`, но не может сделать этого, потому что не знает типа извлеченного объекта. Фактически компилятор считает, что извлеченный объект имеет тип `Object`. Конечно, можно привести объект к нужному типу с помощью префикса `(Account)`, но этим вы забираете у компилятора задачу контроля типов и перекладываете ее на себя. К тому же вам везде придется добавлять префикс `(Account)`.

Проблема решается путем использования обобщенного типа.

```
//Это хороший код, используйте
его в качестве шаблона
ArrayList<Account> things = new
ArrayList<Account>();
things.add(new Account());
Account myAccount = things.get(0)
```

Добавление фразы `<Account>` в двух местах сообщает компилятору о том, что коллекция `things` содержит экземпляры класса `Account`. Следовательно, присвоение извлеченного объекта переменной `myAccount` выполняется без проблем.

Обобщенные типы добавлены в Java 5. И очень скоро программисты стали вставлять в коллекции весьма сложные, причудливые типы. В конце концов, ничто не запрещает объявлять обобщенные типы в обобщенных типах. Например, коллекция `ArrayList` может содержать набор массивов, каждый из которых содержит коллекцию строк, т.е. можно объявить коллекцию массивов коллекций.

```
ArrayList<ArrayList<String>[]> mess =
  new ArrayList<ArrayList<String>[]>();
```

Разработчики языка Java решили, что это довольно полезный трюк, и упростили его использование. Начиная с Java 7 в операторе `new` можно использовать оператор `<>`, который обозначает повторное использование обобщенного типа. Приведенную выше инструкцию теперь можно записать так.

```
ArrayList<ArrayList<String>[]>
mess = new ArrayList<>();
```

Как видите, оператор `<>` упрощает код, но главное не это, а то, что уменьшается количество ошибок, которые могут возникнуть при повторном вводе громоздкой конструкции.

Использование обобщенных типов

Посмотрите еще раз на объявление коллекции `ArrayList` в листинге 11.8.

```
ArrayList<String> people = new ArrayList<String>();
```

Начиная с Java 5 каждый класс коллекции может быть обобщенным. Это означает, что объявление коллекции может содержать какой-либо тип в угловых скобках, например `<String>`. Коллекция может содержать объекты только того типа, который указан в угловых скобках. Например, фраза `ArrayList<String> people` в листинге 11.8 сообщает компилятору о том, что коллекция `people` может содержать только объекты типа `String`.



Обобщенные типы можно использовать только в версиях Java начиная с 5. В версиях Java до 5 использовать их нельзя. Версии Java подробно рассматриваются в главе 2.

В листинге 11.8 фраза `ArrayList<String> people` говорит о том, что на переменную `people` можно ссылаться только как на коллекцию объектов типа `String`. Это означает, что любой элемент коллекции `people` обрабатывается компилятором исключительно как объект типа `String`. Например, приведенная ниже инструкция недопустима.

```
people.add(new Room()); //Неправильно!
```

Встретив ее, компилятор сгенерирует сообщение об ошибке, потому что класс `Room` (он определен в листинге 11.5) — это не класс `String`. Ниже приведен правильный код.

```
people.add("Сергей Ивченко");
```

Поскольку литерал `"Сергей Ивченко"` является объектом типа `String`, компилятор обработает код без возражений.



Как упомянуто выше, начиная с Java 7 при создании коллекции можно использовать оператор <>. В частности, в листинге 11.8 при создании коллекции `people` можно опустить повторное вхождение имени типа `String` и записать `ArrayList<String> people = new ArrayList<>()`.

Проверка, есть ли еще данные

Когда коллекция заполняется данными из файла, как в листинге 11.8, вы обычно не знаете заранее, сколько объектов нужно будет записать в коллекцию (если бы знали, то, скорее всего, лучше было бы применить массив, а не коллекцию). Поэтому на каждой итерации цикла нужно проверить, есть ли еще данные в файле. Если их больше нет, нужно выйти из цикла, иначе будет сгенерировано исключение и произойдет аварийное завершение программы.

Класс `Scanner` содержит ряд методов, проверяющих, есть ли в потоке еще как минимум одно значение заданного типа. Это методы `hasNextInt()`, `hasNextDouble()` и др. Есть также метод `hasNext()`, пригодный для данных любого типа. Каждый из

этих методов возвращает `true`, если в потоке еще есть как минимум одно значение. В противном случае метод возвращает `false`.

В листинге 11.8 используется метод `hasNext()`. Он возвращает `true`, если в потоке еще есть что читать. Обратите внимание на то, что файл `names.txt` заканчивается несколькими пустыми строками. Должен ли метод `nextLine()` прочитать пустую строку и записать ее в коллекцию? Оказывается, это зависит не от него, а от метода `hasNext()`. Программа считывает десять непустых строк, а пустые строки игнорирует. Если же добавить после нескольких пустых строк еще одну непустую, программа запишет в коллекцию несколько пустых строк.

Глава 12

Как сохранить хорошую мину при плохой игре

В этой главе...

- Обработка исключений
- Обработайте исключение или передайте его дальше
- Блок finally
- Закрытие файлов

9 сентября 1945 года. Безымянный жучок проник в механическое реле компьютера Mark II Гарвардского университета и нарушил его работу. Это был первый зарегистрированный случай сбоя компьютера вследствие проникновения в него живого насекомого. Впоследствии слово "bug" (жучок), означающее ошибку в работе компьютера, стало чрезвычайно популярным и породило ряд производных словоформ, таких как "debug" (отладка), "buggy" (ненадежный) и др.

9 апреля 1957 года. Герберт Брайт, менеджер Центра обработки данных компании Westinghouse, Питтсбург, получил тяжелую посылку, заполненную толстыми пачками перфокарт. В те времена это было эквивалентно получению электронного письма или компакт-диска. Посылка пришла от команды разработчиков компилятора FORTRAN — первого высокоуровневого языка программирования. Интернета тогда еще и в помине не было, поэтому обмен программным обеспечением происходил посредством старой доброй почтовой службы.

Брайт запустил полученный компилятор на своем компьютере IBM 704 и обработал с его помощью небольшую программу на языке FORTRAN. Это была сложнейшая техническая задача. Брайту помогал целый отдел сотрудников. Машина IBM 704 занимала огромную комнату площадью 200 м², специально построенную для нее. Транзисторов тогда еще не было. Машина работала на вакуумных электронных лампах, потреблявших электроэнергии не меньше, чем завод средних размеров. Оперативная память машины имела объем 32 Кбайт. Операционная система загружалась с бумажной перфоленты в течение нескольких часов. После всех подготовительных работ и ожиданий компилятор вернул абсурдный результат. В те времена никто еще даже не мечтал о том, чтобы компилятор указал, в какой строке находится ошибка. Она оказалась в том, что в одной из инструкций была пропущена запятая. Брайт изготовил новую перфокарту вместо ошибочной, и программа заработала, как от нее ожидалось. Это был первый зарегистрированный случай отладки высокоуровневого кода пользователями компилятора.

22 июля 1962 года. Запуск космического аппарата Mariner I с мыса Канаверал. Это был первый в истории космический аппарат, предназначенный для полета на другую планету Солнечной системы. Через четыре минуты после запуска аппарат был

уничтожен вследствие неполадок в системе управления. Как оказалось, в формуле расчета скорости ракеты-носителя был пропущен дефис.

Приблизительно в это же время в программном обеспечении NASA, предназначенном для управления орбитальным полетом, была обнаружена ошибочная инструкция `DO 10 I=1.10` (вместо `DO 10 I=1,10`). Точка вместо запятой превратила цикл в оператор присваивания.

Так начиналась компьютерная эпоха. Каждое достижение тогда было праздником, а каждая ошибка становилась знаменитой на весь мир. Данная глава посвящена исключениям — одному из средств устранения ошибок. Вам предстоят приключения, не менее захватывающие, чем при программировании аппарата Mariner I, но не рассчитывайте на то, что ваши ошибки сделают вас знаменитым. Скорее наоборот: вы получите шанс стать знаменитым и богатым, если научитесь устранять их в кратчайшие сроки.

Обработка исключений

Исключение — это ошибка, возникшая вследствие того, что компьютер не может выполнить некоторую операцию, например поделить число на нуль или преобразовать строку в число. В большинстве случаев исключения вызваны не ошибками в коде, а неправильными входными данными. Вы можете идеально отладить код программы, но не можете гарантировать, что на ее вход всегда будут поступать данные без ошибок. Безошибочно может работать только компьютер, а пользователь, как и любой другой человек, часто ошибается. Создавая программу, вы должны рассчитывать на то, что входные данные не всегда будут правильными. Ваша задача состоит в том, чтобы при получении неправильных данных программа не завершалась крахом, а сообщала пользователю об ошибке и предлагала ее исправить.

Предположим, вы получили на руки инвентарную опись чего-либо. Это означает, что вы должны пересчитать некоторые предметы и отметить их количество в журнале, на листке бумаги, на экране портативного устройства или в программе с помощью клавиатуры. Остановимся на последнем варианте. Вы подсчитываете количество коробок стоимостью 3,25 рубля каждая и вводите его с клавиатуры в программу, которая умножает стоимость одной коробки на их количество.

Код программы показан в листинге 12.1. Сама по себе программа, казалось бы, работает безупречно. Если ввести целое число, она отобразит на консоли правильный результат. Однако пользователь может ошибиться. Что будет, если пользователь введет не целое число? Или, совсем уж размечтавшись о посторонних вещах, введет слово `три` вместо числа `3`? Очевидно, программа потерпит крах (рис. 12.1). Вы как программист должны предусмотреть все подобные случаи. При любых входных данных программа должна сообщить пользователю об ошибке и предложить ее исправить.

Листинг 12.1. Ввод количества коробок с клавиатуры

```
import static java.lang.System.out;
import java.util.Scanner;
import java.text.NumberFormat;

class InventoryA {
```

```
    public static void main(String args[]) {
        final double boxPrice = 3.25;
        Scanner keyboard = new Scanner(System.in);
        NumberFormat currency = NumberFormat.getCurrencyInstance();

        out.print("Сколько коробок вы насчитали?");
        String numBoxesIn = keyboard.next();
        int numBoxes = Integer.parseInt(numBoxesIn);

        out.print("Общая стоимость равна ");
        out.println(currency.format(numBoxes * boxPrice));
    }
}
```

```
<terminated> InventoryA [Java Application] D:\Program Files\Java\jre7\bin\javaw.exe (19.12.2011 17:15:46)
Сколько коробок вы насчитали? 3
Общая стоимость равна 9,75 руб.

<terminated> InventoryA [Java Application] D:\Program Files\Java\jre7\bin\javaw.exe (19.12.2011 17:22:27)
Сколько коробок вы насчитали? 3.25
Exception in thread "main" java.lang.NumberFormatException: For input string: "3.25"
        at java.lang.NumberFormatException.forInputString(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at InventoryA.main(InventoryA.java:15)

<terminated> InventoryA [Java Application] D:\Program Files\Java\jre7\bin\javaw.exe (19.12.2011 17:28:40)
Сколько коробок вы насчитали? три
Exception in thread "main" java.lang.NumberFormatException: For input string: "три"
        at java.lang.NumberFormatException.forInputString(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at InventoryA.main(InventoryA.java:15)
```

Рис. 12.1. Три запуска листинга 12.1

Ключ к устранению ошибки — имя исключения `NumberFormatException` (исключение форматирования числа), отображенное на консоли рабочей средой Eclipse. Оно означает, что класс исключения `NumberFormatException` находится в пакете `java.lang`. Вызов метода `Integer.parseInt()` привел к отображению на консоли имени класса `NumberFormatException`.



Краткое описание метода `Integer.parseInt()` можно найти в главе 11.

В язык Java встроен механизм *обработки исключений*. Виртуальная машина Java обнаруживает, что операция не может быть выполнена, и генерирует объект исключения, предоставляя программе и пользователю последний шанс исправить ошибку. Объект исключения, являющийся экземпляром класса, производного от класса `Exception`, как горячая картофелина, передается от одного метода к другому, пока некоторый метод его не перехватит. Когда исключение перехвачено, программа выполняет код восстановления, удаляет исключение и продолжает выполнение программы со следующей нормальной инструкции, как будто ничего не случилось. Этот процесс проиллюстрирован на рис. 12.2.

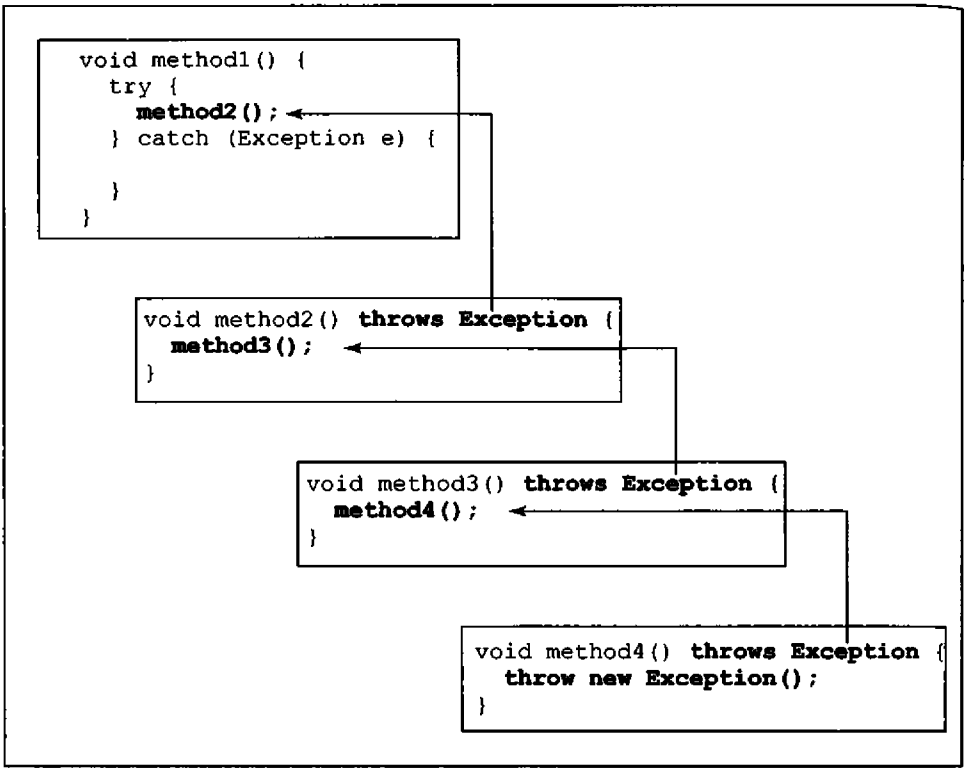


Рис. 12.2. Генерация, передача и перехват исключения

Перехват и обработка исключений выполняется с помощью нескольких ключевых слов Java.

- ✓ **throw** (сгенерировать исключение). Инструкция, создающая объект исключения. В данном примере она применяется для тестирования системы обработки исключений.
- ✓ **throws** (генерирует исключение). Директива, передающая необработанное исключение вызывающему методу.
- ✓ **try** (попытаться выполнить). Блок, содержащий код, в котором может возникнуть исключение. Блок `try` может также содержать вызовы методов, в теле которых возникают исключения.
- ✓ **catch** (перехватить исключение). В блоке `catch` находится код, обрабатывающий исключение. Обработка может состоять из таких операций, как отображение сообщения для пользователя, удаление неправильных данных, проверка других условий и т.п.

Директива `throws`, приведенная в заголовке метода, означает, что метод может передать необработанное исключение вызывающему методу. Он, в свою очередь, также может либо обработать исключение в блоке `catch`, либо передать его дальше, т.е. своему вызывающему методу. На рис. 12.2. исключение передается по цепочке от `method4()`

до method1(), где его, наконец, обрабатывает блок catch. В листинге 12.1 исключение NumberFormatException не обрабатывается и никуда не передается. В результате виртуальная машина Java, не зная, что с ним делать, завершает программу аварийно.



В документации Java API есть раздел, посвященный методу parseInt(). В нем сказано, что, если входная строка не содержит целого числа, метод генерирует исключение NumberFormatException, что мы и видим на рис. 12.1.

Итак, если программа не перехватит и не обработает исключение, его перехватит виртуальная машина Java. В результате программа будет завершена аварийно. Чтобы не допустить этого, нужно добавить в программу код обработки исключений, как показано в листинге 12.2.

Листинг 12.2. Ввод количества коробок с обработкой исключения

```
import static java.lang.System.out;
import java.util.Scanner;
import java.text.NumberFormat;

class InventoryB {

   public static void main(String args[]) {
       final double boxPrice = 3.25;
       Scanner keyboard = new Scanner(System.in);
       NumberFormat currency = NumberFormat.getCurrencyInstance();

       out.print("Сколько коробок вы насчитали? ");
       String numBoxesIn = keyboard.next();

       try {
           int numBoxes = Integer.parseInt(numBoxesIn);
           out.print("Общая стоимость равна ");
           out.println(currency.format(numBoxes * boxPrice));
       } catch (NumberFormatException e) {
           out.println("Это не целое число.");
       }
   }
}
```

На рис. 12.3 показано три сеанса работы программы, приведенной в листинге 12.2. Когда пользователь вместо 3 вводит три, программа отвечает: Это не целое число. Трюк состоит в помещении вызова Integer.parseInt() в блок try. Компьютер наблюдает за исключениями в инструкциях, находящихся в блоке try. Если происходит исключение, компьютер прекращает выполнение блока try и немедленно передает управление блоку catch. Последующие инструкции блока try не выполняются. В листинге 12.2 управление в случае исключения передается блоку catch(NumberFormatException e). В блоке catch компьютер отображает сообщение об ошибке с помощью метода println(). После этого компьютер продолжает выполнять программу, начиная с первой инструкции после блока catch. В блок try компьютер больше не возвращается.

```
<terminated> InventoryB [Java Application] D:\Program
Сколько коробок вы насчитали? 3
Общая стоимость равна 9,75 руб.

<terminated> InventoryB [Java Application] D:\Program
Сколько коробок вы насчитали? три
Это не целое число.

<terminated> InventoryB [Java Application] D:\Program
Сколько коробок вы насчитали? -25
Общая стоимость равна -81,25 руб.
```

Рис. 12.3. Программа обработала исключение, сгенерированное при втором запуске

Параметр блока `catch`

Обратите внимание на список параметров `(NumberFormatException e)` блока `catch` в листинге 12.2. Он выглядит как список параметров метода. Фактически каждый блок `catch` — это маленький метод со своим списком параметров. Операционная система передает блоку `catch` имя типа сгенерированного исключения.

В листинге 12.2 мы ничего не делаем с полученным параметром, однако при необходимости его можно использовать. Сгенерированное исключение — это экземпляр класса `NumberFormatException`. Когда исключение перехвачено блоком `catch`, параметр `e` ссылается на объект исключения. Иными словами, в объекте `e` находится информация об исключении. Можно также вызывать методы объекта исключения.

```
} catch (NumberFormatException e) {
  out.println("Сообщение: ***" + e.getMessage() + "***");
e.printStackTrace();
}
```

Результат работы программы `InventoryB` с этим новым блоком `catch` показан на рис. 12.4. Метод `e.getMessage()` возвращает сообщение, сгенерированное виртуальной машиной Java, а метод `e.printStackTrace()` выводит на консоль записи журнала ошибок, посвященные данному исключению. Эти записи предоставляют вам дополнительную информацию о методах, выполняющихся в момент возникновения исключения (на рис. 12.4 это методы `Integer.parseInt()` и `main()`). Методы `getMessage()` и `printStackTrace()` отображают информацию, помогающую устранить ошибку.

```
<terminated> InventoryB [Java Application] D:\Program Files\Java\jre7\bin\javaw.exe (19.12.2011 22:10:17)
Сколько коробок вы насчитали? три
Сообщение: ***For input string: "три"***
java.lang.NumberFormatException: For input string: "три"
        at java.lang.NumberFormatException.forInputString(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at InventoryB.main(InventoryB.java:17)
```

Рис. 12.4. Результат вызова методов объекта исключения



При смешивании вызовов `System.out.println()` и `printStackTrace()` последовательность вывода информации на консоль непредсказуема. Например, на рис. 12.4 текст `Сообщение:` может появиться после журнала стека. Если последовательность вывода важна для вас, используйте метод `System.err.println()`.

Типы исключений

Что еще может произойти? Невозможность прочитать число — не единственный тип ошибки. Соответственно, кроме `NumberFormatException`, существует много других типов исключений. Вы можете даже создать собственное исключение, которое в этом случае будет называться *пользовательским*. Пример создания и применения пользовательского исключения `OutOfRangeException` приведен в листингах 12.3 и 12.4.

Листинг 12.3. Класс пользовательского исключения

```
@SuppressWarnings("serial")
class OutOfRangeException extends Exception {
}
```

Листинг 12.4. Генерация пользовательского исключения

```
import static java.lang.System.out;
import java.util.Scanner;
import java.text.NumberFormat;

class InventoryC {

   public static void main(String args[]) {
      final double boxPrice = 3.25;
      Scanner keyboard = new Scanner(System.in);
      NumberFormat currency = NumberFormat.getCurrencyInstance();

      out.print("Сколько коробок вы насчитали? ");
      String numBoxesIn = keyboard.next();

      try {
         int numBoxes = Integer.parseInt(numBoxesIn);

         if (numBoxes < 0) {
            throw new OutOfRangeException();
         }

         out.print("Общая стоимость равна ");
         out.println(currency.format(numBoxes * boxPrice));
      } catch (NumberFormatException e) {
         out.println("Это не число.");
      } catch (OutOfRangeException e) {
         out.print(numBoxesIn);
         out.println(" ??? Это невозможно!");
      }
   }
}
```

В листингах 12.3 и 12.4 исправлена ошибка, проявившаяся на рис. 12.3. Посмотрите на последний сеанс программы. Пользователь ввел отрицательное число. Что он имел в виду, вводя отрицательное количество коробок? Может, хотел отобразить тот факт, что было 25 коробок, но их унесли? Однако, скорее всего, он попросту ошибся, потому что в этот момент думал о чем-то другом. Компьютер проверил, число ли это. Определенно, это число. Больше возражений у компьютера не возникло, и он, не задумываясь, умножил его на стоимость коробки. Однако виноват не компьютер, а программист, который не предусмотрел возможность такой ошибки. Чтобы исправить ее, создадим пользовательский класс исключения и будем генерировать его, когда увидим отрицательное число (см. листинг 12.4). Результат показан на рис. 12.5.

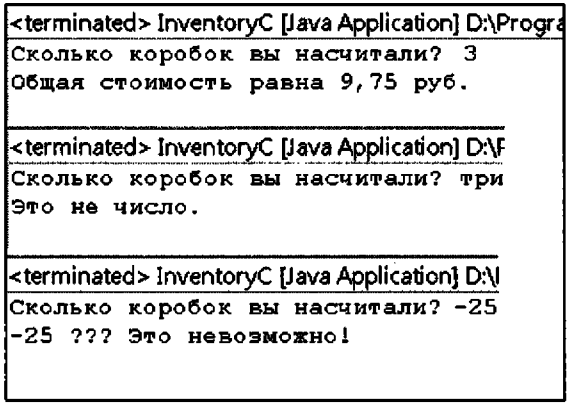


Рис. 12.5. Три сеанса программы, приведенной в листингах 12.3 и 12.4

В листинге 12.3 объявлен класс пользовательского исключения `OutOfRangeException` (исключение, нарушение границы диапазона). Вы можете присвоить ему произвольное имя, однако в Java принято заканчивать имя исключения словом `Exception` (Исключение). Во многих ситуациях ввод отрицательных чисел не является ошибкой, поэтому данное исключение не встроено в Java API. Но в нашем примере отрицательное число недопустимо.

Класс `OutOfRangeException` в листинге 12.3 — самый короткий класс в данной книге. Код класса состоит всего лишь из заголовка и пары фигурных скобок, между которыми ничего нет. Наиболее содержательная фраза `extends Exception` означает, что пользовательский класс является производным от класса `Exception`, который определен в Java API. Это означает, что при создании класса `OutOfRangeException` виртуальная машина Java генерирует исключение, которое может быть перехвачено блоком `catch`. При необходимости можете добавить в конструктор класса любые инструкции, которые, как вы полагаете, нужно выполнить, когда сгенерировано исключение.

В листинге 12.4 код создает экземпляр класса `OutOfRangeException`. В этот момент виртуальная машина генерирует исключение. Экземпляр исключения перехватывается блоком `catch`, который выводит на консоль сообщение для пользователя.



Фраза `@SuppressWarnings("serial")` в листинге 12.3 — это аннотация Java. Аннотации рассматриваются в главе 8. Аннотация `@SuppressWarning` (отменить предупреждения) рассматривается в главе 9.

Кто должен перехватить исключение

Посмотрите еще раз на листинг 12.4. Обратите внимание на то, что с блоком `try` может быть ассоциировано более одного блока `catch`. Когда в блоке `try` генерируется исключение, компьютер проходит по списку блоков `catch` сверху вниз.

По пути вниз для каждого блока `catch` компьютер проверяет, является ли сгенерированное исключение экземпляром класса исключения, приведенного в списке параметров данного блока `catch`.

- ✓ Если нет, компьютер пропускает данный блок `catch` и переходит к следующему.
- ✓ Если да, компьютер выполняет данный блок `catch`. После этого он пропускает остальные блоки `catch`, ассоциированные с данным блоком `try`. Затем компьютер выполняет первую инструкцию, расположенную непосредственно после блоков `try-catch`.

Пример этого процесса можно увидеть в листингах 12.5 и 12.6.

Листинг 12.5. Еще один класс пользовательского исключения

```
@SuppressWarnings("serial")
class NumberTooLargeException extends OutOfRangeException {
}
```

Листинг 12.6. Кто обработает исключение?

```
import static java.lang.System.out;
import java.util.Scanner;
import java.text.NumberFormat;

class InventoryD {

   public static void main(String args[]) {
       final double boxPrice = 3.25;
       Scanner keyboard = new Scanner(System.in);
       NumberFormat currency = NumberFormat.getCurrencyInstance();

       out.print("Сколько коробок вы насчитали? ");
       String numBoxesIn = keyboard.next();

       try {
           int numBoxes = Integer.parseInt(numBoxesIn);

           if (numBoxes < 0) {
               throw new OutOfRangeException();
           }
```

```
            if (numBoxes > 1000) {
                throw new NumberTooLargeException();
            }

            out.print("Общая стоимость равна ");
            out.println(currency.format(numBoxes * boxPrice));
        }

        catch (NumberFormatException e) {
            out.println("Это не число.");
        }

        catch (OutOfRangeException e) {
            out.print(numBoxesIn);
            out.println("??? Это невозможно!");
        }

        catch (Exception e) {
            out.print("Что-то здесь не так, ");
            out.print("но что именно, непонятно.");
            out.println("Тип исключения неизвестен.");
        }

        out.println("Вышли из блоков try-catch.");
    }
}
```

Кроме листингов 12.5 и 12.6, в данной программе используется класс `OutOfRangeException`, приведенный в листинге 12.3.

В листинге 12.6 имитируется сценарий, в котором на вашем складе ограничено пространство. Вы не можете хранить более тысячи коробок одновременно, поэтому нужно проверить, превышает ли значение `numBoxesIn` этот порог. Если превышает, программа генерирует исключение `NumberTooLargeException`.

Однако, чтобы сделать жизнь немного интереснее, в листинге 12.6 нет блока `catch` для исключения `NumberTooLargeException`. Но несмотря на это программа заметила, что коробок слишком много, и сообщила об этом пользователю (рис. 12.6). Как это произошло? Дело в том, что класс исключения `NumberTooLargeException` наследует класс исключения `OutOfRangeException` (см. листинг 12.5). Сгенерировав исключение `NumberTooLargeException`, компьютер проходит по списку блоков `catch`. Дойдя до блока `OutOfRangeException`, компьютер видит, что этот класс наследуется исключением `NumberTooLargeException`, и принимает решение обработать его в данном блоке. Следовательно, блок `catch` перехватывает исключение не только тогда, когда оно **есть** в списке его параметров, но и когда оно **наследует** исключение, приведенное в списке его параметров.

```
<terminated> InventoryD [Java Application] D:\Progra
Сколько коробок вы насчитали? 3
Общая стоимость равна 9,75 руб.
Вышли из блоков try-catch.

<terminated> InventoryD [Java Application] D:\Progra
Сколько коробок вы насчитали? десять
Это не число.
Вышли из блоков try-catch.

<terminated> InventoryD [Java Application] D:\Progra
Сколько коробок вы насчитали? -25
-25 ??? Это невозможно!
Вышли из блоков try-catch.

<terminated> InventoryD [Java Application] D:\Progra
Сколько коробок вы насчитали? 1001
1001 ??? Это невозможно!
Вышли из блоков try-catch.
```

Рис. 12.6. Четыре сеанса работы листинга 12.6

Рассмотрим процесс перехвата исключений подробнее.

- ✓ **Пользователь вводит правильное целое число, например 3.** Все инструкции блока `try` выполняются. После этого компьютер пропускает все блоки `catch` и выполняет первую инструкцию, расположенную непосредственно после них (рис. 12.7).

```
try {
    //Нормальное выполнение без исключений
    }

catch (NumberFormatException e) {
    out.println("Это не число.");
}

catch (OutOfRangeException e) {
    out.print(numBoxesIn);
    out.println("??? Это невозможно!");
}

catch (Exception e) {
    out.print("Что-то здесь не так, ");
    out.print("но что именно, непонятно.");
    out.println("Тип исключения неизвестен.");
}

out.println("Вышли из блоков try-catch.");
```

Рис. 12.7. Не сгенерировано ни одного исключения

- ✓ **Пользователь вводит фразу, которая не является целым числом, например слово десять.** Виртуальная машина Java генерирует исключение `NumberFormatException`. Компьютер пропускает оставшуюся часть блока `try` и выполняет блок `catch`, в списке параметров которого есть класс `NumberFormatException`. После этого компьютер пропускает остальные блоки `catch` и выполняет инструкцию, расположенную непосредственно после них (рис. 12.8).

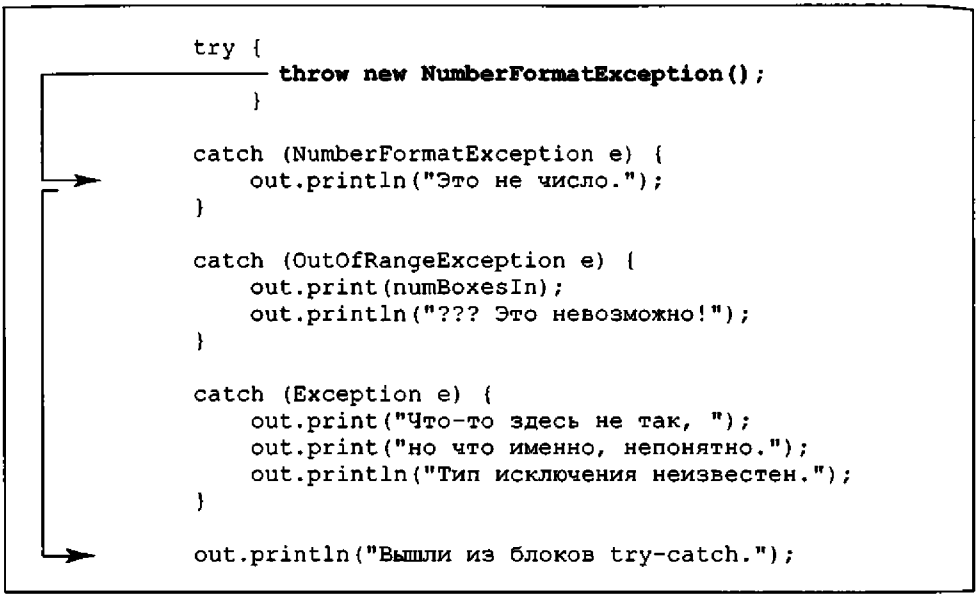


Рис. 12.8. Сгенерировано исключение `NumberFormatException`

- ✓ **Пользователь вводит отрицательное число, например -25.** Программа генерирует исключение `OutOfRangeException`. Компьютер пропускает оставшиеся инструкции блока `try` и первый блок `catch`, потому что `OutOfRangeException` не имеет никакого отношения и не наследует исключение `NumberFormatException`, приведенное в списке параметров первого блока `catch`. Компьютер выполняет второй блок `catch`, в списке параметров которого приведен класс `OutOfRangeException`. Затем компьютер пропускает третий блок `catch` и выполняет инструкцию, расположенную после него (рис. 12.9).
- ✓ **Пользователь вводит слишком большое число, например 1001.** Программа генерирует исключение `NumberTooLargeException`. Компьютер пропускает оставшиеся инструкции блока `try` и первый блок `catch` (потому что исключение `NumberTooLargeException` несовместимо с исключением `NumberFormatException`, указанным в списке параметров первого блока `catch`.

  В соответствии с объявлением класса `NumberTooLargeException` в листинге 12.5, он наследует класс `OutOfRangeException`. Когда компьютер

доходит до второго блока `catch`, он видит, что класс текущего исключения наследует класс параметра блока `catch`. Поэтому он выполняет данный блок. После этого компьютер пропускает третий блок `catch` и выполняет инструкцию, расположенную после него (рис. 12.10).

```
try {
    throw new OutOfRangeException();
}

catch (NumberFormatException e) {
    out.println("Это не число.");
}

catch (OutOfRangeException e) {
    out.print(numBoxesIn);
    out.println("??? Это невозможно!");
}

catch (Exception e) {
    out.print("Что-то здесь не так, ");
    out.print("но что именно, непонятно.");
    out.println("Тип исключения неизвестен.");
}

out.println("Вышли из блоков try-catch.");
```

Рис. 12.9. Сгенерировано исключение `OutOfRangeException`

```
try {
    throw new NumberTooLargeException();
}

catch (NumberFormatException e) {
    out.println("Это не число.");
}

catch (OutOfRangeException e) {
    out.print(numBoxesIn);
    out.println("??? Это невозможно!");
}

catch (Exception e) {
    out.print("Что-то здесь не так, ");
    out.print("но что именно, непонятно.");
    out.println("Тип исключения неизвестен.");
}

out.println("Вышли из блоков try-catch.");
```

Рис. 12.10. Сгенерировано исключение `NumberTooLargeException`

- ✓ **В блоке `try` происходит нечто непредвиденное, например возникает ошибка ввода-вывода и генерируется исключение `IOException` (исключение ввода-вывода).** В принципе, в данном листинге она не может произойти, однако давайте для эксперимента сгенерируем исключение `IOException` искусственно и посмотрим, что произойдет. Это можно сделать с помощью инструкции `throw new IOException()`.

  Когда код сгенерирует исключение `IOException`, компьютер пропустит оставшиеся инструкции блока `try` и два первых блока `catch`, потому что они не имеют отношения к данному исключению. Подойдя к третьему блоку `catch`, компьютер обнаружит, что текущее исключение наследует исключение `Exception`, которое является базовым для всех классов исключений. Поэтому компьютер выполнит третий блок `catch` и перейдет к выполнению инструкции, расположенной непосредственно после этого блока (рис. 12.11).

```
try {
    throw new IOException();
}

catch (NumberFormatException e) {
    out.println("Это не число.");
}

catch (OutOfRangeException e) {
    out.print(numBoxesIn);
    out.println("??? Это невозможно!");
}

catch (Exception e) {
    out.print("Что-то здесь не так, ");
    out.print("но что именно, непонятно.");
    out.println("Тип исключения неизвестен.");
}

out.println("Вышли из блоков try-catch.");
```

Рис. 12.11. Сгенерировано исключение `IOException`

Теперь мы можем сформулировать общее правило выбора подходящего блока `catch`. Когда блок `try` генерирует исключение и компьютер ищет подходящий блок `catch`, он выбирает самый первый сверху, удовлетворяющий одному из следующих условий:

- ✓ класс параметра блока `catch` совпадает с классом текущего исключения;
- ✓ класс параметра блока `catch` является базовым для класса текущего исключения.

Если ниже в списке есть лучшее совпадение, компьютер не найдет его. Предположим, вы добавили блок `catch` с параметром типа `NumberTooLargeException`,

но не до, а после блока `catch` с параметром `OutOfRangeException`. Тогда при исключении `NumberTooLargeException` компьютер выполнит блок с параметром `OutOfRangeException`, хотя есть блок с параметром, тип которого точно совпадает с типом текущего исключения.

Блок `catch` с несколькими типами исключений

Начиная с Java 7, в одном блоке `catch` можно задать перехват исключений нескольких типов. Предположим, в предыдущем примере нужно обрабатывать исключения `NumberFormatException` и `OutOfRangeException` в одном блоке. Для этого можно переписать фрагмент листинга 12.6 следующим образом.

```
try {
        int numBoxes = Integer.parseInt(numBoxesIn);

        if (numBoxes < 0) {
            throw new OutOfRangeException();
        }

        if (numBoxes > 1000) {
            throw new NumberTooLargeException();
        }

        out.print("Общая стоимость равна ");
        out.println(currency.format(numBoxes * boxPrice));
    }

    catch (NumberFormatException|OutOfRangeException e) {
        out.println(" ??? Это невозможно!");
    }

     catch (Exception e) {
        out.print("Что-то здесь не так, ");
        out.print("но что именно, непонятно.");
        out.println("Тип исключения неизвестен.");
    }
```

Символ | сообщает компилятору о том, что блок должен перехватить исключение, принадлежащее любому из указанных типов: `NumberFormatException` или `OutOfRangeException`. Если блок `try` сгенерирует исключение любого из этих типов, компьютер отобразит сообщение `Это невозможно`. Если же блок `try` сгенерирует исключение какого-либо другого типа, будет выполнен следующий блок `catch`, потому что исключение `Exception` является базовым для исключений любого типа.

Не будем чрезмерно осторожничать

Персонаж рассказа А.П. Чехова "Человек в футляре" всегда, даже в солнечный день, ходил с зонтиком, на случай, если вдруг пойдет дождь, и часто повторял фразу "Как бы чего не вышло". Должны ли мы применить подобный подход и попытаться заранее запрограммировать перехват любых возможных исключений? Видимо, нет. Все хорошо в меру. К тому же компилятор Java не позволяет нам впасть в паранойю и

отказывается компилировать код, перехватывающий исключения, которые не могут произойти.

Предположим, в блоке `try` находится безобидная инструкция `i++`. Это вполне допустимо. Однако рассмотрим, что будет, если блок `catch` попытается перехватить исключение `IOException` (исключение ввода-вывода), которое в инструкции `i++` не может произойти.

```
// Неправильный код!
try {
   i++;
} catch (IOException e) {
   e.printStackTrace();
}
```

Инструкция `i++` не выполняет операции ввода-вывода, поэтому в блоке `try` не может произойти исключение `IOException`. Компилятор отреагирует на это тем, что откажется пропустить программу на выполнение и отобразит сообщение `IOException is never thrown` (Исключение IOException никогда не генерируется).

Восстановление работы программы после исключения

В предыдущих примерах блок `catch` отображал на консоли сообщение об ошибке, и на этом лавочка закрывалась. Конечно, было бы лучше, если бы программа не только сообщила об ошибке, но и предоставила пользователю возможность ее исправить. Но как это сделать? Ведь после блока `catch` управление не возвращается в блок `try`. Решение состоит в том, чтобы заключить блоки `try` и `catch` в цикл (рис. 12.7). Тогда все будет повторяться сначала, пока пользователь не введет правильные данные.

Листинг 12.7. Повторение попыток ввести правильные данные

```
import static java.lang.System.out;
import java.util.Scanner;
import java.text.NumberFormat;

class InventoryLoop {

   public static void main(String args[]) {
      final double boxPrice = 3.25;
      boolean gotGoodInput = false;
      Scanner keyboard = new Scanner(System.in);
      NumberFormat currency = NumberFormat.getCurrencyInstance();

      do {
         out.print("Сколько коробок вы насчитали? ");
         String numBoxesIn = keyboard.next();

         try {
            int numBoxes = Integer.parseInt(numBoxesIn);
            out.print("Общая стоимость равна ");
            out.println(currency.format(numBoxes * boxPrice));
            gotGoodInput = true;
```

```
        } catch (NumberFormatException e) {
            out.println();
            out.println("Это не целое число.");
        }
    } while (!gotGoodInput);

    out.println("Отлично!");
  }
}
```

На рис. 12.12 показан результат выполнения листинга 12.7. Первые три попытки неудачные: пользователь вводит строку, которую компьютер не может интерпретировать как целое число. И наконец, при четвертой попытке пользователь вводит нечто такое, против чего код не возражает.

```
<terminated> InventoryLoop [Java Application] D:\Prog
Сколько коробок вы насчитали? 3.5

Это не целое число.
Сколько коробок вы насчитали? три

Это не целое число.
Сколько коробок вы насчитали? чаво?

Это не целое число.
Сколько коробок вы насчитали? 3
Общая стоимость равна 9,75 руб.
Отлично!
```

Рис. 12.12. Выполнение листинга 12.7

Наши друзья — хорошие исключения

У вас могло сложиться впечатление, будто исключения всегда происходят вследствие ошибок и других нежелательных ситуаций. В большинстве случаев это так. Но иногда исключение генерируется при нормальной работе программы, как было задумано программистом. Например, исключение может применяться для обнаружения конца файла. Рассмотрим следующий код, копирующий файл из потока `dataIn` в поток `dataOut`.

```
try {
  while (true) {
    dataOut.writeByte(dataIn.readByte());
  }
} catch (EOFException e) {
    numFilesCopied = 1;
}
```

Копирование выполняется в цикле `while` по одному байту. Условие цикла содержит выражение `true`, поэтому, казалось бы, цикл будет продолжаться бесконечно. Однако программа все же когда-нибудь достигнет конца файла. Когда это произойдет, метод `readByte()` сгенерирует исключение `EOFException` (исключение конца

файла). В результате компьютер выйдет из блока try и одновременно из цикла while. После этого можно сделать все, что нужно, в блоке catch и возобновить нормальный процесс вычислений.

Обработайте исключение или передайте его дальше

Итак, вы дошли до главы 12. Вы многое узнали и проделали много работы. Не устали ли вы? Как насчет того, чтобы вздремнуть несколько минут? Но, наверное, несколько минут — это слишком много, давайте остановимся на пяти секундах.

В листинге 12.8 приведена программа, которая приостанавливает свою работу на пять секунд, давая возможность себе и пользователю немного отдохнуть и собраться с мыслями. Проблема состоит в том, что эта программа не компилируется по причинам, о которых я расскажу ниже.

Листинг 12.8. Вздремнуть нам пока что не удастся

```
// Этот код не компилируется

import static java.lang.System.out;

class NoSleepForTheWeary {

    public static void main(String args[]) {
        out.print("Вздремну-ка я, ");
        out.println("всего пять секунд...");

        takeANap();

        out.println("Я опять бодр и полон сил!");
    }

    static void takeANap() {
        Thread.sleep(5000);
    }
}
```

Стратегический подход, применяемый в листинге 12.8, правильный. Программа вызывает метод sleep() класса Thread, принадлежащего Java API. Число, передаваемое методу, — это количество миллисекунд. Выражение Thread.sleep(5000) означает, что выполнение текущего потока будет приостановлено на пять секунд.

Проблема состоит в том, что метод sleep() может сгенерировать исключение InterruptedException (исключение прерывания). Поэтому при попытке скомпилировать этот код компилятор вернет сообщение о том, что необходимо перехватить и обработать исключение InterruptedException. В результате компилятор не передаст программу на выполнение.



Исключение InterruptedException генерируется, когда какой-либо другой код прерывает паузу данного кода. В Java несколько потоков кода могут выполняться одновременно. Поэтому может случиться так, что один поток

вызывает метод `Thread.sleep()`, а в это время другой поток вызывает метод `interrupt()`. Второй поток хочет временно приостановить первый, чтобы выполнить какую-либо срочную работу, но первый поток и без того уже в режиме паузы. В результате компьютер не понимает, что от него хотят: отменить режим паузы или оставить все как есть. В такой ситуации виртуальная машина Java обычно генерирует исключение.

В языке Java есть исключения двух видов.

- ✓ *Проверяемые исключения* должны быть перехвачены и обработаны в коде, иначе программа будет аварийно завершена.
- ✓ *Непроверяемые исключения* можно не перехватывать и не обрабатывать. Хотя при необходимости обработка такого исключения может быть полезной.

Исключение `InterruptedException` принадлежит к проверяемым. Когда код вызывает метод, который может сгенерировать проверяемое исключение, нужно в этом же коде запрограммировать его перехват и обработку, иначе код не будет компилироваться.

Таким образом, если код метода может сгенерировать проверяемое исключение, необходимо запрограммировать реакцию на него одним из двух способов.

- ✓ Заключить место, в котором может произойти исключение, в блок `try`. Добавить блок `catch` (возможно, пустой), принимающий исключение данного типа.
- ✓ Добавить в заголовок метода директиву `throws` и тип исключения (можно добавить базовый тип `Exception`, обозначающий любое исключение).

Проиллюстрируем эти два пункта в следующих двух листингах. В листинге 12.9 метод `sleep()`, который может сгенерировать исключение, находится в блоке `try`. Непосредственно после него расположен блок `catch`, в списке параметров которого приведен тип `InterruptedException`.

Листинг 12.9. Обработка исключения внутри метода

```
import static java.lang.System.out;

class GoodNightsSleepA {

    public static void main(String args[]) {
        out.print("Немного вздремну. ");
        out.println("Всего пять секунд...");

        takeANap();

        out.println("Я опять бодр и полон сил!");
    }

    static void takeANap() {
```

```
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            out.println("Эй! Кто разбудил меня?");
        }
    }
}
```

Сейчас я должен написать, как обычно в этой книге, что результат выполнения листинга такого-то показан на рисунке таком-то. Но проблема состоит в том, что рис. 12.13 иллюстрирует менее важный случай, когда исключения не произошло. Код компилируется потому, что вызов метода `sleep()` заключен в блок `try` и добавлен блок `catch`. Конечно, интереснее было бы показать, как обрабатывается исключение, но его в данном случае сложно сгенерировать. Нужно создать еще один поток, а в нем создать кнопку, на которой пользователь щелкнет в момент паузы первого потока. Поэтому, чтобы не усложнять пример, ограничимся иллюстрацией более простого случая.

```
<terminated> GoodNightsSleepA [Java Application] D:\Progr
Немного вздремну. Всего пять секунд...
Я опять бодр и полон сил!
```

Рис. 12.13. Вторая строка появится после паузы длительностью пять секунд

Теперь проиллюстрируем второй способ реагирования на исключение, состоящий в добавлении директивы `throws` в заголовок метода (листинг 12.10). Директива `throws` сообщает компилятору о том, что в случае возникновения исключения оно будет передано вызывающему методу.

Листинг 12.10. Использование директивы `throws`

```
import static java.lang.System.out;

class GoodNightsSleepB {

    public static void main(String args[]) {
        out.print("Немного вздремну. ");
        out.println("Всего пять секунд...");

        try {
            takeANap();
        } catch (InterruptedException e) {
            out.println("Эй! Кто разбудил меня?");
        }

        out.println("Я опять бодр и полон сил!");
    }

    static void takeANap() throws InterruptedException {
        Thread.sleep(5000);
    }
}
```

Результат тот же, что и в предыдущем листинге. Его можно увидеть на рис. 12.13. Как и в предыдущем случае, не забывайте, что компьютер "спит" пять секунд, прежде чем отобразить второе сообщение.

Важная часть листинга 12.10 — заголовок метода `takeANap()`. Он содержит директиву `throws InterruptedException`, которая сообщает компилятору о том, что метод `takeANap()` может передать вызывающему методу исключение указанного типа. Этим метод как бы освобождает себя от обязанности обработать исключение и передает эту обязанность вызывающему методу.

Соответственно, теперь обязанность обработать исключение лежит на методе `main()`. Для этого в его теле вызов метода `takeANap()` заключен в блок `try`. Кроме того, в тело метода `main()` добавлен блок `catch` с параметром `InterruptedException`. Теперь компилятор не выдвигает никаких возражений. Обязанность обработать исключение перешла к методу `main()`, и он готов его обработать.

Чтобы лучше понять директиву `throws`, представьте себе игру в волейбол, в которой исключение интерпретируется как волейбольный мяч. Когда игрок подает мяч, он как бы генерирует исключение. Мяч пересекает сетку и попадает к вам. Вы можете либо перебросить его через сетку (т.е. обработать исключение), либо передать другому члену вашей команды, который, в свою очередь, также может либо обработать исключение, либо передать его другому.



Инструкция, расположенная в теле метода, может сгенерировать исключение, не обрабатываемое блоком `catch` вследствие несовпадения типов. Кроме того, исключение может быть сгенерировано инструкцией, не входящей в блок `try`. В любом из этих случаев управление передается вызывающему методу, т.е. за пределы метода, содержащего инструкцию, которая сгенерировала исключение.



В директиве `throws` можно задать несколько типов исключений. Типы исключений должны быть разделены запятыми.

```
throws InterruptedException, IOException, ArithmeticException
```

В Java API определено несколько сотен типов исключений. Некоторые из них наследуют класс `RuntimeException`. Эти исключения относятся к категории непроверяемых (см. ранее). Любое исключение, не наследующее класс `RuntimeException`, относится к категории проверяемых. К этой категории принадлежат исключения, которые компьютеру сложно предвидеть и правильно обработать. Например, исключение `NumberFormatException` (см. листинги 12.2, 12.4 и др.) возникает, когда компьютер не может преобразовать строку в число. В большинстве случаев это невозможно сделать вследствие опечатки в строке. На практике часто возникают исключения `IndexOutOfBoundsException` (индекс нарушил границы) и `NullPointerException` (указатель имеет неопределенное значение). В любом из этих случаев компьютер не может самостоятельно решить, что делать дальше.

В библиотеках Java API определено довольно много проверяемых исключений. Компьютер не может их проигнорировать, и в каждом таком случае он попадает в безвыходную ситуацию. Поэтому компилятор Java настаивает на том, чтобы каждое такое исключение было перехвачено и обработано кодом программы.

Блок finally

Однажды, когда я был еще совсем молодым и только получил водительские права, я взял у родителей автомобиль и поехал к своему другу. По дороге я думал о том, кого встречу у него. От этой мысли меня отвлек сильный удар. Другой автомобиль врезался в правую дверцу моей машины. Произошло исключение типа `RunARedLightException` (проезд на красный свет).

Оба автомобиля остались работоспособными и могли самостоятельно перемещаться. Дорога была оживленной и, чтобы не создавать пробку, мы отъехали на обочину. Я нашел в отсеке под приборной панелью необходимые документы и вышел из машины. Но я был так поглощен и обескуражен происшествием, что не заметил ехавшего прямо на меня автобуса, который вырвал у меня дверцу и бросил меня на капот. Произошло исключение типа `DealWithLawyersException` (работа для адвокатов).

Все закончилось благополучно. Я был весь в синяках, но серьезных травм не получил. Мои родители заплатили за поврежденный чужой автомобиль, поэтому я не понес финансовых потерь (я передал исключение `RunRedLightException` родителям с помощью директивы `throws`).

Этот инцидент поможет нам понять, как обрабатывается исключение. Что произойдет, если во время обработки одного исключения возникнет другое? Ведь инструкции внутри блока `catch` также не застрахованы от исключений.

Одно из решений (как вы дальше увидите, плохое) состоит в том, чтобы поместить еще один блок `try` внутрь блока `catch`. Это предохранит программу от неожиданных инцидентов во время выполнения блока `catch`. Однако для дополнительного блока `try` нужен свой блок `catch`. Таким образом, проблема не решается, а отодвигается, причем уровня вложенности, на котором она будет решена окончательно, не существует.

Другое, лучшее, решение проблемы состоит в создании блока `finally`. Как и блок `catch`, он располагается после блока `try`. Его существенное отличие от блоков `try` и `catch` состоит в том, что все его инструкции всегда выполняются независимо от того, возникло ли исключение и было ли оно обработано. Данная концепция иллюстрируется в листинге 12.11.

Листинг 12.11. Использование блока `finally`

```
import static java.lang.System.out;

class DemoFinally {

    public static void main(String args[]) {
        try {
            doSomething();
        } catch (Exception e) {
         out.println("Исключение перехвачено в main().");
        }
    }

    static void doSomething() {
        try {
```

```
            out.println(0 / 0);
        } catch (Exception e) {
            out.println("Исключение перехвачено в doSomething().");
            out.println(0 / 0);
        } finally {
            out.println("Напечатано в блоке finally.");
        }

        out.println("Будет ли это напечатано?");
    }
}
```

Когда вы думаете о блоке `try`, подумайте и о том, как выполнение программы будет восстановлено после аварийной ситуации. Восстановление программы происходит в блоке `catch`. Затем выполнение продолжается, начиная с первой инструкции после блоков `try-catch`. Если же что-нибудь происходит при выполнении блока `catch`, события развиваются иначе.

Результат работы листинга 12.11 показан на рис. 12.14. Сначала метод `main()` вызывает метод `doSomething()`. После этого метод `doSomething()` начинает создавать многочисленные неприятности. Сначала он пытается поделить нуль на нуль, что невозможно в любом языке программирования. Эта операция порождает исключение `ArithmeticException`, которое перехватывается единственным в методе блоком `catch`.

```
<terminated> DemoFinally [Java Application] D:\Program Files\Ja
Исключение перехвачено в doSomething().
Напечатано в блоке finally.
Исключение перехвачено в main().
```

Рис. 12.14. Выполнение листинга 12.11

Внутри блока `catch` метод `doSomething()` опять пытается поделить нуль на нуль. На этот раз инструкция, выполняющая незаконное деление, не находится внутри защитного блока `try`. Это допустимо, потому что исключение `ArithmeticException` не является проверяемым. Оно наследует класс `RuntimeException`, поэтому перехватывать и обрабатывать его необязательно (см. предыдущий раздел).

В любом случае, перехвачено оно или не перехвачено, второе исключение `ArithmeticException` приводит к тому, что управление выпадает из метода `doSomething()`. Но, прежде чем покинуть метод `doSomething()`, компьютер выполняет блок `finally` — последнюю волю гибнущего блока `try`. Вот почему на рис. 12.14 отображена фраза `Напечатано в блоке finally`.

Интересно отметить, что фразы `Будет ли это напечатано?` на рис. 12.14 нет. Блок `catch` сгенерировал собственное исключение (вместо того чтобы обработать исключение, блок `catch` сгенерировал еще одно). Оно не было перехвачено, поэтому компьютер не продолжил нормальное выполнение инструкций после блоков `try-catch-finally`, а немедленно вышел из метода `doSomething()`.

Итак, компьютер вернулся в метод `main()`. Но из метода `doSomething()` он вышел не в нормальном режиме, а по исключению `ArithmeticException`, которое

произошло в методе `doSomething()`. Поэтому компьютер выполняет блок `catch`, расположенный в методе `main()`. Завершающий аккорд. На этом какофония исключений наконец-то заканчивается.

Закрытие файлов

В 4839 году обитатели планеты Земля откроют капсулу с экземпляром книги *Java для чайников*. Наверняка они заметят, что в некоторых примерах книги есть одна и та же серьезная ошибка: "В главе 8 Барри читает с диска файл `EmployeeInfo.txt`. Программа `DoPayroll` захватывает этот ресурс и не отпускает его до конца сеанса. Как говорят теперь, в пятом тысячелетии, Барри открыл файл, но поленился его закрыть".

Чтобы они так не думали, спешу объяснить, что я не закрыл файл потому, что не хотел перегружать читателя дополнительной информацией. В простых случаях виртуальная машина Java закрывает файлы автоматически при завершении работы программы, однако это происходит не всегда. В 2978 году читатель воспринял пример Барри слишком прямолинейно и не закрыл файл в коде. Это привело в 2980 году к восстанию роботов, краху пост-пост-постиндустриальной экономики и отказу от Американской мечты.

"Во всем этом виноват Барри, — скажет обитатель Земли в 4839 году, — поэтому мы не будем извлекать его из криогенной капсулы".

Как закрыть файл

Надеясь ожить в 4840 году, я добавлю одну строку кода в листинг 8.2. Сейчас листинг 8.2 содержит следующую инструкцию, которая открывает файл.

```
Scanner diskScanner = new Scanner(new File("EmployeeInfo.txt"));
```

В конце метода `main()` я добавлю следующую инструкцию, которая закрывает файл:

```
diskScanner.close()
```

Конечно, эту инструкцию можно вставить в любом месте программы. Наилучшая стратегия состоит в вызове метода `close()` немедленно после обращения к файлу `EmployeeInfo.txt` в последний раз.

В этом месте читатель в 4839 году, возможно, скажет: "Поздно спохватился! Операция закрытия файла не имеет никакого отношения к теме главы 12". Однако в этом будущий читатель не прав.

Ресурсы в заголовке блока `try`

Для закрытия файла достаточно вызвать метод `close()`. Однако есть существенная проблема: исключение может возникнуть до его вызова. Листинг 12.12 почти идентичен листингу 8.2. Добавлена только одна инструкция, отмеченная полужирным шрифтом.

Листинг 12.12. Закрыт ли файл?

```
import java.util.Scanner;
import java.io.File;
import java.io.IOException;
```

```
class DoPayroll {
  public static void main(String args[]) throws IOException {
    Scanner diskScanner = new Scanner(new File("EmployeeInfo.txt"));
    for (int empNum = 1; empNum <= 3; empNum++) {
      payOneEmployee(diskScanner);
    }
    diskScanner.close();
  }

static void payOneEmployee(Scanner aScanner) {
  Employee anEmployee = new Employee();
  anEmployee.setName(aScanner.nextLine());
  anEmployee.setJobTitle(aScanner.nextLine());
  anEmployee.cutCheck(aScanner.nextDouble());
  aScanner.nextLine();
  }
}
```

Листинг 12.12 выглядит прекрасно, но видимость может быть обманчивой. Если в теле метода `payOneEmployee` возникнет исключение, программа будет завершена аварийно до вызова метода `diskScanner.close()`. Я по-прежнему рискую остаться в криогенной капсуле.

Можно заключить часть инструкций в блок `try` и добавить блок `finally`, но когда в программе используются многие ресурсы (файлы, базы данных, сетевые соединения и т.п.), структура блоков `try` и директив `throws` может стать слишком сложной, потому что блоки `try` придется включать также в блоки `catch` и применять другие запутанные комбинации.

Оптимальное решение данной проблемы введено в версии Java 7: ресурсы можно включать в заголовок блока `try` (листинг 12.13).

Листинг 12.13. Открытие файла в заголовке блока `try`

```
import java.util.Scanner;
import java.io.File;
import java.io.IOException;

class DoPayroll {

   public static void main(String args[]) throws IOException {

      try (Scanner diskScanner = new Scanner(new
                                        File("EmployeeInfo.txt"))) {

         for (int empNum = 1; empNum <= 3; empNum++) {
            payOneEmployee(diskScanner);
         }
      }
   }

   static void payOneEmployee(Scanner aScanner) {
      Employee anEmployee = new Employee();
```

```
        anEmployee.setName(aScanner.nextLine());
        anEmployee.setJobTitle(aScanner.nextLine());
        anEmployee.cutCheck(aScanner.nextDouble());
        aScanner.nextLine();
    }
}
```

В листинге 12.13 объявление ресурса `diskScanner` заключено в круглые скобки и помещено после ключевого слова `try`. Это означает, что виртуальная машина Java автоматически закроет объект `diskScanner` после выполнения инструкций в блоке `try`. В заголовке блока `try` можно объявить несколько ресурсов. Тогда виртуальная машина Java закроет их все после выполнения блока `try`. Можно также добавить блоки `catch` и `finally`. Данное средство особенно полезно при использовании соединений с базами данных, потому что позволяет быть уверенным в том, что соединение будет разорвано автоматически в любом случае.

Глава 13

Область видимости

В этой главе...

- Модификаторы доступа к членам классов
- Классы, доступ и части программы
- Как изменить программу, не изменяя классы
- Модификаторы доступа к классам

Я обедаю с коллегами в небольшом ресторанчике рядом с офисом. Один из коллег говорит: "Они могут читать твою электронную почту". Другой говорит: "У них есть полный список сайтов, которые ты посетил. Они знают, что ты ешь за обедом, что покупаешь, что носишь и о чем думаешь. Они знают твои самые большие секреты, скрытые страхи и желания. Не удивлюсь, если они даже знают, когда ты умрешь".

Третий, не забывая накладывать салат в свою тарелку, говорит: "Все идет к тому, что, когда ты высморкаешься, этот факт будет зафиксирован в базе данных. Недавно я открыл один веб-сайт и увидел на нем поздравление с днем рождения. Откуда они узнали, что именно я посетил сайт и что у меня сегодня день рождения?"

"О, да! — говорит первый. — На моей машине есть идентификационный чип. Когда я проезжаю через пункт сбора платежей, нужная сумма вычитается с моей банковской карточки автоматически. Каждый месяц я получаю от компании длинный список платежей, в котором зафиксировано, где и когда я проезжал. В ближайшем будущем, наверное, я буду читать в этом списке, к кому я ездил и что там делал".

Я не вмешиваюсь в разговор и думаю про себя: "Ну и что? Лично я был бы польщен, если бы мой работодатель, мои родственники, местный шериф, большая компания и другие люди интересовались моими делами. Нам часто приходится прилагать огромные усилия, чтобы привлечь чье-либо внимание. Каким покинутым и забытым себя чувствуешь, когда видишь, что никому до тебя нет дела! Многие агентства, записывающие в журнал мои покупки, фиксирующие мои привычки и присылающие рекламные буклеты, часто даже не могут написать или произнести мое имя без ошибки. "Алло, будьте любезны, позовите, пожалуйста, мистера Ларри Берга. Он дома? Ах, да, Барри Берд... Прошу прощения! Конечно, Барри Берд". Наверное, шпионить за людьми — очень скучное занятие. Было бы забавно увидеть на первой полосе *The Times*, что "Барри Берд, автор знаменитого бестселлера *Java для чайников*, сегодня опять одел трусы наизнанку!"

Подумав несколько секунд, я включаюсь в разговор: "А видеокамеры? Теперь они на каждом шагу... Жить становится все интереснее!"

Модификаторы доступа к членам классов

Если вы дошли до этого места в книге, значит, вы уже знаете, что в объектно-ориентированном программировании много внимания уделяется сокрытию одних кодов от других. Программист, пишущий одну часть кода, не должен менять другие части кода, написанные другими программистами. Это обусловлено не требованиями безопасности или секретности, а принципом модульности. Скрывая подробности, вы фактически прячете сложность своего кода и делаете его простым для использования. Ваш код выглядит, как черный ящик с простыми входами и выходами. Вы упрощаете задачу другим людям. Вы облегчаете их жизнь, предохраняете их от ошибок, экономите их деньги, бережете их время и способствуете продлению их жизни. Впрочем, вы и себе облегчаете этим жизнь, потому что уменьшаете вероятность конфликтов имен в своем коде.

В предыдущих главах на каждом шагу встречались примеры использования закрытых полей. Когда поле объявлено с модификатором `private`, оно невидимо за пределами класса и недоступно для изменения. Сокрытие полей и методов улучшает модульность и уменьшает сложность кода.

Но не все поля в примерах данной книги закрытые, есть также немало открытых, т.е. объявленных с модификатором `public`. Как и публичный политик, личные дела которого открыты для всеобщего обозрения, открытое поле доступно всем и каждому, т.е. к нему можно обращаться в любом месте программы. Многие знают, какую зубную пасту использовал Элвис Пресли. Аналогично этому любой программист может сослаться в данной программе на открытую переменную `Elvis`. Разница лишь в том, что открытую переменную можно как читать, так и изменять, а название зубной пасты Элвиса любой его поклонник может только узнать, изменить его он не может.

Ключевые слова `public` и `private` называются *модификаторами доступа*. В примерах этой книги вам часто встречались поля и методы, объявленные без модификаторов доступа. Такие поля и методы имеют доступ, установленный по умолчанию. В большинстве глав книги не имеет значения, что такое доступ, установленный по умолчанию, но не в данной главе. Здесь мы рассмотрим этот вопрос подробнее.

В данной главе рассматривается также еще один модификатор доступа, который не использовался в предыдущих главах. Это модификатор `protected`, определяющий *защищенный доступ*.

Классы, доступ и части программы

Изучая правила доступа, нам придется немного углубиться в специальную терминологию. Некоторые термины уже упоминались в главе 10, но полезно будет повторить их еще раз. Рассмотрим следующий код.

```
class MyClass {
  int myField;          //Поле, член класса.

  void myMethod() {     //Метод, член класса.

    int myOtherField; //Локальная переменная,
                      //не член класса.

  }
}
```

Все, что нам нужно, написано в комментариях кода. Существуют два вида переменных — члены класса и локальные переменные. С локальной переменной все просто: она доступна только в теле метода. Поэтому в данной главе мы больше не будем о ней говорить. Глава посвящена членам класса — полям и методам.

Классы и члены классов

Начиная с этого момента вы должны строго различать, где используется модификатор доступа. В предыдущих главах книги вы размещали их в объявлениях членов класса. Например, можно написать так:

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

Или так:

```
public amountInAccount = 50.22;
```

Использование ключевого слова `public` в этом месте не удивит вас. Но вы, возможно, еще не знаете, что его можно применить и в определении класса, например так:

```
public class Drawing {
```

Ключевое слово `public` имеет в Java два немного разных значения: одно — для членов класса, другое — для классов. В данной главе ключевое слово `public` (и другие модификаторы доступа) рассматривается применительно к членам класса, но в конце главы кратко упоминается и его применение в заголовке класса.

Правила доступа к членам класса

Этот раздел посвящен исключительно членам класса. Но это не значит, что классы в нем игнорируются. В Java все делается только в классах. Каждое поле может быть объявлено только в некотором классе; оно принадлежит этому классу и является его членом. То же самое справедливо и для методов. Важный вопрос, которому посвящена вся данная глава, — можно ли использовать имя некоторого члена класса в другом месте кода. Для ответа на него в первую очередь посмотрите, где находится это место: внутри или вне класса, в котором определен данный член класса.

- ✓ Если член класса объявлен как закрытый (с модификатором `private`), то обратиться к нему можно только внутри данного класса.

```
class SomeClass {
  private int myField = 10;
}

class SomeOtherClass {
  public static void main(String args[]) {
      SomeClass someObject = new SomeClass();

      //Этот код не компилируется
      System.out.println(someObject.myField);
  }
}
```

✓ Если член класса объявлен как открытый (с модификатором `public`), то к нему можно обратиться по имени в любой точке программы.

```
class SomeClass {
  public int myField = 10;
}

class SomeOtherClass {
  public static void main(String args[]) {
      SomeClass someObject = new SomeClass();

      //Этот код компилируется
      System.out.println(someObject.myField);
  }
}
```

На рис. 13.1–13.3 эти правила иллюстрируются в разных ситуациях и с производными классами. На рис. 13.1 показана структура классов, а на рис. 13.2 и 13.3 отмечены классы, в которых доступно открытое или закрытое поле `field`, объявленное в классе `ClassB`.

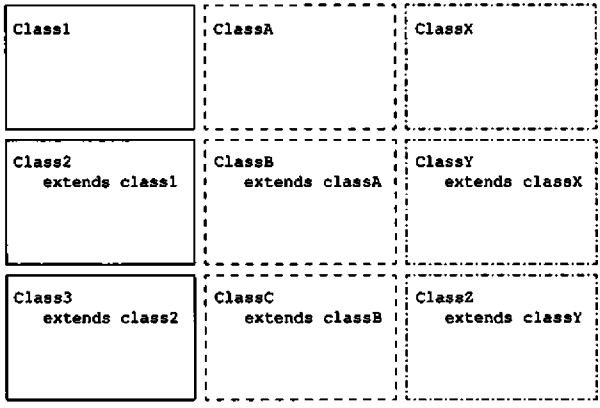


Рис. 13.1. Несколько классов и производных классов

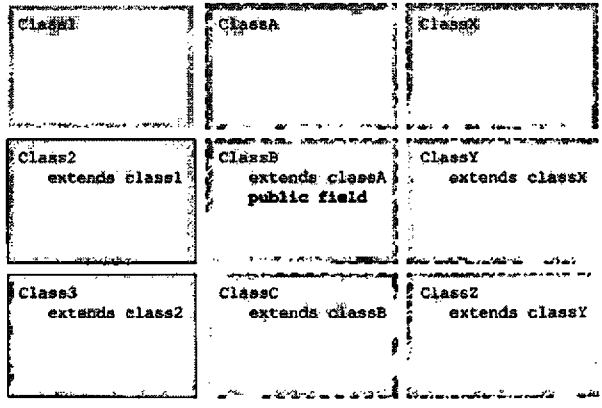


Рис. 13.2. Затенены классы, в которых доступно открытое поле, объявленное в классе `ClassB`

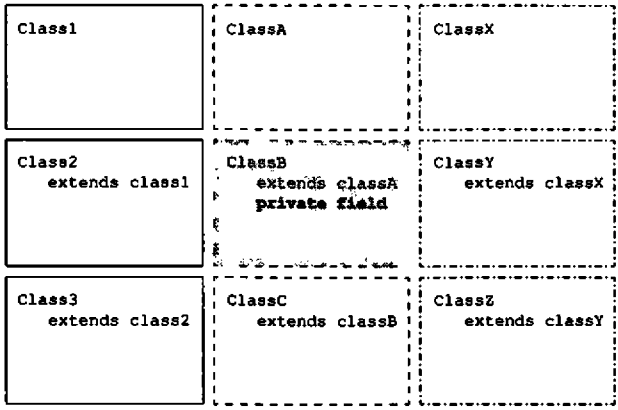


Рис. 13.3. Затенен класс, в котором доступно закрытое поле, объявленное в классе `ClassB`

Пример с рисунком во фрейме

Рассмотрим использование модификаторов доступа на нескольких примерах. В первом примере все поля и методы открыты. При открытом доступе вам не нужно беспокоиться о том, что и где можно использовать.

Код первого примера состоит из нескольких частей. Первая часть — листинг 13.1 — отображает на экране фрейм `ArtFrame`. На поверхности фрейма `ArtFrame` отображается рисунок `Drawing`, представляющий собой небольшую окружность. Если классы `ArtFrame` и `Drawing` доступны, листинг 13.1 отобразит окно, показанное на рис. 13.4.

Листинг 13.1. Отображение фрейма `ArtFrame`

```
import com.burdbrain.drawings.Drawing;
import com.burdbrain.frames.ArtFrame;

class ShowFrame {

    public static void main(String args[]) {
        ArtFrame artFrame = new ArtFrame(new Drawing());

        artFrame.setSize(200, 100);
        artFrame.setVisible(true);
    }
}
```

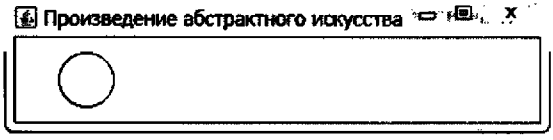


Рис. 13.4. Фрейм `ArtFrame`

Код листинга 13.1 создает экземпляр класса `ArtFrame`. Как и следовало ожидать, класс фрейма `ArtFrame` наследует стандартный библиотечный класс `JFrame`. В главе 9 было сказано, что в Java фреймы по умолчанию невидимы. Поэтому в листинге 13.1 вызывается метод `setVisible()`, который делает фрейм `ArtFrame` видимым.

Обратите внимание на два объявления `import`, которыми начинается листинг 13.1. Первое объявление позволяет применять сокращенное имя класса `Drawing`, находящегося в пакете `com.burdbrain.drawings`, вместо полностью квалифицированного имени. Второе объявление `import` позволяет применять имя `ArtFrame`.



Объявления `import` рассматриваются в главе 4.

Глядя на эти объявления `import`, нетрудно догадаться, что код программы не ограничивается листингом 13.1. Другие части программы находятся в пакетах `com.burdbrain.drawings` и `com.burdbrain.frames`. Чтобы программа работала, я создал классы `Drawing` и `ArtFrame` и поместил их в эти пакеты.

Код класса `Drawing` приведен в листинге 13.2.

Листинг 13.2. Класс `Drawing`

```
package com.burdbrain.drawings;

import java.awt.Graphics;

public class Drawing {
    public int x = 40, y = 40, width = 40, height = 40;

    public void paint(Graphics g) {
        g.drawOval(x, y, width, height);
    }
}
```

Класс `Drawing` содержит несколько полей типа `int` и метод `paint()`. Я всегда стараюсь делать классы как можно меньшими, что и вам рекомендую. Ниже приведено несколько замечаний о классе `Drawing`.

- ✓ **В верхней части кода приведено объявление `package`.** Оно означает, что класс `Drawing` находится в пакете `com.burdbrain.drawings`. Имя пакета взято не с потолка. Соглашения Java требуют, чтобы имена пакетов начинались с инвертированных доменных имен. Я инвертировал имя своего сайта `burdbrain.com` и добавил описательное имя через точку. Имена пакетов должны быть глобально уникальными, потому что заранее невозможно предвидеть, в каких программах будут использоваться классы данного пакета и какие классы будут применены в данном пакете. Что может быть уникальным в этом мире? Ведь даже наши фамилии не уникальны. Но оказывается, есть нечто уникальное — доменные имена. Их уникальность гарантируется адресной системой Интернета.
- ✓ **Класс `Drawing` объявлен с модификатором `public`.** Это означает, что класс объявлен как открытый. Открытые классы чувствительны к

вторжению извне, поэтому добавлять ключевое слово `public` в заголовок определения класса без веской причины не рекомендуется. В листинге 13.2 я был вынужден объявить класс открытым, иначе классы, расположенные не в пакете `com.burdbrain.drawings`, не смогут обращаться к средствам класса. В частности, недопустимой будет следующая инструкция листинга 13.2.



```
ArtFrame artFrame = new ArtFrame(new Drawing());
```

Открытые классы, объявленные с модификатором `public`, рассматриваются далее.

- ✓ **В листинге 13.4 определен метод `paint()`.** В данном случае я использую стандартный трюк для отображения рисунков на экране. Параметр `g` в листинге 13.2 называется *графическим буфером*. Чтобы картинки появились на экране, их нужно нарисовать в графическом буфере, который будет выведен фреймом на экран.

  Рассмотрим этот процесс немного подробнее. В листинге 13.2 метод `paint()` принимает параметр `g` типа `Graphics`. Экземпляр класса `Graphics` является графическим буфером, поэтому все, что вы в него поместите, появится на экране. В классе `Graphics` определено много методов рисования, например метод `drawOval()`, который рисует овал. Для его прорисовки нужно задать четыре параметра: `x` и `y` — координаты левого верхнего угла овала, `height` — высота овала и `width` — ширина овала.

  Метод `paint()` класса `JFrame` автоматически вызывается при каждой перерисовке фрейма, которая происходит при любой операции с фреймом — перемещении, изменении размера, появлении на экране и т.п. Это необходимо для того, чтобы изображение восстанавливалось после каждого изменения фрейма.

Структура папок

Код листинга 13.2 находится в пакете `com.burdbrain.drawings`. Помещая класс в пакет, необходимо создать структуру папок, отображающую имена пакетов. Имена вложенных папок в имени пакета разделены точками.

Например, чтобы создать пакет `com.burdbrain.drawings`, нужно создать три папки: `com`, вложенную в нее `burdbrain` и, опять же, вложенную в `burdbrain` папку `drawings`. Общая структура папок показана на рис. 13.5.



Если компилятор не найдет нужный код в соответствующей папке, он сгенерирует сообщение об ошибке `NoClassDefFoundError` (ошибка, не найдено определение класса). Можете мне поверить, это очень неприятное сообщение. В нем нет ни малейшего намека на то, где его искать или где компилятор надеется его найти. Сохраняйте спокойствие и внимательно все проверьте. Возможно, в имя класса вкралась опечатка. Просмотрите все свои классы и подумайте, есть ли в них что-либо похожее на требуемое имя. Главное — систематичность. Если запаникуете, то будете безрезультатно тыкаться по разным углам в течение многих часов.

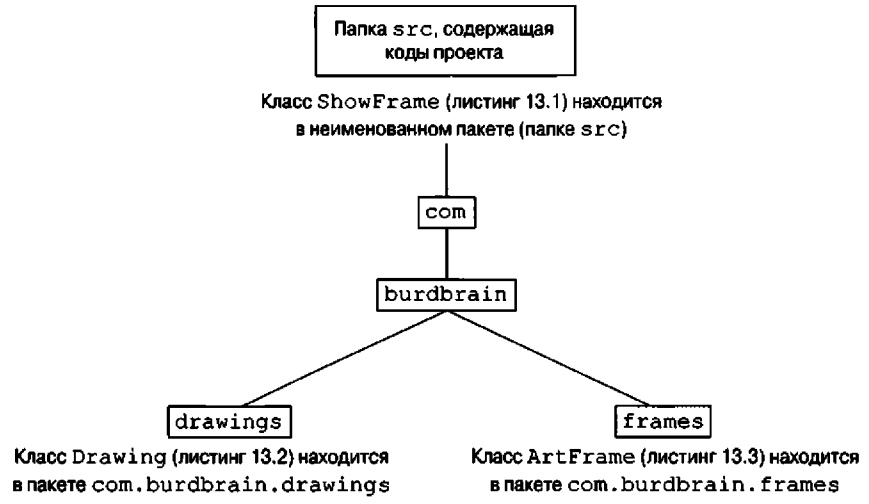


Рис. 13.5. Файлы и папки проекта

Поиск файла в правильном месте

Вы компилируете программу, представленную в листинге 13.1. Компилятор просматривает код и отмечает отсутствующие части программы. В первую очередь он видит, что нужно найти класс `ArtFrame`. Когда он найдет его, то увидит, что нужно найти класс `Drawing`. В листинге 13.1 есть определение только класса `ShowFrame`, а определений `ArtFrame` и `Drawing` в нем нет. Откуда компилятор получает информацию о местоположении классов `ArtFrame` и `Drawing`?

Должен ли компилятор просмотреть весь ваш жесткий диск в поисках этих файлов? Диск может иметь объем сотни гигабайтов или даже терабайтов. Нужно ли искать файлы на сетевых дисках? Потенциально пространство поиска может быть неограниченным.

Для решения этой проблемы в виртуальной машине Java определены маршруты CLASSPATH, т.е. списки мест, в которых компилятор и виртуальная машина могут искать коды. Существует несколько способов установки маршрутов CLASSPATH. Одни программисты создают новый маршрут при каждом запуске программы на Java. Другие создают системную переменную `CLASSPATH` (в Windows и Linux она создается так же, как системная переменная `PATH`). В любом случае виртуальной машине Java необходим список мест, в которых она может искать код. Виртуальная машина не будет просматривать весь жесткий диск и тем более сетевые диски. Поэтому, если она не найдет класс `ArtFrame` или `Drawing` по маршруту CLASSPATH или в папке проекта, она сгенерирует ошибку `NoClassDefFoundError`.

Создание фрейма

В первых трех листингах этой главы представлен пример, состоящий из многих частей. В данном разделе рассматривается последняя, третья, часть примера. Она не очень важна для демонстрации модификаторов доступа, которые являются главной темой данной главы. Поэтому, если вас интересуют только модификаторы доступа,

можете пропустить данный раздел и листинг 13.3. Однако, с другой стороны, вам будет полезно ознакомиться с классами пакета Swing, предназначенными для создания графических пользовательских интерфейсов.

Листинг 13.3. Класс ArtFrame

```
package com.burdbrain.frames;
import java.awt.Graphics;
import javax.swing.JFrame;
import com.burdbrain.drawings.Drawing;

public class ArtFrame extends JFrame {
    private static final long serialVersionUID = 1L;

    Drawing drawing;

    public ArtFrame(Drawing drawing) {
        this.drawing = drawing;
        setTitle("Произведение абстрактного искусства");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    public void paint(Graphics g) {
        drawing.paint(g);
    }
}
```

В листинге 13.3 представлены все инструменты, необходимые для помещения рисунка в фрейм Java. В коде листинга 13.3 используется ряд имен, определенных в Java API. Большинство из них рассмотрено в главах 9 и 10.

Единственное новое средство — метод `paint()`. В листинге 13.3 он всего лишь вызывает еще один метод `paint()`, принадлежащий объекту типа `Drawing`. Объект типа `ArtFrame` создает на экране компьютера *фрейм* — плавающее окно со строкой заголовка и областью рисования. Объект типа `Drawing`, передаваемый фрейму через список параметров конструктора, определяет, что будет нарисовано во фрейме.

Проследив за методом `paint()` класса `ArtFrame`, можно заметить нечто странное: он ниоткуда не вызывается. На самом деле он все же вызывается, но не кодом программы, а машиной Java. Это так называемый *метод обратного вызова*. Он вызывается автоматически каждый раз, когда нужно перерисовать фрейм на экране. Необходимость перерисовки может быть вызвана многими причинами. Например, если фрейм был полностью или частично закрыт каким-либо изображением, а после изображение было удалено и фрейм опять появился на экране, его нужно перерисовать. Перерисовка выполняется также при изменении размеров или перемещении фрейма.

Метод `paint()` объекта типа `ArtFrame` всего лишь вызывает метод `paint()` объекта типа `Drawing`, который рисует во фрейме окружность.

Как изменить программу, не изменяя классы

Предположим, ваш главный поставщик программного обеспечения Burd Brain Consulting продал вам два файла — `Drawing.class` и `ArtFrame.class`. Как пользователь вы не можете видеть коды файлов `Drawing.java` и `ArtFrame.java`. Поэтому

вы не можете изменить классы Drawing и ArtFrame. Предположим также, что вам нужно немного вытянуть окружность (см. рис. 13.4) по горизонтали, чтобы она стала эллипсом. Как это сделать, не изменяя классы, создающие фрейм? На первый взгляд, это невыполнимая задача. Однако на самом деле она легко решается. Для этого нужно создать класс DrawingWide (листинг 13.4), производный от класса Drawing.

Листинг 13.4. Класс DrawingWide

```
import java.awt.Graphics;
import com.burdbrain.drawings.Drawing;

public class DrawingWide extends Drawing {
    int width = 100, height = 30;

    public void paint(Graphics g) {
        g.drawOval(x, y, width, height);
    }
}
```

Чтобы применить код листинга 13.4, необходимо изменить одну строку кода в листинге 13.1. Измененное слово отмечено полужирным шрифтом.

```
ArtFrame artFrame = new ArtFrame(new DrawingWide());
```

Можно также удалить из листинга 13.1 объявление импорта com.burdbrain.drawings.Drawing, потому что в коде этого листинга больше нет ссылки на класс Drawing.

В листинге 13.4 определен класс, производный от класса Drawing. В нем переопределены поля width и height, а также метод paint(). Новый фрейм показан на рис. 13.6.

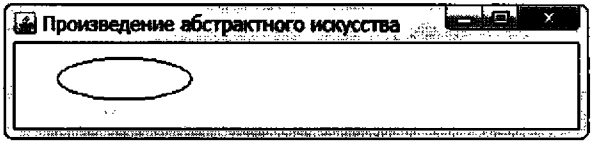


Рис. 13.6. Результат работы класса DrawingWide

Код листинга 13.4 не начинается с объявления пакета package. Это означает, что данный файл находится в неименованном пакете, который на левой панели Eclipse называется default package (пакет, установленный по умолчанию). На жестком диске файл с исходным кодом листинга 13.4 находится в папке src — корневой папке исходных кодов проекта. В ней находятся все файлы пакета default package. Таким образом, программа состоит из трех пакетов.

- ✓ **com.burdbrain.drawings**. В этом пакете находится класс Drawing (листинг 13.2).
- ✓ **com.burdbrain.frames**. В этом пакете находится класс ArtFrame (листинг 13.4).

- ✓ **`default package`**. Неименованный пакет. В него автоматически попадают все классы, определение которых не начинается с объявления пакета. Использовать неименованные пакеты в реальных программах не рекомендуется, потому что это приведет к многочисленным конфликтам имен. Всегда присваивайте пакету какое-либо имя. Сделать это несложно. Однако в данной книге почти во всех примерах используются неименованные пакеты, чтобы не усложнять примеры.

В данный момент проект содержит два класса, рисующие нечто на поверхности фрейма, — старый класс `Drawing` и новый класс `DrawingWide`. Они находятся в разных пакетах. И это неудивительно, поскольку класс `Drawing` разработан компанией Burd Brain Consulting и находится в пакете, имя которого начинается с префикса `com.burdbrain`. А вот класс `DrawingWide` разработан вами, поэтому он не должен находиться в пакете, начинающемся с `com.burdbrain`.

Лучше всего было бы поместить его в отдельный пакет, например в `com.myhomedomain.drawings`, но, чтобы не усложнять пример, оставим его там, куда Eclipse поместила его по умолчанию, — в пакете `default package`. В любом случае производный класс `DrawingWide` компилируется и выполняется, как запланировано.

Доступ, установленный по умолчанию

Задумывались ли вы когда-нибудь над тем, как сложилась бы ваша жизнь, если бы вы не встретили некоторого человека, если бы вам не попала в руки некоторая книга и если бы вы пошли по другой дороге? В данном разделе мы прокрутим часовую стрелку немного назад и посмотрим, что случилось бы, если бы мы пропустили в листинге 13.2 одно слово.

Работа одновременно с разными версиями программ иногда может вызвать головокружение, поэтому начнем обсуждение с того, что у нас есть на данный момент. В первую очередь, у нас есть класс `Drawing`. В нем поля не объявлены как открытые, следовательно, при обращении к ним применяются правила доступа, установленные по умолчанию. Класс `Drawing` находится в пакете `com.burdbrain.drawings`. Удалим одно слово — модификатор `public` в объявлении полей `x`, `y`, `width` и `height` (листинг 13.5). К сожалению, мы не можем отметить полужирным шрифтом то, что удалили. Поэтому скажем словами: удалено слово `public`, находившееся перед спецификатором `int`.

Листинг 13.5. Поля с доступом, установленным по умолчанию

```
package com.burdbrain.drawings;
import java.awt.Graphics;

public class Drawing {
    int x = 40, y = 40, width = 40, height = 40;

    public void paint(Graphics g) {
        g.drawOval(x, y, width, height);
    }
}
```

Кроме того, у нас есть производный класс `DrawingWide` (чтобы вы не искали его код, еще раз приведем его в листинге 13.6). Данный класс находится в неименованном пакете.

Листинг 13.6. Неудачная попытка создать производный класс

```
import java.awt.Graphics;
import com.burdbrain.drawings.Drawing;

public class DrawingWide extends Drawing {
    int width = 100, height = 30;

    public void paint(Graphics g) {
        g.drawOval(x, y, width, height);
    }
}
```

Из-за того что мы пропустили одно слово в листинге 13.5, программа развалилась При попытке скомпилировать листинг 13.6 мы получаем следующие два сообщения об ошибке.

```
x is not public in com.burdbrain.drawings.Drawing;
cannot be accessed from outside package
(поле x в классе com.burdbrain.drawings.Drawing
не открытое, поэтому за пределами пакета
доступа к нему нет)
y is not public in com.burdbrain.drawings.Drawing;
cannot be accessed from outside package
(поле y в классе com.burdbrain.drawings.Drawing
не открытое, поэтому за пределами пакета
доступа к нему нет)
```

Код не компилируется, потому что поле с доступом, установленным по умолча нию, за пределами своего пакета недоступно даже для производного класса. То ж самое справедливо и для метода с доступом, установленным по умолчанию.



Поля и методы класса называются *членами класса*. Все правила доступ применяются к членам класса единообразно, независимо от того, поля эт или методы.



Рассматриваемые в данном разделе правила доступа не применяются к л кальным переменным, т.е. к переменным, объявленным внутри методо К локальной переменной можно обращаться только внутри метода, в к тором она объявлена.



Локальные переменные рассматриваются в главе 10.

В Java члены класса, имеющие доступ, установленный по умолчанию, доступны тол ко в текущем пакете. Иными словами, если член класса объявлен без модификатора д ступа `public`, `private` или `protected`, к нему можно обращаться только внутри пак та, в котором определен данный класс. Это правило иллюстрируется на рис. 13.7 и 13

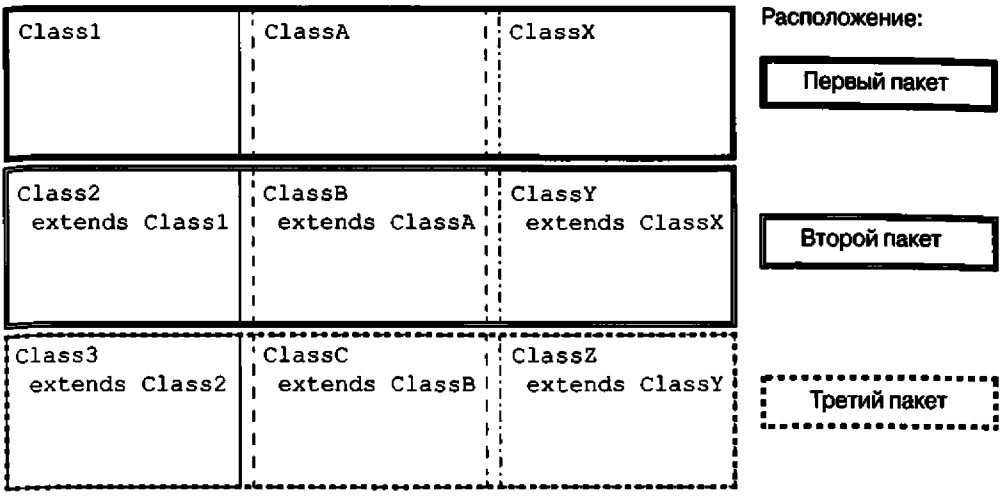


Рис. 13.7. Классы определены в разных пакетах

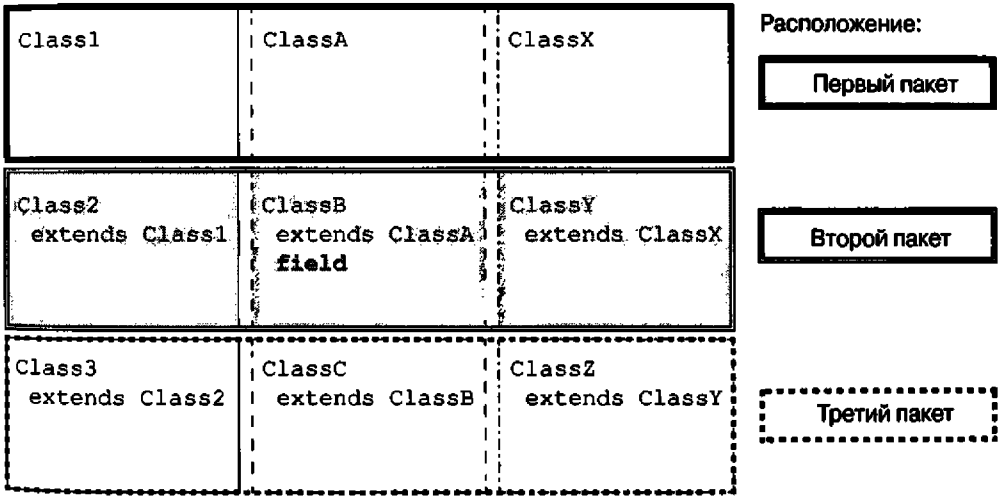


Рис. 13.8. Затенены классы, в которых доступно поле `field`, объявленное без модификаторов доступа



Точки в именах пакетов могут ввести в заблуждение. Они как бы намекают на то, что пакеты вложены друг в друга. Но на самом деле это не совсем так. Вложены только папки в структуре файлов. Не забывайте, что при импорте родительского пакета вложенные пакеты не импортируются. Например, при написании программы, в которой предусмотрено реагирование на щелчок мышью на кнопке, нужны классы, находящиеся в двух разных пакетах. Поэтому нужно написать два объявления импорта: `import java.awt.*` и `import java.awt.event.*`. Это необходимо потому, что при импорте пакета `java.awt` пакет `java.awt.event` не импортируется.

Как проникнуть в пакет

Я люблю получать электронные письма. В худшем случае, если письмо мне не нужно, я могу выбросить его в корзину, щелкнув мышью один раз. В лучшем случае я получу нечто полезное — информацию, фотографии или что-то такое, что предназначено лично для меня.

Сегодня мне повезло. Я получил от компании Burd BrainConsulting класс `DrawingWideBB`, производный от класса `Drawing`. Он похож на класс, приведенный в листинге 13.6, с единственным отличием: класс `DrawingWideBB` (листинг 13.7) находится в пакете `com.burdbrain.drawings`. Чтобы запустить его, нужно изменить в листинге 13.1 следующую строку кода:

```
ArtFrame artFrame = new ArtFrame(new DrawingWideBB());
```

Листинг 13.7. Класс `DrawingWideBB`, производный от `Drawing`

```
package com.burdbrain.drawings;
import java.awt.Graphics;

public class DrawingWideBB extends Drawing {
    int width = 100, height = 30;

    public void paint(Graphics g) {
        g.drawOval(x, y, width, height);
    }
}
```

Если запустить на выполнение листинг 13.7 вместе с листингом 13.5, все заработает прекрасно. Это объясняется тем, что классы `Drawing` и `DrawingWideBB` находятся в одном пакете. Посмотрите еще раз на рис. 13.8 и обратите внимание на то, что затененная область занимает весь пакет. Код класса `DrawingWideBB` имеет полное право пользоваться полями `x` и `y`, которые определены в классе `Drawing` с доступом, установленным по умолчанию, потому что классы `Drawing` и `DrawingWideBB` расположены в одном и том же пакете.



Чтобы класс `DrawingWideBB` (см. листинг 13.7) можно было использовать в программе, нужно сделать в листинге 13.1 два изменения. Во-первых, нужно добавить объявление импорта `import com.burdbrain.drawings.DrawingWideBB`, и, во-вторых, нужно изменить вызов конструктора, написав `new ArtFrame(new DrawingWideBB())`.

Защищенный доступ

Когда я впервые познакомился с Java, я думал, что ключевое слово `protected` (защищенный) означает усиленный, безопасный или что-либо подобное. Раз некоторая сущность является защищенной, значит, к ней тяжело подобраться и ее тяжело повредить. Но оказалось, что это не совсем так. В Java защищенный член класса менее скрытый и менее безопасный, чем имеющий доступ, применяемый по умолчанию. Поэтому пусть вас не обманет слово "защищенный".

Чтобы понять правила доступа к защищенному члену класса, представьте себе поле с доступом, применяемым по умолчанию (т.е. поле, объявленное без ключевого слова `public`, `private` или `protected`). На него можно сослаться только внутри пакета, в котором оно объявлено. Затем представьте себе, что к нему добавлен модификатор `protected`. Что изменилось? Только то, что теперь к нему можно обращаться еще и в производных классах, расположенных в других пакетах, включая все производные от производного класса. Данная концепция иллюстрируется в листингах 13.8 и 13.9.

Листинг 13.8. Защищенные поля

```
package com.burdbrain.drawings;
import java.awt.Graphics;

public class Drawing {
    protected int x = 40, y = 40, width = 40, height = 40;

    public void paint(Graphics g) {
        g.drawOval(x, y, width, height);
    }
}
```

Листинг 13.9. Использование защищенных полей в производном классе

```
import java.awt.Graphics;
import com.burdbrain.drawings.Drawing;

public class DrawingWide extends Drawing {
    int width = 100, height = 30;

    public void paint(Graphics g) {
        g.drawOval(x, y, width, height);
    }
}
```

В листинге 13.8 определен класс `Drawing`, а в листинге 13.9 — класс `DrawingWide`, производный от `Drawing`.

В классе `Drawing` поля `x`, `y`, `width` и `height` объявлены как защищенные. В классе `DrawingWide` объявлены собственные поля `width` и `height`. Они переопределяют одноименные поля, объявленные в базовом классе `Drawing`. Однако переменные `x` и `y` в классе `DrawingWide` ссылаются на поля `x` и `y` базового класса `Drawing`.

Это допустимо даже несмотря на то, что классы `DrawingWide` и `Drawing` определены в разных пакетах. Класс `Drawing` определен в пакете `com.burdbrain.drawings`, а класс `DrawingWide` — в неименованном пакете. Поля `x` и `y` защищенные, поэтому они доступны в производных классах.

Сравните рис. 13.8 и 13.9. Обратите внимание на то, что на рис. 13.9 дополнительно затенен класс `ClassC`. Поскольку он является производным от класса `ClassB`, в нем можно использовать защищенные поля базового класса, даже если он определен в другом пакете.

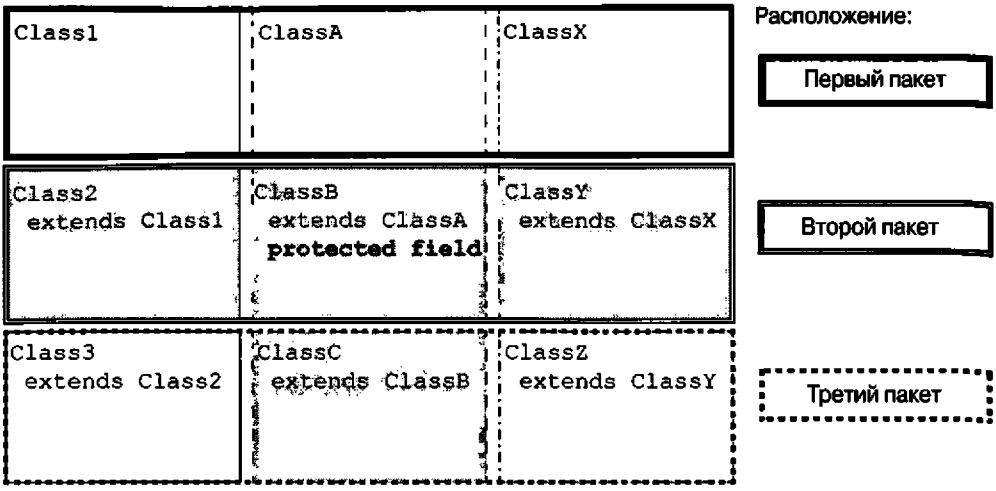


Рис. 13.9. Защищенное поле `field` доступно в производном классе



Вы работали когда-нибудь в команде программистов? Если работали, то знаете, что в каждой команде используется собственный набор имен пакетов. Как же другая команда может использовать классы, созданные вашей командой и находящиеся в ваших пакетах? Удобнее всего это делать с помощью защищенных членов классов. Используйте защищенные члены, если другая команда должна создавать классы, производные от ваших классов.

Включение не производного класса в тот же пакет

Что-то часто эти ребята из Burd Brain Consulting присылают нам одну часть кода за другой. На этот раз они прислали альтернативный вариант класса `ShowFrame` (предыдущий вариант см. в листинге 13.1). Новый вариант — класс `ShowFrameWideBB` — отображает более широкий эллипс (впечатляющее достижение!), но делает это без создания класса, производного от `Drawing`. Вместо этого `ShowFrameWideBB` создает экземпляр класса `Drawing`, а затем изменяет значения полей `width` и `height` данного экземпляра (листинг 13.10).

Листинг 13.10. Рисование более широкого эллипса

```
package com.burdbrain.drawings;
import com.burdbrain.frames.ArtFrame;

class ShowFrameWideBB {

    public static void main(String args[]) {
        Drawing drawing = new Drawing();
        drawing.width = 100;
        drawing.height = 30;

        ArtFrame artFrame = new ArtFrame(drawing);
```

```
        artFrame.setSize(200, 100);
        artFrame.setVisible(true);
    }
}
```

Класс `ShowFrameWideBB`, представленный в листинге 13.10, находится в том же пакете, что и класс `Drawing` (а именно — в пакете `com.burdbrain.drawings`). Но `ShowFrameWideBB` не является производным от класса `Drawing`.

Теперь представьте себе, что вы компилируете `ShowFrameWideBB` совместно с `Drawing` (см. листинг 13.8), т.е. совместно с классом, имеющим защищенные поля. Что произойдет? Все будет хорошо, потому что защищенные члены доступны в двух местах (как ни странно, логически не связанных друг с другом). Посмотрите еще раз на рис. 13.9. Защищенный член доступен, во-первых, в производном классе за пределами пакета и, во-вторых, в любом классе (производном или не производном) внутри пакета.



В листинге 13.10 есть метод `main()`, принадлежащий классу, который находится в пакете `com.burdbrain.drawings`. В большинстве интегрированных сред разработки (включая Eclipse) запуск метода `main()`, расположенного в неименованном пакете, не порождает проблем. Но при запуске программы из командной строки нужно ввести полностью квалифицированное имя класса. Например, чтобы запустить из командной строки программу, приведенную в листинге 13.10, нужно ввести строку `java com.burdbrain.drawings.ShowFrameWideBB`.



На самом деле правила доступа к защищенным членам класса немного сложнее, чем представлено в данном разделе. В спецификации Java можно найти довольно туманное упоминание о кодах, управляющих реализациями объектов. Пока что можете не беспокоиться о таких тонкостях. Возможно, вы достигнете высот профессионализма, так и не встретившись с этим. Даже если вам встретится сообщение об ошибке `variable has protected access` (переменная обладает защищенным доступом), проще будет не изучать спецификацию Java, а устранить ошибку экспериментально.



Спецификация Java рассматривается в главе 3.

Модификаторы доступа к классам

Возможно, почитав о модификаторах доступа к членам классов, вы почувствовали легкое головокружение, несмотря даже на то, что в предыдущих разделах я упомянул не обо всех тонкостях данной темы. Теперь же спешу вас обрадовать: с головокружением наконец-то покончено. По сравнению с повестью о полях и методах, рассказ о доступе к классам значительно короче и проще.

Класс может быть объявлен либо как открытый, либо никак. Открытый класс объявляется следующим образом:

```
public class Drawing
```

Можно также объявить класс без ключевого слова `public`.

```
class ShowFrame
```

Такой класс нельзя называть закрытым (по аналогии с закрытыми членами класса, объявленными с модификатором `private`). Как же его назвать? Да никак! Это просто не открытый класс. Можно назвать его обычным классом. Правда, о нем часто пишут "закрытый", но это не означает, что в его объявлении применяется модификатор `private`.

Открытые классы

Если класс открытый, на него можно ссылаться в любом месте программы. Конечно, это правило применимо с учетом ограничений, обусловленных расположением класса (см. ранее о структуре папок проекта). Необходимо также правильно сослаться на пакет, в котором определен класс. Например, в листинге 13.1 можно написать так.

```
import com.burdbrain.drawings.Drawing;
import com.burdbrain.frames.ArtFrame;
...
ArtFrame artFrame = new ArtFrame(new Drawing());
```

Можно обойтись и без объявлений `import` и написать так.

```
com.burdbrain.frames.ArtFrame artFrame =
  new com.burdbrain.frames.ArtFrame
    (new com.burdbrain.drawings.Drawing());
```

В любом случае компилятор должен откуда-то узнать, что классы `ArtFrame` и `Drawing` находятся в именованных пакетах.

Не открытые классы

Если класс не объявлен как открытый, на него можно ссылаться только в пакете, в котором он определен.

Вернемся к листингу 13.2 и удалим ключевое слово `public`. Пусть определение класса начинается словами `class Drawing`, а не `public class Drawing`.

```
package com.burdbrain.drawings;
import java.awt.Graphics;
class Drawing {
  public int x = 40, y = 40, width = 40, height = 40;
  public void paint(Graphics g) {
      g.drawOval(x, y, width, height);
  }
}
```

Теперь попробуем скомпилировать код листинга 13.7. Все проходит прекрасно, потому что в листинге 13.7 есть следующие строки.

```
package com.burdbrain.drawings;
public class DrawingWideBB extends Drawing
```

Обе части кода находятся в одном и том же пакете `com.burdbrain.drawings`, поэтому при обращении к не открытому классу `Drawing` в классе `DrawingWideBB` проблем не возникает.

Однако теперь попробуем скомпилировать листинг 13.3, который начинается следующей строкой.

```
package com.burdbrain.frames;
```

Она означает, что код находится в пакете `com.burdbrain.frames`, а не `com.burdbrain.drawings`. Поэтому, когда компьютер доходит до строки `Drawing drawing;`, принадлежащей листингу 13.3, он отображает следующее сообщение об ошибке.

```
com.burdbrain.drawings.Drawing is not public
in com.burdbrain.drawings;
cannot be accessed from outside package
(Класс com.burdbrain.drawings.Drawing, определенный
 в com.burdbrain.drawings.Drawing, не открытый;
 доступ к нему за пределами пакета запрещен)
```



Чаще всего окружающие нас вещи не такие простые, какими кажутся. Правило, приведенное в данном разделе, применимо почти ко всем классам в данной книге. Однако в Java есть так называемые *внутренние классы*, которые подчиняются немного другим правилам. К счастью, начинающие программисты очень редко встречаются с внутренними классами. В данной книге внутренний класс можно увидеть только в одном месте — в одном из примеров главы 14. Иногда типы `enum` считаются "замаскированными" внутренними классами. Следовательно, пока что (и еще очень долго) вы можете смело пользоваться правилами, приведенными в данном разделе.

Глава 14

Реагирование на события клавиатуры и мыши

В этой главе...

- Реагирование на щелчок мышью
- Реагирование на другие события
- Внутренние классы

В конце 1980-х годов я купил свою первую мышь, заплатив за нее 100 долларов. Предварительно мы с женой обсудили эту покупку. В то время мышь стоила дорого и была не очень нужна мне, потому что мой компьютер работал в гибридной текстово-оконной среде. Многие ветераны компьютерных технологий до сих пор помнят знаменитый Norton Commander, в котором впервые можно было работать с помощью такого необычного устройства, как мышь.

Однако сейчас на дворе XXI столетие. Последние десять своих мышек я получил бесплатно. Они сами каким-то образом прибегают ко мне. Некоторые подарены друзьями. В последнее время появилась мода дарить экзотические мышки (что еще можно подарить человеку, работающему на компьютере?). Одну мышку мне вручили в магазине во время шоу, привлекающего покупателей. Сейчас, когда я пишу эту книгу, я пользуюсь мышкой "для чайников". Ее мне подарило издательство Wiley, для которого я пишу книгу. Мышка окрашена в желтые и черные тона, как книги серии "для чайников", и наполнена водой. Пузырек воздуха двигается под стеклом в разные стороны в зависимости от движения мышки. Это очень романтичное и завораживающее зрелище, но часто оно отвлекает от работы.

Реагирование на щелчок мышью

В предыдущих главах мы создавали окна, в которых ничего нельзя было делать. Эти окна только отображали некоторую информацию, но ничего не принимали от пользователя. В данной главе наконец-то появится окно с кнопкой, на которой можно будет щелкнуть, в результате чего в программе и на экране что-то произойдет. Код фрейма, содержащего кнопку, приведен в листинге 14.1, а код главного класса, отображающего фрейм, — в листинге 14.2.

Листинг 14.1. Игра в отгадывание случайного числа

```
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Random;
```

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;

class GameFrame extends JFrame implements ActionListener {
   private static final long serialVersionUID = 1L;

   int randomNumber = new Random().nextInt(10) + 1;
   int numGuesses = 0;

   JTextField textField = new JTextField(15);
   JButton button = new JButton("Сделать попытку");
   JLabel label = new JLabel(numGuesses + " попыток");

   public GameFrame() {
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setLayout(new FlowLayout());
      add(textField);
      add(button);
      add(label);
      button.addActionListener(this);
      pack();
      setVisible(true);
   }

   @Override
   public void actionPerformed(ActionEvent e) {
      String textFieldText = textField.getText();

      if (Integer.parseInt(textFieldText)==randomNumber) {
         button.setEnabled(false);
         textField.setText(textField.getText() + "Ура! Угадал!");
         textField.setEnabled(false);
      } else {
         textField.setText("");
         textField.requestFocus();
      }

      numGuesses++;
      String guessWord = (numGuesses == 1) ? " попытка" : " попыток";
      label.setText(numGuesses + guessWord);
   }
}
```

Листинг 14.2. Запуск игры в отгадывание числа

```
class ShowGameFrame {

   public static void main(String args[]) {
       new GameFrame();
   }
}
```

Копии экрана, полученные в сеансе игры, показаны на рис. 14.1 и 14.2. Пользователь играет с программой в отгадывание числа. Программа генерирует секретное целое число (от 1 до 10) и отображает на экране текстовое поле и кнопку. Пользователь пытается угадать сгенерированное число. Для этого он вводит некоторое число в текстовое поле и щелкает на кнопке. После этого, в зависимости от того, угадал ли пользователь, происходит одно из двух.

- ✓ **Если введенное число не равно сгенерированному,** компьютер отображает количество попыток и очищает текстовое поле. Программа готова к очередной попытке.
- ✓ **Если введенное число равно сгенерированному,** программа отображает в текстовом поле сообщение об удачной попытке и затеняет кнопку и текстовое поле. Оба эти элемента интерфейса больше не нужны, поскольку игра закончена. Чтобы сыграть еще раз, нужно заново запустить программу.

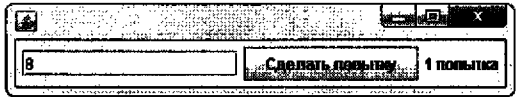


Рис. 14.1. Пользователь не угадал число



Рис. 14.2. Пользователь угадал число

В листинге 14.1 код, создающий фрейм, кнопку и текстовое поле, довольно простой. Мы уже делали это в главах 9 и 10. Для вас новинка — класс текстового поля `JTextField`, но его создание ничем не отличается от создания кнопки или надписи. Как и другие элементы интерфейса, класс `JTextField` определен в пакете `javax.swing`. При создании экземпляра текстового поля можно задать его ширину. В листинге 14.1 значение `15` означает, что будет создано текстовое поле с шириной, равной ширине самой большой буквы, умноженной на 15.



Чтобы надпись отображала грамматически правильное слово, зависящее от количества попыток, в листинге 14.1 используется условный оператор `?` (см. главу 11). Если попытка одна, отображается слово `попытка`, а если не одна, то слово `попыток`.

События и обработка событий

В листинге 14.1 главная новинка для вас — обработка щелчка на кнопке. Все, что пользователь делает при работе с графическим интерфейсом (нажатие клавиши, перемещение указателя мыши, щелчок на кнопке, ввод текста и т.п.), называется *событием*. Код или метод, выполняющийся в результате возникновения некоторого события, называется *обработчиком события*.

В листинге 14.1 обработка щелчка на кнопке выполняется в трех местах кода.

- ✓ Объявление `implements ActionListener` в заголовке класса `GameFrame` означает, что класс реализует интерфейс `ActionListener`, т.е. обеспечивает прослушивание события. Термины *прослушивание* и *прием* события — синонимы. Иногда обработчик называют приемником события, но это неверно. В данном примере приемником служит фрейм.
- ✓ Конструктор класса `GameFrame` добавляет элемент `this` (т.е. фрейм) в список компонентов, прослушивающих событие щелчка на кнопке.
- ✓ В классе `GameFrame` определен метод `actionPerformed()`, являющийся обработчиком события щелчка и выполняющийся при щелчке на кнопке.

Работая совместно, эти три части кода обеспечивают реагирование программы на щелчок на кнопке. Чтобы понять, как это работает, вам нужно ознакомиться с понятием интерфейса, потому что приемником события служит класс, реализующий интерфейс данного события.

Интерфейсы Java

В Java производный класс никогда не наследует более одного базового класса. Следовательно, приведенное ниже объявление неправильное.

```
class DontDoThis extends FirstClass, SecondClass
```

Производный класс может иметь только один базовый класс. Это вполне приемлемо, когда новый класс должен иметь свойства чего-то одного, например должен быть фреймом. Но как быть, если новый класс должен иметь свойства нескольких классов, например быть одновременно фреймом и приемником щелчка на кнопке? Как ему унаследовать свойства одновременно двух классов?

Это можно сделать. В Java для этого используются *интерфейсы*. Интерфейс похож на класс, но отличается от него. (Очень информативное заявление. Чем корова отличается от планеты? Тем, что корова дает молоко, а планета вращается вокруг Солнца.) В любом случае, когда вы слышите слово "интерфейс", представляйте его себе как базовый класс. При этом помните о следующем.

- ✓ **Класс может наследовать только один базовый класс, но может реализовать более одного интерфейса.**

  Например, чтобы фрейм `GameFrame` прослушивал (принимал) не только щелчки, но и нажатия клавиш, нужно объявить его так:

  ```
  class GameFrame extends JFrame implements ActionListener,
                                            ItemListener
  ```

- ✓ **У методов интерфейса нет собственных тел.**

  Ниже приведен фрагмент определения интерфейса `ActionListener`, скопированный из API.

  ```
  package java.awt.event;
  import java.util.EventListener;
  ```

```
public interface ActionListener extends EventListener {
   public void actionPerformed(ActionEvent e);
}
```

Как видите, у метода `actionPerformed()` нет тела. Вместо фигурных скобок и выполняемых инструкций в данном определении есть только точка с запятой.



Метод без тела называется *абстрактным методом*.

- ✓ **При реализации интерфейса нужно определить тела для всех его методов.**

Вот почему в листинге 14.1 определено тело метода `actionPerformed()`. Сначала мы объявили, что собираемся реализовать интерфейс `ActionListener`, а затем действительно реализовали его. В данном случае реализация интерфейса состоит в определении тела метода `actionPerformed()`.

Когда вы объявляете, что собираетесь реализовать интерфейс, компилятор Java воспринимает ваше заявление вполне серьезно. Если далее в коде вы не сделаете этого, т.е. не определите тела всех методов интерфейса, компилятор сообщит об ошибке и не пропустит программу на выполнение.



Чтобы найти методы, которые нужно определить для реализации некоторого интерфейса, скомпилируйте код. Конечно, он не будет скомпилирован, однако компилятор вернет вам список методов, для которых нужно определить тела.



В главе 8 рассматривается аннотация `@Override` (переопределить). Обычно она используется для сообщения компилятору о том, что данный метод замещает метод, объявленный в базовом классе. Начиная с Java 6 аннотацию `@Override` можно использовать для сообщения о реализации метода интерфейса, что я и сделал в листинге 14.1.

Потоки

Программа на Java может быть *многопотоковой*. Это означает, что она может одновременно делать много разных вещей. Конечно, компьютер выполняет не только написанный вами код программы, но и другие коды, которые вы чаще всего даже не видите и никогда не увидите. Операционная система автоматически организует одновременное выполнение многих программ. Но чтобы внутри одной программы одновременно решалось несколько задач, нужно запустить их в разных *потоках*. Например, когда компьютер выполняет метод `main()`, он может проверить, нет ли еще потоков, которые можно запустить на выполнение, чтобы не терять времени, пока выполняется метод `main()`.

Виртуальная машина Java автоматически создает поток обработки событий. Когда выполняется код программы, одновременно (в фоновом режиме) выполняется код прослушивания событий, который принимает щелчок мышью и запускает обработчик события. Этот процесс иллюстрируется на рис. 14.3.

Поток кода	Поток обработки событий

`setLayout(new FlowLayout());`
`add(textField);`
`add(button);`
`add(label);`
 	
`button.addActionListener(this);`
`pack();`
`setVisible(true);`
	Щелкнул ли пользователь мышью?
	...
	Щелкнул ли пользователь мышью?
	...
	Щелкнул ли пользователь мышью?
	Ага! Наконец, щелкнул! Значит, запускаем метод `actionPerformed()`.

Рис. 14.3. Потоки Java

Поток обработки событий видит, что произошел щелчок, и думает: "Что я должен сделать в связи с этим?" Конечно, в коде он быстро находит ответ: "Вызвать метод `actionPerformed()`". Поток обработки событий выполняет код, похожий на приведенный ниже.

```
if(щелчок_на_кнопке) {
   object1.actionPerformed(информация_о_щелчке)
   object2.actionPerformed(информация_о_щелчке)
   object3.actionPerformed(информация_о_щелчке)
}
```

Поток обработки событий должен найти ответ на еще один вопрос: "Где находится нужный метод `actionPerformed()`?" Кроме того, поток должен знать, на какие события нужно реагировать, а какие должны быть проигнорированы. Этот вопрос решается просто: поток отреагирует только на те события, прослушивание которых включено в код программы. Иными словами, к событию должен быть подключен его приемник с помощью метода `addActionListener()`. В листинге 14.1 подключение приемника выполняется с помощью следующей инструкции:

```
button.addActionListener(this);
```

Эта инструкция приказывает потоку обработки событий добавить метод `actionPerformed()` в список методов, вызываемых при щелчке на кнопке. В данном случае приемником служит экземпляр фрейма, обозначенный ключевым словом `this`. В классе фрейма определен метод `actionPerformed()`, который становится обработчиком события.

Чтобы компьютер вызвал метод `actionPerformed()`, его нужно зарегистрировать в потоке обработки событий. Регистрация выполняется путем вызова метода `addActionListener()`, принадлежащего объекту, на котором ожидается щелчок. В листинге 14.1 щелчок ожидается на кнопке `button`, поэтому метод `addActionListener()` принадлежит объекту кнопки.

Ключевое слово `this`

В Java ключевое слово `this` обозначает текущий экземпляр класса, в котором оно находится. В данном примере (см. листинг 14.1), слово `this` находится в классе `GameFrame`, следовательно, оно обозначает фрейм. Вызов метода

addActionListener(this) находится в конструкторе класса GameFrame, поэтому при создании экземпляра фрейма происходит подключение приемника к событию. В данном случае приемником служит текущий фрейм.

```
button.addActionListener(экземпляр_фрейма);
```

Вызвав метод button.addActionListener(this), мы приказываем потоку обработки событий добавить метод actionPerformed() в список методов, вызываемых при щелчке на кнопке button. Класс GameFrame должен содержать метод actionPerformed(), потому что он объявлен в интерфейсе ActionListener, который реализуется классом GameFrame.

Тело метода actionPerformed()

В методе actionPerformed() используется несколько трюков из арсенала Java API.

- ✓ Каждый экземпляр JTextField (как и JLabel) имеет собственный набор методов доступа, включая getText() и setText(). Вызов getText() возвращает строку символов, отображаемую данным элементом интерфейса. Вызов setText() записывает заданную строку символов в элемент интерфейса. В листинге 14.1 метод getText() извлекает из текстового поля число, а метод setText() записывает в текстовое поле либо пустую строку, либо сообщение о том, что число угадано.
- ✓ У каждого класса (в том числе JTextField и jButton) пакета javax.swing есть метод setEnabled(). Вызов setEnabled(false) отключает элемент интерфейса, в результате чего он становится серым и перестает реагировать на щелчки и нажатия клавиш.
- ✓ У каждого класса, определенного в пакете javax.swing, есть метод requestFocus(). При его вызове элемент интерфейса получает фокус ввода. Это означает, что теперь именно этот элемент будет реагировать на нажатие клавиш. В листинге 14.1 инструкция textField.requestFocus() передает фокус текстовому полю. При каждом щелчке на кнопке фокус переходит к ней, однако данная инструкция каждый раз возвращает фокус текстовому полю.



Метод getSource() объекта события e возвращает имя объекта, сгенерировавшего событие. С его помощью можно проверить, действительно ли button — это элемент интерфейса, на котором щелкнул пользователь. Для этого добавьте в тело метода actionPerformed() инструкцию if(e.getSource()==button)println("Это действительно button"). Если во фрейме две кнопки (например, button1 и button2), этим способом можно выяснить, на какой кнопке произошел щелчок. Для этого напишите if(e.getSource()==button1) или if(e.getSource()==button2).

Идентификатор версии

В главе 9 рассматривается применение аннотации SuppressWarnings для отмены предупреждения компилятора о том, что не задан идентификатор версии — число

serialVersionUID. Оно помогает инфраструктуре Java избежать конфликтов версий при передаче объекта с одного места в другое. Например, вы можете передать состояние объекта JFrame на другой компьютер и отобразить фрейм на экране. Принимающий компьютер может проверить номер версии фрейма. Это поможет избежать нежелательных эффектов.

Аннотация SuppressWarnings приказывает компилятору проигнорировать отсутствие номера версии. В листинге 14.1 применен другой подход: объекту JFrame присвоен номер serialVersionUID. Данный фрейм — наша первая версия класса GameFrame, поэтому я присвоил версии номер 1. Однако, поскольку номер версии имеет тип long, литерал имеет вид 1L.

Нужно ли изменять номер версии при каждом изменении кода? Очевидно, нет. В общем случае рекомендуется изменять его, когда в результате изменения кода новая версия становится несовместимой со старой версией других компонентов приложения. Несовместимость означает, что принимающий компьютер или другие классы программы не могут работать с измененным объектом.

Реагирование на другие события

Когда вы знаете, как приложение реагирует на событие одного типа, запрограммировать обработку событий других типов будет несложно. Листинги 14.3 и 14.4 отображают на экране окно, которое конвертирует валюты США и Великобритании. В этих листингах код реагирует на события разных типов. Результат работы программы показан на рис. 14.4–14.6.

Листинг 14.3. Конвертирование местных валют

```
import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.text.NumberFormat;
import java.util.Locale;

import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;

class MoneyFrame extends JFrame implements KeyListener,
                                    ItemListener, MouseListener {
    private static final long serialVersionUID = 1L;

    JLabel fromCurrencyLabel = new JLabel(" ");
    JTextField textField = new JTextField(5);
    JLabel label = new JLabel("           ");
```

```
JComboBox combo = new JComboBox();

NumberFormat currencyUS =
    NumberFormat.getCurrencyInstance(Locale.US);
NumberFormat currencyUK =
    NumberFormat.getCurrencyInstance(Locale.UK);

public MoneyFrame() {
    setLayout(new FlowLayout());

    add(fromCurrencyLabel);
    add(textField);
    combo.addItem("Доллары в фунты");
    combo.addItem("Фунты в доллары");
    add(label);
    add(combo);

    textField.addKeyListener(this);
    combo.addItemListener(this);
    label.addMouseListener(this);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    setSize(300, 100);
    setVisible(true);
}

void setTextOnLabel() {
    String amountString = "";
    String fromCurrency = "";

    try {
        double amount = Double.parseDouble(textField.getText());

        if(combo.getSelectedItem().equals("Доллары в фунты"))
        {
            amountString = " = " + currencyUK.format(amount *
                                                     0.61214);
            fromCurrency = "$";
        }
        if(combo.getSelectedItem().equals("Фунты в доллары"))
        {
            amountString = " = " + currencyUS.format(amount *
                                                     1.63361);
            fromCurrency = "\u00A3";
        }

    } catch (NumberFormatException e) {
    }

    label.setText(amountString);
    fromCurrencyLabel.setText(fromCurrency);
```

```
    }

    @Override
    public void keyReleased(KeyEvent k) {
        setTextOnLabel();
    }

    @Override
    public void keyPressed(KeyEvent k) {
    }

    @Override
    public void keyTyped(KeyEvent k) {
    }

    @Override
    public void itemStateChanged(ItemEvent i) {
        setTextOnLabel();
    }

    @Override
    public void mouseEntered(MouseEvent m) {
        label.setForeground(Color.red);
    }

    @Override
    public void mouseExited(MouseEvent m) {
        label.setForeground(Color.black);
    }

    @Override
    public void mouseClicked(MouseEvent m) {
    }

    @Override
    public void mousePressed(MouseEvent m) {
    }

    @Override
    public void mouseReleased(MouseEvent m) {
    }
}
```

Листинг 14.4. Отображение фрейма `MoneyFrame`

```
class ShowMoneyFrame {
   public static void main(String args[]) {
      new MoneyFrame();
   }
}
```

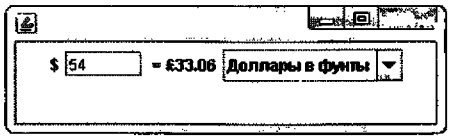


Рис. 14.4. Конвертирование долларов США в британские фунты

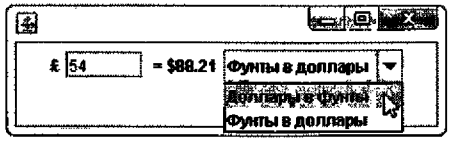


Рис. 14.5. Использование раскрывающегося списка

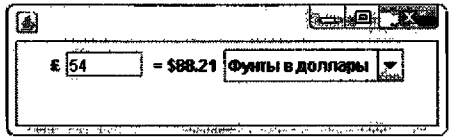


Рис. 14.6. Конвертирование британских фунтов в доллары США

Листинг 14.3 довольно длинный. Это обусловлено тем, что нужно определить все методы, заявленные в реализуемых интерфейсах. Большинство из этих методов пустые, поэтому код класса не такой уж сложный, что видно из приведенного ниже описания.

```
class MoneyFrame extends JFrame implements
        KeyListener, ItemListener, MouseListener {

   Объявление переменных.

   Конструктор класса MoneyFrame.

   Определение метода setTextOnLabel().

   Определение методов, необходимых для
                реализации трех интерфейсов.
}
```

В листинге 14.3 конструктор класса добавляет в окно `MoneyFrame` четыре компонента.

- ✓ **Надпись.** На рис. 14.4 эта надпись отображает символ доллара.
- ✓ **Текстовое поле.** На рис. 14.4 в это поле введено число `54`.
- ✓ **Еще одна надпись.** На рис. 14.4 эта надпись отображает строку `£33.06`.
- ✓ **Раскрывающийся список.** На рис. 14.4 в раскрывающемся списке отображен элемент `Доллары в фунты`. На рис. 14.5 пользователь выбирает нужный элемент, а на рис. 14.6 выбран элемент `Фунты в доллары`.

Класс MoneyFrame реализует три интерфейса: KeyListener (Приемник событий клавиатуры), ItemListener (Приемник событий элемента) и MouseListener (Приемник событий мыши). Поэтому код может прослушивать события трех разных типов. Каждый интерфейс позволяет прослушивать события одного типа, но разных объектов.

- **KeyListener**. В классе, реализующем интерфейс KeyListener, должно быть определено три метода — keyReleased (отпускание клавиши), keyPressed (нажатие клавиши) и keyTyped (ввод клавиши), которые запускаются при возникновении события. Например, когда пользователь отпускает клавишу, поток обработки событий вызывает метод keyReleased().

  В листинге 14.3 метод keyReleased() вызывает метод setTextOnLabel(). Этот метод проверяет, какой элемент сейчас выбран в раскрывающемся списке. Если выбран элемент Доллары в фунты, метод setTextOnLabel конвертирует доллары в фунты (т.е. вычисляет количество фунтов) и выводит на экран соответствующие надписи. Если же выбран элемент Фунты в доллары, метод setTextOnLabel вычисляет количество долларов.



  В методе setTextOnLabel() используется строка \u00A3. Это код символа британского фунта стерлингов, записанный в системе Unicode. Если в окне локализации операционной системы установить в качестве валюты фунт стерлингов, символ фунта будет отображаться автоматически. Класс Locale позволяет локализовать валюту и другие параметры операционной системы, зависящие от языка или страны.

  Данную программу нельзя использовать в реальной жизни, потому что курсы валют в ней жестко закодированы, хотя в реальности они постоянно изменяются.

- **ItemListener**. Класс, реализующий интерфейс ItemListener, должен определить метод itemStateChanged (изменение состояния элемента). Поток обработки событий вызывает метод itemStateChanged() при выборе элемента раскрывающегося списка пользователем.

  В теле метода itemStateChanged() выполняется вызов метода setTextOnLabel(), поэтому при выборе элемента раскрывающегося списка программа перерисовывает надписи.

- **MouseListener**. Класс, реализующий интерфейс MouseListener, должен определить методы mouseEntered (наведение указателя мыши), mouseExited (отведение указателя мыши), mouseClicked (щелчок мышью), mousePressed (нажатие кнопки мыши) и mouseReleased (отпускание кнопки мыши). Реализация MouseListener существенно отличается от реализации ActionListener. При реализации интерфейса ActionListener (см. листинг 14.1) поток обработки событий реагирует только на щелчок мышью. В то же время при реализации интерфейса MouseListener он реагирует на разные операции с мышью — нажатие кнопки, отпускание кнопки и т.п.

В листинге 14.3 методы `mouseEnered()` и `mouseExited()` вызываются, когда указатель мыши входит в область компонента или выходит из нее. Откуда программа знает, что компонентом в данном случае является надпись, а не текстовое поле или раскрывающийся список? Это видно из того, что метод `addMouseListener()`, добавляющий приемник события мыши, вызывается через переменную `label`, которая обозначает надпись. Иными словами, вызываемый метод `addMouseListener()` принадлежит объекту `label`.

Посмотрите на методы `mouseEntered()` и `mouseExited()` в листинге 14.3. Они вызывают библиотечный метод `setForeground()`, который изменяет цвет надписи. Поэтому при наведении указателя цвет надписи изменяется на красный, а при отведении указателя цвет опять становится черным.

Цвета `red` (красный) и `black` (черный) определены в библиотечном классе `Color` как статические поля класса.

В листинге 14.3 определено несколько методов, которые не используются. Например, когда класс реализует интерфейс `MouseListener`, в нем должен быть определен метод `mouseRelease()`. Он нужен не потому, что программа должна что-либо сделать при отпускании кнопки мыши, а потому, что, объявив реализацию интерфейса, вы пообещали компилятору определить все его методы. В данном примере (см. листинг 14.3) метод `mouseRelease()` пуст, но компилятор не возражает. Формально вы определили метод, а то, что он ничего не делает, для компилятора не важно.

Внутренние классы

В Java можно определить один класс внутри другого. С точки зрения пользователя, листинг 14.5 делает то же самое, что листинг 14.1. Однако в листинге 14.5 класс `GameFrame` содержит класс `MyActionListener`.

Листинг 14.5. Класс в классе

```
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Random;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;

class GameFrame extends JFrame {
  private static final long serialVersionUID = 1L;
  int randomNumber = new Random().nextInt(10) + 1;
  int numGuesses = 0;
  JTextField textField = new JTextField(5);
  JButton button = new JButton("Сделать попытку);
```

```
  JLabel label = new JLabel(numGuesses + " попыток");

  public GameFrame() {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLayout(new FlowLayout());
    add(textField);
    add(button);
    add(label);
    button.addActionListener(new MyActionListener());
    pack();
    setVisible(true);
  }

  class MyActionListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
      String textFieldText = textField.getText();
      if (Integer.parseInt(textFieldText) == randomNumber) {
          button.setEnabled(false);
          textField.setText(textField.getText() + " Угадал!");
          textField.setEnabled(false);
      } else {
        textField.setText("");
        textField.requestFocus();
      }

      numGuesses++;
      String guessWord = (numGuesses == 1) ? " попытка" :
                                             " попыток";
      label.setText(numGuesses + guessWord);
    }
  }
}
```

Класс `MyActionListener` в листинге 14.5 — это *внутренний класс*. Почти во всем он аналогичен обычному классу. Различие между ними лишь в том, что код внутреннего класса может ссылаться на поля внешнего класса. Например, в нескольких инструкциях в определении класса `MyActionListener` используется имя `textField`, определенное во внешнем классе `GameFrame`.

Обратите внимание на то, что в листинге 14.5 класс `MyActionListener` используется только один раз — в вызове метода `button.addActionListener()`. Поэтому возникает вопрос: нужно ли имя предмету, который используется только один раз? Нельзя ли вместо имени подставить сам предмет? Так оно и есть! Определение внутреннего класса можно вставить в инструкцию вызова метода `button.addActionListener()`. Таким образом, в круглых скобках вместо имени можно подставить довольно длинное определение класса (листинг 14.6)! Такой класс называется *анонимным внутренним классом*. Вам такая конструкция может показаться очень странной, но она часто используется на практике.

Листинг 14.6. Класс без имени (анонимный внутренний класс)

```
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Random;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;

class GameFrame extends JFrame {
  private static final long serialVersionUID = 1L;
  int randomNumber = new Random().nextInt(10) + 1;
  int numGuesses = 0;
  JTextField textField = new JTextField(5);
  JButton button = new JButton("Сделать попытку");
  JLabel label = new JLabel(numGuesses + " попыток");

  public GameFrame() {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLayout(new FlowLayout());
    add(textField);
    add(button);
    add(label);
    button.addActionListener(new ActionListener() {

      @Override
      public void actionPerformed(ActionEvent e) {
        String textFieldText = textField.getText();
        if (Integer.parseInt(textFieldText) == randomNumber) {
          button.setEnabled(false);
          textField.setText(textField.getText() + " Угадал!");
          textField.setEnabled(false);
        } else {
          textField.setText("");
          textField.requestFocus();
        }

        numGuesses++;
        String guessWord = (numGuesses == 1) ? " попытка" :
                                               " попыток";
        label.setText(numGuesses + guessWord);
      }
  });
    pack();
    setVisible(true);
  }

  class MyActionListener implements ActionListener {
    @Override
```

```
    public void actionPerformed(ActionEvent e) {
      String textFieldText = textField.getText();
      if (Integer.parseInt (textFieldText) == randomNumber) {
        button.setEnabled(false);
        textField.setText(textField.getText() + " Угадал!");
        textField.setEnabled(false);
      } else {
        textField.setText("");
        textField.requestFocus();
      }

      numGuesses++;
      String guessWord = (numGuesses == 1) ? " попытка" :
                                             " попыток";
      label.setText(numGuesses + guessWord);
    }
  }
}
```

Внутренние классы довольно удобны для определения обработчиков событий, таких как метод `actionPerformed()` в примерах данной главы. Самая сложная проблема при их использовании — правильно отследить скобки (круглые и фигурные) и отступы. Поэтому я вам советую пока что отказаться от них. Сначала пишите коды без внутренних классов, как в листинге 14.1. А затем, когда вы привыкнете к структуре классов, создайте внутренний класс и вставьте его в вызов метода.

Глава 15

Аплеты

В этой главе...

- Пример простого аплета
- Анимированный аплет
- Реагирование на события в аплете

Когда платформа Java впервые ворвалась в компьютерный мир в 1995 году, ее популярность была обусловлена главным образом возможностью создавать аплеты. *Аплет* — это программа на Java, которая выполняется в браузере и создает окно на веб-странице. На странице аплет занимает прямоугольную область и может отображать рисунки и анимацию, принимать информацию от пользователя, воспроизводить аудиофайлы и т.п. Фактически аплет может делать почти все, что делает полнофункциональная компьютерная программа. Его возможности несколько ограничены только требованиями безопасности. Например, аплет не может сохранять информацию на диске без разрешения пользователя.

Пример простого аплета

В листингах 15.1 и 15.2 приведен код простого аплета Java, который отображает прямоугольник и фразу `Java для чайников`. Результат работы этого кода показан на рис. 15.1.

Листинг 15.1. Класс аплета

```
import javax.swing.JApplet;

public class SimpleApplet extends JApplet {
   private static final long serialVersionUID = 1L;

   public void init() {
      setContentPane(new DummiesPanel());
   }
}
```

Листинг 15.2. Класс, рисующий содержимое аплета

```
import javax.swing.JPanel;
import java.awt.Font;
import java.awt.Graphics;

class DummiesPanel extends JPanel {
    private static final long serialVersionUID = 1L;
```

```
    public void paint(Graphics myGraphics) {

        myGraphics.drawRect(50, 60, 250, 75);
        myGraphics.setFont(new Font("Dialog", Font.BOLD, 24));
        myGraphics.drawString("Java для чайников",55,100);
    }
}
```



Рис. 15.1. Выполнение аплета в окне программы Applet Viewer

От обычной программы Java аплет отличается тем, что его запускает веб-браузер, а не операционная система или виртуальная машина Java. Браузер открывает страницу HTML, видит на ней аплет и автоматически его запускает. Страница HTML должна содержать ссылку на аплет (листинг 15.3).

Листинг 15.3. Ссылка на аплет, приведенная на странице HTML

```
<applet code=SimpleApplet width=350 height=200></applet>
```

Во время отладки аплета его удобнее отображать не на веб-странице в браузере, а с помощью специальной утилиты Applet Viewer (см. рис. 15.1). При запуске аплета в рабочей среде Eclipse он выполняется в окне Applet Viewer.

Выполнение аплета

Просматривая листинги 15.1 и 15.2, вы увидите нечто странное: у аплета нет метода `main()`. Это объясняется тем, что аплет не является полнофункциональной программой Java. Аплет — это класс, содержащий методы, вызываемые (непосредственно или косвенно) браузером при отображении веб-страницы на экране. В первую очередь браузер вызывает метод `init()`, показанный в листинге 15.1. Метод `inti()`, в свою очередь, вызывает метод `setContentPane()`, который создает и рисует панель `DummiesPanel`, код которой приведен в листинге 15.2.

Обратите внимание на метод `paint()` в листинге 15.2. Браузер вызывает его автоматически при каждой перерисовке панели аплета. Тело метода `paint()` содержит инструкции, рисующие содержимое аплета на экране.



Список методов аплета, вызываемых браузером, приведен далее.

Открытый класс

Класс `SimpleApplet` в листинге 15.1 объявлен с ключевым словом `public`, поэтому класс аплета является открытым. Если опустить ключевое слово `public`, компилятор вернет сообщение об ошибке. Каждый класс, наследующий `JApplet`, должен быть открытым. Если бы он не был открытым, браузер не смог бы вызывать методы класса.



Любой класс может иметь тип доступа `default` (по умолчанию) и `public` (открытый). Ссылаться на класс, имеющий тип доступа `default`, может только код, расположенный в том же пакете, что и определение класса. Браузер пытается вызвать методы, расположенные внутри класса аплета. Браузер, естественно, находится не в том же пакете, в котором находится класс аплета, поэтому аплет должен быть объявлен с ключевым словом `public`. Если ключевое слово `public` опущено, значит, класс аплета имеет тип доступа `default` и код браузера не может вызывать методы аплета.



Типы доступа `public` и `default` более подробно рассматриваются в главе 13.

Классы Java API

В листинге 15.2 используется ряд инструментов, которые определены в стандартных библиотечных классах Java API.

- ✓ **Метод `drawRect()`** рисует незакрашенный прямоугольник. Размеры и расположение прямоугольника передаются методу `drawRect()` через список параметров. Согласно листингу 15.2 левый верхний угол прямоугольника имеет координаты 50×60 пикселей, а правый нижний угол — 220×75 пикселей. Координаты отсчитываются от левого верхнего угла панели.

  В данном примере прямоугольник обрамляет слова `Java для чайников`. Для этого я подобрал размеры прямоугольника экспериментально. Однако в некоторых задачах нужно автоматически настраивать размеры рамок исходя из размеров надписей, зависящих от параметров шрифта. Для решения этой задачи полезен библиотечный класс `FontMetrics`, возвращающий геометрические параметры шрифтов.

- ✓ **Класс `Font`** определяет параметры шрифта. В листинге 15.2 для надписи задаются следующие параметры: гарнитура шрифта `Dialog`, полужирное начертание (`Font.BOLD`) и высота 24 пункта. На практике чаще используются другие гарнитуры, такие как `Times New Roman`, `Arial`, `Courier New` и др.

- ✓ **Метод `drawString()`** отображает на экране строку символов. В листинге 15.2 он отображает строку `Java для чайников`. Шрифт задан в предыдущей инструкции с помощью метода `setFont()`. Числа `55` и `100` — это координаты левого верхнего угла надписи.

Анимированный аплет

Рассматриваемый в данном разделе аплет более сложный, чем предыдущий, потому что в нем изображение постоянно изменяется. Соответственно, код аплета тоже сложнее (листинги 15.4 и 15.5). Чтобы анимировать изображение, нужно многое сделать, в частности создать и запустить таймер, организовать прослушивание событий таймера и т.д. Однако ваши усилия не будут тщетны, потому что вы сможете использовать эту же инфраструктуру анимации в других задачах.

Листинг 15.4. Аплет `Odometer`

```
import javax.swing.JApplet;
import javax.swing.Timer;
import java.awt.Color;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class Odometer extends JApplet implements ActionListener {
    private static final long serialVersionUID = 1L;
    Timer timer;

    public void init() {
        OdometerPanel panel = new OdometerPanel();
        this.setBackground(Color.white);
        setContentPane(panel);
    }

    public void start() {
        if (timer == null) {
            timer = new Timer(100, this);
            timer.start();
        } else {
            timer.restart();
        }
    }

    public void stop() {
        if (timer != null) {
            timer.stop();
            timer = null;
        }
    }

    public void actionPerformed(ActionEvent e) {
        repaint();
    }
}
```

Листинг 15.5. Панель аплета

```
import javax.swing.JPanel;
import java.awt.Font;
import java.awt.Graphics;
```

```
class OdometerPanel extends JPanel {
    private static final long serialVersionUID = 1L;
    long hitCount = 239472938472L;

    public void paint(Graphics myGraphics) {
        myGraphics.setFont(new Font("Monospaced", Font.BOLD, 24));
        myGraphics.drawString("Вы - посетитель номер " +
                              Long.toString(hitCount++), 50, 50);
    }
}
```

Чтобы вывести этот аплет в браузере, скомпилируйте его в среде Eclipse. Поместите в папку `bin` файл `Odometer.htm`, содержащий следующий код:

```
<applet code=Odometer width=600 height=200></applet>
```

Откройте файл `Odometer.htm` как локальную веб-страницу в любом браузере. На рис. 15.2 показан файл `Odometer.htm`, открытый в браузере Opera. Обратите внимание на то, что число, видимое на рис. 15.2, не равно начальному значению переменной `hitCount`, заданному в листинге 15.5. Это объясняется тем, что каждые 100 мс аплет добавляет единицу к значению `hitCount` и отображает новое значение на экране. Число на экране постоянно изменяется.

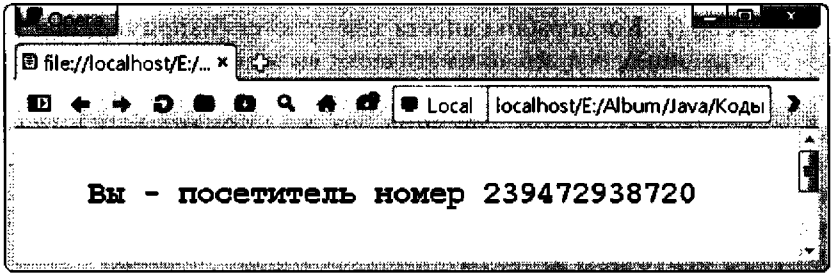


Рис. 15.2. Аплет `Odometer` на веб-странице

Методы, используемые в аплете

Большинство методов, используемых в листингах 15.4 и 15.5, стандартны для всех аплетов. В библиотечных классах `JApplet` и `JPanel` есть определения этих методов, поэтому вам не нужно их объявлять. В коде нужно явно переопределить только те методы, которые необходимо настроить под решаемую задачу.

Ниже приведены методы `JApplet` и `JPanel`, вызываемые браузером автоматически при отображении аплета на странице.

- ✓ **`init()`**. Браузер автоматически вызывает метод `init()` при первом посещении страницы, содержащей аплет. Предположим, вы закрыли браузер. Когда вы вновь запустите браузер и повторно посетите страницу с аплетом, браузер вновь вызовет метод `init()`. Если же посетить страницу повторно, не закрывая браузер, он не вызовет метод `init()`.

- ✓ **`start()`**. Браузер вызывает метод `start()` непосредственно после метода `init()` при первом посещении страницы, а также при каждом повторном посещении страницы или активизации аплета. Если аплет выполняет некоторую непрерывную работу, в тело метода `start()` можно поместить начало или инициализацию этой работы. Например, если аплет содержит анимацию, в метод `start()` можно поместить запуск таймера.
- ✓ **`paint()`**. В листинге 15.5 браузер вызывает метод `paint()` непосредственно после вызова метода `start()`. Методы `start()` и `paint()` могут принадлежать к разным классам, поэтому последовательность их вызова в данном примере определяется только вызовом метода `setContentPane()` в листинге 15.4. Метод `paint()` выполняет перерисовку панели при каждом изменении размеров или перемещении. Более подробно метод `paint()` рассматривается в главе 13.

  Браузер вызывает метод `paint()` несколько раз. Предположим, вы закрыли другим окном браузер (вместе с аплетом), свернули окно браузера или переместили границу окна браузера таким образом, что стала видна только часть аплета. После этого, когда вы вновь откроете или развернете окно браузера и весь аплет снова станет виден на экране, браузер вызовет метод `paint()`, чтобы перерисовать содержимое панели.
- ✓ **`stop()`**. Когда работа аплета завершается (например, при закрытии страницы), браузер автоматически вызывает метод `stop()`. Предположим, вы щелкнули на ссылке, в результате чего браузер переходит на другую страницу. При этом браузер автоматически вызовет метод `stop()`. Потом, когда вы вновь откроете страницу с аплетом, браузер вызовет метод `start()`.

Содержимое методов аплета

В листингах 15.4 и 15.5 используется стандартный способ создания анимации в аплете. Ниже приведено его краткое описание.

- ✓ Аплет реализует интерфейс `ActionListener`, прослушивающий события таймера.
- ✓ Метод `start()` создает таймер с помощью следующего кода:

  ```
  new Timer(100, this)
  ```

  В листинге 15.4 таймер генерирует событие `ActionEvent` каждые 100 мс.

  При возникновении события `ActionEvent` виртуальная машина Java вызывает метод `actionPerformed()`. В каком классе она находит метод `actionPerformed()`? Ответ на этот вопрос дает ключевое слово `this`, которое в данном случае обозначает текущий аплет — объект типа `Odometer` (поскольку код `new Timer(100,this)` находится в определении класса `Odometer`). Итак, таймер генерирует событие, а событие запускает метод `actionPerformed()`. Как все просто!

- ✓ Метод `actionPerformed()` запускает метод `repaint()`. За кулисами при запуске метода `repaint()` виртуальная машина Java всегда вызывает метод `paint()` какого-либо класса. В данном примере код этого класса приведен в листинге 15.5. Его метод `paint()` каждые 100 мс перерисовывает на панели фразу `Вы - посетитель номер` и отображает новое число.
- ✓ Это может продолжаться вечно. Однако ничего вечного в нашем мире нет. В конце концов вам надоест смотреть на изменяющееся число, и вы закроете страницу или браузер. Однако, прежде чем "умереть", браузер автоматически вызовет метод `stop()`, который выполнит необходимые завершающие операции.

Если бы код не был таким стандартным, я почувствовал бы себя виноватым за то, что объясняю его слишком кратко. Вы можете почти без изменений использовать его для приведения в движение своих аплетов. Для этого всего лишь скопируйте листинги 15.4 и 15.5 в свой проект и заполните методы `init()` и `paint()` нужным кодом.

Ниже приведен ряд рекомендаций по заполнению методов `init()` и `paint()`.

- ✓ **Метод `init()` должен содержать конфигурационный код аплета, т.е. код, который выполняется один раз при загрузке аплета.**

  В листинге 15.4 конфигурационный код выполняет ряд подготовительных операций с панелью.

  - Создание панели путем вызова конструктора класса `OdometerPanel`.
  - Заполнение фона аплета белым цветом, чтобы он не контрастировал с фоном веб-страницы. Заполнение выполняется методом `this.setBackground()`. Ключевое слово `this` обозначает текущий аплет. Если вы хотите увидеть область, занимаемую аплетом на веб-странице, задайте заполнение фона другим цветом.
  - Размещение панели в аплете путем вызова метода `setContentPane()`.
- ✓ **Метод `paint()` создает один кадр анимации.** Программа многократно вызывает метод `paint()`, что позволяет создать иллюзию движения.

  В листинге 15.5 метод `paint()` устанавливает шрифт графического буфера `myGraphics`. Если важна производительность, это можно сделать один раз, а не при каждой перерисовке. Метод `drawString()` отображает на экране изменяющееся значение переменной `hitCount`. Изменение переменной `hitCount` выполняется в списке параметров `drawString()` с помощью оператора `++`. Значение типа `long` преобразуется в значение `String` с помощью статического метода `toString()` класса `Long`.

Для отладки аплета поместите в его код вызовы `System.out.println()`. Если вы работаете в интегрированной среде Eclipse, то строки, выводимые методом `println()`, будут видны в окне консоли Eclipse.

Реагирование на события в аплете

Рассмотрим пример аплета с интерактивными элементами интерфейса, реагирующими на действия пользователя. Данный аплет очень похож на пример, приведенный в главе 14 (см. листинг 14.1). Фактически отличаются они только тем, что вместо фрейма теперь используется аплет. Я сделал так не потому, что я ленивый (хотя я действительно ленивый), но главным образом для того, чтобы продемонстрировать тот факт, что аплеты почти ничем не отличаются от фреймов. Почти всегда можно, немного подправив код фрейма, сделать из него аплет. Код аплета с интерактивными элементами приведен в листингах 15.6 и 15.7.

Листинг 15.6. Аплет с игрой в угадывание числа

```
import javax.swing.JApplet;

public class GameApplet extends JApplet {
    private static final long serialVersionUID = 1L;

    public void init() {
        setContentPane(new GamePanel());
    }
}
```

Листинг 15.7. Панель аплета

```
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Random;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

class GamePanel extends JPanel implements ActionListener {
   private static final long serialVersionUID = 1L;
   int randomNumber = new Random().nextInt(10) + 1;
   int numGuesses = 0;
   JTextField textField = new JTextField(5);
   JButton button = new JButton("Сделать попытку");
   JLabel label = new JLabel(numGuesses + " попыток");

   GamePanel() {
       setBackground(Color.WHITE);
       add(textField);
       add(button);
       add(label);
       button.addActionListener(this);
   }
```

```
    public void actionPerformed(ActionEvent e) {
        String textFieldText = textField.getText();
        if (Integer.parseInt(textFieldText) == randomNumber) {
            button.setEnabled(false);
            textField.setText(textField.getText() + " Угадал!");
            textField.setEnabled(false);
        } else {
            textField.setText("");
            textField.requestFocus();
        }
        numGuesses++;
        String guessWord = (numGuesses == 1) ? " попытка" :
                                               " попыток";
        label.setText(numGuesses + guessWord);
    }
}
```

Чтобы выполнить этот аплет в браузере, поместите в папку `bin` файл `GameApplet.htm`, содержащий только следующий код:

```
<applet code="GameApplet" width=225 height=50></applet>
```

На рис. 15.3 и 15.4 показано выполнение аплета в браузере Firefox. Как видите, аплет выглядит так же, как фрейм, созданный листингом 14.1. Единственное существенное различие между ними состоит в том, что аплет выполняется в браузере как часть веб-страницы, а фрейм отображается в окне настольного приложения.

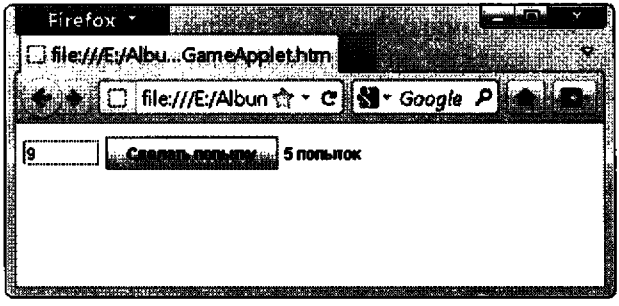


Рис. 15.3. Пользователь не угадал число

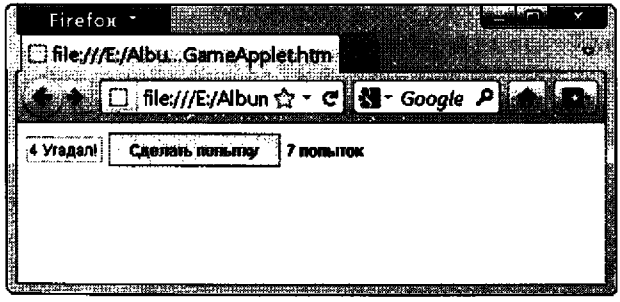


Рис. 15.4. Пользователь угадал число

Подойдем к анализу листинга 15.7 с обратной стороны: рассмотрим не то, что в нем есть, а то, чего в нем нет. Чтобы превратить листинг 14.1 в листинг 15.7, я изменил в нем всего несколько строк.



- ✓ **Удален вызов метода `setLayout()`, который определяет компоновку окна.** По умолчанию применяется компоновка `FlowLayout`, которая нужна в аплете.

  Компоновки рассматриваются в главе 9.
- ✓ **Удален вызов метода `pack()`,** потому что ширину и высоту аплета определяют атрибуты `width` (ширина) и `height` (высота) дескриптора `applet`, приведенного в документе HTML.
- ✓ **Нет вызова метода `setVisible()`.** По умолчанию аплет видимый.

Еще одно различие нужно отметить между листингами 14.2 и 15.6. Как и в других аплетах, в листинге 15.6 нет метода `main()`. Вместо него используется метод `init()`. Метод `main()` не нужен, потому что инструкция `new GameApplet()`, создающая объект аплета, не используется. Объект аплета создается браузером, он же вызывает метод `init()` при загрузке аплета. Это стандартный сценарий выполнения аплета Java в браузере.

Глава 16

Соединение с базой данных

В этой главе...

- JDBC и Java DB
- Создание записей базы данных
- Извлечение данных

Когда я читаю лекции по Java для профессиональных программистов, я всегда слышу от них одно и то же: "Нам не нужны буковки, бегающие по экрану, и мерцающие кнопки. Покажите нам, как обратиться к базе данных. Это то, что для нас действительно важно". Вскоре вы, конечно же, тоже станете профессиональным программистом, поэтому я расскажу вам, как код Java взаимодействует с базами данных.

JDBC и Java DB

Когда я начал работать с базой данных, самой сложной проблемой для меня была установка соединения с ней. Я написал весь необходимый код на Java (ну ладно, признаюсь, скопировал из других книг). Часть задачи, касающаяся кода Java, оказалась самой легкой. Намного тяжелее было подключить код к базе данных, работающей в операционной системе.

Сложность этой задачи обусловлена тем, что способ подключения кода Java к базе данных зависит от операционной системы и типа системы управления базой данных (СУБД). Во многих книгах подробно рассматриваются инфраструктура и конфигурационные параметры подключения, не имеющие никакого отношения к Java, однако эти книги помогают мало, потому что на каждом компьютере что-нибудь сконфигурировано нестандартным способом. И вот теперь я тоже пишу главу о соединении с базой данных. Что еще остается делать автору книги о Java?

К счастью, пакет JDK (Java Development Kit — набор инструментов разработки приложений на Java) поставляется со встроенной базой данных Java DB, основанной на инфраструктуре баз данных Apache Derby. Инфраструктура Java DB безопасная, легкая и стандартная. К тому же она выполняется совместно с другими инструментами Java JDK. Java DB для пользователей появилась начиная с версии Java 6.

Наличие системы Java DB облегчает мне жизнь тем, что предоставляет всем читателям книги общую базу данных, которую они могут использовать при работе с примерами. Эта база данных бесплатная и не требует конфигурирования.

А что делать, если заказчик использует не Java DB, а базу данных другого типа? Сейчас на рынке информационных технологий можно найти огромное количество самых разных баз данных, включая MySQL, PostgreSQL, SQLite, Oracle, Microsoft Access, DB2 и др. Инфраструктура Java предоставляет решение и на этот случай. Для установки соединения с базами данных других типов можно использовать стандартные

библиотечные классы JDBC (Java Database Connectivity — соединение с базами данных на Java). Эти классы предоставляют доступ к большинству современных систем управления базами данных. Нужно лишь найти драйвер используемой базы данных и отредактировать несколько строк кода в примерах данной главы.

Создание записей базы данных

Инфраструктура JDBC находится в пакетах `java.sql` и `javax.sql`. В примерах данной главы используются классы пакета `java.sql`. Первый пример приведен в листинге 16.1. В данном примере каждая запись базы данных содержит следующую информацию о счете: имя клиента, адрес клиента и сумма на счету. Приведенная программа создает таблицу и вставляет в нее две записи о клиентах.

Листинг 16.1. Создание базы данных и таблицы; вставка данных

```
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.Connection;
import java.sql.SQLException;

class CreateTable {

   public static void main(String args[]) {
      final String DRIVER = "org.apache.derby.jdbc.EmbeddedDriver";
      final String CONNECTION = "jdbc:derby:AccountDatabase;
                                 create=true";

      try {
         Class.forName(DRIVER).newInstance();
      } catch (InstantiationException e) {
         e.printStackTrace();
      } catch (IllegalAccessException e) {
         e.printStackTrace();
      } catch (ClassNotFoundException e) {
         e.printStackTrace();
      }

      try (Connection connection =
              DriverManager.getConnection(CONNECTION);

           Statement statement = connection.createStatement()) {

        statement.executeUpdate(
          "create table ACCOUNTS                          "
          + "   (NAME VARCHAR(32) NOT NULL PRIMARY KEY, "
          + "    ADDRESS VARCHAR(32),                     "
          + "    BALANCE FLOAT)                        ");

        statement.executeUpdate(
```

```
            "insert into ACCOUNTS values                    "
            + "  ('Барри Берд', 'улица Киберов, 222', 24.02)");

          statement.executeUpdate(
            "insert into ACCOUNTS values                    "
            + "  ('Джо Доу', 'улица Лудитов, 112', 55.63)");

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```



Если хотите применить MySQL вместо Java DB, внесите в листинг 16.1 следующие изменения: замените значение `DRIVER` на `com.mysql.jdbc.Driver`, а значение `CONNECTION` — на `jdbc:mysql://localhost/AccountDatabase;create=true`. Аналогичные изменения нужно внести для DB2, Oracle и других баз данных.

Чтобы выполнить код листинга 16.1, нужно файл с драйвером базы данных поместить в место, где Java сможет его найти. В данном примере мы устанавливаем соединение с базой данных Java DB, другое название которой — Apache Derby. Драйвер базы данных находится в файле `derby.jar`, который находится в папке `db/lib` пакета JDK. Чтобы файл `derby.jar` был доступен программам на Java, добавьте его маршрут в переменную `CLASSPATH` операционной системы Windows.



Можно также добавить `derby.jar` в список ссылок проекта. Для этого выберите в Eclipse команду Project⇨Properties⇨Java Build Path⇨Libraries⇨Add External JARs (Проект⇨Свойства⇨Маршруты сборки Java⇨Библиотеки⇨Добавить внешние файлы JAR). Найдите и выделите файл `derby.jar` и щелкните на кнопке Открыть.

При выполнении листинга 16.1 кажется, что ничего не произошло. Программа скомпилировалась, запустилась и завершилась; при этом внешне ничего не изменилось. Это объясняется тем, что в программе нет инструкций вывода на консоль. Однако откройте Проводник Windows и загляните в папку `16-01`. Вы увидите, что в ней появилась папка `AccountDatabase`. Это база данных, созданная программой.



Можете также проанализировать результат работы листинга 16.1 более подробно. Откройте файл журнала `derby.log`, находящийся в папке `16-01`. В нем вы увидите описание этапов работы базы данных. В появившейся папке `derbyDB` есть дополнительные файлы журналов. Файлы с расширением `.dat` содержат информацию, хранящуюся в базе данных. Можно сделать эти файлы видимыми в рабочей среде Eclipse. Для этого выделите имя проекта на левой панели Eclipse и выберите в меню команду File⇨Refresh (Файл⇨Обновить).

Использование команд SQL

Наиболее важная часть листинга 16.1 — три вызова метода `executeUpdate()`, выполняющих три команды SQL (одну команду `create` и две команды `insert`). При каждом вызове метод `executeUpdate()` получает в списке параметров строку (переменную типа `String`). Чтобы строки было легче читать, я разбил их на несколько частей, отделенных друг от друга символом +.

Оператор + может выполнять в Java разные операции в зависимости от того, где он находится. Для чисел оператор + означает суммирование, а для строк — конкатенацию, т.е. объединение двух строк в одну.



В код можно вставить строковый литерал произвольной длины. Когда во время ввода курсор дойдет до правой границы экрана, продолжайте вводить строку. Код сам сдвинется влево. Однако слишком длинные строки неудобны. Если вы хотите видеть всю строку без прокрутки, разбейте ее на несколько частей с помощью операторов +, как в листинге 16.1.



Разбить строку Java на несколько частей, просто нажимая клавишу <Enter>, нельзя. Между символами « (двойная кавычка) не должно быть разрывов строк. Иными словами, весь строковый литерал должен находится в одной строке кода.

Если вы знакомы с языком SQL (Structured Query Language — язык структурированных запросов), то вы легко заметите, что методу `executeUpdate()` в качестве параметра передается команда SQL. Если же не знакомы, почитайте какую-нибудь книгу по SQL. В данной главе я не объясняю команды SQL (`create table` и `insert table`), используемые в примере, потому что это команды SQL, а не Java. Пока что представляйте их себе как строковые литералы, передаваемые методу `executeUpdate()`. Команда `create` создает таблицу в базе данных, а команда `insert` вставляет в нее запись. Таким образом, при написании программы на Java можно запускать команды SQL. Для этого их нужно представить как строковые литералы и передать методу `executeUpdate()`.



Код данного примера строго придерживается спецификации JDBC v.1.0. Более поздние версии JDBC поддерживают *прокручиваемые результирующие наборы*. Они позволяют применять методы наподобие `insertRow()`, которые выполняют те же операции, что и команды SQL. Поэтому в более поздних версиях JDBC можно манипулировать базами данных с помощью методов Java, отказавшись от команд SQL.

Подключение и отключение базы данных

Кроме вызова методов `executeUpdate()`, код листинга 16.1 выполняет ряд вспомогательных операций. Ниже приведено описание каждой части кода.

- ✓ **`Class.forName()`.** Поиск драйвера базы данных.

  Для сообщения программы Java с базой данных необходимо промежуточное программное обеспечение, которое называется *драйвером базы данных*. Существует огромное количество драйверов, причем многие из

них очень дорогие. В листинге 16.1 используется небольшой бесплатный драйвер Derby JDBC Embedded. Его код находится в классе `EmbeddedDriver`, который расположен в пакете `org.apache.derby.jdbc`.

Чтобы можно было использовать класс `EmbeddedDriver`, необходимо вызвать метод `Class.forName()`. Как ни странно это звучит, в Java API есть класс `Class`. Он содержит информацию о стандартных классах, доступных для виртуальной машины Java. В листинге 16.1 метод `Class.forName()` находит класс `org.apache.derby.jdbc.EmbeddedDriver`. После загрузки экземпляра `EmbeddedDriver` можно продолжить процесс подключения к базе данных.

- ✓ **`DriverManager.getConnection()`.** Установка сеанса работы с конкретной базой данных.

  Метод `getConnection()` принадлежит классу `DriverManager`. В листинге 16.1 метод `getConnection()` создает экземпляр базы данных `AccountDatabase` и открывает соединение с ней. Может случиться, что при вызове метода `getConnection()` база данных `AccountDatabase` уже существует. Строка `create=true` означает, что в этом случае вызов метода `getConnection()` ни на что не влияет.

  В листинге 16.1 обратите внимание на двоеточия в параметрах метода `getConnection()`. Код не только присваивает базе данных имя `AccountDatabase`, но и сообщает классу `Drivermanager`, какой протокол необходимо использовать для соединения с базой данных. Код `jdbc:derby:` играет ту же роль, что и префикс `http:` в веб-адресе — он приказывает компьютеру использовать протокол `jdbc` для коммуникации по протоколу `derby`, который обеспечивает подключение к базе данных `AccountDatabase`.

- ✓ **`connection.createStatement()`.** Создание команды SQL.

  В протоколе JDBC объект `statement`, представляющий команду SQL, создается один раз. После этого его можно использовать много раз для запуска разных команд SQL с помощью метода `statement.executeUpdate()`. При каждом вызове методу передаются разные строковые литералы, поэтому выполняются разные команды SQL. Объект `statement` создается путем вызова метода `connection.createStatement()`.

- ✓ **`try...catch`.** Обработка исключений Java.

  Исключения рассматриваются в главе 12. При работе с базой данных могут возникать *проверяемые исключения*, вызванные разными неполадками в соединениях. Существуют два вида исключений: проверяемые и непроверяемые. Проверяемое исключение отличается тем, что оно обязательно должно быть перехвачено и обработано. Если в коде есть инструкция, которая может сгенерировать проверяемое исключение, она должна быть помещена в блок `try`, иначе компилятор сообщит об ошибке. Вызов метода `Class.forName()` может порождать исключения трех типов. Почти любая операция с базой данных может порождать исключение типа `SQLException`.

- ✓ **Блоки try с ресурсами.** Освобождение ресурсов при неожиданном исключении.

  Как говорит моя супруга, вы не можете предъявлять претензии к другим, если сами создаете беспорядок. Каждое соединение с базой данных блокирует ряд ресурсов компьютера. Когда вы закончили пользоваться некоторыми ресурсами, вы должны освободить их. Это можно сделать явно, вызвав метод `close()`. Обращение к ресурсам происходит в блоке `try-catch`, следовательно, метод `close()` нужно вызвать также в блоке `try-catch`.

  Но есть одна проблема. Когда что-либо происходит и система генерирует исключение, программа мгновенно "выпрыгивает" из блока `try`, и метод `close()` не будет вызван. Можно поместить вызов `close()` еще и в блок `catch`, но в нем также может быть сгенерировано исключение. В результате вы рискуете оставить соединение с базой данных открытым, даже после завершения программы. Предприняв еще несколько попыток, вы исчерпаете ресурс открытых соединений и не только сами ничего больше не сможете сделать, но и заблокируете базу данных для других пользователей.

  Для решения этой проблемы в Java 7 были введены блоки `try` с ресурсами. От обычных блоков `try` они отличаются тем, что перед открывающей фигурной скобкой блока добавляется список инструкций, создающих ресурсы, используемые в блоке `try`. В листинге 16.1 в список инструкций включены вызовы методов `getConnection()` и `createStatement()`. Инструкции в списке, как обычно, разделены точками с запятыми.

  При завершении блока `try` виртуальная машина Java автоматически закрывает и освобождает ресурсы, открытые в списке. Кроме того, виртуальная машина заботится обо всех проблемах, связанных с неудачными попытками правильно обработать исключения.

Извлечение данных

Чтобы от базы данных была какая-либо польза, нужно уметь извлекать из нее информацию. В данном разделе мы выполним запрос к базе данных `AccountDatabase`, созданной и заполненной информацией в листинге 16.1. Код запроса к базе данных приведен в листинге 16.2.

Листинг 16.2. Запрос к базе данных

```
import static java.lang.System.out;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.ResultSet;
import java.text.NumberFormat;
```

```
class GetData {

   public static void main(String args[]) {
      NumberFormat currency = NumberFormat.getCurrencyInstance();
      final String DRIVER = "org.apache.derby.jdbc.EmbeddedDriver";
      final String CONNECTION = "jdbc:derby:AccountDatabase";

      try {
         Class.forName(DRIVER).newInstance();
      } catch (InstantiationException e) {
         e.printStackTrace();
      } catch (IllegalAccessException e) {
         e.printStackTrace();
      } catch (ClassNotFoundException e) {
         e.printStackTrace();
      }

      try (Connection connection =
         DriverManager.getConnection(CONNECTION);

         Statement statement = connection.createStatement();
         ResultSet resultset =
            statement.executeQuery("select * from ACCOUNTS")) {
         while(resultset.next()) {
            out.print(resultset.getString("NAME"));
            out.print(", ");
            out.print(resultset.getString("ADDRESS"));
            out.print(" ");
            out.println(currency.format(resultset.
                        getFloat("BALANCE")));
         }
      } catch (SQLException e) {
         e.printStackTrace();
      }
   }
}
```



Чтобы создать соединение с MySQL (вместо Java DB), внесите в листинг 16.2 следующие изменения: присвойте переменной `DRIVER` значение `com.mysql.jdbc.Driver`, а переменной `CONNECTION` — значение `jdbc:mysql://localhost/AccountDatabase;create=true`. Аналогичные изменения необходимы также для работы с DB2, Oracle и базами данных других типов.

Скопируйте папку `AccountDatabase` из каталога `16-01` в `16-02`. Папка `AccountDatabase` — это база данных, созданная и заполненная информацией в предыдущем примере. Запустите на выполнение код листинга 16.2. Результат показан на рис. 16.1. Код выполняет запрос к базе данных, получает от нее две записи и отображает их на консоли.

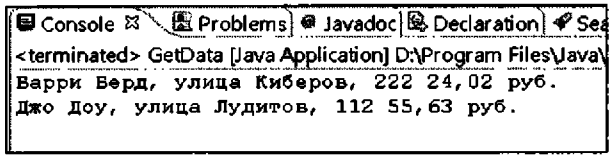


Рис. 16.1. Извлечение информации из базы данных

Листинг 16.2 начинается, как и листинг 16.1, с вызова методов `forName()`, `getConnection()` и `createStatement()`. Затем код вызывает метод `executeQuery()` и передает ему команду SQL в качестве параметра. Те, кто знают SQL, легко увидят, что указанная команда SQL извлекает все данные из таблицы `ACCOUNTS`, созданной листингом 16.1.

Результат запроса, возвращенный методом `executeQuery()`, имеет тип `java.sql.ResultSet`. В этом главное различие между методами `executeUpdate()` и `executeQuery()`: они оба выполняют запрос к базе данных, но первый возвращает результирующий набор записей, а второй не возвращает. Фактически возвращенный результирующий набор является таблицей базой данных. Как и реальная таблица, он состоит из строк и столбцов. Каждая строка называется *записью* и содержит данные об одном счете — имя клиента, адрес клиента и сумму на счете.

После вызова метода `executeQuery()` и получения результирующего набора код проходит по записям таблицы. Для этого используется цикл, проверяющий условие `resultset.next()` в начале каждой итерации. Метод `resultset.next()` решает две задачи.

- ✓ Он перемещает курсор базы данных на следующую запись (если она существует). *Курсор* — это указатель на запись базы данных.
- ✓ Если записей больше нет, он возвращает значение `false`, используемое для выхода из цикла.

Если условие `resultset.next()` равно `true`, значит, в результирующем наборе есть еще одна запись. В этом случае компьютер входит в тело цикла, извлекает запись и отображает ее на консоли. В противном случае компьютер выходит из цикла.

Остановимся на моменте, когда курсор указывает на запись в результирующем наборе и выполняется тело цикла в листинге 16.2. Компьютер извлекает данные из записи результирующего набора путем вызова методов `getString()` и `getFloat()`. В листинге 16.1 в таблицу `ACCOUNTS` (СЧЕТА) добавлены столбцы `NAME` (ИМЯ), `ADDRESS` (АДРЕС) и `BALANCE` (СУММА НА СЧЕТЕ). Соответственно, в листинге 16.2 программа извлекает данные из этих полей путем вызова методов `getString()` и `getFloat()`. Имена столбцов передаются этим методам. Получив данные, программа выводит их на консоль с помощью методов `println()`.



Каждый экземпляр результирующего набора `ResultSet` имеет ряд методов `get...`, позволяющих извлекать данные разного типа: `getArray()`, `getBigDecimal()`, `getBlob()`, `getInt()`, `getObject()`, `getTimestamp()` и др.

Часть V

Великолепные десятки

The 5th Wave — Рич Теннант



В этой части...

Вот вы и подошли к концу книги. Пришло время закончить наш разговор. После долгих размышлений я решил завершить его рядом полезных советов и методик, которые помогут вам стать профессиональным программистом. Я приведу здесь несколько сайтов, которые будут полезны вам при дальнейшем изучении Java. Если вы прочитали не все предыдущие главы или даже не прочитали ни единого слова из них, но немного знакомы с Java, данная часть все равно будет для вас полезной. Если же вы читали предыдущие главы, в данной части вы найдете применение полученным знаниям.

Глава 17

Десять способов избежать ошибок

В этой главе...

- Правильное использование регистра букв
- Выход из блока switch
- Сравнение двух значений
- Добавление элемента в графический интерфейс
- Добавление приемников событий
- Определение конструкторов
- Исправление нестатических ссылок
- Соблюдение границ массива
- Указатели на null
- Помогите виртуальной машине Java найти классы

Не ошибается только тот, кто ничего не делает. Нет ничего зазорного в том, чтобы совершить ошибку. Но это совсем не значит, что к ошибкам можно относиться снисходительно. Нужно приложить все усилия к тому, чтобы их было как можно меньше. Особенно это справедливо в программировании, потому что одна ошибка делает неработоспособной всю программу.

Правильное использование регистра букв

Язык Java чувствителен к регистру букв, поэтому при выборе имен переменных, классов и членов классов нужно соблюдать стиль именования, привычный для других программистов. Чем менее странной и более привычной будет ваша рабочая среда, тем меньше ошибок вы совершите и тем более качественной будет программа.

- ✓ Имена классов должны начинаться с буквы в верхнем регистре, а имена переменных и членов класса — в нижнем.
- ✓ Ключевые слова Java всегда пишутся только в нижнем регистре. Например, `if` нельзя написать как `If` или `IF`.
- ✓ Если в коде используются имена библиотечных классов и членов классов, регистр каждой буквы должен быть тот же, что и в библиотеке.
- ✓ Проследите за тем, чтобы каждое имя, используемое в разных частях программы, состояло из букв в одном и том же регистре. Например, если переменная объявлена как `myAccount`, нельзя в другом месте обратиться к ней как к `myaccount`. Если регистр хоть одной буквы изменится, компилятор будет считать их двумя разными переменными.

Чувствительность идентификаторов Java к регистру рассматривается в главе 3.

Выход из блока switch

В большинстве случаев после выполнения набора инструкций, принадлежащих конкретному условию case, нужно завершить выполнение блока switch. Для этого после набора инструкций нужно расположить оператор break. Если его нет, компьютер выполнит также следующий набор инструкций, после него — еще один, и так до тех пор, пока не встретит оператор break или закрывающую фигурную скобку блока switch. Например, если переменная verse равна 3, приведенный ниже код выведет на консоль все три строки.

```
switch (verse) {
case 3:
  out.println("Строка_1");
case 2:
  out.println("Строка_2");
case 1:
  out.println("Строка_3");
}
```

Более подробно оператор break описан в главе 5.

Сравнение двух значений

Не забывайте, что для сравнения двух значений нужно использовать два знака равенства, а не один. Например, приведенное ниже условие правильное:

```
if (inputNumber == randomNumber)
```

А вот следующее условие неправильное:

```
if (inputNumber = randomNumber)
```

Операторы сравнения рассматриваются в главе 5.

Добавление элемента в графический интерфейс

Ниже приведен конструктор фрейма.

```
public SimpleFrame() {
  JButton button = new JButton("Ввести данные");
  setTitle("Программа ввода данных");
  setLayout(new FlowLayout());
  add(button);
  button.addActionListener(this);
  setSize(300, 100);
  setVisible(true);
}
```

Создавая любой элемент интерфейса, не забывайте вызывать метод add(), добавляющий его в фрейм. Будет обидно, если вы проделаете всю работу по созданию кнопки, но забудете добавить ее во фрейм, в результате чего она так и не появится на экране. Создание кнопки во фрейме рассматривается в главе 9.

Добавление приемников событий

В предыдущем разделе приведен код конструктора класса `SimpleFrame`. Если вы забудете вызвать метод `addActionListener()`, то при щелчке на кнопке ничего не произойдет. Пользователь может еще раз щелкнуть на ней, на этот раз сильнее, но результат будет тот же. Прослушивание событий рассматривается в главе 14.

Определение конструкторов

В класс можно добавить конструктор с параметрами.

```
public Temperature(double number)
```

Но не забывайте, что в этом случае компилятор не создаст конструктор без параметров, определенный по умолчанию. Следовательно, вы не сможете создать объект с помощью конструктора без параметров, как показано ниже.

```
Temperature roomTemp = new Temperature();
```

Это можно будет сделать, только если вы определите собственный конструктор без параметров. Более подробно конструкторы рассматриваются в главе 9.

Исправление нестатических ссылок

Приведенный ниже код не компилируется.

```
class WillNotWork {
  String greeting = "Привет!";
  public static void main(String args[]) {
    System.out.println(greeting);
  }
}
```

Ошибка состоит в том, что метод `main()` статический, а поле `greeting` не статическое. Метод `main()` принадлежит классу, а не экземпляру класса, а поле `greeting`, наоборот, — экземпляру класса, а не классу. Какой экземпляр поля `greeting` имел в виду программист, когда вставил его в метод `main()`? Очевидно, что ответа на этот вопрос не существует. Правила использования статических и нестатических полей приведены в главе 10.

Соблюдение границ массива

Не забывайте, что при объявлении массива с десятью элементами они нумеруются от 0 до 9. Предположим, массив `guests` создается следующим образом:

```
int guests[] = new int[10];
```

Тогда на элементы массива можно ссылаться так: `guests[0]`, `guests[1]`... `guests[9]`. На этом массив заканчивается. Если написать `guests[10]`, произойдет ошибка, потому что элемента номер 10 не существует. Массивы рассматриваются в главе 11.

Указатели на null

Примеры данной книги мало подвержены исключениям типа NullPointerException (исключение, указатель на null), потому что они специально сделаны простыми и неприхотливыми. Однако в реальной жизни исключения этого типа — кошмар любого программиста. Чаще всего исключение NullPointerException возникает, когда метод должен был вернуть объект, но ничего не вернул. Рассмотрим простой пример.

```
import static java.lang.System.out;
import java.io.File;
class ListMyFiles {
  public static void main(String args[]) {
    File myFile = new File("\\windows");
    String dir[] = myFile.list();
    for (String fileName : dir) {
      out.println(fileName);
    }
  }
}
```

Предполагается, что программа отобразит на консоли список всех файлов, находящихся в папке windows (использование двойной обратной косой рассматривается в главе 8).

Однако что произойдет, если программа не найдет папку windows или если в строке приведено неправильное имя папки?

```
File myFile = new File("&*%$!!");
```

В этом случае оператор new File() возвратит null. Соответственно, переменной myFile будет присвоено значение null. В следующей инструкции переменная dir не будет создана, и цикл станет невыполнимым. Программа сгенерирует исключение NullPointerException и (если код не перехватывает исключение) будет аварийно завершена.

Чтобы избежать подобных ситуаций, внимательно знакомьтесь с документацией по используемым библиотечным классам. Если код вызывает метод, который может вернуть null, добавьте блок обработки исключений. Обработка исключений рассматривается подробнее в главе 12.

Помогите виртуальной машине Java найти классы

Ошибка NoClassDefFoundError (не найдено определение класса) может возникнуть по многим причинам, но чаще всего она происходит, когда Java не может найти нужный файл с определением класса. Чтобы избежать этой ошибки, придерживайтесь следующих правил:

- ✓ папка проекта должна содержать все файлы Java с классами, используемыми в коде;

- ✓ если используются именованные пакеты, папка проекта должна содержать соответствующую иерархию вложенных папок;
- ✓ правильно установите переменную `CLASSPATH` в параметрах операционной системы.

Дополнительная информация о том, где Java ищет нужные файлы, приведена в главе 13.

Глава 18

Десять сайтов, посвященных Java

В этой главе...

- Веб-сайты этой книги
- Сайты Java
- Новости, обзоры, примеры кодов
- Работа
- Сайты для каждого

Интернет — неисчерпаемый источник полезной информации по любой теме, в том числе по Java. В данной главе перечислены десять наиболее полезных и забавных сайтов, которые помогут вам эффективнее использовать Java. Все они бесплатны, и ни на одном из них не используются рекламные жучки, всплывающие окна и другие вредные выдумки.

Веб-сайты этой книги

1. Сайт, посвященный этой книге и поддерживаемый ее автором, находится по такому адресу:

   `http://users.drew.edu/bburd/JavaForDummies/`

2. На веб-странице книги на сайте издательства "Диалектика" содержатся коды примеров и другая полезная информация:

   `http://www.dialektika.com/books/978-5-8459-1769-0.html`

Сайты Java

3. Официальный сайт Oracle находится по такому адресу:

   `http://oracle.com/technetwork/java`

   Русская версия сайта Oracle имеет такой адрес:

   `http://oracle.com/ru/technetworks/java/index.html`

4. Для загрузки программного обеспечения Java посетите сайт `http://java.com`.
5. Профессиональные программисты и разработчики обмениваются опытом работы на Java на сайтах `http://java.net` и `http://developers.sun.ru`.

Новости, обзоры, примеры кодов

6. Многочисленные популярные статьи крупных экспертов по Java есть на сайте `http://javaportal.ru/java/articles`.
7. Популярный форум, посвященный Java, имеет адрес `http://javatalks.ru`.

Работа

8. Объявления о вакансиях программистов на Java в разных городах России можно найти на следующих сайтах:

```
http://jooble.ru/search-vacancy-rossiya/ctg-all/kw-java
http://www.trud.com/rabota-ru--programmisty_java.html
```

Сайты для каждого

Эти два сайта не посвящены исключительно Java, однако любой список источников информации по компьютерным технологиям был бы без них неполным.

9. Популярный сайт, посвященный компьютерным технологиям и программированию, — `http://slashdot.org`.
10. Хранилище SourceForge (`http://sourceforge.net`) содержит 200 тысяч бесплатных проектов и приложений, включая исходные коды.

Часть VI

Приложения

The 5th Wave — Рич Теннант



В этой части...

В данной части мы рассмотрим процесс загрузки и установки программного обеспечения, необходимого для программирования на Java, и работу с программой Eclipse, с помощью которой удобно редактировать и выполнять приложения Java. Эти вопросы не связаны с грамматикой языка Java, поэтому я вынес их в приложения.

Установка интегрированной среды разработки

В этом приложении...

- Загрузка и установка JDK
- Загрузка и установка Eclipse
- Конфигурирование Eclipse

Существует много разных сред разработки приложений Java, однако для работы с данной книгой рекомендуется установить только пакет JDK и программу Eclipse.

Загрузка и установка JDK

Если вы работаете в операционной системе Windows, Linux или Solaris, можете загрузить пакет JDK (Java Development Kit — набор инструментов разработки приложений Java) с официального сайта компании Oracle (`http://oracle.com`). Пользователям Mac предоставляются другие возможности.

Java на платформе Windows, Linux и Solaris

На момент написания данного раздела точный адрес веб-страницы загрузки JDK для Windows, Linux и Solaris был следующий:

```
http://oracle.com/technetwork/java/javase/downloads
```

Видимо, он не останется неизменным вечно. Однако название компании Oracle вряд ли изменится к тому моменту, когда вы прочтете эти строки, поэтому, если по указанному выше адресу вы не найдете нужные ресурсы, посетите сайт `http://oracle.com` и откройте страницу `downloads`. В любом случае учитывайте следующее.

- ✓ **Компания Oracle поставляет три разных издания Java — SE, EE и ME.**

  Язык программирования — это набор синтаксических и грамматических правил, определяющих способ написания инструкций для компьютера в исходном коде программы. Библиотека API (Application Programming Interface — интерфейс программирования приложений) — это набор стандартных классов, предназначенных для решения общих задач на данном языке.

  Есть только один язык Java, но для него есть три разные официальные библиотеки API. Загрузив Java, вы получаете определенный набор инструментов для работы с исходным кодом и одну из трех библиотек API. Ниже приведено краткое описание каждого издания.

- **Издание SE** (Standard Edition — стандартное издание) содержит все, что нужно для работы на настольном компьютере, включая обработку текста, математические вычисления, ввод и вывод, коллекции объектов и т.п.

  Для работы с примерами данной книги достаточно иметь издание SE.

- **Издание EE** (Enterprise Edition — корпоративное издание) содержит средства, полезные при работе с мощными корпоративными серверами. В него включены инструменты для работы с веб-серверами и базами данных, средства обмена сообщениями между клиентами и серверами, средства управления распределенными системами и т.п.

- **Издание ME** (Micro Edition — миниатюрное издание) предназначено для мобильных устройств, таких как смартфоны, планшетные компьютеры, букридеры, плееры и пр. Это издание предоставляет ограниченный набор средств, необходимых для специализированных устройств, менее мощных, чем настольные компьютеры.



Для работы с данной книгой вам необходимо издание SE. Кроме того, учитывайте следующее.

- ✓ **Существуют разные версии пакета Java. Каждая версия имеет ряд обновлений.**

  Когда-то номера версий Java начинались с единицы. Первой была версия 1.0, затем появилась версия 1.1, после чего началось нечто странное: появилась Java 2, v.1.2, после нее — Java 2, v.1.3 и т.д. Как произошел прыжок от 1.4 до 5.0, до сих пор никто не может понять.

  В конце концов, первое число решили убрать, и появились версии Java 6 и Java 7 (на данный момент это последняя версия).

  В первый момент появления версии поставляется ее обновление номер 1 (update 1 или короче — `u1`). Постепенно номер обновления увеличивается. На данный момент поставляется версия Java SE 7u1.

- ✓ **Существуют два типа загрузок Java.**

  На сайте Oracle вы увидите в каждой версии и в каждом издании два пакета — JRE (Java Runtime Environment — среда выполнения Java) и JDK (Java Development Kit — набор инструментов разработки приложений Java). Пакет JRE необходим для выполнения существующих программ Java. Возможно, на вашем компьютере уже установлена некоторая версия JRE. Для разработки программ необходим пакет JDK.

  В пакете JDK есть все необходимое для разработки и выполнения программ Java. Таким образом, JDK содержит JRE. Вы должны загрузить и установить JDK. Пакет JRE вам не нужен.

- ✓ **Java может поставляться с другими инструментами.**

  Все они представлены на странице загрузки на сайте Oracle. Вы можете бесплатно загрузить интегрированную рабочую среду NetBeans, средства JavaFX, исходные коды Java, документацию и другие материалы.

Документация по Java SE очень полезна, потому что в ней можно найти справочник по стандартным библиотечным классам Java, без которого невозможно создавать профессиональные программы. Остальные материалы нужны только для решения специальных задач, например для создания сценариев JavaFX для веб-страниц.

Загрузите с сайта `http://oracle.com` инсталляционный файл Java SE JDK и установите его на своем компьютере. В Windows инсталляционный файл имеет расширение `.exe`. Запустите его на выполнение, и он установит JDK на ваш компьютер. В Linux инсталляционный файл имеет расширение `.rpm` или `.bin`. Установите его на компьютере с помощью менеджера пакетов Linux.

Не пропустите расширение файла

Предположим, вы загрузили файл `jdk-7u1-windows-x64.exe`. Это инсталляционный файл Java JDK 7u1 для операционной системы Windows на 64-разрядном компьютере. Однако проводник Windows может скрывать зарегистрированные расширения файлов. Вы не увидите расширение `.exe` и не поймете, что это за файл: выполняемый файл, который нужно запустить, или файл какого-либо другого типа. Поэтому рекомендуется отменить сокрытие расширений. Для этого откройте панель управления Windows, откройте диалоговое окно **Свойства папки** и снимите флажок **Скрывать расширения зарегистрированных типов файлов**. Учитывайте, что в разных версиях Windows (XP, Vista или 7) названия окон и флажка могут быть немного разными.

Java для Macintosh

В первую очередь нужно выяснить, установлен ли JDK на компьютере. Это можно сделать одним из двух способов.

- ✓ **На основе номера версии операционной системы.**

  Версии Macintosh OS X 10.5 (кодовое имя — "Leopard") и OS X 10.6 ("Snow Leopard") поставляются с установленными JDK. Если на вашем компьютере установлена любая из этих версий (возможно, с дополнительными точками, например 10.6.4), значит, у вас все готово для разработки приложений Java.

  Версия Macintosh OS X 10.7 ("Lion") поставляется без JDK. Фактически ни одна версия JDK не одобрена компанией Apple как пригодная для OS X 10.7. Поэтому, если вы купили версию OS X 10.7 как самую последнюю и самую лучшую, вы получили самый последний продукт, но не самый лучший для работы с Java. Однако не отчаивайтесь и продолжайте читать данное приложение.

- ✓ **Выполните на компьютере тесты, позволяющие выяснить наличие JDK и номер его версии.**

  Выполнить указанные тесты можно несколькими способами. Далее приведены два наиболее популярных способа.

Утилита Java Preferences

С ее помощью можно выяснить наличие пакета JDK и его версию. Для этого выполните следующие операции.

1. **На панели инструментов выберите команду Finder.**
   Активизируется окно Finder.
2. **На боковой панели окна Finder выберите пункт Applications (Приложения).**
   На главной панели окна Finder появится список приложений.
3. **Дважды щелкните на пункте Utilities (Утилиты).**
   На главной панели появится список утилит.
4. **В списке утилит найдите пункт Java Preferences (Предпочтения Java).**
   Если его нет, не спешите огорчаться. Это не обязательно означает, что JDK не установлен. В данном случае перейдите к тесту, описанному в следующем разделе.
5. **Дважды щелкните на пункте Java Preferences.**
   После довольно длительной задержки компьютер выведет окно с информацией о JDK. Мой компьютер отобразил версию Java SE 6 и загадочный номер 1.6.0_22-b04-307. Вполне подойдет.



Для работы с примерами книги необходима версия SE 5 (иногда она пишется как 1.5.0) или выше.

Если утилита Java Preferences не решила проблему, воспользуйтесь встроенным в Mac командным окном (терминалом) Unix, как описано в следующем разделе.

Выяснение версии JDK с помощью командного терминала

1. **В поле поиска Spotlight введите ключевое слово `Terminal`.**
2. **Когда пункт Terminal появится в списке, выделите его и нажмите клавишу <Enter>.**
   Откроется окно Terminal. Обычно в нем приведен простой черный текст на белом фоне.
3. **Введите приведенную ниже команду и нажмите клавишу <Enter>.**

   ```
   javac -version
   ```

   На своем компьютере в окне Terminal я получил такой ответ.

   ```
   javac 1.6.0_22
   ```

   Если на вашем компьютере вы увидите номер `1.5.0` или выше, значит, установлена версия JDK, пригодная для работы с примерами книги.



В Macintosh окно Terminal по-разному реагирует на регистр букв. Например, чтобы увидеть список пользователей, в командной строке терминала можно ввести `w` или `W`. Чтобы увидеть справочную страницу с описанием команды `w`, можно ввести `man w`, `MAN w` или даже `mAn w`. Но если ввести `man W`, терминал вернет сообщение о том, что информация о команде `W` не найдена. Аналогично в команде `javac -version` все буквы слова `version` должны быть в нижнем регистре. Если ввести `javac -Version`, терминал вернет сообщение о том, что флажка `Version` не существует.

Если терминал сообщит что-то вроде "команда не найдена", значит, возможно, на вашем компьютере выполняется версия Mac OS X 10.7.0 или более поздняя. Если терминал возвращает номер версии ниже `1.5.0`, значит, установлена слишком старая, непригодная версия JDK.

Что делать, если нужная версия JDK не установлена

Как обычно, есть несколько способов решения этой проблемы.

- ✓ В меню Apple выберите команду Software Update (Обновление программного обеспечения). В открывшемся окне задайте обновление версии JDK.
- ✓ На сайте `http://developer.apple.com` введите в поле поиска ключевое слово `Java`. Найдите версию JDK, пригодную для вашей версии Mac OS X. Возможно, к моменту, когда вы читаете эти строки, ребята из Apple снизошли до нас и добавили JDK в OS X.
- ✓ Поищите Java и Mac OS X на других сайтах. Если вам повезет, вы найдете пакет JDK сторонних производителей, подходящий для вашей операционной системы. Если вам не очень повезет, вы найдете подходящий JDK, но который тяжело установить и с которым сложно работать. При его установке будьте осторожны. Я нашел подходящие пакеты по указанному ниже адресу, но не устанавливал и не проверял их, поэтому ручаться за них не могу.

  `http://openjdk.java.net/projects/bsd-port`

Загрузка и установка Eclipse

IDE (Integrated Development Environment — интегрированная среда разработки) — это программа, которая предоставляет удобные инструменты для создания программного обеспечения. Разрабатывать программы (в том числе программы на Java) можно и без интегрированной среды, но тогда вы потратите намного больше усилий и времени, чем при ее использовании (некоторые опытные разработчики не согласны с этим, но то, в чем они правы, относится только к очень опытным разработчикам).

Согласно официальному сайту Eclipse Foundation программа Eclipse — это "универсальный инструмент разработки, бесплатная расширяемая интегрированная среда создания программного обеспечения на любой платформе вообще и ни на какой в частности". И действительно, программа Eclipse очень гибкая и разнообразная. Многие разработчики считают, что Eclipse особенно удобна для разработки программ на Java, но она не менее удобна и для работы на C++, PHP и на многих других языках. Существуют даже варианты Eclipse, вообще не имеющие ничего общего с разработкой программного обеспечения, например веб-браузер Lively, вкладки которого взяты из Eclipse.

Загрузка Eclipse

Чтобы загрузить Eclipse из Интернета, выполните следующие операции.

1. **Откройте сайт `http://eclipse.org`.**
2. **Найдите вариант Eclipse, предназначенный для вашей операционной системы.**

На момент, когда я пишу эти строки, на странице `http://eclipse.org` нарисован большой прямоугольник с надписью `Загрузите Eclipse сейчас. Загрузить?` Однако что там будет написано завтра, неизвестно.

После щелчка на гиперссылке `Загрузить` появляется раскрывающийся список с тремя наиболее распространенными операционными системами: Windows, Linux и Mac OS X. Выберите операционную систему, установленную на вашем компьютере.

3. **Выберите подходящий пакет Eclipse.**

   На данный момент сайт предлагает пакеты Eclipse IDE for Java Developers, Eclipse IDE for Java EE Developers, Eclipse Classic, Eclipse IDE for C/C++ Developers и др. Для вас лучше всего подойдет Eclipse IDE for Java Developers.

4. **Выберите 32- или 64-разрядную версию.**

   Для большинства пакетов Eclipse возможен выбор между 32- и 64-разрядной версиями. Если на компьютере установлена 64-разрядная операционная система, можете выбрать 32- или 64-разрядную Eclipse, но лучше 64-разрядную. Если же установлена 32-разрядная операционная система, можно выбрать только 32-разрядную Eclipse.



Разрядности JDK и Eclipse должны быть одинаковыми — обе 32-разрядные или обе 64-разрядные. Смешивать их нельзя.

5. **Щелкните на гиперссылке `Download` (Загрузить).**

   На странице может быть предложено несколько зеркальных сайтов загрузки. Выберите ближайший к вашему городу.

Установка Eclipse

Процесс установки Eclipse зависит от операционной системы и типа файла, который вы загрузили.

- ✓ **Вы работаете в Windows и загрузили саморазворачивающийся архив `.exe`.**

  Дважды щелкните в проводнике Windows на файле `.exe` и разверните файл в любую папку. Рекомендуется скопировать содержимое в папку `Program Files`, в которой находятся остальные программы.

- ✓ **Вы работаете в Windows и загрузили файл архива `.zip`.**

  Извлеките содержимое архива в любую папку.

  Для этого откройте проводник Windows, найдите загруженный файл `.zip` и дважды щелкните на нем. Архив откроется в программе архивации. Перетащите папку `eclipse` в любое место файловой системы. Как и в предыдущем случае, рекомендуется перетащить ее в папку `Program Files`.

Инсталлятор и системный реестр Windows при установке Eclipse не используются. Развернув содержимое архива в любую папку на жестком диске, можете сразу запустить программу Eclipse. Для этого дважды щелкните на файле `eclipse.exe`.

- ✓ **Вы работаете в Mac OS X.**

  Если вы загрузили файл `.tar.gz`, найдите его в папке `Downloads` и дважды щелкните на нем. Двойной щелчок приведет к извлечению содержимого архива. После этого в папке `Downloads` должна появится папка `Eclipse`. Перетащите ее в папку `Applications`.

  Если вы загрузили файл `.dmg`, ваш браузер должен открыть его. Если же не открыл, найдите файл `.dmg` в папке `Downloads` и дважды щелкните на нем. Выполняйте инструкции, выводимые на экран. После этого перетащите папку `Eclipse` в папку `Applications`.

- ✓ **Вы работаете в Linux.**

  Вы загрузили либо архив `.tar.gz`, либо саморазворачивающийся архив `.bin`. Извлеките содержимое архива или дважды щелкните на саморазворачивающемся архиве.

Конфигурирование Eclipse

После установки Eclipse нужно сконфигурировать рабочую среду. Для этого выполните ряд операций.

1. **Запустите программу Eclipse.**

   При первом запуске на экране появится диалоговое окно Workspace Launcher (Установка рабочего пространства). В этом окне задается вопрос, в каком месте файловой системы вы хотите сохранять коды, создаваемые с помощью Eclipse.

2. **В диалоговом окне Workspace Launcher щелкните на кнопке OK, чтобы подтвердить маршрут, предложенный по умолчанию.**

   Можете не щелкать на кнопке OK и задать другой маршрут. Это не играет особой роли, потому что потом его можно будет изменить.

   При первом запуске Eclipse в конкретном рабочем пространстве появится диалоговое окно Welcome (Заставка). В нем есть несколько значков и строк текста.

3. **При наведении указателя на значки появляются подсказки с их именами. Найдите значок Workbench (Рабочее пространство).**

4. **Щелкните на значке Workbench, чтобы открыть главный экран Eclipse.**

   Будет активизирован главный экран Eclipse с заданным рабочим пространством. На панели Package Explorer (Проводник проектов) откройте папку с примерами данной книги (рис. A.1).

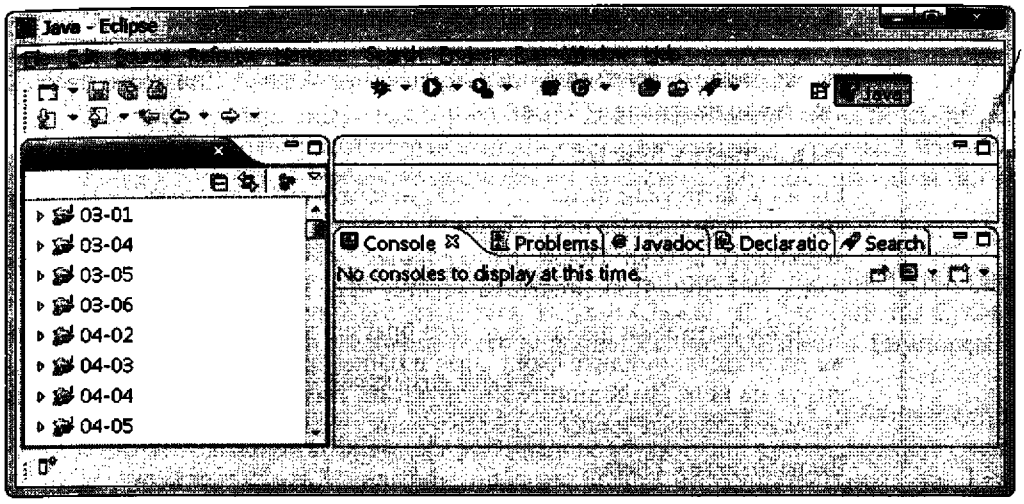


Рис. А.1. Интегрированная рабочая среда Eclipse

Теперь все готово для экспериментов с готовыми примерами или создания собственного приложения Java.

Использование Eclipse

В этом приложении...

- Работа с примерами книги
- Создание собственного проекта

Работа с примерами книги

1. **Запустите программу Eclipse.**
2. **Выберите в главном меню команду File⇨Switch Workspace⇨Other (Файл⇨Переключить рабочее пространство⇨Другое).**
   Активизируется диалоговое окно Workspace Launcher (Установка рабочего пространства).
3. **Введите маршрут нового рабочего пространства (рис. Б.1).**
   Можете ввести любой маршрут.

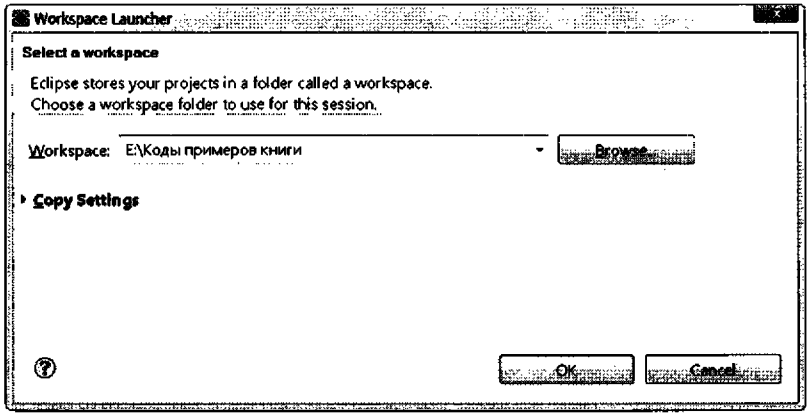


Рис. Б.1. Задание маршрута рабочего пространства

4. **Щелкните на кнопке OK.**
   На диске `E:\` будет создана папка `Коды примеров книги`. Она будет служить рабочим пространством Eclipse при работе с примерами книги.
5. **Загрузите из Интернета и скопируйте в папку рабочего пространства все примеры книги.**
6. **Выберите команду File⇨Import⇨General⇨Existing Projects into Workspace (Файл⇨Импорт⇨Общие⇨Существующие проекты в рабочее пространство). Щелкните на кнопке Next (Далее).**
7. **С помощью кнопки Browse (Обзор) выберите папку `E:\Коды примеров книги`.**

8. **Щелкните на кнопках Select All (Выделить все) и Finish (Готово).**
   На панели Project Explorer (Проводник проектов) появится список проектов, содержащих примеры книги (рис. Б.2). Первое число означает номер главы, а второе — номер листинга.

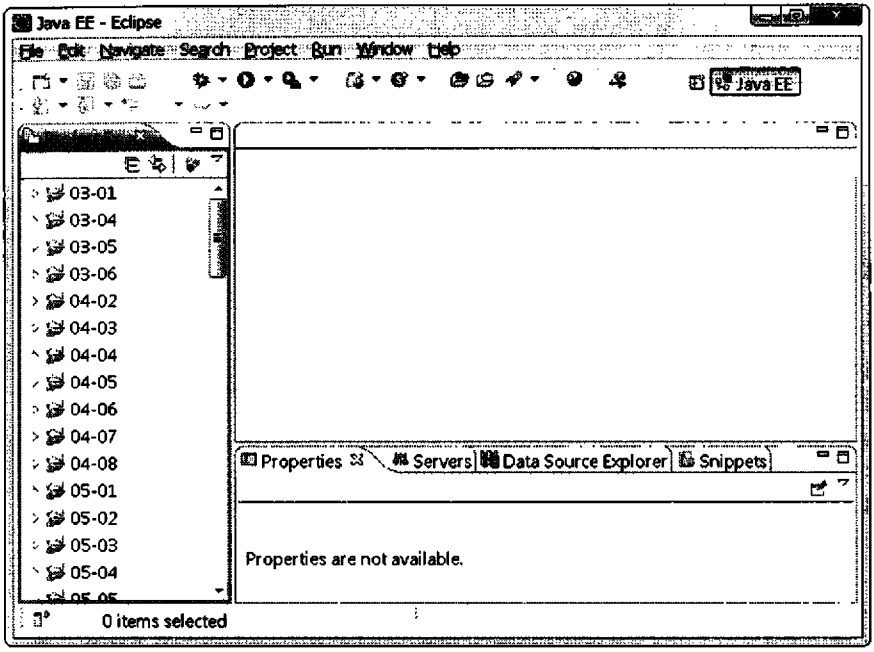


Рис. Б.2. Список проектов с примерами

9. **На панели Project Explorer разверните узел `03-01/src/default package` и дважды щелкните на исходном файле `Displayer.java`. Нажмите клавишу <F5>, чтобы загрузить проект (если Eclipse попросит подтвердить загрузку).**

10. **Щелкните на кнопке Run (Выполнить), чтобы запустить программу на выполнение.**
    При первом запуске проекта активизируется диалоговое окно Save and Launch (Сохранение и запуск). Если оно появилось, щелкните на кнопке OK, чтобы подтвердить предложенные по умолчанию параметры сохранения и запуска.
    На панели Console (Консоль) будет отображен результат работы программы (рис. Б.3).

Чтобы поэкспериментировать с примером, отредактируйте исходный код и еще раз щелкните на кнопке Run. Чтобы загрузить другой проект, закройте все окна исходных кодов текущего проекта, разверните узел другого проекта на панели Project Explorer, откройте на панели редактора исходные коды другого проекта (они находятся в папке `src/default package`), нажмите клавишу <F5>, отредактируйте исходные коды и щелкните на кнопке Run.

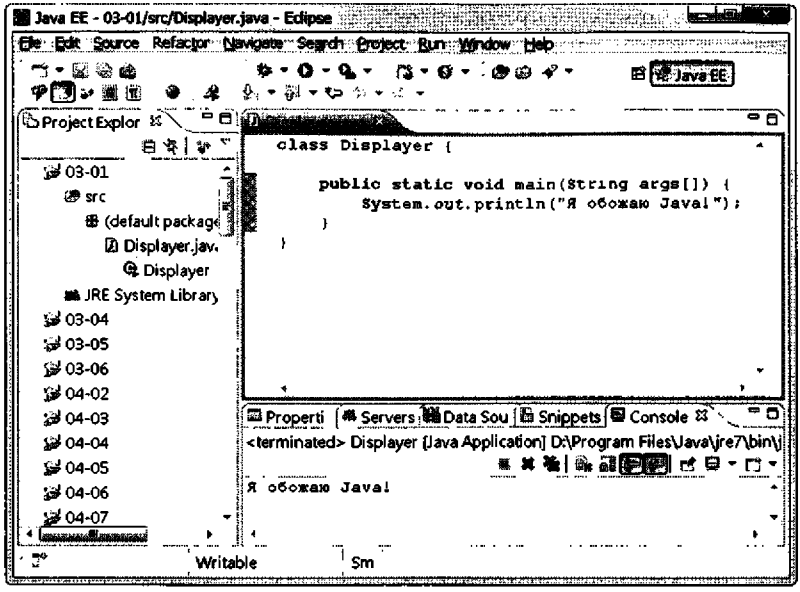


Рис. Б.3. На консоли отображен результат работы программы

Создание собственного проекта

1. **Запустите Eclipse.**
2. **Закройте окна исходных кодов текущего проекта.**
3. **Выберите в главном меню команду File⇨New⇨Project (Файл⇨Создать⇨Проект).**
4. **В окне New Project выделите пункт Java Project и щелкните на кнопке Next (Далее).**
5. **В окне New Java Project введите имя проекта (например, `Proj_1`) и щелкните на кнопке Finish (рис. Б.4).**

   На панели Project Explorer появился пустой проект `Proj_1`. Теперь нужно создать в папке исходных кодов `src` пакет, а в пакете — класс, содержащий метод `main()`.
6. **Щелкните правой кнопкой мыши на папке `src` и выберите в контекстном меню команду New⇨Package (Создать⇨Пакет).**
7. **В диалоговом окне New Java Package (Новый пакет Java) введите имя нового пакета, например `pac1`, и щелкните на кнопке Finish.**

   На левой панели в узле `src` появится вложенный узел `pac1`.
8. **Щелкните правой кнопкой мыши на узле `pac1` и выберите в контекстном меню команду New⇨Class (Создать⇨Класс).**

   Активизируется диалоговое окно New Java Class (рис. Б.5).
9. **В поле Name (Имя) введите имя класса, например `Class_1`.**
10. **Установите флажок public static void main, потому что проект должен содержать хотя бы один класс, в котором определен метод `main()`.**
11. **Щелкните на кнопке Finish.**

В пакете `pac1` появится пустой класс `Class_1` (рис. Б.6).

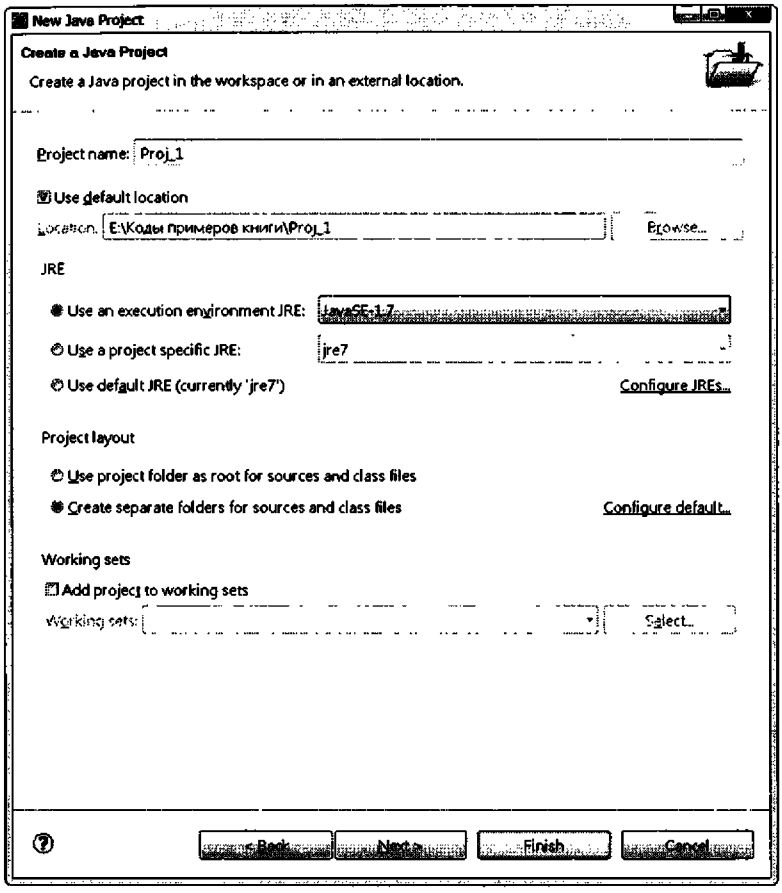


Рис. Б.4. Задание параметров нового проекта

12. Введите в тело метода `main()` следующую инструкцию, которая отображает на консоли строку `Это класс Class_1`, и щелкните на кнопке Run (Выполнить).

```
System.out.println("Это класс Class_1");
```

На консоли появится результат работы программы (рис. Б.7).

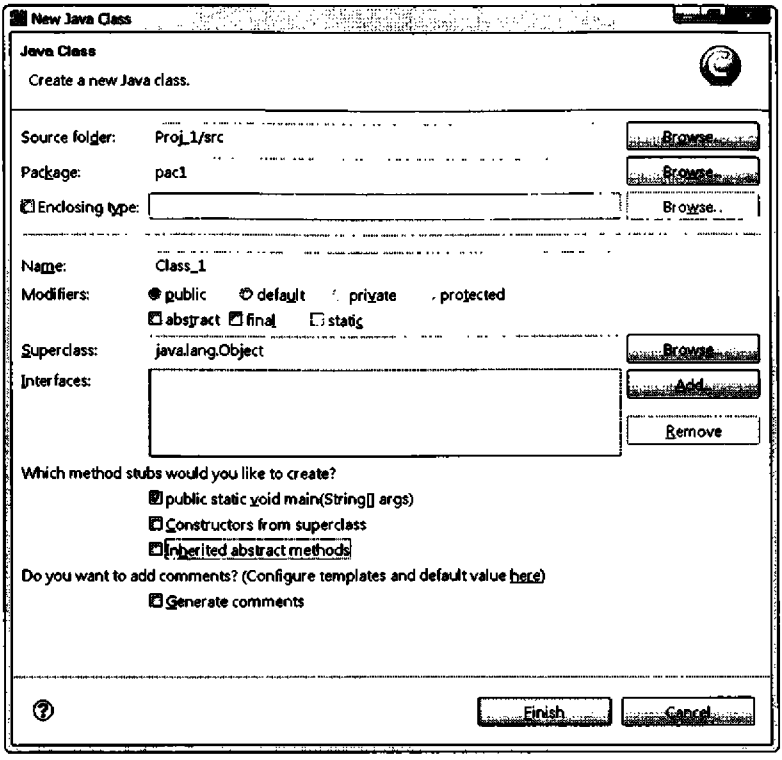


Рис. Б.5. Создание класса

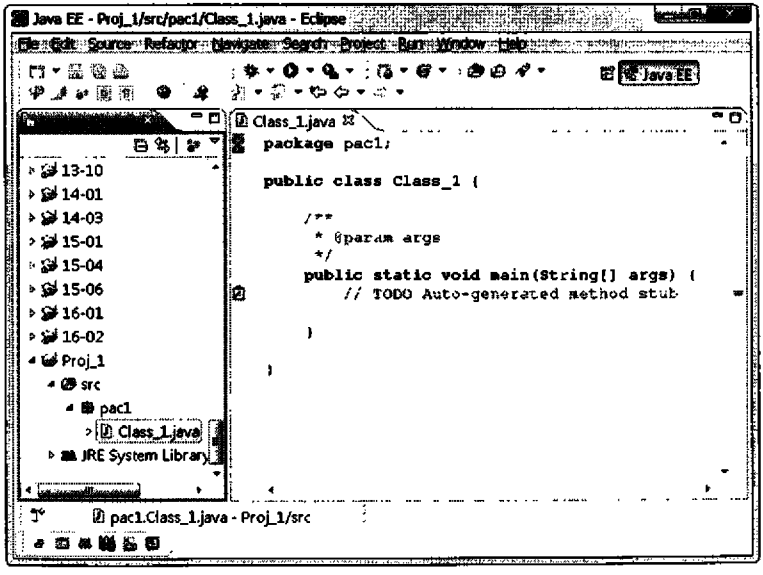


Рис. Б.6. Новый класс, пока что пустой

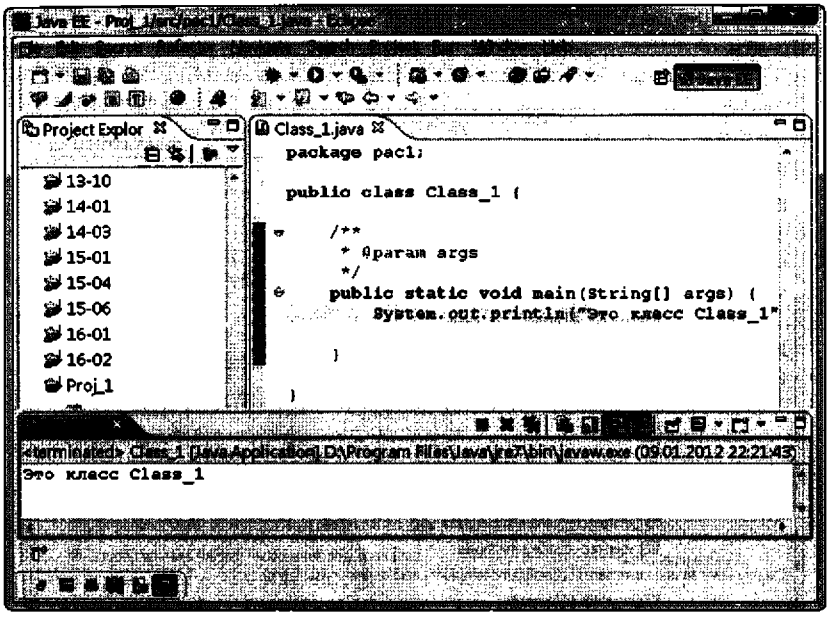


Рис. Б.7. Результат работы нового проекта

Предметный указатель

A

API, 50
Applet Viewer, 316

C

CLASSPATH, 286

E

Eclipse, 46

J

Java DB, 325
Javadoc, 64
JCP, 51
JDBC, 325
JDK, 44
JRE, 44
JUnit, 147

O

OpenJDK, 51

S

SDK, 44
SQL, 328

U

UTF-16, 82

А

Абстрактный метод, 303
Анимация, 318
Аннотация, 179; 198
Анонимный внутренний класс, 312
Аплет, 315
  анимация, 318
Аргумент командной строки, 241

Б

База данных, 325
Базовый класс, 31; 171
Байт, 75
Байтовый код, 39
Бит, 75
Блок, 100; 130
Булев тип, 82

В

Ветвление, 95
Виртуальная машина, 37; 40
Внутренний класс, 297; 311
Возвращаемое значение, 142
Вызов метода, 55
Выражение, 83
Высокоуровневый язык, 26

Г

Главный метод, 57
Грамматика, 50
Графический буфер, 285
Группирующий разделитель, 161

Д

Двоичный код, 40
Декремент, 89
Драйвер базы данных, 328

З

Заголовок метода, 141
Закрытие файла, 276
Закрытое поле, 154
Защищенный доступ, 292
Значение, 72

И

Идентификатор, 51; 73
  версии, 305
Импорт, 86
Имя переменной, 73
Индекс массива, 226
Инициализация, 79; 138
  цикла, 124
Инкремент, 89
Инструкция, 74
  if, 95; 98; 111
Интегрированная среда разработки, 37; 46
Интерфейс, 302
Исключение, 207; 254
  непроверяемое, 271
  перехват, 261
  проверяемое, 271
Исходный код, 38
Итерация, 122

К

Класс, 30; 33; 54; 133
  ArrayList, 247
  BigDecimal, 149
  DecimalFormat, 202
  DriverManager, 329
  Exception, 255
  FontMetrics, 317
  JApplet, 317
  JOptionPane, 108
  NumberFormat, 240
  PrintStream, 234
  Random, 98
  RuntimeException, 273
  Scanner, 97
  внутренний, 297
  открытый, 296
Ключевое слово, 51
  catch, 256
  enum, 182
  final, 189
  finally, 274
  private, 280
  public, 280
  return, 221
  static, 207
  super, 192
  switch, 113
  this, 217; 304
  throw, 256
  throws, 207
  try, 256
Код, 25
Коллекция, 245
Командная строка, 241
Комментарий, 62
Компилятор, 37
Конкатенация, 87
Консоль, 241
Константа, 190
Конструктор, 181
  неявный, 193
Конфликт имен, 52
Кроссплатформенность, 42
Курсор базы данных, 332

Л

Литерал, 73
Логический оператор, 106
Локальная переменная, 216

М

Массив, 226
Машинный код, 40
Метод, 55
  доступа, 153
  обратного вызова, 287
Многопоточность, 303
Модификатор доступа, 151; 154; 280

Н

Наследование, 31; 169
Неименованный пакет, 289
Неявный конструктор, 193

О

Обобщенный тип, 249
Обработчик события, 301
Объект, 30; 133
Объектно-ориентированное
    программирование, 28
Объявление, 76
Оператор, 74
  логический, 106
  присваивания, 93
  равенства, 100
  сравнения, 102
  условный, 241
Открытое поле, 154
Открытый класс, 296
Отступ, 60; 101

П

Пакет, 196
Передача
  по значению, 219
  по ссылке, 221
Переменная, 71
Перечисление, 182
Поиск в массиве, 233
Поле, 135
Пользовательский идентификатор, 51
Пользовательское исключение, 259
Постфиксный инкремент, 90
Поток, 303
Префиксный инкремент, 90
Приемник события, 302
Примитивный тип, 80
Присваивание, 74; 93
Программа, 25
Производный класс, 31; 171

Прослушивание события, 302
Псевдослучайное число, 98

Р

Рекурсивный метод, 57

С

Символьный тип, 81
Синтаксис, 50
Случайное число, 98
Событие, 299
Сравнение, 102
Ссылочный тип, 83
Стартовое число, 98
Статический инициализатор, 210
Строка форматирования, 149
Структура программы, 61
Счетчик цикла, 121

Т

Табулостоп, 230
Текстовый формат, 162
Тестовый модуль, 147
Тип, 73; 75
  boolean, 82
  char, 81

У

Управляющая последовательность, 230
Условный оператор
  тернарный, 241

Ф

Фабричный метод, 240
Фигурные скобки, 60
Фокус ввода, 305
Формат вывода, 148
Фрейм, 84; 196; 283
  создание, 286

Ц

Целочисленный тип, 78
Цикл
  do, 126
  for, 122; 231
  while, 119

Ч

Член класса, 141; 290

Э

Экземпляр класса, 31

...FOR Dummies™

BESTSELLING BOOK SERIES

Java для чайников®

5-е издание

...ДЛЯ чайников™

СЕРИЯ КОМПЬЮТЕРНЫХ КНИГ ОТ ДИАЛЕКТИКИ

Шпаргалка

Ключевые слова Java

Ключевое слово	Назначение
abstract	Модификатор, указывающий на то, что подробности класса, метода или интерфейса должны быть реализованы в другом месте кода
assert	Проверка истинности условия в процессе отладки программы
boolean	Примитивный булев тип, имеющий два значения: **true** и **false**
break	Выход из блока **switch** или из любого цикла
byte	Примитивный тип; целое число длиной 8 бит
case	Метка маршрута выполнения в блоке **switch**
catch	Блок, выполняющийся при исключении в блоке **try**
char	Примитивный тип переменной, содержащей символ в формате UTF-16
class	Определение класса
const	В Java нельзя использовать это ключевое слово; нельзя также объявить переменную с именем **const**; константу можно создать с помощью ключевого слова **final**
continue	Принудительное завершение текущей итерации цикла и выполнение следующей итерации сначала
default	Маршрут выполнения в блоке **switch**, выбираемый, когда управляющая переменная не равна ни одному варианту **case**
do	Бесконечный цикл; может быть завершен оператором **break** или с помощью условия **while**
double	Примитивный тип; число с плавающей точкой длиной 64 бит
else	Заголовок блока, расположенного после блока **if** и выполняющегося, если условие равно **false**
enum	Специальный тип; переменная, которая может принимать только ограниченный набор именованных значений
extends	Объявление базового класса в заголовке производного класса
false	ЛОЖЬ; одно из двух булевых значений
final	Модификатор, означающий, что значение переменной не может быть изменено, класс не может быть базовым или метод не может быть переопределен
finally	Блок, выполняющийся после блока **try** в любом случае, независимо от того, было ли сгенерировано исключение (однако, если завершается поток **try** или закрывается процесс виртуальной машины Java, блок **finally** может быть пропущен)
float	Примитивный тип; число с плавающей точкой длиной 32 бит
for	Цикл, определяемый тремя инструкциями — начальным значением счетчика, условием выполнения и приращением счетчика
goto	В Java использование этого ключевого слова запрещено; однако, поскольку это все же ключевое слово, его нельзя использовать в качестве идентификатора
if	Условие ветвления программы
implements	Объявление интерфейса в заголовке класса
import	Задание имен пакетов и классов, позволяющее сокращать имена в коде

...FOR Dummies ™

BESTSELLING BOOK SERIES

Java для чайников®

5-е издание

Шпаргалка

...для чайников ™

СЕРИЯ КОМПЬЮТЕРНЫХ КНИГ ОТ ДИАЛЕКТИКИ

Ключевые слова Java

Ключевое слово	Назначение
instanceof	Проверка, является ли объект экземпляром класса
int	Примитивный тип; целое число длиной 32 бит
interface	Определение интерфейса
long	Примитивный тип; целое число длиной 64 бит
native	Позволяет программисту использовать код, написанный на другом языке (на любом из языков, отличных от Java)
new	Создание экземпляра класса
null	Значение “пусто”; отсутствие значения; неопределенное значение
package	Группа логически связанных классов, обладающих общими именами
private	Модификатор доступа, ограничивающий область видимости рамками класса
protected	Модификатор доступа, ограничивающий область видимости рамками текущего класса и всех производных классов
public	Модификатор доступа, не ограничивающий область видимости
return	Завершение выполнения метода и, возможно, возврат значения
short	Примитивный тип; целое число длиной 16 бит
static	Модификатор, определяющий принадлежность поля или метода классу, а не экземпляру класса
strictfp	Ограничение представления слишком больших или малых чисел типа **float** и **double** при выполнении промежуточных вычислений
super	Ссылка на базовый класс; вызов конструктора базового класса
switch	Выбор маршрута выполнения кода в зависимости от значения управляющей переменной
synchronized	Синхронизация двух потоков
this	Ссылка на текущий экземпляр класса
throw	Генерация исключения
throws	Указывает на то, что метод может передавать необработанные исключения выше по стеку вызовов
transient	Указывает на то, что при сериализации объекта значение переменной можно не сохранять
true	ИСТИНА; одно из двух булевых значений
try	Защищенный блок инструкций; защищенность означает, что возникшее исключение будет обработано
void	Спецификатор метода, сообщающий о том, что метод не возвращает никакого значения
volatile	Определение правил использования переменной в нескольких потоках одновременно
while	Одна из разновидностей циклов; повторение выполнения блока инструкций при истинности условия

Бестселлер для начинающих теперь полностью обновлен для Java 7!

Java — современный объектно-ориентированный язык программирования. Программа, написанная на Java, способна выполняться практически на любом компьютере. Зная Java, можно создавать мощные мультимедийные приложения для любой платформы. Десятки тысяч программистов начинали изучать Java с помощью предыдущих изданий этой книги. Теперь ваша очередь! Независимо от того, на каком языке вы программировали раньше (и даже если вы никогда прежде не программировали), вы быстро научитесь создавать современные кроссплатформенные приложения.

- ***Мир Java.*** Познакомьтесь с платформой Java, новыми средствами версии Java 7 и современными инструментами разработки.
- ***Строительные блоки.*** Научитесь создавать мощные программы на основе классов — хорошо изолированных блоков кода.
- ***Не повторяйтесь, но повторяйте.*** Научитесь использовать существующие фрагменты кодов в новых программах.
- ***Не зацикливайтесь, но зацикливайте.*** Научитесь использовать циклы и условные конструкции для ветвления программ.
- ***Будьте классным и стильным.*** Освойте правильный стиль кодирования с помощью классов и принципов инкапсуляции, наследования и полиморфизма.
- ***Экспериментируйте.*** Применяйте переменные и методы, массивы и коллекции. Программируйте обработку событий. Создавайте обработчики исключений.

Коды всех примеров можно скачать с веб-страницы книги:
http://www.dialektika.com/books/978-5-8459-1769-0.html

Об авторе

Барри Берд — доктор философии, профессор кафедры математики и информатики в Университете Дрю, автор многочисленных книг и статей по сетевым технологиям и программированию.



Основные темы книги:

- *ключевые концепции Java*
- *грамматика языка*
- *повторное использование кода*
- *циклы и условные конструкции*
- *принципы объектно-ориентированного программирования*
- *обработка исключений*
- *создание аплетов Java*
- *как избежать распространенных ошибок*

ДИАЛЕКТИКА

Посетите "Диалектику" в Интернете
по адресу: http://www.dialektika.com



ISBN 978-5-8459-1769-0

