

JAVA[®]

БИБЛИОТЕКА ПРОФЕССИОНАЛА

Том 1. Основы

ОДИННАДЦАТОЕ ИЗДАНИЕ



КЕЙ ХОРСТМАНН

Java[®]



Библиотека профессионала

Том 1. Основы



Core Java[®]

Volume I – Fundamentals

Eleventh Edition

Cay S. Horstmann



Pearson

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town

Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City

São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo



Java[®]

Библиотека профессионала

Том 1. Основы

Одиннадцатое издание

Кей Хорстманн



Москва • Санкт-Петербург
2019

ББК 32.973.26-018.2.75

X82

УДК 681.3.07

ООО "Диалектика"

Зав. редакцией С.Н. Тригуб

Перевод с английского и редакция И.В. Берштейна

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

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

Хорстманн, Кей С.

X82 Java. Библиотека профессионала, том 1. Основы. 11-е изд. : Пер. с англ. — СПб. : ООО "Диалектика", 2019. — 864 с. : ил. — Парал. тит. англ.

ISBN 978-5-907114-79-1 (рус., том 1)

ISBN 978-5-907144-30-9 (рус., многотом)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

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

Copyright © 2019 by Dialektika Computer Publishing Ltd.

Authorized Russian translation of the English edition of *Core Java Volume I: Fundamentals*, 11th Edition (ISBN 978-0-13-516630-7) © 2019 Pearson Education Inc.

Portions copyright © 1996-2013 Oracle and/or its affiliates. All Rights Reserved.

This translation is published and sold by permission of Pearson Education Inc., which owns or controls all rights to publish and sell the same.

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

Научно-популярное издание

Кей С. Хорстманн

Java. Библиотека профессионала, том 1 Основы. 11-е издание

Подписано в печать 05.03.2019. Формат 70х100/16

Гарнитура Times

Усл. печ. л. 69,66. Уч.-изд. л. 54,1

Тираж 500 экз. Заказ № 2231

Отпечатано в АО "Первая Образцовая типография"

Филиал "Чеховский Печатный Двор"

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

Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-907114-79-1 (рус., том 1)

ISBN 978-5-907144-30-9 (рус., многотом)

ISBN 978-0-13-516630-7 (англ.)

© 2019 ООО "Диалектика"

© 2019 Pearson Education Inc.

Оглавление

Предисловие	15
Глава 1. Введение в язык Java	21
Глава 2. Среда программирования на Java	35
Глава 3. Основные языковые конструкции Java	51
Глава 4. Объекты и классы	125
Глава 5. Наследование	195
Глава 6. Интерфейсы, лямбда-выражения и внутренние классы	273
Глава 7. Исключения, утверждения и протоколирование	339
Глава 8. Обобщенное программирование	393
Глава 9. Коллекции	437
Глава 10. Программирование графики	509
Глава 11. Компоненты пользовательского интерфейса в Swing	567
Глава 12. Параллелизм	661
Глава 13. Библиотека JavaFX	759
Приложение А. Ключевые слова Java	849
Предметный указатель	851

Содержание

Предисловие	15
Глава 1. Введение в язык Java	21
1.1. Программная платформа Java	21
1.2. Характерные особенности Java	22
1.2.1. Простота	23
1.2.2. Объектно-ориентированный характер	23
1.2.3. Поддержка распределенных вычислений в сети	24
1.2.4. Надежность	24
1.2.5. Безопасность	24
1.2.6. Независимость от архитектуры компьютера	25
1.2.7. Переносимость	25
1.2.8. Интерпретируемость	26
1.2.9. Производительность	26
1.2.10. Многопоточность	27
1.2.11. Динамичность	27
1.3. Апплеты и Интернет	28
1.4. Краткая история развития Java	29
1.5. Распространенные заблуждения относительно Java	32
Глава 2. Среда программирования на Java	35
2.1. Установка комплекта Java Development Kit	35
2.1.1. Загрузка комплекта JDK	36
2.1.2. Установка комплекта JDK	37
2.1.3. Установка библиотек и документации	39
2.2. Применение инструментов командной строки	40
2.3. Применение IDE	45
2.4. Утилита JShell	48
Глава 3. Основные языковые конструкции Java	51
3.1. Простая программа на Java	51
3.2. Комментарии	55
3.3. Типы данных	55
3.3.1. Целочисленные типы данных	56
3.3.2. Числовые типы данных с плавающей точкой	57
3.3.3. Тип данных <code>char</code>	58
3.3.4. Юникод и тип <code>char</code>	59
3.3.5. Тип данных <code>boolean</code>	60
3.4. Переменные и константы	61
3.4.1. Объявление переменных	61
3.4.2. Инициализация переменных	62
3.4.3. Константы	63
3.4.4. Перечислимые типы	64
3.5. Операции	64
3.5.1. Арифметические операции	64
3.5.2. Математические функции и константы	65

3.5.3. Преобразование числовых типов	67
3.5.4. Приведение типов	68
3.5.5. Сочетание арифметических операций с присваиванием	69
3.5.6. Операции инкремента и декремента	69
3.5.7. Операции отношения и логические операции	70
3.5.8. Поразрядные логические операции	70
3.5.9. Круглые скобки и иерархия операций	71
3.6. Символьные строки	72
3.6.1. Подстроки	72
3.6.2. Сцепление строк	73
3.6.3. Принцип постоянства символьных строк	73
3.6.4. Проверка символьных строк на равенство	75
3.6.5. Пустые и нулевые строки	76
3.6.6. Кодовые точки и единицы	76
3.6.7. Прикладной программный интерфейс API класса String	77
3.6.8. Оперативно доступная документация на API	80
3.6.9. Построение символьных строк	82
3.7. Ввод и вывод	84
3.7.1. Чтение вводимых данных	84
3.7.2. Форматирование выводимых данных	86
3.7.3. Файловый ввод и вывод	91
3.8. Управляющая логика	93
3.8.1. Область видимости блоков	93
3.8.2. Условные операторы	93
3.8.3. Неопределенные циклы	97
3.8.4. Определенные циклы	100
3.8.5. Оператор switch для многовариантного выбора	104
3.8.6. Операторы прерывания логики управления программой	106
3.9. Большие числа	108
3.10. Массивы	111
3.10.1. Объявление массивов	112
3.10.2. Доступ к элементам массива	113
3.10.3. Цикл в стиле for each	114
3.10.4. Копирование массивов	115
3.10.5. Параметры командной строки	116
3.10.6. Сортировка массивов	116
3.10.7. Многомерные массивы	119
3.10.8. Неровные массивы	122
Глава 4. Объекты и классы	125
4.1. Введение в ООП	126
4.1.1. Классы	126
4.1.2. Объекты	128
4.1.3. Идентификация классов	128
4.1.4. Отношения между классами	129
4.2. Применение predefined классов	130
4.2.1. Объекты и объектные переменные	131
4.2.2. Класс LocalDate из библиотеки Java	134
4.2.3. Модифицирующие методы и методы доступа	135
4.3. Определение собственных классов	139
4.3.1. Класс Employee	139
4.3.2. Использование нескольких исходных файлов	142
4.3.3. Анализ класса Employee	142

4.3.4. Первые действия с конструкторами	143
4.3.5. Объявление локальных переменных с помощью ключевого слова var	144
4.3.6. Обработка пустых ссылок на объекты	145
4.3.7. Явные и неявные параметры	146
4.3.8. Преимущества инкапсуляции	147
4.3.9. Привилегии доступа к данным в классе	149
4.3.10. Закрытые методы	150
4.3.11. Конечные поля экземпляра	150
4.4. Статические поля и методы	151
4.4.1. Статические поля	151
4.4.2. Статические константы	152
4.4.3. Статические методы	153
4.4.4. Фабричные методы	154
4.4.5. Метод main()	154
4.5. Параметры методов	157
4.6. Конструирование объектов	163
4.6.1. Перегрузка	163
4.6.2. Инициализация полей по умолчанию	164
4.6.3. Конструктор без аргументов	164
4.6.4. Явная инициализация полей	165
4.6.5. Имена параметров	166
4.6.6. Вызов одного конструктора из другого	167
4.6.7. Блоки инициализации	167
4.6.8. Уничтожение объектов и метод finalize()	171
4.7. Пакеты	172
4.7.1. Именованые пакетов	172
4.7.2. Импорт классов	172
4.7.3. Статический импорт	174
4.7.4. Ввод классов в пакеты	174
4.7.5. Область видимости пакетов	177
4.7.6. Путь к классам	179
4.7.7. Указание пути к классам	181
4.8. Архивные JAR-файлы	182
4.8.1. Создание JAR-файлов	182
4.8.2. Файл манифеста	183
4.8.3. Исполняемые JAR-файлы	184
4.8.4. Многоверсионные архивные JAR-файлы	184
4.8.5. Примечание к параметрам командной строки	186
4.9. Документирующие комментарии	187
4.9.1. Вставка комментариев	188
4.9.2. Комментарии к классам	188
4.9.3. Комментарии к методам	189
4.9.4. Комментарии к полям	189
4.9.5. Комментарии общего характера	190
4.9.6. Комментарии к пакетам	191
4.9.7. Извлечение комментариев	191
4.10. Рекомендации по разработке классов	192
Глава 5. Наследование	195
5.1. Классы, суперклассы и подклассы	196
5.1.1. Определение подклассов	196
5.1.2. Переопределение методов	197
5.1.3. Конструкторы подклассов	199

5.1.4. Иерархии наследования	203
5.1.5. Полиморфизм	203
5.1.6. Представление о вызовах методов	205
5.1.7. Предотвращение наследования: конечные классы и методы	207
5.1.8. Приведение типов	209
5.1.9. Абстрактные классы	211
5.1.10. Защищенный доступ	216
5.2. Глобальный суперкласс Object	217
5.2.1. Переменные типа Object	218
5.2.2. Метод equals()	218
5.2.3. Проверка объектов на равенство и наследование	219
5.2.4. Метод hashCode()	223
5.2.5. Метод toString()	225
5.3. Обобщенные списочные массивы	231
5.3.1. Объявление списочных массивов	232
5.3.2. Доступ к элементам списочных массивов	234
5.3.3. Совместимость типизированных и базовых списочных массивов	237
5.4. Объектные оболочки и автоупаковка	238
5.5. Методы с переменным числом параметров	242
5.6. Классы перечислений	243
5.7. Рефлексия	245
5.7.1. Класс Class	246
5.7.2. Основы обработки исключений	248
5.7.3. Ресурсы	249
5.7.4. Анализ функциональных возможностей классов с помощью рефлексии	251
5.7.5. Анализ объектов во время выполнения с помощью рефлексии	257
5.7.6. Написание кода универсального массива с помощью рефлексии	263
5.7.7. Вызов произвольных методов и конструкторов	266
5.8. Рекомендации по применению наследования	270
Глава 6. Интерфейсы, лямбда-выражения и внутренние классы	273
6.1. Интерфейсы	274
6.1.1. Понятие интерфейса	274
6.1.2. Свойства интерфейсов	280
6.1.3. Интерфейсы и абстрактные классы	281
6.1.4. Статические и закрытые методы	282
6.1.5. Методы с реализацией по умолчанию	283
6.1.6. Разрешение конфликтов с методами по умолчанию	284
6.1.7. Интерфейсы и обратные вызовы	286
6.1.8. Интерфейс Comparator	289
6.1.9. Клонирование объектов	290
6.2. Лямбда-выражения	296
6.2.1. Причины для употребления лямбда-выражений	296
6.2.2. Синтаксис лямбда-выражений	298
6.2.3. Функциональные интерфейсы	300
6.2.4. Ссылки на методы	302
6.2.5. Ссылки на конструкторы	305
6.2.6. Область видимости переменных	306
6.2.7. Обработка лямбда-выражений	308
6.2.8. Еще о компараторах	311
6.3. Внутренние классы	312
6.3.1. Доступ к состоянию объекта с помощью внутреннего класса	313

6.3.2. Специальные синтаксические правила для внутренних классов	316
6.3.3. О пользе, необходимости и безопасности внутренних классов	317
6.3.4. Локальные внутренние классы	320
6.3.5. Доступ к конечным переменным из внешних методов	321
6.3.6. Анонимные внутренние классы	322
6.3.7. Статические внутренние классы	325
6.4. Загрузчики служб	329
6.5. Прокси-классы	331
6.5.1. О применении прокси-классов	332
6.5.2. Создание прокси-объектов	332
6.5.3. Свойства прокси-классов	336
Глава 7. Исключения, утверждения и протоколирование	339
7.1. Обработка ошибок	340
7.1.1. Классификация исключений	341
7.1.2. Объявление проверяемых исключений	343
7.1.3. Порядок генерирования исключений	345
7.1.4. Создание классов исключений	347
7.2. Перехват исключений	348
7.2.1. Перехват одного исключения	348
7.2.2. Перехват нескольких исключений	350
7.2.3. Повторное генерирование и связывание исключений в цепочку	351
7.2.4. Блок оператора finally	352
7.2.5. Оператор try с ресурсами	355
7.2.6. Анализ элементов трассировки стека	356
7.3. Рекомендации по обработке исключений	361
7.4. Применение утверждений	364
7.4.1. Понятие утверждения	364
7.4.2. Разрешение и запрет утверждений	365
7.4.3. Проверка параметров с помощью утверждений	366
7.4.4. Документирование предположений с помощью утверждений	367
7.5. Протоколирование	368
7.5.1. Элементарное протоколирование	369
7.5.2. Усовершенствованное протоколирование	369
7.5.3. Смена диспетчера протоколирования	371
7.5.4. Локализация	373
7.5.5. Обработчики протоколов	374
7.5.6. Фильтры	378
7.5.7. Средства форматирования	378
7.5.8. "Рецепт" протоколирования	378
7.6. Рекомендации по отладке программ	387
Глава 8. Обобщенное программирование	393
8.1. Назначение обобщенного программирования	394
8.1.1. Преимущества параметров типа	394
8.1.2. На кого рассчитано обобщенное программирование	395
8.2. Определение простого обобщенного класса	396
8.3. Обобщенные методы	398
8.4. Ограничения на переменные типа	399
8.5. Обобщенный код и виртуальная машина	402
8.5.1. Стирание типов	402
8.5.2. Преобразование обобщенных выражений	403

8.5.3. Преобразование обобщенных методов	404
8.5.4. Вызов унаследованного кода	406
8.6. Ограничения и пределы обобщений	407
8.6.1. Параметрам типа нельзя приписывать простые типы	407
8.6.2. Во время выполнения можно запрашивать только базовые типы	407
8.6.3. Массивы параметризованных типов недопустимы	408
8.6.4. Предупреждения о переменном числе аргументов	409
8.6.5. Нельзя создавать экземпляры переменных типа	410
8.6.6. Нельзя строить обобщенные массивы	410
8.6.7. Переменные типа в статическом контексте обобщенных классов недействительны	412
8.6.8. Нельзя генерировать или перехватывать экземпляры обобщенного класса в виде исключений	412
8.6.9. Преодоление ограничения на обработку проверяемых исключений	413
8.6.10. Остерегайтесь конфликтов после стирания типов	415
8.7. Правила наследования обобщенных типов	416
8.8. Подстановочные типы	417
8.8.1. Понятие подстановочного типа	418
8.8.2. Ограничения супертипа на подстановки	419
8.8.3. Неограниченные подстановки	422
8.8.4. Захват подстановок	423
8.9. Рефлексия и обобщения	425
8.9.1. Обобщенный класс Class	425
8.9.2. Сопоставление типов с помощью параметров Class<T>	427
8.9.3. Сведения об обобщенных типах в виртуальной машине	427
8.9.4. Литералы типов	431
Глава 9. Коллекции	437
9.1. Каркас коллекций в Java	437
9.1.1. Разделение интерфейсов и реализаций коллекций	438
9.1.2. Интерфейс Collection	440
9.1.3. Итераторы	441
9.1.4. Обобщенные служебные методы	443
9.2. Интерфейсы в каркасе коллекций Java	446
9.3. Конкретные коллекции	448
9.3.1. Связные списки	450
9.3.2. Списочные массивы	458
9.3.3. Хеш-множества	459
9.3.4. Древовидные множества	463
9.3.5. Одно- и двухсторонние очереди	467
9.3.6. Очереди по приоритету	468
9.4. Отображения	470
9.4.1. Основные операции над отображениями	470
9.4.2. Обновление записей в отображении	473
9.4.3. Представления отображений	474
9.4.4. Слабые хеш-отображения	476
9.4.5. Связные хеш-множества и отображения	477
9.4.6. Перечислимые множества и отображения	478
9.4.7. Хеш-отображения идентичности	479
9.5. Представления и оболочки	481
9.5.1. Мелкие коллекции	481
9.5.2. Поддиапазоны	482
9.5.3. Немодифицируемые представления	483

9.5.4. Синхронизированные представления	484
9.5.5. Проверяемые представления	485
9.5.6. О необязательных операциях	485
9.6. Алгоритмы	489
9.6.1. Назначение обобщенных алгоритмов	490
9.6.2. Сортировка и перетасовка	491
9.6.3. Двоичный поиск	493
9.6.4. Простые алгоритмы	495
9.6.5. Групповые операции	496
9.6.6. Взаимное преобразование коллекций и массивов	497
9.6.7. Написание собственных алгоритмов	498
9.7. Унаследованные коллекции	499
9.7.1. Класс Hashtable	500
9.7.2. Перечисления	500
9.7.3. Таблицы свойств	501
9.7.4. Стеки	504
9.7.5. Битовые множества	505
Глава 10. Программирование графики	509
10.1. История развития инструментальных средств для разработки GUI на Java	509
10.2. Отображение фреймов	511
10.2.1. Создание фрейма	511
10.2.2. Свойства фрейма	513
10.3. Отображение данных в компоненте	517
10.3.1. Двухмерные формы	521
10.3.2. Окрашивание цветом	528
10.3.3. Применение шрифтов	530
10.3.4. Воспроизведение изображений	536
10.4. Обработка событий	537
10.4.1. Общее представление об обработке событий	537
10.4.2. Пример обработки событий от щелчков на экранных кнопках	539
10.4.3. Краткое обозначение приемников событий	543
10.4.4. Классы адаптеров	544
10.4.5. Действия	546
10.4.6. События от мыши	551
10.4.7. Иерархия событий в библиотеке AWT	557
10.5. Прикладной интерфейс Preferences API	560
Глава 11. Компоненты пользовательского интерфейса в Swing	567
11.1. Библиотека Swing и проектный шаблон “модель–представление–контроллер”	568
11.2. Введение в компоновку пользовательского интерфейса	572
11.2.1. Диспетчеры компоновки	572
11.2.2. Граничная компоновка	574
11.2.3. Сеточная компоновка	576
11.3. Ввод текста	577
11.3.1. Текстовые поля	578
11.3.2. Метки и пометка компонентов	580
11.3.3. Поля для ввода пароля	581
11.3.4. Текстовые области	582
11.3.5. Панели прокрутки	583

11.4. Компоненты для выбора разных вариантов	585
11.4.1. Флажки	585
11.4.2. Кнопки-переключатели	588
11.4.3. Границы	592
11.4.4. Комбинированные списки	594
11.4.5. Регулируемые ползунки	598
11.5. Меню	604
11.5.1. Создание меню	605
11.5.2. Пиктограммы в пунктах меню	607
11.5.3. Пункты меню с флажками и кнопками-переключателями	608
11.5.4. Всплывающие меню	609
11.5.5. Клавиши быстрого доступа и оперативные клавиши	611
11.5.6. Разрешение и запрет доступа к пунктам меню	613
11.5.7. Панели инструментов	618
11.5.8. Всплывающие подсказки	620
11.6. Расширенные средства компоновки	621
11.6.1. Диспетчер сеточно-контейнерной компоновки	622
11.6.2. Специальные диспетчеры компоновки	632
11.7. Диалоговые окна	636
11.7.1. Диалоговые окна для выбора разных вариантов	636
11.7.2. Создание диалоговых окон	641
11.7.3. Обмен данными	645
11.7.4. Диалоговые окна для выбора файлов	651
Глава 12. Параллелизм	661
12.1. Назначение потоков исполнения	662
12.2. Состояния потоков исполнения	667
12.2.1. Новые потоки исполнения	667
12.2.2. Исполняемые потоки	667
12.2.3. Блокированные и ожидающие потоки исполнения	668
12.2.4. Завершенные потоки исполнения	669
12.3. Свойства потоков исполнения	670
12.3.1. Прерывание потоков исполнения	670
12.3.2. Потоковые демоны	673
12.3.3. Именованное потоков исполнения	674
12.3.4. Обработчики необрабатываемых исключений	674
12.3.5. Приоритеты потоков исполнения	675
12.4. Синхронизация	676
12.4.1. Пример состояния гонок	676
12.4.2. Объяснение причин, приводящих к состоянию гонок	679
12.4.3. Объекты блокировки	681
12.4.4. Объекты условий	684
12.4.5. Ключевое слово synchronized	689
12.4.6. Синхронизированные блоки	693
12.4.7. Принцип монитора	694
12.4.8. Поля и переменные типа volatile	695
12.4.9. Поля и переменные типа final	697
12.4.10. Атомарность операций	697
12.4.11. Взаимные блокировки	699
12.4.12. Локальные переменные в потоках исполнения	702
12.4.13. Причины, по которым методы stop() и suspend() не рекомендованы к применению	703

12.5. Потокобезопасные коллекции	705
12.5.1. Блокирующие очереди	705
12.5.2. Эффективные отображения, множества и очереди	712
12.5.3. Атомарное обновление записей в отображениях	713
12.5.4. Групповые операции над параллельными хеш-отображениями	717
12.5.5. Параллельные представления множеств	719
12.5.6. Массивы, копируемые при записи	720
12.5.7. Алгоритмы обработки параллельных массивов	720
12.5.8. Устаревшие потокобезопасные коллекции	721
12.6. Задачи и пулы потоков исполнения	722
12.6.1. Интерфейсы Callable и Future	723
12.6.2. Исполнители	725
12.6.3. Управление группами задач	727
12.6.4. Архитектура вилочного соединения	732
12.7. Асинхронные вычисления	735
12.7.1. Завершаемые будущие действия	735
12.7.2. Составление завершаемых будущих действий	738
12.7.3. Длительные задачи в обратных вызовах пользовательского интерфейса	744
12.8. Процессы	751
12.8.1. Построение процесса	752
12.8.2. Выполнение процесса	753
12.8.3. Дескрипторы процессов	755
Глава 13. Библиотека JavaFX	759
13.1. Отображение данных на сцене	759
13.1.1. Первое JavaFX-приложение	759
13.2.2. Рисование геометрических форм	763
13.2.3. Текст и изображения	767
13.3. Обработка событий	771
13.3.1. Реализация обработчиков событий	772
13.3.2. Реагирование на изменения свойств	772
13.3.3. События от мыши и клавиатуры	775
13.4. Компоновка	782
13.4.1. Панели компоновки	783
13.4.2. Язык FXML	789
13.4.3. Стилиевые таблицы CSS	795
13.5. Элементы управления пользовательского интерфейса	800
13.5.1. Элементы управления вводом текста	800
13.5.2. Элементы управления выбором разных вариантов	804
13.5.3. Меню	811
13.5.4. Простые диалоговые окна	819
13.5.5. Специальные элементы управления	828
13.6. Свойства и привязки	832
13.6.1. Свойства в библиотеке JavaFX	832
13.6.2. Привязки	835
13.7. Длительные задачи в обратных вызовах пользовательского интерфейса	841
Приложение А. Ключевые слова Java	849
Предметный указатель	851

Предисловие

К читателю

В конце 1995 года язык программирования Java вырвался на просторы Интернета и моментально завоевал популярность. Технология Java обещала стать *универсальным связующим звеном*, соединяющим пользователей с информацией, откуда бы она ни поступала — от веб-серверов, баз данных, поставщиков информации или любого другого источника, который только можно вообразить. И действительно, у Java есть все, чтобы выполнить эти обещания. Это весьма основательно сконструированный язык, получивший широкое признание. Его встроенные средства защиты и безопасности оснастили как программистов, так и пользователей программ на Java. Язык Java изначально обладал встроенной поддержкой для решения таких сложных задач, как сетевое программирование, взаимодействие с базами данных и параллелизм.

С 1995 года было выпущено одиннадцать главных версий комплекта Java Development Kit. За последние двадцать лет прикладной программный интерфейс (API) языка Java увеличился от 200 до более 4 тысяч классов и теперь охватывает самые разные предметные области, включая конструирование пользовательских интерфейсов, управление базами данных, интернационализацию, безопасность и обработку данных в формате XML.

Книга, которую вы держите в руках, является первым томом одиннадцатого издания. С выходом каждого издания ее главный автор старался как можно быстрее следовать очередному выпуску комплекта Java Development Kit, каждый раз переписывая ее, чтобы вы могли воспользоваться преимуществами новейших средств Java. Настоящее издание обновлено с учетом новых языковых средств, появившихся в версиях Java Standard Edition (SE) 9, 10 и 11.

Как и все предыдущие издания этой книги, настоящее издание по-прежнему *адресуется серьезным программистам, которые хотели бы пользоваться Java для разработки настоящих проектов*. Автор этой книги представляет себе вас, дорогой читатель, как грамотного специалиста с солидным опытом программирования на других языках, кроме Java, и надеется, что вам не нравятся книги, которые полны игрушечных примеров вроде программ управления тостерами или животными в зоопарке либо “прыгающим текстом”. Ничего подобного вы не найдете в этой книге. Цель автора — помочь вам понять язык Java и его библиотеки в полной мере, а не создать иллюзию такого понимания.

В этой книге вы найдете массу примеров кода, демонстрирующих почти все обсуждаемые языковые и библиотечные средства. Эти примеры намеренно сделаны как можно более простыми, чтобы сосредоточиться на основных моментах. Тем не менее в большинстве в своем они совсем не игрушечные, не срезают острых углов и могут послужить вам неплохой отправной точкой для разработки собственного кода.

Автор предполагает, что вы стремитесь (и даже жаждете) узнать обо всех расширенных средствах, которые Java предоставляет в ваше распоряжение. Поэтому в первом томе настоящего издания подробно рассматриваются следующие темы.

- Объектно-ориентированное программирование.
- Рефлексия и прокси-классы.
- Интерфейсы и внутренние классы.
- Обработка исключений.
- Обобщенное программирование.
- Каркас коллекций.
- Модель приемников событий.
- Проектирование графического пользовательского интерфейса.
- Параллельное программирование.

В связи со стремительным ростом библиотеки классов Java одного тома оказалось недостаточно для описания всех языковых средств Java, о которых следует знать серьезным программистам. Поэтому книга была разделена на два тома. В первом томе, который вы держите в руках, главное внимание уделяется фундаментальным понятиям языка Java, а также основам программирования пользовательского интерфейса. Второй том посвящен средствам разработки приложений масштаба предприятия и усовершенствованному программированию пользовательских интерфейсов. В нем вы найдете подробное обсуждение следующих вопросов.

- Поточный прикладной программный интерфейс API.
- Обработка файлов и регулярные выражения.
- Базы данных.
- Обработка данных в формате XML.
- Аннотации.
- Интернационализация.
- Сетевое программирование.
- Расширенные компоненты графического пользовательского интерфейса.
- Усовершенствованная графика.
- Платформенно-ориентированные методы.

При написании книги ошибки и неточности неизбежны, и автору книги очень важно знать о них. Но он, конечно, предпочел бы узнать о каждой из них только один раз. Поэтому перечень часто задаваемых вопросов, исправлений, ошибок и обходных приемов был размещен по адресу <http://horstmann.com/corejava>, куда вы можете обращаться за справкой.

Краткий обзор книги

В главе 1 дается краткий обзор тех функциональных возможностей языка Java, которыми он отличается от других языков программирования. В ней сначала поясняется, что было задумано разработчиками Java и в какой мере им удалось воплотить задуманное в жизнь. Затем приводится краткая история развития языка Java и показывается, как он стал тем, чем он есть в настоящее время.

В главе 2 сначала поясняется, как загрузить и установить инструментарий JDK, а также примеры программ к этой книге. Затем рассматривается весь процесс компиляции и запуска трех типичных программ на Java (консольного приложения, графического приложения и апплета) только средствами JDK, текстового редактора, специально ориентированного на Java, а также интегрированной среды разработки на Java.

В главе 3 начинается обсуждение языка программирования Java и излагаются самые основы: переменные, циклы и простые функции. Если у вас имеется опыт программирования на C или C++, вам нетрудно будет усвоить материал этой главы, поскольку синтаксис этих языковых средств, по существу, ничем не отличается в Java. А если вам приходилось программировать на языках, не похожих на C, например на Visual Basic, прочитайте эту главу с особым вниманием.

Ныне объектно-ориентированное программирование (ООП) — господствующая методика программирования, и ей в полной мере отвечает язык Java. В главе 4 представлены понятие *инкапсуляции* — первой из двух фундаментальных составляющих объектной ориентации, а также механизмы, реализующие ее в языке Java: классы и методы. В дополнение к правилам языка Java здесь также приводятся рекомендации по правильному объектно-ориентированному проектированию. И, наконец, будет представлен замечательный инструмент *javadoc*, форматирующий комментарии из исходного кода в набор веб-страниц с перекрестными ссылками. Если у вас имеется опыт программирования на C++, можете лишь бегло просмотреть эту главу. А тем, кому раньше не приходилось программировать на объектно-ориентированных языках, придется потратить больше времени на усвоение принципов ООП, прежде чем изучать Java дальше.

Классы и инкапсуляция — это лишь часть методики ООП, и поэтому в главе 5 представлен еще один ее краеугольный камень — *наследование*. Наследование позволяет модифицировать существующий класс в соответствии с конкретными потребностями программирующего. Это — основополагающий прием программирования на Java. Механизм наследования в Java очень похож на аналогичный механизм в C++. Опять же программирующие на C++ могут сосредоточить основное внимание лишь на языковых отличиях в реализации наследования.

В главе 6 поясняется, как пользоваться в Java понятием *интерфейса*. Интерфейсы дают возможность выйти за пределы простого наследования, описанного в главе 5. Владение интерфейсами позволит в полной мере воспользоваться объектно-ориентированным подходом к программированию на Java. После интерфейсов рассматриваются *лямбда-выражения* в качестве краткого способа выражения блока кода, который может быть выполнен впоследствии. И, наконец, рассматривается также удобное языковое средство Java, называемое *внутренними классами*.

Глава 7 посвящена *обработке исключений* — надежному механизму Java, призванному учитывать тот факт, что непредвиденные ситуации могут возникать и в грамотно написанных программах. Исключения обеспечивают эффективный способ отделения кода нормальной обработки от кода обработки ошибок. Но даже после оснащения прикладной программы проверкой всех возможных исключительных ситуаций в ней все-таки может произойти неожиданный сбой. Во второй части этой главы будет представлено немало полезных советов по организации отладки программ. Кроме того, здесь рассматривается весь процесс отладки на конкретном примере.

В главе 8 дается краткий обзор *обобщенного программирования*. Обобщенное программирование делает прикладные программы легче читаемыми и более безопасными. В этой главе будет показано, как применяется строгая типизация, исключается потребность в неприглядном и небезопасном приведении типов и как преодолеваются трудности на пути совместимости с предыдущими версиями Java.

Глава 9 посвящена каркасу коллекций на платформе Java. Всякий раз, когда требуется сначала собрать множество объектов, а в дальнейшем извлечь их, приходится обращаться к коллекции, которая наилучшим образом подходит для конкретных условий, вместо того чтобы сбрасывать их в обычный массив. В этой главе будут продемонстрированы те преимущества, которые дают стандартные, предварительно подготовленные коллекции.

В **главе 10** представлено введение в программирование графических пользовательских интерфейсов. Будет показано, как создаются окна, как в них выполняется раскраска, рисуются геометрические фигуры, форматируется текст многими шрифтами и как изображения выводятся на экран. Далее здесь будет пояснено, как писать прикладной код, реагирующий на такие события, как щелчки мышью или нажатия клавиш.

Глава 11 посвящена более подробному обсуждению инструментальных средств Swing. Набор инструментов Swing позволяет строить межплатформенный графический пользовательский интерфейс. В этой главе вы ознакомитесь с различными видами экранных кнопок, текстовых компонентов, рамок, ползунков, комбинированных списков, меню и диалоговых окон. Но знакомство с некоторыми из более совершенных компонентов Swing будет отложено до второго тома настоящего издания.

Глава 12 посвящена обсуждению параллельного программирования, которое позволяет выполнять программируемые задачи параллельно. Это очень важное и любопытное применение технологии Java в эпоху многоядерных процессоров, которые нужно загрузить работой, чтобы они не простаивали.

В **главе 13**, завершающей первый том настоящего издания, приведено краткое введение в библиотеку JavaFX, позволяющую разрабатывать современный графический пользовательский интерфейс настольных приложений.

В **приложении А** перечислены зарезервированные слова языка Java.

Условные обозначения

Как это принято во многих книгах по программированию, моноширинным шрифтом выделяется исходный код примеров.



НА ЗАМЕТКУ! Этой пиктограммой выделяются примечания.



СОВЕТ. Этой пиктограммой выделяются советы.



ВНИМАНИЕ! Этой пиктограммой выделяются предупреждения о потенциальной опасности.



НА ЗАМЕТКУ C++! В этой книге имеется немало примечаний к синтаксису C++, где разъясняются отличия между языками Java и C++. Вы можете пропустить их, если у вас нет опыта программирования на C++ или же если вы склонны воспринимать этот опыт как страшный сон, который лучше забыть.

Язык Java сопровождается огромной библиотекой в виде прикладного программного интерфейса (API). При упоминании вызова какого-нибудь метода из прикладного программного интерфейса API в первый раз в конце соответствующего раздела приводится его краткое описание. Эти описания не слишком информативны, но, как мы надеемся, более содержательны, чем те, что представлены в официальной

оперативно доступной документации на прикладной программный интерфейс API. Имена интерфейсов выделены **полужирным моноширинным** шрифтом, а число после имени класса, интерфейса или метода обозначает версию JDK, в которой данное средство было внедрено, как показано ниже.

Название прикладного программного интерфейса 9

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

Листинг 1.1. Исходный код из файла `InputTest/InputTest.java`

Примеры исходного кода

Все примеры исходного кода, приведенные в этой книге, доступны по соответствующей ссылке в архивированном виде на посвященном ей веб-сайте по адресу <http://horstmann.com/corejava>. Код примеров можно также загрузить с веб-страницы русского издания книги по адресу: <http://www.williamspublishing.com/Books/978-5-907114-79-1.html>. Подробнее об установке комплекта для разработки приложений на Java (Java Development Kit — JDK) и примеров кода речь пойдет в главе 2.

Благодарности

Написание книги всегда требует значительных усилий, а ее переписывание не намного легче, особенно если учесть постоянные изменения в технологии Java. Чтобы сделать книгу полезной, необходимы совместные усилия многих преданных делу людей, и автор книги с удовольствием выражает признательность всем, кто внес свой посильный вклад в настоящее издание книги.

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

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

Среди рецензентов этого и предыдущих изданий хотелось бы отметить Чака Аллисона (Chuck Allison) из университета долины Юты, Ланса Андерсона (Lance Anderson, Oracle), Пола Андерсона (Paul Anderson, Anderson Software Group), Алека Битона (IBM), Клиффа Берга, Эндрю Бинстока (Andrew Binstock, Oracle), Джошуа Блоха (Joshua Bloch), Дэвида Брауна (David Brown), Корки Карпрайта (Corky Cartwright), Френка Коена (Frank Cohen, PushToTest), Криса Крейна (Chris Crane, devXsolution), доктора

Николаса Дж. Де Лилло (Dr. Nicholas J. De Lillo) из Манхеттенского колледжа, Ракеша Дхупара (Rakesh Dhoopar, Oracle), Дэвида Гири (David Geary), Джима Гиша (Jim Gish, Oracle), Брайана Гетца (Brian Goetz, Oracle), Анжелу Гордон (Angela Gordon) и Дэна Гордона (Dan Gordon, Electric Cloud), Роба Гордона (Rob Gordon), Джона Грэя (John Gray) из Хартфордского университета, Камерона Грегори (Cameron Gregory, olabs.com), Марти Холла (Marty Hall, coreservlets.com, Inc.), Винсента Харди (Vincent Hardy, Adobe Systems), Дэна Харки (Dan Harkey) из университета штата Калифорния в Сан-Хосе, Вильяма Хиггинса (William Higgins, IBM), Владимира Ивановича (Vladimir Ivanovic, PointBase), Джерри Джексона (Jerry Jackson, CA Technologies), Тима Киммета (Tim Kimmeter, Walmart), Криса Лаффра (Chris Laffra), Чарли Лаи (Charlie Lai, Apple), Анжелику Лангер (Angelika Langer), Дуга Лэнгстона (Doug Langston), Ханг Лау (Hang Lau) из университета имени Макгилла, Марка Лоуренса (Mark Lawrence), Дуга Ли (Doug Lea, SUNY Oswego), Грегори Лонгшора (Gregory Longshore), Боба Линча (Bob Lynch, Lynch Associates), Филиппа Милна (Philip Milne, консультанта), Марка Моррисси (Mark Morrissey) из научно-исследовательского института штата Орегон, Махеша Нилаканта (Mahesh Neelakanta) из Атлантического университета штата Флорида, Хао Фам (Hao Pham), Пола Филиона (Paul Philion), Блейка Рагсдейла (Blake Ragsdell), Стюарта Реджеса (Stuart Reges) из университета штата Аризона, Рича Розена (Rich Rosen, Interactive Data Corporation), Питера Сандерса (Peter Sanders) из университета ЭССИ (ESSI), г. Ницца, Франция, доктора Пола Сангеру (Dr. Paul Sanghera) из университета штата Калифорния в Сан-Хосе и колледжа имени Брукса, Пола Сэвинка (Paul Sevinc, Teamup AG), Деванг Ша (Devang Shah, Oracle), Бредли А. Смита (Bradley A. Smith), Стивена Стелтинга (Steven Stelling, Oracle), Кристофера Тэйлора (Christopher Taylor), Люка Тэйлора (Luke Taylor, Valtech), Джорджа Тхируватукала (George Thiruvathukal), Кима Топли (Kim Topley, StreamingEdge), Джанет Трауб (Janet Traub), Пола Тиму (Paul Тума, консультанта), Питера Ван Дер Линдена (Peter van der Linden), Кристиана Улленбума (Christian Ullenboom), Берта Уолша (Burt Walsh), Дана Ксю (Dan Xu, Oracle) и Джона Завгрена (John Zavgren, Oracle).

Кей Хорстманн, Сан-Франциско, шт. Калифорния, июнь 2018 г.

От издательства

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

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

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

Наши электронные адреса:

E-mail: info@williamspublishing.com

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

Введение в язык Java

В этой главе...

- ▶ Программная платформа Java
- ▶ Характерные особенности Java
- ▶ Апплеты Java и Интернет
- ▶ Краткая история развития Java
- ▶ Распространенные заблуждения относительно Java

На появление первой версии Java в 1996 году откликнулись не только специализирующиеся на вычислительной технике газеты и журналы, но даже такие солидные издания, как *The New York Times*, *The Washington Post* и *Business Week*. Java — единственный язык программирования, удостоившийся десятиминутного репортажа на Национальном общественном радио в США. Для разработки и сопровождения программных продуктов *только* на этом языке программирования был учрежден венчурный фонд в 100 миллионов долларов. Это было удивительное время. Тем временам и последующей истории развития языка Java посвящена эта глава.

1.1. Программная платформа Java

В первом издании этой книги о Java было сказано следующее: “Как язык программирования, Java перевыполнил рекламные обещания. Определенно, Java — хороший язык программирования. Несомненно, это один из лучших языков, доступных серьезным программистам. Потенциально Java имел все предпосылки, чтобы стать великим языком программирования, но теперь время для этого уже, вероятно, упущено. Как только появляется новый язык программирования, сразу же возникает неприятная проблема его совместимости с созданным раньше программным обеспечением”.

По поводу этого абзаца редактор первого издания книги долго спорил с одним из руководителей компании Sun Microsystems, где первоначально был разработан язык Java. Но и сейчас, по прошествии длительного времени, такая оценка кажется

правильной. Действительно, Java обладает целым рядом преимуществ, о которых речь пойдет далее в этой главе. Но более поздние дополнения далеко не так изящны, как исходный вариант этого языка, и виной тому — пресловутые требования совместимости.

Как уже отмечалось в первом издании книги, Java никогда не был только языком. Хорошие языки — не редкость, а появление некоторых из них вызвало в свое время настоящую сенсацию в области вычислительной техники. В отличие от них, Java — это программная платформа, включающая в себя мощную библиотеку, большой объем кода, пригодного для повторного использования, а также среду для выполнения программ, которая обеспечивает безопасность, независимость от операционной системы и автоматическую сборку “мусора”.

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

1.2. Характерные особенности Java

Создатели Java составили официальное техническое описание, в котором объяснялись цели и достоинства нового языка. В этом документе приведено одиннадцать характерных особенностей Java. Этот язык

- простой;
- объектно-ориентированный;
- распределенный;
- надежный;
- безопасный;
- не зависящий от архитектуры компьютера;
- переносимый;
- интерпретируемый;
- высокопроизводительный;
- многопоточный;
- динамичный.

В данном разделе приводятся цитаты из официального описания Java, раскрывающие особенности этого языка программирования, а также комментарии к ним, основывающиеся на личном опыте работы автора с текущей версией Java.



НА ЗАМЕТКУ! Упомянутое выше официальное описание Java можно найти по адресу www.oracle.com/technetwork/java/langenv-140151.html. Там же описаны характерные особенности Java. А краткий обзор одиннадцати характерных особенностей Java приведен по адресу <http://horstmann.com/corejava/java-an-overview/7Gosling.pdf>.

1.2.1. Простота

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

Синтаксис Java, по существу, представляет собой упрощенный вариант синтаксиса C++. В этом языке отсутствует потребность в файлах заголовков, арифметике (и даже в синтаксисе) указателей, структурах, объединениях, перегрузке операций, виртуальных базовых классах и т.п. (Отличия Java от C++ упоминаются на протяжении всей книги в специальных врезках.) Но создатели Java не стремились исправить все недостатки языка C++. Например, синтаксис оператора `switch` в Java остался неизменным. Зная C++, нетрудно перейти на Java.

Когда был выпущен язык Java, C++ был отнюдь не самым распространенным языком программирования. Многие разработчики пользовались языком Visual Basic и его средой программирования путем перетаскивания. Этим разработчикам Java показался непростым языком, поэтому им потребовалось несколько лет для овладения средами разработки на Java. В настоящее время среды разработки на Java продвинулись далеко вперед по сравнению с большинством других языков программирования.

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

На то время это было впечатляющим достижением. Разумеется, с тех пор библиотеки разрослись до гигантских размеров. Но теперь существует отдельная платформа Java Micro Edition с компактными библиотеками, более подходящая для встроенных устройств.

1.2.2. Объектно-ориентированный характер

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

Объектно-ориентированный подход к программированию вполне упрочился на момент появления языка Java. Действительно, особенности Java, связанные с объектами, сравнимы с языком C++. Основное отличие между ними заключается в механизме множественного наследования, который в Java заменен более простым понятием интерфейсов. Язык Java обладает большими возможностями для самоанализа при выполнении, чем C++ (подробнее об этом речь пойдет в главе 5).

1.2.3. Поддержка распределенных вычислений в сети

“Язык Java предоставляет разработчику обширную библиотеку программ для передачи данных по протоколу TCP/IP, HTTP и FTP. Приложения на Java способны открывать объекты и получать к ним доступ по сети с такой же легкостью, как и в локальной файловой системе, используя URL для адресации”.

В настоящее время эта особенность Java воспринимается как само собой разумеющееся, но в 1995 году подключение к веб-серверу из программы на C++ или Visual Basic считалось немалым достижением.

1.2.4. Надежность

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

Компилятор Java выявляет такие ошибки, которые в других языках обнаруживаются только на этапе выполнения программы. Кроме того, программисты, потратившие многие часы на поиски ошибки, вызвавшей нарушения данных в памяти из-за неверного указателя, будут обрадованы тем, что в работе с Java подобные осложнения не могут даже в принципе возникнуть.

1.2.5. Безопасность

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

Ниже перечислены некоторые виды нарушения защиты, которые с самого начала предотвращает система безопасности Java.

- Намеренное переполнение стека выполняемой программы — один из распространенных способов нарушения защиты, используемых вирусами и “червями”.
- Повреждение данных на участках памяти, находящихся за пределами пространства, выделенного процессу.
- Несанкционированное чтение файлов и их модификация.

Изначально в Java было принято весьма ответственное отношение к загружаемому коду. Ненадежный и небезопасный код выполнялся в среде “песочницы”, где он не мог оказывать отрицательного влияния на главную систему. Пользователи могли быть уверены, что в их системе не произойдет ничего плохого из-за выполнения кода Java независимо от его происхождения, поскольку он просто не мог проникнуть из “песочницы” наружу.

Но модель безопасности в Java сложна. Вскоре после выпуска первой версии Java Development Kit группа специалистов по безопасности из Принстонского университета обнаружила в системе безопасности едва заметные программные ошибки, которые позволяли совершать атаки на главную систему из ненадежного кода.

Эти ошибки были быстро исправлены. Но, к сожалению, злоумышленники ухитрились найти незначительные прорехи в реализации архитектуры безопасности. И компании Sun Microsystems, а затем и компании Oracle пришлось потратить немало времени на устранение подобных прорех.

После целого ряда крупномасштабных атак производители браузеров и специалисты из компании Oracle стали осмотрительнее. Теперь модули Java, подключаемые к браузерам, больше не доверяют удаленному коду, если отсутствует цифровая подпись этого кода и согласие пользователей на его выполнение.



НА ЗАМЕТКУ! Оглядывая назад, можно сказать, что модель безопасности в Java оказалась не такой удачной, как предполагалось изначально, но и тогда она явно опережала свое время. Альтернативный механизм доставки кода, предложенный корпорацией Microsoft, опирался только на цифровые подписи для обеспечения безопасности. Очевидно, что этого было явно недостаточно — любой пользователь программного обеспечения корпорации Microsoft может подтвердить, что даже программы широко известных производителей часто завершаются аварийно, создавая тем самым опасность повреждения данных.

1.2.6. Независимость от архитектуры компьютера

“Компилятор генерирует объектный файл, формат которого не зависит от архитектуры компьютера. Скомпилированная программа может выполняться на любых процессорах, а для ее работы требуется лишь исполняющая система Java. Код, генерируемый компилятором Java, называется байт-кодом. Он разработан таким образом, чтобы его можно было легко интерпретировать на любой машине или оперативно преобразовать в собственный машинный код”.

Генерирование кода для виртуальной машины не считалось в то время каким-то новшеством. Подобный принцип уже давно применялся в таких языках программирования, как Lisp, Smalltalk и Pascal.

Очевидно, что байт-код, интерпретируемый с помощью виртуальной машины, всегда будет работать медленнее, чем машинный код. Но эффективность байт-кода можно существенно повысить за счет динамической компиляции во время выполнения программы.

У виртуальной машины имеется еще одно важное преимущество по сравнению с непосредственным выполнением программы. Она существенно повышает безопасность, поскольку в процессе работы можно оценить последствия выполнения каждой конкретной команды.

1.2.7. Переносимость

“В отличие от C и C++, ни один из аспектов спецификации Java не зависит от реализации. Разрядность примитивных типов данных и арифметические операции над ними строго определены”.

Например, тип `int` в Java всегда означает 32-разрядное целое число, а в C и C++ тип `int` может означать как 16-, так и 32-разрядное целое число. Единственное ограничение состоит в том, что разрядность типа `int` не может быть меньше разрядности типа `short int` и больше разрядности типа `long int`. Фиксированная разрядность числовых типов данных позволяет избежать многих неприятностей, связанных с выполнением программ на разных компьютерах. Двоичные данные хранятся и передаются в неизменном формате, что также позволяет избежать недоразумений,

связанных с разным порядком следования байтов на различных платформах. Символьные строки сохраняются в стандартном формате Юникода.

“Библиотеки, являющиеся частью системы, предоставляют переносимые интерфейсы. Например, в Java предусмотрен абстрактный класс `Window` и его реализации для операционных систем `Unix`, `Windows` и `Macintosh`”.

Пример класса `Window`, по-видимому, был выбран не совсем удачно. Всякий, когда-либо пытавшийся написать программу, которая одинаково хорошо работала бы под управлением операционных систем `Windows`, `Mac OS` и десятка разновидностей ОС `Unix`, знает, что это очень трудная задача. Разработчики версии `Java 1.0` приняли героическую попытку решить эту задачу, предоставив простой набор инструментальных средств, приспособляющий обычные элементы пользовательского интерфейса к различным платформам. К сожалению, в конечном итоге получилась библиотека, работать с которой было непросто, а результаты оказывались едва ли приемлемыми в разных системах. Этот первоначальный набор инструментов для построения пользовательского интерфейса был в дальнейшем не раз изменен, хотя переносимость программ на разные платформы по-прежнему остается проблемой.

Тем не менее со всеми задачами, которые не имеют отношения к пользовательским интерфейсам, библиотеки `Java` отлично справляются, позволяя разработчикам работать, не привязываясь к конкретной платформе. В частности, они могут пользоваться файлами, регулярными выражениями, XML-разметкой, датами и временем, базами данных, сетевыми соединениями, потоками исполнения и прочими средствами, не опираясь на базовую операционную систему. Программы на `Java` не только становятся переносимыми, но и прикладные программные интерфейсы `Java API` нередко оказываются более высокого качества, чем их платформенно-ориентированные аналоги.

1.2.8. Интерпретируемость

“Интерпретатор `Java` может выполнять байт-код непосредственно на любой машине, на которую перенесен интерпретатор. А поскольку процесс компоновки носит в большей степени пошаговый и относительно простой характер, процесс разработки программ может быть заметно ускорен, став более творческим”.

С этим можно согласиться, но с большой натяжкой. Всем, кто имеет опыт программирования на `Lisp`, `Smalltalk`, `Visual Basic`, `Python`, `R` или `Scala`, хорошо известно, что на самом деле означает “ускоренный и более творческий” процесс разработки. Опробуя что-нибудь, вы сразу же видите результат. Но такой опыт просто отсутствует в работе со средами разработки на `Java`. Но так было до версии `Java 9`, когда появилось инструментальное средство `jshell`, поддерживающее быстрое экспериментальное программирование в диалоговом режиме.

1.2.9. Производительность

“Обычно интерпретируемый байт-код имеет достаточную производительность, но бывают ситуации, когда требуется еще более высокая производительность. Байт-код можно транслировать во время выполнения программы в машинный код для того процессора, на котором выполняется данное приложение”.

На ранней стадии развития `Java` многие пользователи были не согласны с утверждением, что производительности “более чем достаточно”. Но теперь динамические компиляторы (называемые иначе *JIT-компиляторами*) настолько усовершенствованы,

что могут конкурировать с традиционными компиляторами, а в некоторых случаях они даже дают выигрыш в производительности, поскольку имеют больше доступной информации. Так, например, динамический компилятор может отслеживать код, который выполняется чаще, и оптимизировать по быстрдействию только эту часть кода. Динамическому компилятору известно, какие именно классы были загружены. Он может сначала применить встраивание, когда некоторая функция вообще не переопределяется на основании загруженной коллекции классов, а затем отменить, если потребуется, такую оптимизацию.

1.2.10. Многопоточность

“К преимуществам многопоточности относится более высокая интерактивная реакция и поведение программ в реальном масштабе времени”.

В настоящее время все большее значение приобретает распараллеливание выполняемых задач, поскольку действие закона Мура подходит к концу. В нашем распоряжении теперь имеются не более быстродействующие процессоры, а большее их количество, и поэтому мы должны загрузить их полезной работой, чтобы они не простаивали. Но, к сожалению, в большинстве языков программирования проявляется поразительное пренебрежение этой проблемой.

И в этом отношении Java опередил свое время. Он стал первым из основных языков программирования, где поддерживалось параллельное программирование. Как следует из упомянутого выше официального описания Java, побудительная причина к такой поддержке была несколько иной. В то время многоядерные процессоры были редкостью, а веб-программирование только начинало развиваться, и поэтому процессорам приходилось долго простаивать в ожидании ответа от сервера. Параллельное программирование требовалось для того, чтобы пользовательский интерфейс не застыл в ожидании подобных ответов. И хотя параллельное программирование никогда не было простым занятием, в Java сделано немало, чтобы этот процесс стал более управляемым.

1.2.11. Динамичность

“Во многих отношениях язык Java является более динамичным, чем языки C и C++. Он был разработан таким образом, чтобы легко адаптироваться к постоянно изменяющейся среде. В библиотеки можно свободно включать новые методы и объекты, ни коим образом не затрагивая приложения, пользующиеся библиотеками. В языке Java совсем не трудно получить информацию о ходе выполнения программы”.

Это очень важно в тех случаях, когда требуется добавить код в уже выполняющуюся программу. Ярким тому примером служит код, загружаемый из Интернета для выполнения браузером. Сделать это на C или C++ не так-то просто, но разработчики Java были хорошо осведомлены о динамических языках программирования, которые позволяли без особого труда развивать выполняющуюся программу. Их достижение состояло в том, что они внедрили такую возможность в основной язык программирования.



НА ЗАМЕТКУ! После первых успехов Java корпорация Microsoft выпустила программный продукт под названием J++, включавший в себя язык программирования, очень похожий на Java, а также виртуальную машину. В настоящее время корпорация Microsoft прекратила поддержку J++ и сосредоточила свои усилия на другом языке, который называется C# и чем-то напоминает Java, хотя в нем используется другая виртуальная машина. Языки J++ и C# в этой книге не рассматриваются.

1.3. Апплеты и Интернет

Первоначальная идея была проста — пользователи загружают байт-коды Java по сети и выполняют их на своих машинах. Программы Java, работающие под управлением веб-браузеров, называются *апплетами*. Для использования апплета требуется веб-браузер, поддерживающий язык Java и способный интерпретировать байт-код. Лицензия на исходные коды языка Java принадлежит компании Oracle, настаивающей на неизменности как самого языка, так и структуры его основных библиотек, и поэтому апплеты Java должны запускаться в любом браузере, который поддерживает Java. Посещая всякий раз веб-страницу, содержащую апплет, вы получаете последнюю версию этой прикладной программы. Но важнее другое: благодаря средствам безопасности виртуальной машины вы избавляетесь от необходимости беспокоиться об угрозах, исходящих от вредоносного кода.

Ввод апплета на веб-странице осуществляется практически так же, как и встраивание изображения. Апплет становится частью страницы, а текст обтекает занимаемое им пространство. Изображение, формируемое апплетом, становится *активным*. Оно реагирует на действия пользователя, изменяет в зависимости от них свой внешний вид и выполняет обмен данными между компьютером, на котором выполняется апплет, и компьютером, где этот апплет постоянно хранится.

На рис. 1.1 приведен апплет Jmol, предназначенный для отображения моделей молекул. Чтобы лучше понять структуру молекулы, ее можно повернуть или изменить масштаб изображения, пользуясь мышью. В то время, когда были изобретены апплеты, подобного рода непосредственного манипулирования не удавалось добиться на имевшихся тогда статических веб-страницах, где присутствовали лишь несовершенные сценарии JavaScript и отсутствовал холст в виде элемента HTML-разметки.

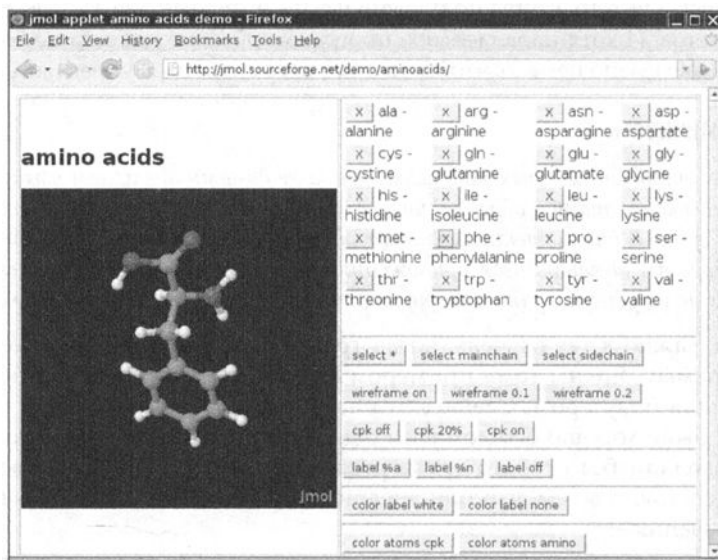


Рис. 1.1. Апплет Jmol

Когда апплеты только появились, они наделали немало шума. Многие считают, что именно привлекательность апплетов стала причиной ошеломляющего успеха Java. Но первоначальный энтузиазм быстро сменился разочарованием. В разных версиях

браузеров Netscape и Internet Explorer поддерживались разные версии языка Java, причем некоторые из них заметно устарели. Эта неприятная ситуация создавала все больше препятствий при разработке апплетов, в которых можно было бы воспользоваться преимуществами последней версии Java. Поэтому вместо апплетов стали применять технологию Flash от компании Adobe Systems, когда требовалось получить динамические эффекты в браузере. В дальнейшем, когда язык Java стали преследовать серьезные проблемы безопасности, в браузерах и подключаемых к ним модулях на Java постепенно начали употребляться все более ограничительные меры безопасности. Ныне, чтобы добиться работоспособности апплетов в браузере, требуется немалое умение и самоотдача. Так, если посетить веб-сайт с апплетом Jmol по адресу <http://jmol.sourceforge.net/demo/aminoacids/>, то на нем можно обнаружить сообщение, предупреждающее о необходимости настроить браузер таким образом, чтобы он разрешал выполнять апплеты.

1.4. Краткая история развития Java

В этом разделе кратко изложена история развития языка Java. В основу этого материала положены различные опубликованные первоисточники (в частности, интервью с создателями языка Java в июльском выпуске электронного журнала *SunWorld* за 1995 г.).

История Java восходит к 1991 году, когда группа инженеров из компании Sun Microsystems под руководством Патрика Нотона (Patrick Naughton) и члена совета директоров (и разностороннего специалиста) Джеймса Гослинга (James Gosling) занялась разработкой языка, который можно было бы использовать для программирования бытовых устройств, например, контроллеров для переключения каналов кабельного телевидения. Подобные устройства не обладают большими вычислительными мощностями и объемом оперативной памяти, и поэтому новый язык должен был быть простым и способным генерировать очень компактный код. Кроме того, разные производители могут выбирать разные процессоры для контроллеров, поэтому было очень важно не привязываться к конкретной их архитектуре. Проект создания нового языка получил кодовое название “Green”.

Стремясь получить компактный и независимый от платформы код, разработчики решили создать переносимый язык, способный генерировать промежуточный код для виртуальной машины. Большинство сотрудников компании Sun Microsystems имели опыт работы с операционной системой Unix, поэтому в основу разрабатываемого ими языка был положен язык C++, а не Lisp, Smalltalk или Pascal. Как сказал Гослинг в своем интервью: “Язык — это всегда средство, а не цель”. Сначала Гослинг решил назвать его Oak (Дуб). (Возможно потому, что он любил смотреть на дуб, росший прямо под окнами его рабочего кабинета в компании Sun Microsystems.) Затем сотрудники компании узнали, что слово “Oak” уже используется в качестве имени ранее созданного языка программирования, и изменили название на Java. Этот выбор был сделан по наитию.

В 1992 г. в рамках проекта “Green” была выпущена первая продукция под названием “*7”. Это было устройство интеллектуального дистанционного управления. К сожалению, ни одна из компаний-производителей электронной техники не заинтересовалась этой разработкой. Затем группа стала заниматься созданием устройства для кабельного телевидения, которое могло бы осуществлять новые виды услуг, например, включать видеосистему по требованию. И снова они не получили ни одного контракта. Примечательно, что одной из компаний, согласившихся все-таки с ними

сотрудничать, руководил Джим Кларк (Jim Clark) — основатель компании Netscape, впоследствии сделавшей очень много для развития языка Java.

Весь 1993 год и половину 1994 года продолжались безрезультатные поиски покупателей продукции, разработанной в рамках проекта “Green”, получившего новое название — “First Person, Inc.”. Патрик Хотон, один из основателей группы, в основном занимавшийся маркетингом, налетал в общей сложности более 300 тысяч миль, пытаясь продать разработанную технологию. Работа над проектом “First Person, Inc.” была прекращена в 1994 г.

Тем временем в рамках Интернета начала развиваться система под названием World Wide Web (Всемирная паутина). Ключевым элементом этой системы был браузер, превращающий гипертекстовые данные в изображение на экране. В 1994 году большинство пользователей применяли некоммерческий веб-браузер Mosaic, разработанный в суперкомпьютерном центре университета штата Иллинойс в 1993 г. Частично этот браузер был написан Марком Андреессеном (Mark Andreessen), подрадившимся работать за 6,85 доллара в час. В то время Марк заканчивал университет, и браузер был его дипломной работой. (Затем он достиг известности и успеха как один из основателей и ведущих специалистов компании Netscape.)

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

Сам браузер был разработан Патриком Хотоном и Джонатаном Пэйном (Johnatan Payne). Позднее он был доработан и получил имя HotJava. Чтобы продемонстрировать все возможности Java, браузер был написан на этом языке. Разработчики браузера HotJava наделили свой продукт способностью выполнять код на веб-страницах. Программный продукт, подтверждавший действенность новой технологии, был представлен 23 мая 1995 года на выставке SunWorld '95 и вызвал всеобщий интерес к Java, сохраняющийся и по сей день.

Компания Sun Microsystems выпустила первую версию Java в начале 1996 г. Пользователи быстро поняли, что версия Java 1.0 не подходит для разработки серьезных приложений. Конечно, эту версию можно применять для реализации визуальных эффектов на веб-страницах, например, написать апплет, выводящий на страницу случайно “прыгающий” текст, но версия Java 1.0 была еще сырой. В ней даже отсутствовали средства вывода на печать. Грубо говоря, версия Java 1.0 еще не была готова. В следующей версии, Java 1.1, были устранены наиболее очевидные недостатки, улучшены средства рефлексии и реализована новая модель событий для программирования графических пользовательских интерфейсов (GUI). Но, несмотря на это, ее возможности были все еще ограничены.

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

Кроме Standard Edition, были предложены еще два варианта: Micro Edition (“микроредакция”) для портативных устройств, например для мобильных телефонов, и Enterprise Edition (редакция для корпоративных приложений). В этой книге основное внимание уделяется редакции Standard Edition.

Версии 1.3 и 1.4 редакции Standard Edition стали результатом поэтапного усовершенствования первоначально выпущенной версии Java 2. Они обладали новыми возможностями, повышенной производительностью и, разумеется, содержали намного меньше ошибок. В процессе развития Java многие взгляды на апплеты и клиентские приложения были пересмотрены. В частности, оказалось, что на Java удобно разрабатывать высококачественные серверные приложения.

В версии 5.0 язык Java подвергся наиболее существенной модификации с момента выпуска версии 1.1. (Первоначально версия 5.0 имела номер 1.5, но на конференции JavaOne в 2004 г. была принята новая нумерация версий.) После многолетних исследований были добавлены обобщенные типы (которые приблизительно соответствуют шаблонам C++), хотя при этом не были выдвинуты требования модификации виртуальной машины. Ряд других языковых элементов, например, циклы в стиле `for each`, автоупаковка и аннотации, были явно “навеяны” языком C#.

Версия 6 (без суффикса .0) была выпущена в конце 2006 г. Опять же сам язык не претерпел существенных изменений, но были внесены усовершенствования, связанные с производительностью, а также произведены расширения библиотек.

По мере того как в центрах обработки данных все чаще стали применяться аппаратные средства широкого потребления вместо специализированных серверов, для компании Sun Microsystems наступили тяжелые времена, и в конечном итоге в 2009 году она была приобретена компанией Oracle. Разработка последующих версий Java приостановилась на долгое время. И только в 2011 году компания Oracle выпустила новую версию Java 7 с простыми усовершенствованиями.

В 2014 году была выпущена версия Java 8 с наиболее существенными изменениями почти за двадцать лет существования этого языка. В версии Java 8 предпринята попытка внедрить стиль функционального программирования, чтобы упростить выражение вычислений, которые могут выполняться параллельно. Все языки программирования должны развиваться, чтобы оставаться актуальным, и в этом отношении язык Java проявил себя с наилучшей стороны.

История главного нововведения в версии Java 9 относится еще к 2008 году, когда Марк Рейнгольд (Mark Reinhold), главный разработчик платформы Java, предпринял усилия, чтобы каким-то образом справиться с громоздкой, монолитной платформой Java. Чтобы добиться поставленной цели, пришлось внедрить *модули* — самостоятельные блоки кода, предоставляющие конкретные функциональные возможности. На разработку и реализацию модульной системы, вполне вписывающейся в платформу Java, ушло одиннадцать лет, но еще предстоит убедиться, насколько хорошо она подходит для приложений и библиотек Java. Версия Java 9, выпущенная в 2017 году, обладает и другими привлекательными языковыми средствами, описываемыми в настоящем издании.

Начиная с 2018 года, версии Java выпускаются через каждые шесть месяцев, чтобы ускорить внедрение новых языковых средств. Хотя некоторые версии (например, Java 11) рассчитаны на поддержку в долгосрочной перспективе.

В табл. 1.1 сведены данные об этапах развития языка и библиотек Java. Как видите, размеры прикладного программного интерфейса API значительно увеличились.

Таблица 1.1. Этапы развития языка Java

Версия	Год выпуска	Новые языковые средства	Количество классов и интерфейсов
1.0	1996	Выпуск самого языка	211
1.1	1997	Внутренние классы	477
1.2	1998	Отсутствуют	1524
1.3	2000	Отсутствуют	1840
1.4	2002	Утверждения	2723
5.0	2004	Обобщенные классы, цикл в стиле <code>for each</code> , автоупаковка, аргументы переменной длины, метаданные, перечисления, статический импорт	3279
6	2006	Отсутствуют	3793
7	2009	Оператор <code>switch</code> со строковыми метками ветвей, ромбовидный оператор, двоичные литералы, усовершенствованная обработка исключений	4024
8	2014	Лямбда-выражения, интерфейсы с методами по умолчанию, потоки данных и библиотеки даты и времени	4240
9	2017	Модули, различные усовершенствования языка и библиотек	6005

1.5. Распространенные заблуждения относительно Java

В завершение этой главы перечислим некоторые распространенные заблуждения, связанные с языком Java.

Язык Java — это расширение языка HTML.

Java — это язык программирования, а HTML — способ описания структуры веб-страниц. У них нет ничего общего, за исключением дескрипторов HTML-разметки, позволяющих размещать на веб-страницах апплеты, написанные на Java.

Я пользуюсь XML, и поэтому мне не нужен язык Java.

Java — это язык программирования, а XML — средство описания данных. Данные, представленные в формате XML, можно обрабатывать программными средствами, написанными на любом языке программирования, но лишь в прикладном программном интерфейсе Java API содержатся превосходные средства для обработки таких данных. Кроме того, на Java реализованы многие программы независимых производителей, предназначенные для работы с XML-документами. Более подробно этот вопрос обсуждается во втором томе данной книги.

Язык Java легко выучить.

Нет ни одного языка программирования, сопоставимого по функциональным возможностям с Java, который можно было бы легко выучить. Простейшие программы написать несложно, но намного труднее выполнить серьезные задания. Обратите также внимание на то, что в этой книге обсуждению самого языка Java посвящено лишь несколько глав. А в остальных главах рассматривается работа с библиотеками, содержащими тысячи классов и интерфейсов, а также многие тысячи методов. Правда, вам

совсем не обязательно помнить каждый из них, но все же нужно знать достаточно много, чтобы применять Java на практике.

Java со временем станет универсальным языком программирования для всех платформ.

Теоретически это возможно, но на практике существуют области, где вполне укоренились другие языки программирования. Так, язык Objective C и его последователь Swift трудно заменить на платформе iOS для мобильных устройств. Все, что происходит в браузере, обычно находится под управлением сценариев на JavaScript. Прикладные программы для Windows написаны на C++ или C#. А язык Java лучше всего подходит для разработки серверных и межплатформенных клиентских приложений.

Java — это всего лишь очередной язык программирования.

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

Почему так происходит? Успех любого языка программирования определяется в большей степени его *системной поддержкой*, чем изяществом его синтаксиса. Существуют ли стандартные библиотеки, позволяющие сделать именно то, что требуется программисту? Разработана ли удобная среда для создания и отладки программ? Интегрирован ли язык и его инструментарий в остальную вычислительную инфраструктуру? Язык Java достиг успехов в области серверных приложений, поскольку его библиотеки классов позволяют легко сделать то, что раньше было трудно реализовать, например, поддерживать работу в сети и организовать многопоточную обработку. Тот факт, что язык Java способствовал сокращению количества ошибок, связанных с указателями, также свидетельствует в его пользу. Благодаря этому производительность труда программистов повысилась. Но не в этом кроется причина его успеха.

Java является патентованным средством, поэтому пользоваться им не рекомендуется.

Сразу же после создания Java компания Sun Microsystems предоставляла бесплатные лицензии распространителям и конечным пользователям. Несмотря на то что компания Sun Microsystems полностью контролировала распространение Java, в работу над этим языком и его библиотеками были вовлечены многие другие компании. Исходный код виртуальной машины и библиотек доступен всем желающим, но его можно использовать лишь для ознакомления, а вносить в него изменения запрещено. До настоящего момента язык Java имел “закрытый исходный код, но прекрасно работал”.

Ситуация изменилась кардинально в 2007 году, когда компания Sun Microsystems объявила, что последующие версии Java будут доступны на условиях General Public License (GPL) — той же лицензии открытого кода, по которой распространяется ОС Linux. Компания Oracle проявила свою приверженность к сохранению открытости кода Java, но с одной важной оговоркой — патентованием. Всякий может получить патент на использование и видоизменение кода Java по лицензии GPL, но только на настольных и серверных платформах. Так, если вы желаете использовать Java во встроенных системах, для этой цели вам потребуется другая лицензия, которая, скорее всего, повлечет за собой определенные отчисления. Впрочем, срок действия таких патентов истекает в следующем десятилетии, после чего язык Java станет совершенно бесплатным.

Программы на Java работают под управлением интерпретатора, а следовательно, серьезные приложения будут выполняться слишком медленно.

В начале развития Java программы на этом языке действительно интерпретировались. Теперь в состав всех виртуальных машин Java входит динамический компилятор. Критические участки кода будут выполняться не медленнее, чем если бы они были написаны на C++, а в некоторых случаях даже быстрее.

Все программы на Java выполняются на веб-страницах.

Все апплеты, написанные на Java, действительно выполняются в окне веб-браузера. Но большинство программ на Java являются независимыми приложениями, которые никак не связаны с веб-браузером. Фактически многие программы на Java выполняются на веб-серверах и генерируют код для веб-страниц.

Программы на Java представляют большую опасность для системы защиты.

В свое время было опубликовано несколько отчетов об ошибках в системе защиты Java. Большинство из них касалось реализаций языка Java в конкретных браузерах. Исследователи восприняли это как вызов и принялись искать глобальные недостатки в системе защиты Java, чтобы доказать неадекватность модели безопасности апплетов. Но их усилия не увенчались успехом. Обнаруженные ошибки в конкретных реализациях вскоре были исправлены. В дальнейшем предпринимались и более серьезные попытки взлома системы защиты Java, на которые сначала в компании Sun Microsystems, а затем и в компании Oracle реагировали слишком медленно. Намного оперативнее (и даже слишком оперативно) отреагировали производители браузеров, запретив выполнение прикладного кода на Java по умолчанию. Чтобы оценить значение этого факта, вспомните о миллионах вирусных атак из исполняемых файлов в Windows и макросы редактора Word, действительно вызвавшие немало хлопот. При этом критика недостатков самой платформы для совершения подобных атак была поразительно беззубой. И даже через двадцать лет после создания платформа Java остается намного более безопасной, чем любая другая платформа, доступная для выполнения прикладного кода.

Язык JavaScript — упрощенная версия Java.

JavaScript — это язык сценариев, которые можно использовать на веб-страницах. Он был разработан компанией Netscape и сначала назывался LiveScript. Синтаксис JavaScript напоминает синтаксис Java, но на этом их сходство и заканчивается (за исключением названия, конечно). В частности, Java — *строго типизированный язык*, а его компилятор перехватывает многие ошибки, возникающие в связи с неверным употреблением типов. А в языке JavaScript подобные ошибки обнаруживаются лишь на стадии выполнения программы, что намного затрудняет их устранение.

Пользуясь Java, можно заменить компьютер недорогим устройством для доступа к Интернету.

Когда появился язык Java, некоторые были уверены, что так и случится. И хотя в продаже появились сетевые компьютеры, оснащенные средствами Java, пользователи не спешат заменить свои мощные и удобные персональные компьютеры устройствами без жестких дисков с ограниченными возможностями. Ныне основными вычислительными средствами для многих конечных пользователей стали мобильные телефоны и планшетные компьютеры. Большая часть этих мобильных устройств работает на платформе Android, производной от Java. Поэтому вам будет легче программировать на платформе Android, научившись программировать на Java.

Среда программирования на Java

В этой главе...

- ▶ Установка комплекта Java Development Kit
- ▶ Выбор среды для разработки программ
- ▶ Использование инструментов командной строки
- ▶ Применение интегрированной среды разработки
- ▶ Описание утилиты JShell

Из этой главы вы узнаете, как устанавливать комплект инструментальных средств разработки Java Development Kit (JDK), а также компилировать и запускать на выполнение разнотипные программы: консольные программы, графические и веб-приложения. Мы будем пользоваться инструментальными средствами JDK, набирая команды в окне командной оболочки. Но многие программисты предпочитают удобства, предоставляемые интегрированной средой разработки (IDE). В этой главе будет показано, как пользоваться бесплатно доступной IDE для компиляции и выполнения программ, написанных на Java. Освоить IDE и пользоваться ими нетрудно, но они долго загружаются и предъявляют большие требования к вычислительным ресурсам компьютера, так что применять их для разработки небольших программ не имеет смысла. Овладев приемами, рассмотренными в этой главе, и выбрав подходящие инструментальные средства для разработки программ, вы можете перейти к главе 3, с которой, собственно, начинается изучение языка Java.

2.1. Установка комплекта Java Development Kit

Наиболее полные и современные версии комплекта Java Development Kit (JDK) от компании Oracle доступны для операционных систем Solaris, Linux, Mac OS X

и Windows. Версии, находящиеся на разных стадиях разработки для многих других платформ, лицензированы и поставляются производителями соответствующих платформ.

2.1.1. Загрузка комплекта JDK

Для загрузки комплекта Java Development Kit на свой компьютер вам нужно перейти на веб-страницу по адресу <http://www.oracle.com/technetwork/java/javase/downloads/index.html>, приложив немного усилий, чтобы разобраться в обозначениях и сокращениях и найти нужное программное обеспечение. И в этом вам помогут сведения, приведенные в табл. 2.1.

Таблица 2.1. Обозначения и сокращения программных средств Java

Наименование	Сокращение	Пояснение
Java Development Kit	JDK	Программное обеспечение для тех, кто желает писать программы на Java
Java Runtime Environment	JRE	Программное обеспечение для потребителей, желающих выполнять программы на Java
Standard Edition	SE	Платформа Java для применения в настольных системах и простых серверных приложениях
Enterprise Edition	EE	Платформа Java для сложных серверных приложений
Micro Edition	ME	Платформа Java для применения в мобильных устройствах
JavaFX	—	Альтернативный комплект инструментальных средств для построения GUI, входивший в дистрибутив Java SE от компании Oracle до версии Java 11
OpenJDK	—	Бесплатная реализация Java SE с открытым кодом, входящая в дистрибутив Java SE от компании Oracle
Java 2	J2	Устаревшее обозначение версий Java, выпущенных в 1998–2006 гг.
Software Development Kit	SDK	Устаревшее обозначение версий JDK, выпущенных в 1998–2006 гг.
Update	u	Обозначение, принятое в компании Oracle для выпусков с исправлениями ошибок вплоть до версии Java 8
NetBeans	—	Интегрированная среда разработки от компании Oracle

Сокращение JDK вам уже знакомо. Оно, как нетрудно догадаться, означает Java Development Kit, т.е. комплект инструментальных средств для разработки программ на Java. Некоторые трудности может вызвать тот факт, что в версиях 1.2–1.4 этот пакет называется Java SDK (Software Development Kit). Иногда вы встретите ссылки на старый термин. Вплоть до главы 10 упоминается также среда Java Runtime Environment (JRE), включающая в себя виртуальную машину, хотя и без компилятора. Но для разработчиков она не подходит, а предназначена для конечных пользователей программ на Java, которым компилятор ни к чему. Далее вам будет встречаться обозначение Java SE. Оно означает Java Standard Edition, т.е. стандартную редакцию Java, в отличие от редакций Java EE (Enterprise Edition) для предприятий и Java ME (Micro Edition) для встроженных устройств.

Иногда вам может встретиться обозначение Java 2, которое было введено в 1998 г., когда специалисты компании Sun Microsystems по маркетингу поняли, что очередной дробный номер выпуска никак не отражает глобальных отличий между JDK 1.2

и предыдущими версиями. Но поскольку такое мнение сформировалось уже после выхода в свет JDK, то было решено, что номер версии 1.2 останется за *комплектom разработки*. Последующие выпуски JDK имеют номера 1.3, 1.4 и 5.0. *Платформа* же была переименована с Java на Java 2. Таким образом, данный комплект разработки называется Java 2 Standard Edition Software Development Kit Version 5.0, или J2SE SDK 5.0.

Правда, в 2006 году нумерация версий была упрощена. Следующая версия Java Standard Edition получила название Java SE 6, а последовавшие за ней версии — Java SE 7 и Java SE 8. Но в то же время они получили “внутренние” номера 1.6.0, 1.7.0 и 1.8.0 соответственно. Эта незначительная путаница в обозначениях дошла и до версии Java SE 9, когда был сначала присвоен номер версии 9, а затем 9.0.1. (А почему не 9.0.0 для первоначальной версии? Еще любопытнее то обстоятельство, что в обозначении номера версии требуется опускать конечные нули, чтобы оставить недолговечный промежуток между главной версией и первым ее обновлением для системы безопасности.)



НА ЗАМЕТКУ! В остальной части этой книги сокращение SE опущено. Поэтому Java 9, по существу, означает Java SE 9.

До версии Java 9 существовали 32- и 64-разрядные версии комплекта Java Development Kit. Но теперь 32-разрядные версии в компании Oracle больше не выпускаются. Чтобы пользоваться современной версией комплекта JDK от компании Oracle, придется установить 64-разрядную версию соответствующей операционной системы.

В Linux необходимо сделать выбор между файлом RPM и архивным файлом с расширением **.tar.gz**. Предпочтение лучше отдать последнему, чтобы распаковать архив в любом удобном месте. Итак, выбирая подходящий комплект JDK, необходимо принять во внимание следующее.

- Для дальнейшей работы потребуется комплект JDK (Java SE Development Kit), а не JRE.
- В Linux лучше выбрать архивный файл с расширением **.tar.gz**.

Сделав выбор, примите условия лицензионного соглашения и загрузите файл с выбранным комплектом JDK.



НА ЗАМЕТКУ! Компания Oracle предлагает комплект, в который входит комплект инструментальных средств разработки Java Development Kit и интегрированная среда разработки NetBeans. Рекомендуется пока что воздержаться от всех подобных комплектов, установив только Java Development Kit. Если в дальнейшем вы решите воспользоваться NetBeans, загрузите эту IDE по адресу <https://netbeans.org>.

2.1.2. Установка комплекта JDK

После загрузки JDK нужно установить этот комплект и выяснить, где он был установлен, поскольку эти сведения понадобятся в дальнейшем. Ниже вкратце поясняется порядок установки JDK на разных платформах.

- Если вы работаете в Windows, запустите на выполнение программу установки. Вам будет предложено место для установки JDK. Рекомендуется не принимать предлагаемый каталог. По умолчанию это каталог `c:\Program Files\Java\jdk-11.0.x`. Удалите Program Files из пути для установки данного пакета.

- Если вы работаете в Mac OS, запустите на выполнение стандартный установщик. Он автоматически установит программное обеспечение в каталоге `/Library/Java/JavaVirtualMachines/jdk-11.0.x.jdk/Contents/Home`. Найдите этот каталог с помощью утилиты Finder.
- Если вы работаете в Linux, распакуйте архивный файл с расширением `.tar.gz`, например, в своем рабочем каталоге или же в каталоге `/opt`. А если вы устанавливаете комплект JDK из файла RPM, убедитесь в том, что он установлен в каталоге `/usr/java/jdk-11.0.x`.

В этой книге делаются ссылки на каталог `jdk`, содержащий комплект JDK. Так, если в тексте указана ссылка `jdk/bin`, она обозначает обращение к каталогу `/opt/jdk-11.0.4/bin` или `c:\Java\jdk-11.0.4\bin`.

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

- В Linux добавьте приведенную ниже строку в конце файла `~/.bashrc` или `~/.bash_profile`.

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

Непрерывно укажите правильный путь к JDK, например `/opt/jdk-11.0.4`.

В ОС Windows 10 введите **environment** (окружение) в поле поиска диалогового окна Windows Settings (Параметры Windows) и установите флажок **Edit environment variables for your account** (Править переменные окружения для вашей учетной записи; рис. 2.1). В итоге должно появиться диалоговое окно **Environment Variables** (Переменные окружения), которое может быть скрыто за диалоговым окном **Windows Settings**. Если же вам не удастся найти его, попробуйте выполнить команду **sysdm.cpl** из диалогового окна **Run** (Выполнить), которое открывается нажатием клавиши `<Windows+R>`, а затем выберите вкладку **Advanced** (Дополнительно) и щелкните на кнопке **Environment Variables**. Найдите и выберите переменную **Path** из списка **User Variables** (Пользовательские переменные). Щелкните на кнопке **Edit** (Править), а затем на кнопке **New** (Создать) и введите элемент с каталогом `jdk\bin` (рис. 2.2). Сохраните сделанные установки. В итоге подсказка в любом вновь открытом окне командной строки будет начинаться с правильного пути.

Правильность сделанных установок можно проверить следующим образом. Откройте окно терминала, или командной строки, или оболочки. Как вы это сделаете, зависит от операционной системы. Введите следующую строку:

```
java -version
```

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

```
javac 9.0.4
```

Если вместо этого появится сообщение вроде `"java:command not found"` (`java:command не найдено`) или `"The name specified is not recognized as an internal or external command, operable program or batch file"` (Указанное имя не распознано ни как внутренняя или внешняя команда, ни как действующая программа или командный файл), следует еще раз проверить, правильно ли выполнена установка и задан путь к JDK.



Рис. 2.1. Установка свойств системы в Windows 10

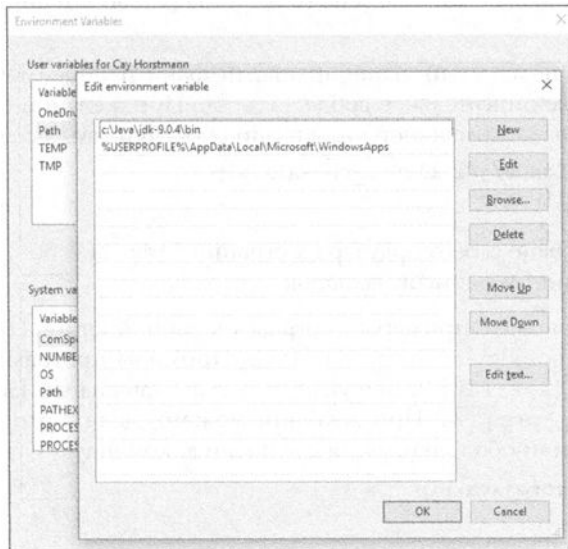


Рис. 2.2. Установка переменной окружения **Path** в Windows 10

2.1.3. Установка библиотек и документации

Исходные файлы библиотек поставляются в комплекте JDK в виде архива, хранящегося в файле `src.zip`. Распакуйте этот файл, чтобы получить доступ к исходному коду. С этой целью выполните следующие действия.

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

2. Создайте каталог `javasrc` в своем начальном каталоге. При желании можете сделать это в окне терминала, командной строки или оболочки, введя следующую команду:

```
mkdir javasrc
```

3. Найдите архивный файл `src.zip` в каталоге `jdk/lib`.
4. Распакуйте архивный файл `src.zip` в каталог `javasrc`. При желании можете сделать это в окне терминала, командной строки или оболочки, введя следующие команды:

```
cd javasrc
jar xvf jdk/src.zip
cd ..
```



СОВЕТ. Архивный файл `src.zip` содержит исходный код всех общедоступных библиотек. Чтобы получить дополнительный исходный код (компилятора, виртуальной машины, собственных методов и закрытых вспомогательных классов), посетите веб-страницу по адресу <http://openjdk.java.net>.

Документация содержится в отдельном от JDK архиве. Ее можно загрузить по адресу <http://www.oracle.com/technetwork/java/javase/downloads/index.html>. Для этого выполните следующие действия.

1. Загрузите архивный файл документации под названием `jdk-11.0.x_doc-all.zip`.
2. Распакуйте упомянутый выше архивный файл и переименуйте каталог `doc` на нечто более описательное вроде `javadoc`. При желании можете сделать это в окне терминала, командной строки или оболочки, введя следующие команды:

```
jar xvf Downloads/jdk-11.0.x_doc-all.zip
mv docs jdk-11-docs
```

3. Перейдите в окне своего браузера к странице `jdk-11-docs/index.html` и введите эту страницу в список закладок.

Кроме того, установите примеры программ к данной книге. Их можно загрузить по адресу <http://horstmann.com/corejava>. Примеры программ упакованы в архивный файл `corejava.zip`. Распакуйте их в свой начальный каталог. Они расположатся в каталоге `corejava`. При желании можете сделать это в окне терминала, командной строки или оболочки, введя следующую команду:

```
jar xvf Downloads/corejava.zip
```

2.2. Применение инструментов командной строки

Если у вас имеется опыт работы в IDE Microsoft Visual Studio, значит, вы уже знакомы со средой разработки, состоящей из встроенного текстового редактора, меню для компиляции и запуска программ, а также отладчика. В комплект JDK не входят средства, даже отдаленно напоминающие интегрированную среду разработки (IDE). Все команды выполняются из командной строки. И хотя такой подход к разработке программ на Java может показаться обременительным, мастерское владение им является весьма существенным навыком. Если вы устанавливаете платформу Java впервые,

вам придется найти и устранить выявленные неполадки, прежде чем устанавливать IDE. Но выполняя даже самые элементарные действия самостоятельно, вы получаете лучшее представление о внутреннем механизме работы IDE.

После того как вы освоите самые элементарные действия для компиляции и выполнения программ на Java, вам, скорее всего, потребуется IDE. Подробнее об этом речь пойдет в следующем разделе.

Выберем сначала самый трудный путь, вызывая компилятор и запуская программы на выполнение из командной строки.

1. Откройте окно командной оболочки.
2. Перейдите к каталогу `corejava/v1ch02/Welcome`. (Напомним, что каталог `corejava` был специально создан для хранения исходного кода примеров программ из данной книги, как пояснялось в разделе 2.1.3.)
3. Введите следующие команды:

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

На экране должен появиться результат, приведенный на рис. 2.3.

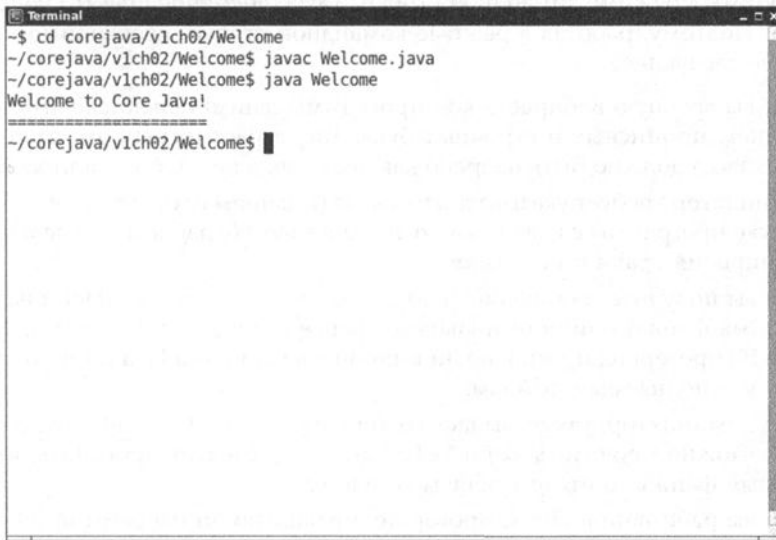


Рис. 2.3. Компиляция и выполнение программы `Welcome.java`

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

Что же произошло? Служебная программа (иначе — утилита) `javac` — это компилятор Java. Она скомпилировала исходный код из файла `Welcome.java` и преобразовала его в байт-код, сохранив последний в файле `Welcome.class`. А утилита `java` запускает виртуальную машину Java. Она выполняет байт-код, который компилятор поместил в указанный файл с расширением `.class`.

Программа `Welcome` очень проста и лишь выводит сообщение на экран. Исходный код этой программы приведен в листинге 2.1, а о том, как она работает, речь пойдет в следующей главе.

Листинг 2.1. Исходный код из файла `Welcome.java`

```
1  /**
2   * Эта программа отображает приветствие автора книги
3   * @version 1.30 2014-02-27
4   * @author Cay Horstmann
5   */
6  public class Welcome
7  {
8      public static void main(String[] args)
9      {
10         String greeting = "Welcome to Core Java!";
11         System.out.println(greeting);
12         for (int i = 0; i < greeting.length(); i++)
13             System.out.print("=");
14         System.out.println();
15     }
16 }
17 }
```

В эпоху визуальных сред разработки программ многие программисты просто не умеют работать в режиме командной строки. Такое неумение чревато неприятными ошибками. Поэтому, работая в режиме командной строки, необходимо принимать во внимание следующее.

- Если вы вручную набираете код программы, внимательно следите за употреблением прописных и строчных букв. Так, в рассмотренном выше примере имя класса должно быть набрано как `Welcome`, а не `welcome` или `WELCOME`.
- Компилятор требует указывать *имя файла* (в данном случае `Welcome.java`). При запуске программы следует указывать *имя класса* (в данном случае `Welcome`) без расширения `.java` или `.class`.
- Если вы получите сообщение "Bad command or file name" (Неверная команда или имя файла) или упоминавшееся ранее сообщение "javac:command not found", проверьте, правильно ли выполнена установка Java и верно ли указаны пути к исполняемым файлам.
- Если компилятор `javac` выдаст сообщение "cannot read: Welcome.java" (невозможно прочитать файл `Welcome.java`), следует проверить, имеется ли нужный файл в соответствующем каталоге.
- Если вы работаете в Linux, проверьте, правильно ли набраны прописные буквы в имени файла `Welcome.java`. А в Windows просматривайте содержимое каталогов по команде `dir`, а не средствами графического интерфейса Проводника Windows. Некоторые текстовые редакторы (в частности, Notepad) сохраняют текст в файлах с расширением `.txt`. Если вы пользуетесь таким редактором для редактирования содержимого файла `Welcome.java`, он сохранит его в файле `Welcome.java.txt`. По умолчанию Проводник Windows скрывает расширение `.txt`, поскольку оно предполагается по умолчанию. В этом случае следует переименовать файл, воспользовавшись командой `ren`, или повторно сохранить его, указав имя в кавычках, например "Welcome.java".
- Если при запуске программы вы получаете сообщение об ошибке типа `java.lang.NoClassDefFoundError`, проверьте, правильно ли вы указали имя файла.
- Если вы получите сообщение касательно имени `welcome`, начинающегося со строчной буквы `w`, еще раз выполните команду `java Welcome`, написав это имя

с прописной буквы **W**. Не забывайте, что в Java учитывается регистр символов. Если же вы получите сообщение по поводу ввода имени `Welcome/java`, значит, вы случайно ввели команду `java Welcome.java`. Повторите команду `java Welcome`.

- Если вы указали имя `Welcome`, а виртуальная машина не в состоянии найти класс с этим именем, проверьте, не установлена ли каким-нибудь образом переменная окружения `CLASSPATH` в вашей системе. Эту переменную, как правило, не стоит устанавливать глобально, но некоторые неудачно написанные установщики программного обеспечения в Windows это делают. Последуйте той же самой процедуре, что и для установки переменной окружения `PATH`, но на этот раз удалите установку переменной окружения `CLASSPATH`.



СОВЕТ. Отличное учебное пособие имеется по адресу <https://docs.oracle.com/javase/tutorial/getStarted/cupojava>. В нем подробно описываются скрытые препятствия, которые нередко встречаются на пути начинающих программировать на Java.



НА ЗАМЕТКУ! В версии JDK 11 вместо команды `javac` с единственным исходным файлом можно пользоваться сценариями оболочки, начинающимися со строки `#!/path/to/java` с шебангом (т.е. со знаками решетки и восклицания в самом начале).

Программа `Welcome` не особенно впечатляет. Поэтому рассмотрим пример графического приложения. Это приложение представляет собой очень простую программу, которая загружает и выводит на экран изображение из файла. Как и прежде, скомпилируем и выполним это приложение в режиме командной строки.

1. Откройте окно терминала или командной оболочки.
2. Перейдите к каталогу `corejava/v1ch02/ImageViewer`.
3. Введите следующие команды:

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

На экране появится новое окно приложения `ImageViewer`. Выберите команду меню `File⇒Open` (Файл⇒Открыть) и найдите файл изображения, чтобы открыть его. (В одном каталоге с данной программой находится несколько графических файлов.) Изображение появится в окне (рис. 2.4). Чтобы завершить выполнение программы, щелкните на кнопке `Close` (Закреть) в строке заголовка текущего окна или выберите команду меню `File⇒Exit` (Файл⇒Выйти).

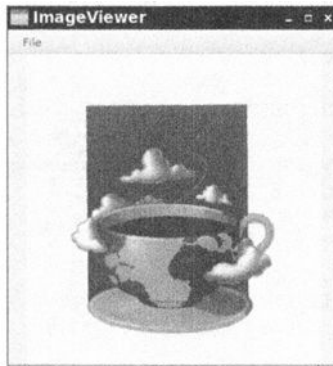


Рис. 2.4. Окно выполняющегося приложения `ImageViewer`

Бегло просмотрите исходный код данной программы, приведенный в листинге 2.2. Эта программа заметно длиннее, чем первая, но и она не слишком сложна, особенно если представить себе, сколько строк кода на языке С или С++ нужно было бы написать, чтобы получить такой же результат. Написанию приложений с графическим пользовательским интерфейсом (GUI), подобных данной программе, посвящена глава 10.

Листинг 2.2. Исходный код из файла **ImageViewer/ImageViewer.java**

```
1  import java.awt.*;
2  import java.io.*;
3  import javax.swing.*;
4
5  /**
6   * Программа для просмотра изображений.
7   * @version 1.31 2018-04-10
8   * @author Cay Horstmann
9   */
10 public class ImageViewer
11 {
12     public static void main(String[] args)
13     {
14         EventQueue.invokeLater(() -> {
15             var frame = new ImageViewerFrame();
16             frame.setTitle("ImageViewer");
17             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18             frame.setVisible(true);
19         });
20     }
21 }
22
23 /**
24 * Фрейм с текстовой меткой для вывода изображения.
25 */
26 class ImageViewerFrame extends JFrame
27 {
28     private static final int DEFAULT_WIDTH = 300;
29     private static final int DEFAULT_HEIGHT = 400;
30
31     public ImageViewerFrame()
32     {
33         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34
35         // использовать метку для вывода изображений на экран
36         var label = new JLabel();
37         add(label);
38
39         // установить селектор файлов
40         var chooser = new JFileChooser();
41         chooser.setCurrentDirectory(new File("."));
42
43         // установить строку меню
44         var menuBar = new JMenuBar();
```

```
45     setJMenuBar(menuBar);
46
47     var menu = new JMenu("File");
48     menuBar.add(menu);
49
50     var openItem = new JMenuItem("Open");
51     menu.add(openItem);
52     openItem.addActionListener(event -> {
53         // отобразить диалоговое окно селектора файлов
54         int result = chooser.showOpenDialog(null);
55
56         // если файл выбран, задать его в качестве
57         // пиктограммы для метки
58         if (result == JFileChooser.APPROVE_OPTION)
59         {
60             String name = chooser.getSelectedFile().getPath();
61             label.setIcon(new ImageIcon(name));
62         }
63     });
64
65     var exitItem = new JMenuItem("Exit");
66     menu.add(exitItem);
67     exitItem.addActionListener(event -> System.exit(0));
68 }
69 }
```

2.3. Применение IDE

В предыдущем разделе было показано, каким образом программа на Java компилируется и выполняется из командной строки. И хотя это очень полезный навык, для повседневной работы следует выбрать интегрированную среду разработки (IDE). Такие среды стали настолько эффективными и удобными, что профессионально разрабатывать программное обеспечение без их помощи просто не имеет смысла. К числу наиболее предпочтительных относятся IDE Eclipse, NetBeans и IntelliJ IDEA. В этой главе будет показано, как приступить к работе с IDE Eclipse. Но вы вольны выбрать другую IDE для проработки материала данной книги.

Прежде всего загрузите IDE Eclipse по адресу <http://www.eclipse.org/downloads/>, где имеются версии IDE Eclipse для Linux, Mac OS X, Solaris и Windows. Выполните программу установки и выберите установочный набор Eclipse IDE for Java Developers (IDE Eclipse для разработчиков программ на Java).

Чтобы приступить к написанию программы на Java в IDE Eclipse, выполните следующие действия.

1. После запуска Eclipse выберите из меню команду File⇒New Project (Файл⇒Создать проект).
2. Выберите вариант Java Project (Проект Java) в диалоговом окне мастера проектов (рис. 2.5).
3. Щелкните на кнопке Next (Далее). Сбросьте флажок Use default location (Использовать место по умолчанию). Щелкните на кнопке Browse (Обзор) и перейдите к каталогу corejava/vlch02/Welcome (рис. 2.6).

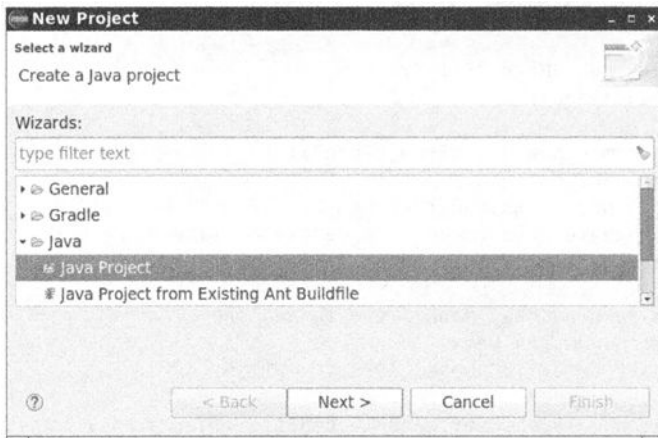


Рис. 2.5. Диалоговое окно Eclipse для создания нового проекта

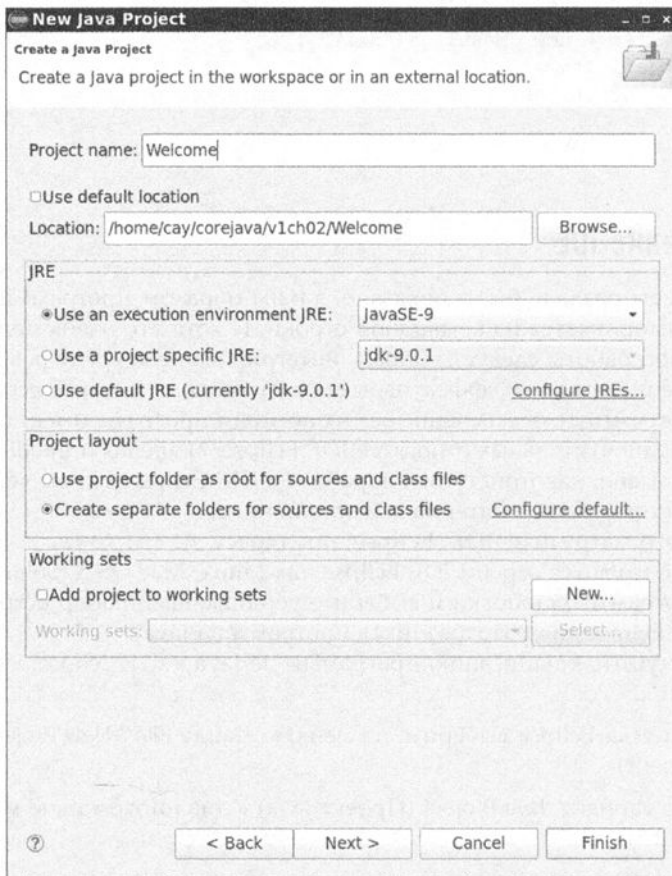


Рис. 2.6. Настройка проекта в Eclipse

- Щелкните на кнопке Finish (Готово). В итоге будет создан новый проект.
- Щелкайте по очереди на треугольных кнопках слева от имени проекта до тех пор, пока не найдете файл `Welcome.java`, а затем дважды щелкните на нем. В итоге появится окно с исходным кодом программы, как показано на рис. 2.7.

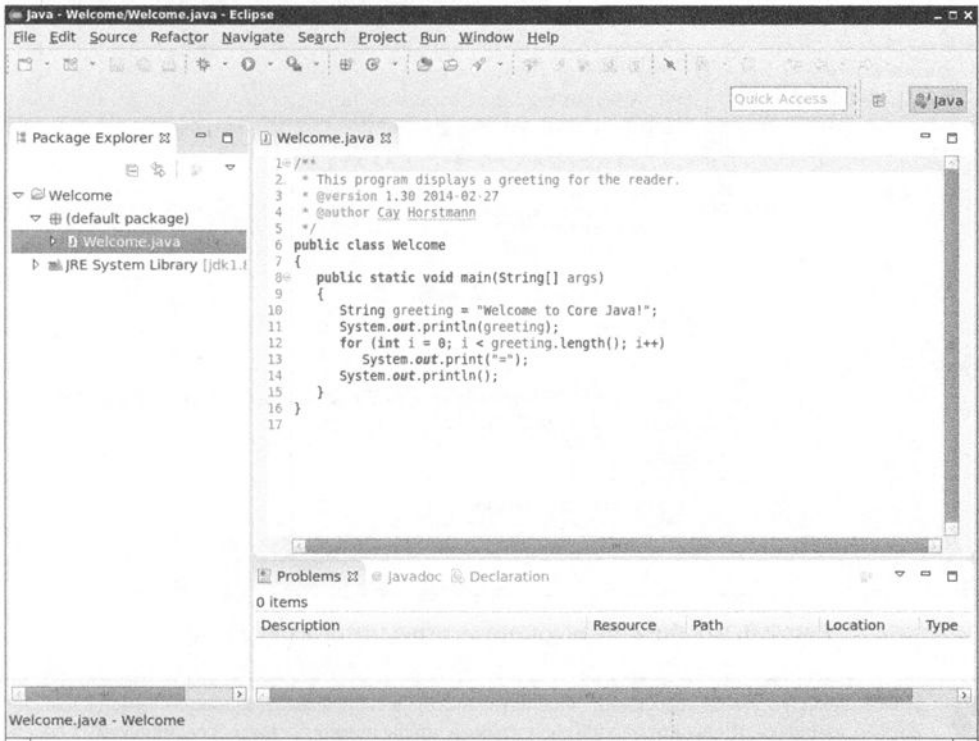


Рис. 2.7. Редактирование исходного кода в Eclipse

- Щелкните правой кнопкой мыши на имени проекта (Welcome) на левой панели. Выберите в открывшемся контекстном меню команду `Run`⇒`Run As`⇒`Java Application` (Выполнить⇒Выполнить как⇒Приложение Java). В нижней части окна появится окно для вывода результатов выполнения программы.

Рассматриваемая здесь программа состоит из нескольких строк кода, и поэтому в ней вряд ли имеются ошибки или опечатки. Но для того, чтобы продемонстрировать порядок обработки ошибок, допустим, что в имени строчного типа данных `String` вместо прописной буквы набрана строчная:

```
string greeting = "Welcome to Core Java!";
```

Обратите внимание на волнистую линию под словом `string`. Перейдите на вкладку `Problems` (Ошибки) ниже исходного кода и щелкайте на треугольных кнопках до тех пор, пока не увидите сообщение об ошибке в связи с неверно указанным типом данных `string` (рис. 2.8). Щелкните на этом сообщении. Курсор автоматически перейдет на соответствующую строку кода в окне редактирования, где вы можете быстро исправить допущенную ошибку.

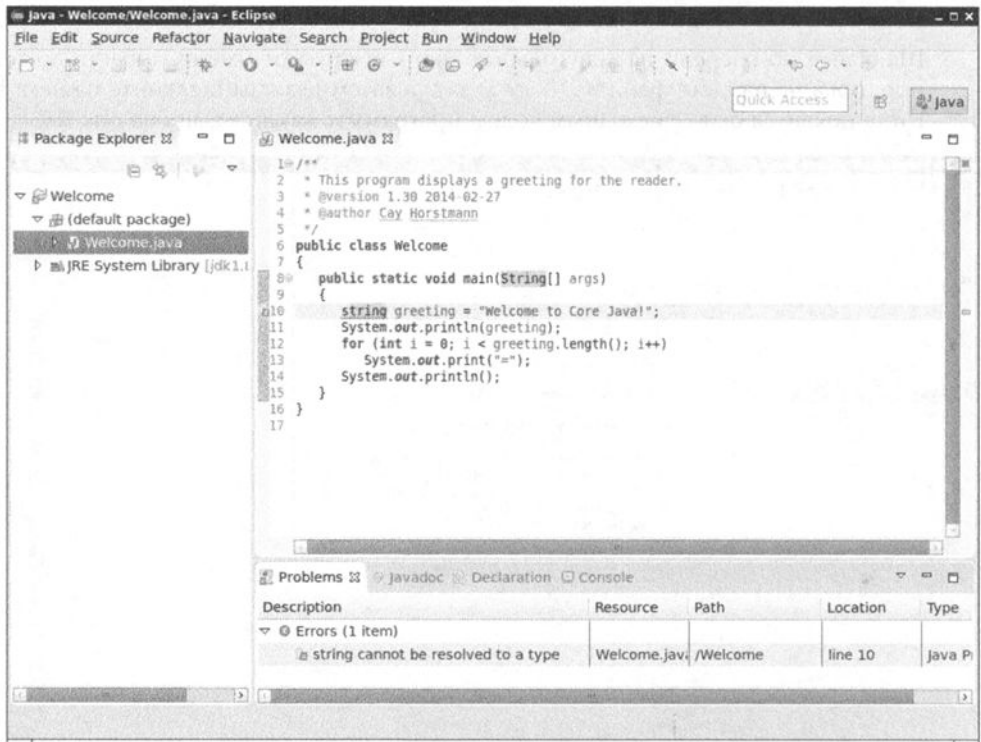


Рис. 2.8. Сообщение об ошибке, выводимое в окне Eclipse



СОВЕТ. Зачастую вывод сообщений об ошибках в Eclipse сопровождается пиктограммой с изображением лампочки. Щелкните на этой пиктограмме, чтобы получить список рекомендуемых способов исправления ошибки.

2.4. Утилита JShell

В предыдущем разделе было показано, как компилировать и выполнять прикладную программу на Java. В версии Java 9 внедрен еще один способ работы на платформе Java. Для этой цели утилита JShell предоставляет цикл “чтение–вычисление–вывод” (REPL). Вы вводите выражение на языке Java, а утилита JShell вычисляет его, выводит получаемый результат и ожидает от вас ввода следующего выражения. Чтобы запустить утилиту JShell на выполнение, достаточно ввести команду `jshell` в окне терминала, командной строки или оболочки (рис. 2.9).

Утилита JShell начинает свое выполнение с приветствия и последующей подсказки:

```
| Welcome to JShell -- Version 9.0.1
| For an introduction type: /help intro1
```

```
jshell>
```

¹ | Добро пожаловать в JShell -- версии 9.0.1
| Для ознакомления введите команду: `/help intro`

Теперь введите следующее выражение:

```
"Core Java".length()
```

```

Terminal ~$
Fichier Édition Affichage Rechercher Terminal Aide

~$ jshell
| Welcome to JShell -- Version 9.0.1
| For an introduction type: /help intro

jshell> "Core Java".length()
$1 ==> 9

jshell> 5 * $1 - 3
$2 ==> 42

jshell> int answer = 6 * 7
answer ==> 42

jshell> Math.
E               IEEEremainder(  PI               abs(
acos(           addExact(      asin(             atan(
atan2(          cbrt(          ceil(             class
copySign(       cos(           cosh(         decrementExact(
exp(            expm1(         floor(        floorDiv(
floorMod(       fma(           getExponent(  hypot(
incrementExact( log(           log10(       log1p(
max(            min(           multiplyExact( multiplyFull(
multiplyHigh(   negateExact(  nextAfter(   nextDown(
nextUp(         pow(           random()     rint(
round(          scalb(        signum(      sin(
sinh(           sqrt(          subtractExact( tan(
tanh(           toDegrees(    toIntExact(  toRadians(
ulp(

jshell> Math.

```

Рис. 2.9. Выполнение утилиты JShell

Утилита JShell ответит на это выводом получаемого результата. В данном случае это количество символов в строке "Core Java", как показано ниже.

```
$1 ==> 9
```

Обратите внимание на то, что для получения такого результата вам *не* нужно вводить оператор `System.out.println`. Утилита JShell автоматически выведет значение каждого вводимого вами выражения.

В выводимом результате `$1` обозначает, что полученный результат доступен для последующих вычислений. Так, если ввести следующее выражение:

```
5 * $1 - 3
```

то в ответ будет получен такой результат:

```
$2 ==> 42
```

Если переменной требуется пользоваться неоднократно, ей можно присвоить более запоминающееся имя. Но для этого придется следовать правилам синтаксиса Java, указав как тип данных, так и имя переменных. (Более подробно синтаксис Java рассматривается в главе 3.) Например:


```
jshell> int answer = 6 * 7
answer ==> 42
```

Еще одним полезным средством является автозаполнение нажатием клавиши табуляции. Чтобы опробовать его, введите

```
Math.
```

и нажмите клавишу табуляции. В итоге будет выведен перечень всех методов, которые могут быть вызваны для переменной `generator`:

```
jshell> Math.
E               IEEEremainder(   PI               abs(
acos(           addExact(        asin(           atan(
atan2(          cbirt(           ceil(           class
copySign(       cos(            cosh(         decrementExact(
exp(            expm1(           floor(        floorDiv(
floorMod(       fma(            getExponent(  hypot(
incrementExact( log(           log10(        log1p(
max(            min(            multiplyExact( multiplyFull(
multiplyHigh(   negateExact(   nextAfter(   nextDown(
nextUp(         pow(            random()     rint(
round(          scalb(          signum(      sin(
sinh(           sqrt(            subtractExact( tan(
tanh(           toDegrees(      toIntExact(  toRadians(
ulp(
```

А теперь введите `1` и снова нажмите клавишу табуляции. Имя метода будет автоматически дополнено до `log`, а в итоге получен более короткий список.

```
jshell> Math.log
log(      log10(      log1p(
```

Вызов метода можно далее заполнить вручную:

```
jshell> Math.log10(0.001)
$3 ==> -3.0
```

Чтобы повторить команду, нажимайте клавишу `<↑>` до тех пор, пока не появится строка, которую требуется выполнить снова или отредактировать. С помощью клавиш `<←>` и `<→>` можно переместить курсор и ввести или удалить символы. Отредактировав строку, нажмите клавишу `<Enter>`. Например, введите числовое значение `0.001` и замените его значением `1000`, а затем нажмите клавишу `<Enter>`:

```
jshell> Math.log10(1000)
$4 ==> 3.0
```

Утилита JShell упрощает и делает интересным изучение языка Java и его библиотек, не прибегая к тяжеловесной среде разработки и не возясь с такими длинными операторами, как `public static void main`.

Итак, в этой главе были рассмотрены механизмы компиляции и запуска программ, написанных на Java. Теперь вы готовы перейти к главе 3, чтобы приступить непосредственно к изучению языка Java.

Основные языковые конструкции Java

В этой главе...

- ▶ Простая программа на Java
- ▶ Комментарии
- ▶ Типы данных
- ▶ Переменные и константы
- ▶ Операции
- ▶ Символьные строки
- ▶ Ввод и вывод
- ▶ Управляющая логика
- ▶ Большие числа
- ▶ Массивы

Будем считать, что вы успешно установили комплект JDK и запустили простые программы, примеры которых были рассмотрены в главе 2. Теперь самое время приступить непосредственно к программированию на Java. Из этой главы вы узнаете, каким образом в Java реализуются основные понятия программирования, в том числе типы данных, условные операторы и циклы.

3.1. Простая программа на Java

Рассмотрим самую простую программу, какую только можно написать на Java. В процессе выполнения она лишь выводит сообщение на консоль.

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

Этому примеру стоит посвятить столько времени, сколько потребуется, чтобы привыкнуть к особенностям языка и рассмотреть характерные черты программ на Java, которые будут еще не раз встречаться во всех приложениях. Прежде всего следует обратить внимание на то, что в языке Java *учитывается регистр букв*. Так, если вы перепутаете их (например, наберете **Main** вместо **main**), рассматриваемая здесь программа выполняться не будет.

А теперь проанализируем исходный код данной программы построчно. Ключевое слово `public` называется *модификатором доступа*. Такие модификаторы управляют обращением к коду из других частей программы. Более подробно этот вопрос будет рассматриваться в главе 5. Ключевое слово `class` напоминает нам, что все элементы программ на Java находятся в составе классов. Классы будут подробно обсуждаться в следующей главе, а до тех пор будем считать их некими контейнерами, в которых реализована программная логика, определяющая порядок работы приложения. Классы являются стандартными блоками, из которых состоят все настольные и веб-приложения, написанные на Java. *Все*, что имеется в программе на Java, должно находиться в пределах класса.

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

Согласно принятым условным обозначениям, имя класса должно начинаться с прописной буквы (именно так сформировано имя `FirstSample`). Если имя состоит из нескольких слов, каждое из них должно начинаться с прописной буквы. (Это так называемое *смешанное написание* в “верблюжьем стиле” — *CamelCase*.)

Файл, содержащий исходный текст, должен называться так же, как и открытый (`public`) класс, и иметь расширение `.java`. Таким образом, код рассматриваемого здесь класса следует разместить в файле `FirstSample.java`. (Как упоминалось выше, регистр букв непременно учитывается, поэтому имя `firstsample.java` для обозначения исходного файла рассматриваемой здесь программы не годится.)

Если вы правильно назвали файл и не допустили ошибок в исходном тексте программы, то в результате компиляции получите файл, содержащий байт-код данного класса. Компилятор Java автоматически назовет этот файл `FirstSample.class` и сохранит его в том же каталоге, где находится исходный файл. Осталось лишь запустить программу на выполнение с помощью загрузчика Java, набрав следующую команду:

```
java FirstSample
```

(Расширение `.class` не указывается!) При выполнении этой программы на экран выводится символьная строка “We will not use 'Hello, World!'” (Мы не будем пользоваться избитой фразой “Здравствуй, мир!”).

Когда для запуска скомпилированной программы на выполнение используется команда

```
java ИмяКласса
```

виртуальная машина Java всегда начинает свою работу с выполнения метода `main()` указанного класса (Термином *метод* в Java принято обозначать функцию.) Следовательно, для нормального выполнения программы в классе должен присутствовать метод `main()`. Разумеется, в класс можно добавить и другие методы и вызывать их из метода `main()`. (Мы покажем, как создавать такие методы, в следующей главе.)



НА ЗАМЕТКУ! В соответствии со спецификацией языка Java (Java Language Specification) метод `main()` должен быть объявлен как `public`. (Спецификация языка Java является официальным документом. Ее можно загрузить по адресу <https://docs.oracle.com/javase/specs>. Некоторые версии загрузчика Java допускали выполнение программ, даже когда метод `main()` не имел модификатора доступа `public`. Эта ошибка была внесена в список замеченных ошибок, доступный на сайте по адресу <https://bugs.sun.com/bugdatabase/index.jsp>, и получила идентификационный номер 4252539. Но она была помечена в 1999 г. меткой "исправлению не подлежит". Разработчики из компании Sun Microsystems разъяснили, что спецификация виртуальной машины Java не требует, чтобы метод `main()` был открытым (см. веб-страницу по адресу <https://docs.oracle.com/javase/specs/jvms/se8/html>), а попытка исправить эту ошибку "может вызвать потенциальные осложнения". К счастью, здравый смысл в конечном итоге возобладал. Загрузчик Java в версии Java SE 1.4 требует, чтобы метод `main()` был открытым (`public`).)

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

Обратите внимание на фигурные скобки в исходном коде рассматриваемой здесь программы. Как и в C/C++, в Java фигурные скобки используются для выделения *блоков* программы. Код любого метода в Java должен начинаться открывающей фигурной скобкой (`{`) и завершаться закрывающей фигурной скобкой (`}`).

Стиль расстановки фигурных скобок всегда вызывал споры. Обычно мы стараемся располагать скобки одну под другой, выравнивая их с помощью пробелов. А поскольку пробелы не имеют значения для компилятора Java, вы можете употреблять какой угодно вам стиль расположения фигурных скобок. В дальнейшем, рассматривая различные операторы цикла, мы обсудим применение фигурных скобок более подробно.

Можете пока что не обращать внимания на ключевые слова `static void`, считая их просто необходимой частью программы на Java. К концу главы 4 вам полностью раскроется подлинный смысл этих ключевых слов. А до тех пор запомните, что каждое приложение на Java должно *непрерывно* содержать метод `main()`, объявление которого приведено ниже.

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



НА ЗАМЕТКУ C++! Если вы программируете на C++, то, безусловно, знаете, что такое класс. Классы в Java похожи на классы в C++, но у них имеются существенные отличия. Так, в языке Java все функции являются методами того или иного класса. (Термин *метод* в Java соответствует термину *функция-член* в C++.) Следовательно, в Java должен существовать класс, которому принадлежит метод `main()`. Вероятно, вы знакомы с понятием *статических функций-членов* в C++. Это функции-члены, определенные в классе и не принадлежащие ни одному из объектов этого класса. Метод `main()` в Java всегда является статическим. И, наконец, как и в C/C++, ключевое слово `void` означает, что метод не возвращает никакого значения. В отличие от C/C++, метод `main()` не передает операционной системе код завершения. Если этот метод корректно завершает свою работу, код завершения равен 0. А для того чтобы выйти из программы с другим кодом завершения, следует вызвать метод `System.exit()`.

Обратите внимание на следующий фрагмент кода:

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

Фигурные скобки обозначают начало и конец *тела* метода, которое в данном случае состоит из единственной строки кода. Как и в большинстве других языков программирования, операторы Java можно считать равнозначными предложениям в обычном языке. В языке Java каждый оператор должен *непрерывно* оканчиваться точкой с запятой. В частности, символ конца строки не означает конец оператора, поэтому оператор может занимать столько строк, сколько потребуется.

В рассматриваемом здесь примере кода при выполнении метода `main()` на консоль выводится одна текстовая строка. Для этой цели используется объект `System.out` и вызывается его метод `println()`. Обратите внимание на то, что метод отделяется от объекта точкой. В общем случае вызов метода в Java принимает приведенную ниже синтаксическую форму, что равнозначно вызову функции.

объект.метод(параметры)

В данном примере при вызове метода `println()` в качестве параметра ему передается символьная строка. Этот метод выводит символьную строку на консоль, дополняя ее символом перевода строки. В языке Java, как и в C/C++, строковый литерал заключается в двойные кавычки. (Порядок обращения с символьными строками будет рассмотрен далее в этой главе.)

Методам в Java, как и функциям в любом другом языке программирования, можно вообще не передавать параметры или же передавать один или несколько *параметров*, которые иногда еще называют *аргументами*. Даже если у метода отсутствуют параметры, после его имени обязательно ставят пустые скобки. Например, при вызове метода `println()` без параметров на экран выводится пустая строка. Такой вызов выглядит следующим образом:

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



НА ЗАМЕТКУ! У объекта `System.out` имеется метод `print()`, который выводит символьную строку, не добавляя к ней символ перехода на новую строку. Например, при вызове метода `System.out.print("Hello")` выводится строка "Hello" и в конце нее ставится курсор. А следующие выводимые на экран данные появятся сразу после буквы `o`.

3.2. Комментарии

Комментарии в Java, как и в большинстве других языков программирования, игнорируются при выполнении программы. Таким образом, в программу можно добавить столько комментариев, сколько потребуется, не опасаясь увеличить ее объем. В языке Java предоставляются три способа выделения комментариев в тексте. Чаще всего для этой цели используются два знака косой черты (`//`), а комментарий начинается сразу после знаков `//` и продолжается до конца строки. Если же требуются комментарии, состоящие из нескольких строк, каждую их строку следует начинать знаками `//`, как показано ниже.

```
System.out.println("We will not use 'Hello, World!'");  
// Мы не будем пользоваться избитой фразой "Здравствуй, мир!".  
// Остроумно, не правда ли?
```

Кроме того, для создания больших блоков комментариев можно использовать разделители `/*` и `*/`. И, наконец, третьей разновидностью комментариев можно пользоваться для автоматического формирования документации. Эти комментарии начинаются знаками `/**` и оканчиваются знаками `*/`, как показано в листинге 3.1. Подробнее об этой разновидности комментариев и автоматическом формировании документации речь пойдет в главе 4.

Листинг 3.1. Исходный код из файла `FirstSample/FirstSample.java`

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



ВНИМАНИЕ! В языке Java комментарии, выделяемые знаками `/*` и `*/`, не могут быть вложенными. Это означает, что фрагмент кода нельзя исключить из программы, просто закомментировав его парой знаков `/*` и `*/`, поскольку в этом коде могут, в свою очередь, присутствовать разделители `/*` и `*/`.

3.3. Типы данных

Язык Java является *строго типизированным*. Это означает, что тип каждой переменной должен быть *непрерывно* объявлен. В языке Java имеются восемь *простых*, или *примитивных*, типов данных. Четыре из них представляют целые числа, два — действительные числа с плавающей точкой, один — символы в Юникоде (как поясняется далее, в разделе 3.3.3), а последний — логические значения.



НА ЗАМЕТКУ! В языке Java предусмотрен пакет для выполнения арифметических операций с произвольной точностью. Но так называемые “большие числа” в Java являются объектами и не считаются новым типом данных. Далее в этой главе будет показано, как обращаться с ними.

3.3.1. Целочисленные типы данных

Целочисленные типы данных служат для представления как положительных, так и отрицательных чисел без дробной части. В языке Java имеются четыре целочисленных типа (табл. 3.1).

Таблица 3.1. Целочисленные типы данных в Java

Тип	Требуемый объем памяти (в байтах)	Диапазон допустимых значений (включительно)
<code>int</code>	4	От <code>-2147483648</code> до <code>2147483647</code> (т.е. больше 2 млрд)
<code>short</code>	2	От <code>-32768</code> до <code>32767</code>
<code>long</code>	8	От <code>-9223372036854775808</code> до <code>-9223372036854775807</code>
<code>byte</code>	1	От <code>-128</code> до <code>127</code>

Как правило, наиболее удобным оказывается тип `int`. Так, если требуется представить в числовом виде количество обитателей планеты, то нет никакой нужды прибегать к типу `long`. Типы `byte` и `short` используются главным образом в специальных приложениях, например, при низкоуровневой обработке файлов или ради экономии памяти при формировании больших массивов данных, когда во главу угла ставится размер информационного хранилища.

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

Длинные целые числа указываются с суффиксом **L** (например, `4000000000L`), шестнадцатеричные числа — с префиксом **0x** (например, `0xCAFE`), восьмеричные — с префиксом **0**. Так, `010` — это десятичное число `8` в восьмеричной форме. Такая запись иногда приводит к недоразумениям, поэтому пользоваться восьмеричными числами не рекомендуется.



НА ЗАМЕТКУ C++! В языках C и C++ разрядность таких целочисленных типов, как `int` и `long`, зависит от конкретной платформы. Так, на платформе 8086 с 16-разрядным процессором разрядность целочисленного типа `int` составляет 2 байта, а на таких платформах с 32-разрядным процессором, как Pentium или SPARC, — 4 байта. Аналогично разрядность целочисленного типа `long` на платформах с 32-разрядным процессором составляет 4 байта, а на платформах с 64-разрядным процессором — 8 байт. Именно эти отличия затрудняют написание межплатформенных программ. А в Java разрядность всех числовых типов данных не зависит от конкретной платформы. Следует также иметь в виду, что в Java отсутствуют беззнаковые (**unsigned**) разновидности целочисленных типов `int`, `long`, `short` или `byte`.

Начиная с версии Java 7, числа можно указывать и в двоичной форме с префиксом **0b** или **0B**. Например, **0b1001** — это десятичное число 9 в двоичной форме. Кроме того, начиная с версии Java 7, числовые литералы можно указывать со знаками подчеркивания, как, например, **1_000_000** (или **0b1111_0100_0010_0100_0000**) для обозначения одного миллиона. Знаки подчеркивания добавляются только ради повышения удобочитаемости больших чисел, а компилятор Java просто удаляет их.



НА ЗАМЕТКУ! Если вы оперируете целочисленными значениями, которые могут вообще не быть отрицательными и которым требуется дополнительный бит, то можете, проявив внимательность, интерпретировать целочисленные значения со знаком как целочисленные значения без знака. Например, вместо числовых значений в пределах от **-128** до **127** переменная **b** типа **byte** может представлять числовые значения пределах от 0 до 255. И эти значения можно хранить в переменных типа **byte**. Характер двоичных арифметических операций сложения, вычитания и умножения таков, что они будут выполняться нормально, при условии, что они не приведут к переполнению. А для выполнения остальных арифметических операций рекомендуется вызвать метод **Byte.toUnsignedInt(b)**, чтобы получить сначала значение типа **int** в пределах от 0 до 255, а затем обработать это числовое значение и привести его обратно к типу **byte**. В классах **Integer** и **Long** имеются методы для выполнения операций деления без знака с остатком.

3.3.2. Числовые типы данных с плавающей точкой

Типы данных с плавающей точкой представляют числа с дробной частью. В языке Java имеются два числовых типа данных с плавающей точкой. Они приведены в табл. 3.2.

Таблица 3.2. Числовые типы данных с плавающей точкой в Java

Тип	Требуемый объем памяти (в байтах)	Диапазон допустимых значений (включительно)
float	4	Приблизительно $\pm 3,40282347\text{E}+38\text{F}$ (6-7 значащих десятичных цифр)
double	8	Приблизительно $\pm 1,7976931348623157\text{E}+308\text{F}$ (15 значащих десятичных цифр)

Название **double** означает, что точность таких чисел вдвое превышает точность чисел типа **float**. (Некоторые называют их *числами с двойной точностью*.) Для большинства приложений тип **double** считается более удобным, а ограниченной точности чисел типа **float** во многих случаях оказывается совершенно недостаточно. Числовыми значениями типа **float** следует пользоваться лишь в работе с библиотекой, где они непременно требуются, или же в том случае, если такие значения приходится хранить в большом количестве.

Числовые значения типа **float** указываются с суффиксом **F**, например **3.14F**. А числовые значения с плавающей точкой, указываемые без суффикса **F** (например, **3.14**), всегда рассматриваются как относящиеся к типу **double**. Для их представления можно (но не обязательно) использовать суффикс **D**, например **3.14D**.



НА ЗАМЕТКУ! Числовые литералы с плавающей точкой могут быть представлены в шестнадцатеричной форме. Например, числовое значение $0.125 = 2^{-3}$ можно записать как **0x1.0p-3**. В шестнадцатеричной форме для выделения показателя степени числа служит обозначение **p**, а не **e**, поскольку **e** — шестнадцатеричная цифра. Обратите внимание на то, что дробная часть числа записывается в шестнадцатеричной форме, а показатель степени — в десятичной, тогда как основание показателя степени — 2, но не 10.

Все операции над числами с плавающей точкой производятся по стандарту IEEE 754. В частности, в Java имеются три специальных значения с плавающей точкой.

- Положительная бесконечность.
- Отрицательная бесконечность.
- Не число (NaN).

Например, результат деления положительного числа на 0 равен положительной бесконечности. А вычисление выражения 0/0 или извлечение квадратного корня из отрицательного числа дает нечисловой результат NaN.



НА ЗАМЕТКУ! В языке Java существуют константы `Double.POSITIVE_INFINITY`, `Double.NEGATIVE_INFINITY` и `Double.NaN` (а также соответствующие константы типа `float`), но на практике они редко применяются. В частности, чтобы убедиться, что некоторый результат равен константе `Double.NaN`, нельзя выполнить приведенную ниже проверку.

```
if (x == Double.NaN) // никогда не будет истинно
```

Все величины, не являющиеся числами, считаются разными. Но в то же время можно вызвать метод `Double.isNaN()`, как показано ниже.

```
if (Double.isNaN(x)) // проверить, не является ли числом значение
                    // переменной x
```



ВНИМАНИЕ! Числа с плавающей точкой нельзя использовать в финансовых расчетах, где ошибки округления недопустимы. Например, в результате выполнения оператора `System.out.println(2.0 - 1.1)` будет выведено значение `0.8999999999999999`, а не `0.9`, как было бы логично предположить. Подобные ошибки связаны с внутренним двоичным представлением чисел. Как в десятичной системе счисления нельзя точно представить результат деления `1/3`, так и в двоичной системе невозможно точно представить результат деления `1/10`. Если же требуется исключить ошибки округления, то следует воспользоваться классом `BigDecimal`, рассматриваемым далее в этой главе.

3.3.3. Тип данных char

Первоначально тип `char` предназначался для описания отдельных символов, но теперь это уже не так. Ныне одни символы в Юникоде могут быть описаны единственным значением типа `char`, а для описания других требуются два значения типа `char`. Подробнее об этом речь пойдет в следующем разделе.

Литеральные значения типа `char` должны быть заключены в одиночные кавычки. Например, литеральное значение 'A' является символьной константой, которой соответствует числовое значение 65. Не следует путать символ 'A' со строкой "A", состоящей из одного символа. Кодовые единицы Юникода могут быть представлены шестнадцатеричными числами в пределах от `\u0000` до `\uFFFF`. Например, значение `\u2122` соответствует знаку торговой марки ([™]), а значение `\u03C0` — греческой букве π.

Кроме префикса `\u`, который превращает кодовую единицу в Юникоде, существует также несколько специальных управляющих последовательностей символов, приведенных табл. 3.3. Эти управляющие последовательности можно вводить в символьные константы или строки, например `'\u2122'` или `"Hello\n"`. Управляющие последовательности, начинающиеся с префикса `\u` (и никакие другие), можно даже указывать за пределами символьных констант или строк, заключаемых в кавычки.

Приведенная ниже строка кода вполне допустима, потому что последовательности `\u005B` и `\u005D` соответствуют кодировке знаков [и].

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

Таблица 3.3. Управляющие последовательности специальных символов

Управляющая последовательность	Назначение	Значение в Юникоде
<code>\b</code>	Возврат на одну позицию	<code>\u0008</code>
<code>\t</code>	Табуляция	<code>\u0009</code>
<code>\n</code>	Переход на новую строку	<code>\u000a</code>
<code>\r</code>	Возврат каретки	<code>\u000d</code>
<code>\"</code>	Двойная кавычка	<code>\u0022</code>
<code>\'</code>	Одинарная кавычка	<code>\u0027</code>
<code>\\</code>	Обратная косая черта	<code>\u005c</code>



ВНИМАНИЕ! Управляющие последовательности символов в Юникоде обрабатываются перед синтаксическим анализом кода. Например, управляющая последовательность `"\u0022+\u0022"` не является символьной строкой, состоящей из знака +, заключаемого в кавычки (`U+0022`). Вместо этого значение `\u0022` преобразуется в знак " перед синтаксическим анализом, в результате чего получается пустая строка `"+""`.

Более того, следует избегать употребления префикса `\u` в комментариях. Так, если ввести в исходном коде программы следующий комментарий:

```
// \u00A0 это знак новой строки
```

то возникнет синтаксическая ошибка, поскольку значение `\u00A0` заменяется знаком новой строки при компиляции программы. Аналогично следующий комментарий:

```
// войти в каталог c:\users
```

приводит к синтаксической ошибке, поскольку за префиксом `\u` не следуют четыре шестнадцатеричные цифры.

3.3.4. Юникод и тип `char`

Для того чтобы полностью уяснить тип `char`, нужно иметь ясное представление о принципах кодировки в Юникоде. Кодировка в Юникоде была изобретена для преодоления ограничений традиционных кодировок символов. До появления Юникода существовало несколько стандартных кодировок: ASCII, ISO 8859-1, KOI-8, GB18030, BIG-5 и т.д. При этом возникали два затруднения. Во-первых, один и тот же код в различных кодировках соответствовал разным символам. Во-вторых, в языках с большим набором символов использовался код различной длины: часто употребляющиеся символы представлялись одним байтом, а остальные символы — двумя, тремя и большим количеством байтов.

Для разрешения этих затруднений была разработана кодировка в Юникоде. В результате исследований, начавшихся в 1980-х годах, выяснилось, что двухбайтового кода более чем достаточно для представления всех символов, использующихся во всех языках мира. И еще оставался достаточный резерв для любых мыслимых расширений. В 1991 году была выпущена спецификация Unicode 1.0, в которой использовалось меньше половины из возможных 65536 кодов. В языке Java изначально были

приняты 16-разрядные символы в Юникоде, что дало ему еще одно преимущество над другими языками программирования, где используются 8-разрядные символы.

Но впоследствии случилось непредвиденное: количество символов превысило допустимый для кодировки предел 65536. Причиной тому стали чрезвычайно большие наборы иероглифов китайского, японского и корейского языков. Поэтому в настоящее время 16-разрядного типа `char` недостаточно для описания всех символов в Юникоде.

Для того чтобы стало понятнее, каким образом данное затруднение разрешается в Java, начиная с версии Java 5, необходимо ввести ряд терминов. В частности, *кодовой точкой* называется значение, связанное с символом в кодировке. Согласно стандарту на Юникод, кодовые точки записываются в шестнадцатеричной форме и предваряются символами `U+`. Например, кодовая точка латинской буквы `A` равна `U+0041`. В Юникоде кодовые точки объединяются в 17 *кодowych плоскостей*. Первая кодовая плоскость, называемая *основной многоязыковой плоскостью*, состоит из “классических” символов в Юникоде с кодовыми точками от `U+0000` до `U+FFFF`. Шестнадцать дополнительных плоскостей с кодовыми точками от `U+10000` до `U+10FFFF` содержат *дополнительные символы*.

Кодировка UTF-16 — это способ представления в Юникоде всех кодовых точек кодом переменной длины. Символы из основной многоязыковой плоскости представляются 16-битовыми значениями, называемыми *кодowymi единицами*. Дополнительные символы обозначаются последовательными парами кодовых единиц. Каждое из значений кодируемой подобным образом пары попадает в область 2048 неиспользуемых значений из основной многоязыковой плоскости. Эта так называемая *область подстановки* простирается в пределах от `U+D800` до `U+DBFF` для первой кодовой единицы и от `U+DC00` до `U+DFFF` для второй кодовой единицы. Такой подход позволяет сразу определить, соответствует ли значение коду конкретного символа или является частью кода дополнительного символа. Например, математическому коду символа `O`, обозначающего множество октонионов, соответствует кодовая точка `U+1D546` и две кодовые единицы — `U+D835` и `U+DD46` (с описанием алгоритма кодировки UTF-16 можно ознакомиться, обратившись по адресу <https://tools.ietf.org/html/rfc2781>).

В языке Java тип `char` описывает *кодovou единицу* в кодировке UTF-16. Начинаящим программировать на Java рекомендуется пользоваться кодировкой UTF-16 лишь в случае крайней необходимости. Старайтесь чаще пользоваться символьными строками как абстрактными типами данных (подробнее о них речь пойдет далее, в разделе 3.6).

3.3.5. Тип данных `boolean`

Тип данных `boolean` имеет два логических значения: `false` и `true`, которые служат для вычисления логических выражений. Преобразование значений типа `boolean` в целочисленные и наоборот невозможно.



НА ЗАМЕТКУ C++! В языке C++ вместо логических значений можно использовать числа и даже указатели. Так, нулевое значение эквивалентно логическому значению `false`, а ненулевые значения — логическому значению `true`. А в Java представить логические значения посредством других типов нельзя. Следовательно, программирующий на Java защищен от недоразумений, подобных следующему:

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

Эта строка кода на C++ компилируется и затем выполняется проверка по условию, которая всегда дает логическое значение **false**. А в Java наличие такой строки приведет к ошибке на этапе компиляции, поскольку целочисленное выражение `x = 0` нельзя преобразовать в логическое.

3.4. Переменные и константы

Как и в каждом языке программирования, переменные служат в Java для хранения значений. А константы являются переменными, значения которых не изменяются. В последующих разделах поясняется, как объявлять переменные и константы.

3.4.1. Объявление переменных

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

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

Обратите внимание на точку с запятой в конце каждого объявления. Она необходима, поскольку объявление в языке Java считается полным оператором, а все операторы в Java завершаются точкой с запятой.

Имя переменной должно начинаться с буквы и представлять собой сочетание букв и цифр. Термины *буквы* и *цифры* в Java имеют более широкое значение, чем в большинстве других языков программирования. Буквами считаются символы 'A'–'Z', 'a'–'z', '_', '\$' и любой другой символ кодировки в Юникоде, соответствующий букве. Например, немецкие пользователи в именах переменных могут использовать букву 'ä', а греческие пользователи — букву 'п'. Аналогично цифрами считаются как обычные десятичные цифры, '0'–'9', так и любые символы кодировки в Юникоде, используемые для обозначения цифры в конкретном языке. Символы вроде '+' или '©', а также пробел нельзя использовать в именах переменных. Все символы в имени переменной важны, причем *регистр букв* также *учитывается*. Длина имени переменной не ограничивается.



СОВЕТ! Если вам действительно интересно знать, какие именно символы в Юникоде считаются "буквами" в Java, воспользуйтесь для этого методами `isJavaIdentifierStart()` и `isJavaIdentifierPart()` из класса `Character`.



СОВЕТ! Несмотря на то что знак \$ считается достоверным в Java, пользоваться им для именования элементов прикладного кода не рекомендуется. Ведь он служит для обозначения имен, формируемых компилятором Java и другими инструментальными средствами.

В качестве имен переменных нельзя использовать и зарезервированные слова Java. Начиная с версии Java 9, единственный знак подчеркивания `_` нельзя употреблять в качестве имени переменной. А в последующих версиях Java он может найти применение в качестве метасимвола.

В одной строке программы можно разместить несколько объявлений переменных:

```
int i,j; // обе переменные — целочисленные
```

Но придерживаться такого стиля программирования все же не рекомендуется. Ведь если объявить каждую переменную в отдельной строке, то читать исходный код программы будет намного легче.



НА ЗАМЕТКУ! Как упоминалось ранее, в Java различаются прописные и строчные буквы. Так, переменные `hireday` и `hireDay` считаются разными. Вообще говоря, употреблять в коде две переменные, имена которых отличаются только регистром букв, не рекомендуется. Но иногда для переменной трудно подобрать подходящее имя. Одни программисты в подобных случаях дают переменной имя, совпадающее с именем типа, но отличающееся регистром букв. Например:

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

А другие программисты предпочитают использовать в имени переменной префикс `a`:

```
Box aBox;
```

3.4.2. Инициализация переменных

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

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

Чтобы присвоить ранее объявленной переменной какое-нибудь значение, следует указать слева ее имя, поставить знак равенства (=), а справа написать любое допустимое на языке Java выражение, задающее требуемое значение:

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

При желании переменную можно объявить и инициализировать одновременно. Например:

```
int vacationDays = 12;
```

И, наконец, объявление переменной можно размещать в любом месте кода Java. Так, приведенный ниже фрагмент кода вполне допустим.

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

Тем не менее при написании программ на Java переменную рекомендуется объявлять как можно ближе к тому месту кода, где предполагается ее использовать.



НА ЗАМЕТКУ! Начиная с версии 10, объявлять типы локальных переменных необязательно, если их можно вывести из первоначального значения. В таком случае вместо типа переменной следует указать ключевое слово `var`, как показано ниже.

```
var vacationDays = 12; // переменная vacationDays типа int  
var greeting = "Hello"; // переменная greeting типа String
```

Такой способ объявлять переменные будет использоваться в примерах кода, начиная со следующей главы.



НА ЗАМЕТКУ C++! В языках C и C++ различают *объявление* и *определение* переменной. Ниже приведен пример определения переменной.

```
int i = 10;
```

Объявление переменной выглядит следующим образом:

```
extern int i;
```

А в Java объявления и определения переменных не различаются.

3.4.3. Константы

В языке Java для обозначения констант служит ключевое слово `final`, как показано в приведенном ниже фрагменте кода.

```
public class Constants
{
    public static void main(String[] args)
    {
        final double CM_PER_INCH = 2.54;
        double paperWidth = 8.5;
        double PaperHeight = 11;
        System.out.println("Paper size in centimeters: "
            + paperWidth * CM_PER_INCH
            + "by" + paperHeight * CM_PER_INCH);
    }
}
```

Ключевое слово `final` означает, что присвоить данной переменной какое-нибудь значение можно лишь один раз, после чего изменить его уже нельзя. Использовать в именах констант только прописные буквы необязательно, но такой стиль способствует удобочитаемости кода.

В программах на Java часто возникает потребность в константах, доступных нескольким методам в одном классе. Обычно они называются *константами класса*. Константы класса объявляются с помощью ключевых слов `static final`. Ниже приведен пример применения константы класса в коде.

```
public class Constants2
{
    public static final double CM_PER_INCH = 2.54;
    public static void main(String[] args)
    {
        double paperWidth = 8.5;
        double paperHeight = 11;
        System.out.println("Paper size in centimeters: "
            + paperWidth * CM_PER_INCH + " by "
            + paperHeight * CM_PER_INCH);
    }
}
```

Обратите внимание на то, что константа класса задается *за пределами* метода `main()`, поэтому ее можно использовать в других методах того же класса. А если константа объявлена как `public`, то методы из других классов также могут получить к ней доступ. В данном примере это можно сделать по ссылке `Constants2.CM_PER_INCH`.



НА ЗАМЕТКУ C++! В языке Java слово `const` является зарезервированным, но в настоящее время оно уже не употребляется. Для объявления констант следует использовать ключевое слово `final`.

3.4.4. Перечислимые типы

Иногда переменной должны присваиваться лишь значения из ограниченного набора. Допустим, продается пицца четырех размеров: малого, среднего, большого и очень большого. Конечно, размеры можно представить целыми числами (1, 2, 3 и 4) или буквами (S, M, L и X). Но такой подход чреват ошибками. В процессе написания программы можно присвоить переменной недопустимое значение, например 0 или m.

В подобных случаях можно воспользоваться *перечислимым типом*. Перечислимый тип имеет конечный набор именованных значений. Например:

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

После этого можно определить переменные данного типа, как показано ниже.

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

Переменная типа `Size` может содержать только predefined значения. Допускается также пустое значение `null`, указывающее на то, что в данной переменной не установлено никакого значения. Более подробно перечислимые типы рассматриваются в главе 5.

3.5. Операции

Операции служат для объединения значений. Как будет показано в последующих разделах, в Java предоставляется богатый набор арифметических и логических операций и математических функций.

3.5.1. Арифметические операции

Для обозначения арифметических операций сложения, вычитания, умножения и деления в Java употребляются обычные знаки подобных операций: `+`, `-`, `*` и `/` соответственно. Операция `/` обозначает целочисленное деление, если оба ее аргумента являются целыми числами. В противном случае эта операция обозначает деление чисел с плавающей точкой. Остаток от деления целых чисел обозначается символом `%`. Например, `15/2` равно 7, `15%2` равно 1, а `15.0/2` — 7.5. Следует, однако, иметь в виду, что в результате целочисленного деления на ноль генерируется исключение, в то время как результатом деления на ноль чисел с плавающей точкой является бесконечность или NaN.



НА ЗАМЕТКУ! Одной из заявленных целей языка Java является переносимость. Вычисления должны приводить к одинаковому результату, независимо от того, какая виртуальная машина их выполняет. Для арифметических операций над числами с плавающей точкой соблюдение этого требования неожиданно оказалось непростой задачей. Для хранения числовых значений типа `double` используются 64 бита, но в некоторых процессорах применяются 80-разрядные регистры с плавающей точкой. Эти регистры обеспечивают дополнительную точность на промежуточных стадиях вычисления. Рассмотрим в качестве примера следующее выражение:

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

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

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

В этом случае все команды в теле метода **main()** будут выполнять точные операции над числами с плавающей точкой. А если пометить ключевым словом **strictfp** класс, то во всех его методах должны выполняться точные операции с плавающей точкой.

Многое при подобных вычислениях зависит от особенностей работы процессоров Intel. По умолчанию в промежуточных результатах может использоваться расширенный показатель степени, но не расширенная мантисса. (Процессоры компании Intel поддерживают округление мантиссы без потери производительности.) Следовательно, вычисления по умолчанию отличаются от точных вычислений лишь тем, что в последнем случае возможно переполнение.

Если сказанное выше кажется вам слишком сложным, не отчаивайтесь. Переполнение при вычислениях с плавающей точкой, как правило, не возникает. А в примерах, рассматриваемых в этой книге, ключевое слово **strictfp** использоваться не будет.

3.5.2. Математические функции и константы

В состав класса **Math** входит целый набор математических функций, которые нередко требуются для решения практических задач. В частности, чтобы извлечь квадратный корень из числа, применяется метод **sqrt()**:

```
double x = 4;  
double y = Math.sqrt(x);  
System.out.println(y); // выводит числовое значение 2.0
```



НА ЗАМЕТКУ! У методов **println()** и **sqrt()** имеется едва заметное, но существенное отличие. Метод **println()** оперирует объектом **System.out**, определенным в классе **System**, тогда как метод **sqrt()** определен в классе **Math** и не оперирует никаким объектом. Такие методы называются статическими и будут рассматриваться в главе 4.

В языке Java не поддерживается операция возведения в степень. Для этой цели следует вызвать метод **pow()** из класса **Math**. В результате выполнения приведенной ниже строки кода переменной y присваивается значение переменной x , возведенное в степень a (x^a). Оба параметра метода **pow()**, а также возвращаемое им значение относятся к типу **double**.

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


Метод `floorMod()` предназначен для решения давней проблемы с остатками от целочисленного деления. Рассмотрим в качестве примера выражение $n \% 2$. Как известно, если число n является четным, то результат данной операции равен 0, а иначе — 1. Если же число n оказывается отрицательным, то результат данной операции будет равен -1. Почему? Когда были созданы первые компьютеры, то кем-то были установлены правила целочисленного деления с остатком и для отрицательных операндов. Оптимальное (или эвклидово) правило такого деления известно в математике уже многие сотни лет и состоит в следующем: остаток всегда должен быть ³ 0. Но вместо того, чтобы обратиться к справочнику по математике, первопроходцы в области вычислительной техники составили правила, которые им показались вполне благоразумными, хотя на самом деле они неудобные.

Допустим, вычисляется положение часовой стрелки на циферблате. Корректируя это положение, нужно нормализовать его числовое значение в пределах от 0 до 11. Сделать это совсем не трудно следующим образом:

$(\text{положение} + \text{коррекция}) \% 12$

Но что если корректируемое значение окажется отрицательным? В таком случае отрицательным может стать и полученный результат. Следовательно, в последовательность вычислений придется ввести ветвление или же составить такое выражение:

$((\text{положение} + \text{коррекция}) \% 12 + 12) \% 12$

Но и то и другое довольно хлопотно. Дело упрощается благодаря методу `floorMod()`. Так, в результате следующего вызова:

`floorMod(положение + коррекция, 12)`

всегда получается значение в пределах от 0 до 11. (К сожалению, метод `floorMod()` выдает и отрицательные результаты при отрицательных делителях, хотя такое нечасто случается на практике.)

В состав класса `Math` входят также перечисленные ниже методы для вычисления обычных тригонометрических функций.

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

Кроме того, в него включены методы для вычисления экспоненциальной и обратной к ней логарифмической функции (натурального и десятичного логарифмов):

```
Math.exp()  
Math.log()  
Math.log10()
```

И, наконец, в данном классе определены также следующие две константы как приближенное представление чисел π и e .

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



СОВЕТ. При вызове методов для математических вычислений класс `Math` можно не указывать явно, включив вместо этого в начало исходного файла следующую строку кода:

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

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

```
System.out.println("The square root of \u03C0 is " + sqrt(PI));
```

Более подробно вопросы статического импорта обсуждаются в главе 4.



НА ЗАМЕТКУ! Для повышения производительности методов из класса **Math** применяются процедуры из аппаратного модуля, выполняющего операции с плавающей точкой. Если для вас важнее не быстродействие, а предсказуемые результаты, пользуйтесь классом **StrictMath**. В нем реализуются алгоритмы из свободно распространяемой библиотеки **fdlibm** математических функций (www.netlib.org/fdlibm), гарантирующей идентичность результатов на всех платформах.



НА ЗАМЕТКУ! В классе **Math** предоставляется ряд методов, обеспечивающих более надежное выполнение арифметических операций. Математические операции негласно возвращают неверные результаты, когда вычисление приводит к переполнению. Например, результат умножения одного миллиарда на три (`1000000000 * 3`) оказывается равным `-1294967296`, поскольку максимальное значение типа `int` лишь ненамного превышает два миллиарда. А если вместо этого вызвать метод `Math.multiplyExact(1000000000, 3)`, то будет сгенерировано исключение. Это исключение можно перехватить, чтобы нормально завершить выполнение программы, а не позволить ей продолжаться негласно с неверным результатом умножения. В классе **Math** имеются также методы `addExact()`, `subtractExact()`, `incrementExact()`, `decrementExact()`, `negateExact()` с параметрами типа `int` и `long`.

3.5.3. Преобразование числовых типов

Нередко возникает потребность преобразовать один числовой тип в другой. На рис. 3.1 представлены допустимые преобразования числовых типов.

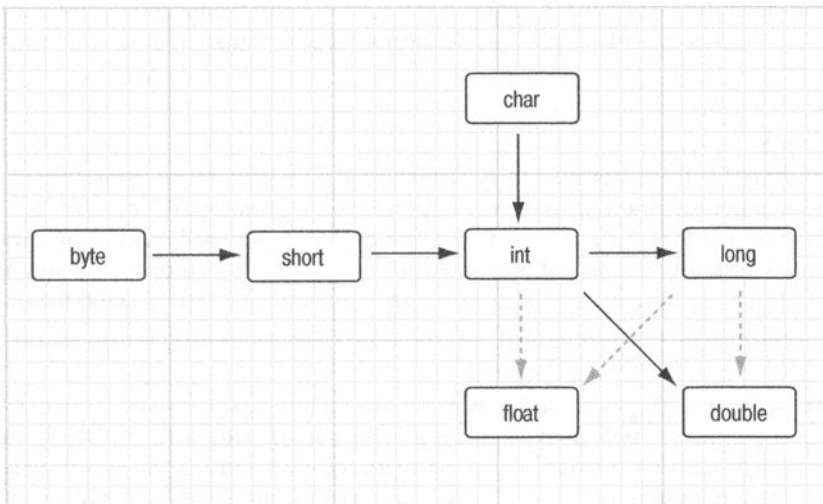


Рис. 3.1. Допустимые преобразования числовых типов

Шесть сплошных линий со стрелками на рис. 3.1 обозначают преобразования, которые выполняются без потери данных, а три штриховые линии со стрелками — преобразования, при которых может произойти потеря точности. Например, количество цифр в длинном целом числе 123456789 превышает количество цифр, которое может быть представлено типом `float`. Число, преобразованное в тип `float`, имеет тот же порядок, но несколько меньшую точность:

```
int n = 123456789;
float f = n; // значение переменной f равно 1.234567892E8
```

Если два числовых значения объединяются бинарной операцией (например, `n+f`, где `n` — целочисленное значение, а `f` — значение с плавающей точкой), то перед выполнением операции оба операнда преобразуются в числовые значения одинакового типа по следующим правилам.

- Если хотя бы один из операндов относится к типу `double`, то и второй операнд преобразуется в тип `double`.
- Иначе, если хотя бы один из операндов относится к типу `float`, то и второй операнд преобразуется в тип `float`.
- Иначе, если хотя бы один из операндов относится к типу `long`, то и второй операнд преобразуется в тип `long`.
- Иначе оба операнда преобразуются в тип `int`.

3.5.4. Приведение типов

Как пояснялось выше, значения типа `int` при необходимости автоматически преобразуются в значения типа `double`. С другой стороны, в ряде ситуаций числовое значение типа `double` должно рассматриваться как целое. Преобразования числовых типов в Java возможны, но они могут, конечно, сопровождаться потерей данных. Такие преобразования называются *приведением типов*. Синтаксически приведение типов задается парой скобок, в которых указывается желательный тип, а затем имя переменной:

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

В результате приведения к целому типу числового значения с плавающей точкой, хранящегося в переменной `x`, значение переменной `nx` становится равным 9, поскольку дробная часть числа при этом отбрасывается. Если же требуется округлить число с плавающей точкой до ближайшего целого числа (что во многих случаях намного полезнее), то для этой цели служит метод `Math.round()`, как показано ниже.

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

Теперь значение переменной `nx` становится равным 10. При вызове метода `round()` по-прежнему требуется выполнять приведение типов `(int)`. Дело в том, что значение, возвращаемое методом `round()`, относится к типу `long` и поэтому может быть присвоено переменной типа `int` только с явным приведением. Иначе существует вероятность потери данных.



ВНИМАНИЕ! При попытке приведения типов результат может выйти за пределы диапазона допустимых значений. И в этом случае произойдет усечение. Например, при вычислении выражения `(byte) 300` получается значение 44.



НА ЗАМЕТКУ C++! Приведение логических значений к целым и наоборот невозможно. Такое ограничение предотвращает появление ошибок. А в тех редких случаях, когда действительно требуется представить логическое значение в виде целого, можно составить условное выражение вроде `b ? 1 : 0`.

3.5.5. Сочетание арифметических операций с присваиванием

В языке Java предусмотрена сокращенная запись бинарных арифметических операций (т.е. операций, предполагающих два операнда). Например, следующая строка кода:

```
x += 4;
```

равнозначна такой строке кода:

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

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



НА ЗАМЕТКУ! Если в результате выполнения операции получается значение иного типа, чем у левого операнда, то полученный результат приводится именно к этому типу. Так, если переменная `x` относится к типу `int`, то следующая операция достоверна:

```
x += 3.5;
```

а переменной `x` присваивается результат (`int`) `(x + 3.5)` приведения к типу `int`.

3.5.6. Операции инкремента и декремента

Программисты, конечно, знают, что одной из самых распространенных операций с числовыми переменными является добавление или вычитание единицы. В языке Java, как и в языках C и C++, для этой цели предусмотрены операции инкремента и декремента. Так, в результате операции `n++` к текущему значению переменной `n` прибавляется единица, а в результате операции `n--` значение переменной `n` уменьшается на единицу. Таким образом, после выполнения приведенного ниже фрагмента кода значение переменной `n` становится равным 13.

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

Операции `++` и `--` изменяют значение переменной, поэтому их нельзя применять к самим числам. Например, выражение `4++` считается недопустимым.

Существуют два вида операций инкремента и декремента. Выше продемонстрирована постфиксная форма, в которой символы операции размещаются после операнда. Но есть и префиксная форма: `++n`. Обе эти операции изменяют значение переменной на единицу. Их отличие проявляется только тогда, когда эти операции присутствуют в выражениях. В префиксной форме сначала изменяется значение переменной, и для дальнейших вычислений уже используется новое значение, а в постфиксной форме используется прежнее значение этой переменной, и лишь после данной операции оно изменяется, как показано в приведенных ниже примерах кода.

```
int m = 7;  
int n = 7;
```

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

Пользоваться операциями инкремента и декремента в выражениях не рекомендуется, поскольку это зачастую запутывает код и приводит к досадным ошибкам.

3.5.7. Операции отношения и логические операции

В состав Java входит полный набор операций отношения. Для проверки на равенство служат знаки `==`. Например, выражение `3 == 7` дает в результате логическое значение `false`. Для проверки на неравенство служат знаки `!=`. Так, выражение `3 != 7` дает в итоге логическое значение `true`. Кроме того, в Java поддерживаются обычные операции сравнения: `<` (меньше), `>` (больше), `<=` (меньше или равно) и `>=` (больше или равно).

В языке Java, как и в C++, знаки `&&` служат для обозначения логической операции И, а знаки `||` — для обозначения логической операции ИЛИ. Как обычно, знак восклицания (!) означает логическую операцию отрицания. Операции `&&` и `||` задают порядок вычисления по сокращенной схеме: если первый операнд определяет значение всего выражения, то остальные операнды не вычисляются. Рассмотрим для примера два выражения, объединенных логической операцией `&&`:

```
выражение_1 && выражение_2
```

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

```
x != 0 && 1/x > x+y // не делить на ноль
```

Таким образом, деление на ноль не происходит. Аналогично значение выражение `выражение_1 || выражение_2` оказывается истинным, если истинным является значение первого выражения. В этом случае вычислять второе выражение нет нужды.

В языке Java имеется также тернарная операция `?:`, которая иногда оказывается полезной. Ниже приведена ее общая форма.

```
условие ? выражение_1 : выражение_2
```

Если условие истинно, то вычисляется первое выражение, а если оно ложно — второе выражение. Например, вычисление выражения `x < y ? x : y` дает в итоге меньшее из значений переменных `x` и `y`.

3.5.8. Поразрядные логические операции

Работая с любыми целочисленными типами данных, можно применять операции, непосредственно обрабатывающие двоичные разряды, или биты, из которых состоят целые числа. Это означает, что для определения состояния отдельных битов числа можно использовать маски. В языке Java имеются следующие поразрядные операции: `&` (И), `|` (ИЛИ), `^` (исключающее ИЛИ), `~` (НЕ). Так, если `n` — целое число, то приведенное ниже выражение будет равно единице только в том случае, если четвертый бит в двоичном представлении числа равен единице.

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

Используя поразрядную операцию `&` в сочетании с соответствующей степенью 2, можно замаскировать все биты, кроме одного.



НА ЗАМЕТКУ! При выполнении поразрядной операции `&` и `|` над логическими переменными типа `boolean` получаются логические значения. Эти операции аналогичны логическим операциям `&&` и `||`, за исключением того, что вычисление производится по полной схеме, т.е. обрабатываются все элементы выражения.

В языке Java поддерживаются также операции `>>` и `<<`, сдвигающие двоичное представление числа вправо или влево. Эти операции удобны в тех случаях, если требуется сформировать двоичное представление для поразрядного маскирования:

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

Имеется даже операция `>>>`, заполняющая старшие разряды нулями, в то время как операция `>>` восстанавливает в старших разрядах знаковый бит. А такая операция, как `<<<`, в Java отсутствует.



ВНИМАНИЕ! Значение в правой части операций поразрядного сдвига сокращается по модулю 32 (если левая часть является целочисленным значением типа `long`, то правая часть сокращается по модулю 64). Например, выражение `1<<35` равнозначно выражению `1<<3` и дает в итоге значение 8.



НА ЗАМЕТКУ! В языках C и C++ не определено, какой именно сдвиг выполняет операция `>>`: арифметический (при котором знаковый бит восстанавливается) или логический (при котором старшие разряды заполняются нулями). Разработчики вольны выбрать тот вариант, который покажется им наиболее эффективным. Это означает, что результат выполнения операции сдвига вправо в C/C++ определен лишь для неотрицательных чисел. А в Java подобная неоднозначность устранена.

3.5.9. Круглые скобки и иерархия операций

В табл. 3.4 представлены сведения о предшествовании, или приоритетности, операций. Если скобки не используются, то сначала выполняются более приоритетные операции. Операции, находящиеся на одном уровне иерархии, выполняются слева направо, за исключением операций, имеющих правую ассоциативность, как показано в табл. 3.4. Например, операция `&&` приоритетнее операции `||`, поэтому выражение `a && b || c` равнозначно выражению `(a && b) || c`. Операция `+=` ассоциируется справа налево, а следовательно, выражение `a += b += c` означает `a += (b += c)`. В данном случае значение выражения `b += c` (т.е. значение переменной `b` после прибавления к нему значения переменной `c`) прибавляется к значению переменной `a`.

Таблица 3.4. Приоритетность операций

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

Операции	Ассоциативность
&	Слева направо
^	Слева направо
	Слева направо
&&	Слева направо
	Слева направо
?:	Справа налево
= += -= *= /= %= = ^= <<= >>= >>>=	Справа налево



НА ЗАМЕТКУ C++! В отличие от C и C++, в Java отсутствует операция-запятая. Но в первой и третьей части оператора цикла `for` можно использовать список выражений, разделяемых запятыми.

3.6. Символьные строки

По существу, символьная строка Java представляет собой последовательность символов в Юникоде. Например, строка `"Java\u2122"` состоит из пяти символов: букв J, a, v, a и знака [™]. В языке Java отсутствует встроенный тип для символьных строк. Вместо этого в стандартной библиотеке Java содержится класс `String`. Каждая символьная строка, заключенная в кавычки, представляет собой экземпляр класса `String`:

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

3.6.1. Подстроки

С помощью метода `substring()` из класса `String` можно выделить подстроку из отдельной символьной строки. Например, в результате выполнения приведенного ниже фрагмента кода формируется подстрока `"Hel"`.

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



НА ЗАМЕТКУ! Как и в языках C и C++, в Java кодовые единицы и кодовые точки отсчитываются в символьных строках от нуля.

Второй параметр метода `substring()` обозначает позицию символа, который не следует включать в состав подстроки. В данном примере требуется скопировать символы на трех позициях, 0, 1 и 2 (т.е. от позиции 0 до позиции 2 включительно), поэтому при вызове метода `substring()` указываются значения 0 и 3, обозначающие копируемые символы от позиции 0 и до позиции 2 включительно, но исключая позицию 3.

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

3.6.2. Сцепление строк

В языке Java, как и в большинстве других языков программирования, предоставляется возможность объединить две символьные строки, используя знак `+` операции сцепления.

```
String expletive = "Expletive";
String PG13 = "deleted";
String message = expletive + PG13;
```

В приведенном выше фрагменте кода переменной `message` присваивается символьная строка `"Expletivedeleted"`, сцепленная из двух исходных строк. (Обратите внимание на отсутствие пробела между словами в этой строке. Знак `+` операции сцепления соединяет две строки *точно* в том порядке, в каком они были заданы в качестве операндов.)

При сцеплении символьной строки со значением, не являющимся строковым, это значение преобразуется в строковое. (Как станет ясно из главы 5, каждый объект в Java может быть преобразован в символьную строку.) В приведенном ниже примере кода переменной `rating` присваивается символьная строка `"PG13"`, полученная путем сцепления символьной строки с числовым значением, автоматически преобразуемым в строковое.

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

Такая возможность широко применяется в операторах вывода. Например, приведенная ниже строка кода вполне допустима для вывода результата в нужном формате, т.е. с пробелом между сцепляемыми строками.

```
System.out.println("The answer is " + answer);
```

Если требуется соединить вместе две символьные строки, разделяемые каким-нибудь знаком, то для этой цели можно воспользоваться статическим методом `join()`, как показано ниже.

```
String all = String.join(" / ", "S", "M", "L", "XL");
// в итоге переменная all содержит строку "S / M / L / XL"
```

В версии 11 появился метод `repeat()`, позволяющий повторять сколько требуется заданную символьную строку:

```
String repeated = "Java".repeat(3);
// повторить строку "Java", чтобы
// получить в итоге строку "JavaJavaJava"
```

3.6.3. Принцип постоянства символьных строк

В классе `String` отсутствуют методы, которые позволяли бы *изменять* символы в существующей строке. Так, если требуется заменить символьную строку в переменной `greeting` с `"Hello"` на `"Help!"`, этого нельзя добиться одной лишь заменой двух последних символов. Программирующим на C это покажется, по меньшей мере, странным. “Как же видоизменить строку?” — спросят они. В языке Java можно внести необходимые изменения в строку, выполнив сцепление подстроки, которую требуется сохранить, с заменяющими символами, как показано ниже. В итоге переменной `greeting` присваивается символьная строка `"Help!"`.

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


Программируя на Java, нельзя изменять отдельные символы в строке, поэтому в документации на этот язык объекты типа `String` называются *неизменяемыми*, т.е. постоянными. Как число 3 всегда равно 3, так и строка "Hello" всегда состоит из последовательности кодовых единиц символов 'H', 'e', 'l', 'l' и 'o'. Изменить ее состав нельзя. Но, как мы только что убедились, можно изменить содержимое строковой переменной `greeting` и заставить ее ссылаться на другую символьную строку подобно тому, как числовой переменной, в которой хранится число 3, можно присвоить число 4.

Не приводит ли это к снижению эффективности кода? Казалось бы, намного проще изменять символы, чем создавать новую строку заново. Возможно, это и так. В самом деле, создавать новую строку путем сцепления символьных строк "Hel" и "p!" неэффективно. Но у неизменяемых строк имеется одно существенное преимущество: компилятор может сделать строки *совместно используемыми*.

Чтобы стал понятнее принцип постоянства символьных строк, представьте, что в общем пуле находятся разные символьные строки. Строковые переменные указывают на объекты в этом пуле. При копировании строковой переменной оригинал и копия содержат одну и ту же общую последовательность символов. Одним словом, создатели языка Java решили, что эффективность совместного использования памяти перевешивает неэффективность редактирования строк путем выделения и сцепления подстрок.

Посмотрите на свои программы. Чаще всего вы сравниваете символьные строки, чем изменяете их. Разумеется, бывают случаи, когда непосредственные манипуляции символьными строками оказываются более эффективными. Одна из таких ситуаций возникает, когда требуется составить строку из отдельных символов, поступающих из файла или вводимых с клавиатуры. Для подобных ситуаций в языке Java предусмотрен отдельный класс `StringBuffer`, рассматриваемый далее, в разделе 3.6.9, а в остальном достаточно и средств, предоставляемых в классе `String`.



НА ЗАМЕТКУ C++! Когда программирующие на C обнаруживают символьные строки в программе на Java, они обычно попадают в тупик, поскольку привыкли рассматривать строки как массивы символов:

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

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

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

При замене содержимого переменной `greeting` другой символьной строкой в коде Java выполняется примерно следующее:

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

Разумеется, теперь переменная `greeting` указывает на строку "Help!". И даже самые убежденные поклонники языка C должны признать, что синтаксис Java более изящный, чем последовательность вызовов функции `strncpy()`. А что, если присвоить переменной `greeting` еще одно строковое значение, как показано ниже?

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

Не возникнут ли при этом утечки памяти? К счастью, в Java имеется механизм автоматической сборки "мусора". Если память больше не нужна, она в конечном итоге освобождается.

Если вы программируете на C++ и пользуетесь классом `string`, определенным в стандарте ANSI C++, вам будет намного легче работать с объектами типа `String` в Java. Объекты класса `string` в C++ также обеспечивают автоматическое выделение и освобождение памяти. Управление

памятью осуществляется явным образом с помощью конструкторов, деструкторов и операций присваивания. Но символьные строки в C++ являются изменяемыми, а это означает, что отдельные символы в строке можно видоизменять.

3.6.4. Проверка символьных строк на равенство

Чтобы проверить две символьные строки на равенство, достаточно вызвать метод `equals()`. Так, выражение `s.equals(t)` возвращает логическое значение `true`, если символьные строки `s` и `t` равны, а иначе — логическое значение `false`. Следует, однако, иметь в виду, что в качестве `s` и `t` могут быть использованы строковые переменные или константы. Например, следующее выражение вполне допустимо:

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

А для того чтобы проверить идентичность строк, игнорируя отличия в прописных и строчных буквах, следует вызвать метод `equalsIgnoreCase()`, как показано ниже.

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

Для проверки символьных строк на равенство *нельзя* применять операцию `==`. Она лишь определяет, хранятся ли обе строки в одной и той же области памяти. Разумеется, если обе строки хранятся в одном и том же месте, они должны совпадать. Но вполне возможна ситуация, когда одинаковые символьные строки хранятся в разных местах. Ниже приведен соответствующий пример.

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

Если виртуальная машина всегда обеспечивает совместное использование одинаковых символьных строк, то для проверки их на равенство можно применять операцию `==`. Но совместно использовать можно лишь *константы*, а не символьные строки, получающиеся в результате таких операций, как сцепление или извлечение подстроки методом `substring()`. Следовательно, лучше вообще отказаться от проверки символьных строк на равенство с помощью операции `==`, чтобы исключить в программе наихудшую из возможных ошибок, проявляющуюся лишь время от времени и практически непредсказуемую.



НА ЗАМЕТКУ C++! Если вы привыкли пользоваться классом `string` в C++, будьте особенно внимательны при проверке символьных строк на равенство. В классе `string` операция `==` перегружается и позволяет проверять идентичность содержимого символьных строк. Возможно, создатели Java напрасно отказались от обработки символьных строк подобно числовым значениям, но в то же время это позволило сделать символьные строки похожими на указатели. Создатели Java могли бы, конечно, переопределить операцию `==` для символьных строк таким же образом, как они это сделали с операцией `+`. Что ж, у каждого языка программирования свои недостатки.

Программирующие на C вообще не пользуются операцией `==` для проверки строк на равенство и вместо этого вызывают функцию `strcmp()`. Метод `compareTo()` является в Java точным аналогом функции `strcmp()`. Конечно, для сравнения строк можно воспользоваться следующим выражением:

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

Но, на наш взгляд, применение метода `equals()` делает программу более удобочитаемой.

3.6.5. Пустые и нулевые строки

Пустой считается символьная строка нулевой длины. Чтобы проверить, является ли символьная строка пустой, достаточно составить выражение вида `if (str.length() == 0)` или `if (str.equals(""))`.

Пустая строка является в Java объектом, в котором хранится нулевая (т.е. 0) длина символьной строки и пустое содержимое. Но в переменной типа `String` может также храниться специальное пустое значение `null`, указывающее на то, что в настоящий момент ни один из объектов не связан с данной переменной. (Подробнее пустое значение `null` обсуждается в главе 4.) Чтобы проверить, является ли символьная строка нулевой, т.е. содержит значение `null`, следует задать условие `if (str == null)`.

Иногда требуется проверить, не является ли символьная строка ни пустой, ни нулевой. Для этой цели служит условие `if (str != null && str.length() != 0)`. Но сначала нужно проверить, не является ли символьная строка `str` нулевой. Как будет показано в главе 4, вызов метода для пустого значения `null` считается ошибкой.

3.6.6. Кодовые точки и единицы

Символьные строки в Java реализованы в виде последовательности значений типа `char`. Как пояснялось в разделе 3.3.3, с помощью типа данных `char` можно задавать кодовые единицы, представляющие кодовые точки Юникода в кодировке UTF-16. Наиболее часто употребляемые символы представлены в Юникоде одной кодовой единицей, а дополнительные символы — парами кодовых единиц.

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

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

Чтобы определить истинную длину символьной строки, представляющую собой число кодовых точек, нужно сделать следующий вызов:

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

При вызове метода `s.charAt(n)` возвращается кодовая единица на позиции `n`, где `n` находится в пределах от 0 до `s.length() - 1`. Ниже приведены примеры вызова данного метода.

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

Для получения *i*-й кодовой точки служат выражения

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

А зачем вообще обсуждать кодовые единицы? Рассмотрим следующую символьную строку:

```
O is the set of octonions
```

Для представления символа **О** требуются две кодовые единицы в кодировке UTF-16. В результате приведенного ниже вызова будет получен не код пробела, а вторая кодовая единица символа **О**.

```
char ch = sentence.charAt(1)
```

Чтобы избежать подобных осложнений, не следует применять тип `char`, поскольку он представляет символы на слишком низком уровне.



НА ЗАМЕТКУ! Не следует, однако, думать, что экзотическими символами с кодовыми единицами свыше `U+FFFF` можно пренебречь. Пользователи, любящие употреблять японский эмотикон, могут вводить в строки такие символы, как, например, символ `U+1F37A`, обозначающий пивную кружку.

Если же требуется просмотреть строку посимвольно, т.е. получить по очереди каждую кодовую точку, то можно воспользоваться фрагментом кода, аналогичным приведенному ниже.

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

А организовать просмотр строки в обратном направлении можно следующим образом:

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

Очевидно, что это не совсем удобно. Поэтому проще воспользоваться методом `codePoints()`, который формирует “поток” значений типа `int` для отдельных кодовых точек. (Более подробно потоки данных рассматриваются в главе 2 второго тома настоящего издания.) Полученный в итоге поток данных можно преобразовать в массив, как показано ниже, а затем перебрать его элементы, как поясняется далее, в разделе 3.10.

```
int[] codePoints = str.codePoints().toArray();
```

С другой стороны, чтобы преобразовать массив кодовых точек в символьную строку, можно воспользоваться конструктором класса `String`, как показано в ниже. (Более подробно конструкторы и операция `new` обсуждаются в главе 4.)

```
String str = new String(codePoints, 0, codePoints.length);
```



НА ЗАМЕТКУ! Символьные строки совсем не обязательно реализуются виртуальной машиной в виде последовательностей кодовых единиц. Так, в версии Java 9 для хранения символьных строк, состоящих только из однобайтовых кодовых единиц, употребляется массив типа `byte`, а для хранения всех остальных символьных строк — массив типа `char`.

3.6.7. Прикладной программный интерфейс API класса `String`

В языке Java класс `String` содержит свыше 50 методов. Многие из них оказались очень полезными и применяются очень часто. В приведенном ниже фрагменте описания прикладного программного интерфейса API данного класса перечислены наиболее полезные из них.

Время от времени в качестве вспомогательного материала на страницах этой книги будут появляться фрагменты описания прикладного программного интерфейса Java API. Каждый такой фрагмент начинается с имени класса, например `java.lang.String`, где `java.lang` — пакет (подробнее о пакетах речь пойдет в главе 4). После имени класса следуют имена конкретных методов и их описание.

Обычно в подобных описаниях перечисляются не все, а только наиболее употребительные методы конкретного класса. Полный перечень методов описываемого класса можно найти в оперативно доступной документации, как поясняется в разделе 3.6.8.

Кроме того, приводится номер версии, в которой был реализован класс. Если же тот или иной метод был добавлен позже, то у него имеется отдельный номер версии.

`java.lang.String 1.0`

- `char charAt(int index)`
Возвращает символ, расположенный на указанной позиции. Вызывать этот метод следует только в том случае, если вас интересуют низкоуровневые кодовые единицы.
- `int codePointAt(int index) 5.0`
Возвращает кодовую точку, начало или конец которой находится на указанной позиции.
- `int offsetByCodePoints(int startIndex, int cpCount) 5.0`
Возвращает индекс кодовой точки, которая отстоит на количество `cpCount` кодовых точек от исходной кодовой точки на позиции `startIndex`.
- `int compareTo(String other)`
Возвращает отрицательное значение, если данная строка лексикографически предшествует строке `other`, положительное значение — если строка `other` предшествует данной строке, нулевое значение — если строки одинаковы.
- `IntStream codePoints() 8`
Возвращает кодовые точки из данной символьной строки в виде потока данных. Для их размещения в массиве следует вызвать метод `toArray()`.
- `new String(int[] codePoints, int offset, int count) 5.0`
Создает символьную строку из количества `count` кодовых точек в заданном массиве, начиная с указанной позиции `offset`.
- `boolean isEmpty()`
- `boolean isBlank() 11`
Возвращают логическое значение `true`, если символьная строка пуста или состоит из пробелов.
- `boolean equals(Object other)`
Возвращает логическое значение `true`, если данная строка совпадает с указанной строкой `other`.
- `boolean equalsIgnoreCase(String other)`
Возвращает логическое значение `true`, если данная строка совпадает с указанной строкой `other` без учета регистра букв.
- `boolean startsWith(String prefix)`
- `boolean endsWith(String suffix)`
Возвращают логическое значение `true`, если данная строка начинается указанной подстрокой `prefix` или оканчивается указанной подстрокой `suffix`.
- `int indexOf(String str)`
- `int indexOf(String str, int fromIndex)`
- `int indexOf(int cp)`
- `int indexOf(int cp, int fromIndex)`
Возвращают индекс начала первой подстроки, совпадающей с указанной подстрокой `str`, или же индекс заданной кодовой точки `cp`. Отсчет начинается с позиции 0 или `fromIndex`. Если указанная подстрока `str` отсутствует в данной строке, возвращается значение, равное -1.

java.lang.String 1.0 (окончание)

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

Возвращают начало последней подстроки, равной указанной подстроке *str*, или же индекс заданной кодовой точки *cp*. Отсчет начинается с конца строки или с позиции *fromIndex*. Если указанная подстрока *str* отсутствует в данной строке, возвращается значение, равное -1.

- **int length()**
Возвращает длину строки.
- **int codePointCount(int startIndex, int endIndex) 5.0**

Возвращает количество кодовых точек между позициями *startIndex* и *endIndex* - 1. Неспаренные суррогаты считаются кодовыми точками.

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

Возвращает новую строку, которая получается путем замены всех подстрок, совпадающих с указанной подстрокой *oldString*, заданной строкой *newString*. В качестве параметров типа *CharSequence* могут быть указаны объекты типа *String* или *StringBuilder*.

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

Возвращают новую строку, состоящую из всех кодовых единиц, начиная с позиции *beginIndex* и до конца строки или позиции *endIndex* - 1.

- **String toLowerCase()**
- **toUpperCase()**

Возвращают новую строку, состоящую из всех символов исходной строки. Исходная строка отличается от результирующей тем, что все буквы преобразуются в строчные или прописные.

- **String trim()**
- **String strip() 11**

Возвращают новую строку, из которой исключены все начальные и конечные символы, код которых меньше или равен U+0020 (метод *trim()*), или же пробелы (метод *strip()*), имеющиеся в исходной строке.

- **String join(CharSequence delimiter, CharSequence... elements) 8**

Возвращает новую строку, все элементы которой соединяются через заданный разделитель.

- **String repeat(int count) 11**

Возвращает символьную строку, повторяющую исходную строку столько раз, сколько задается с помощью параметра *count*.



НА ЗАМЕТКУ! В приведенном выше фрагменте описания прикладного программного интерфейса API параметры некоторых методов относятся к типу *CharSequence*. Это тип интерфейса, к которому относятся символьные строки. Подробнее об интерфейсах речь пойдет в главе 6, а до тех пор достаточно знать, что всякий раз, когда в объявлении метода встречается параметр типа *CharSequence*, в качестве этого параметра можно передать аргумент типа *String*.

3.6.8. Оперативно доступная документация на API

Как упоминалось выше, класс `String` содержит немало методов. Более того, в стандартной библиотеке существуют тысячи классов, содержащих огромное количество методов, и запомнить все полезные классы и методы просто невозможно. Поэтому очень важно уметь пользоваться оперативно доступной документацией на прикладной программный интерфейс API, чтобы быстро находить в ней сведения о классах и методах из стандартной библиотеки. Документацию на прикладной интерфейс API можно загрузить и сохранить локально или же перейти к ее оперативно доступному варианту, введя в своем браузере адрес <https://docs.oracle.com/javase/9/docs/api/overview-summary.html>.

Начиная с версии Java 9, в документации на прикладной интерфейс API предоставляется поле поиска (рис. 3.2). Прежние версии содержат фреймы со списками пакетов и классов. Для того чтобы получить эти списки, достаточно выбрать пункт меню `Frames` (Фреймы). Например, чтобы получить дополнительные сведения о методах класса `String`, введите **`String`** в поле поиска и выберите тип `java.lang.String` или найдите во фрейме с именами классов ссылку на этот класс и щелкните на ней. В итоге появится описание класса `String`, как показано на рис. 3.3.

Выполнив прокрутку вниз, вы увидите краткое описание всех методов данного класса. Методы расположены в алфавитном порядке (рис. 3.4). Щелкните на имени интересующего вас метода, чтобы получить его подробное описание (рис. 3.5). Если, например, вы щелкнете на ссылке `compareToIgnoreCase`, то получите описание одноименного метода.

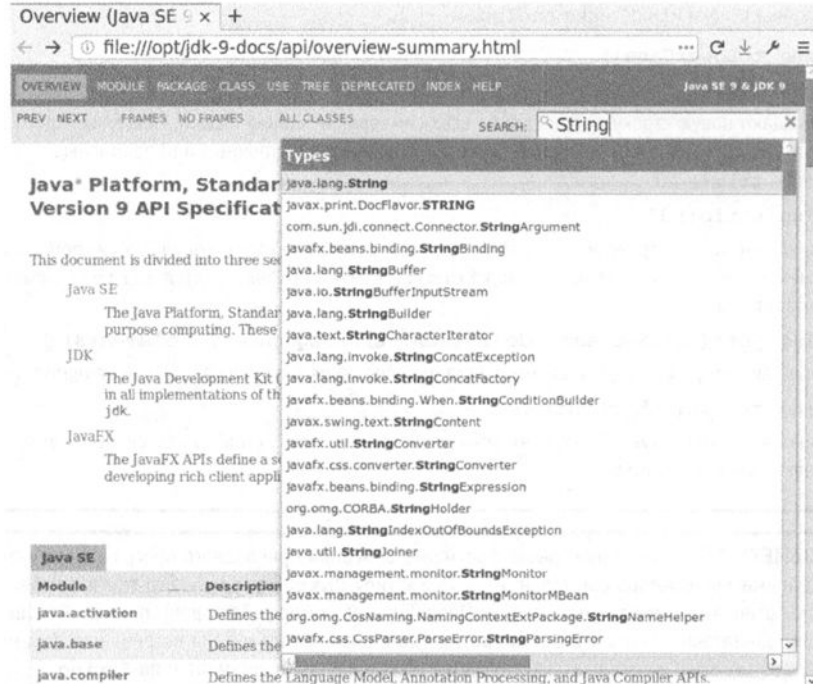


Рис. 3.2. Документация на прикладной интерфейс Java API

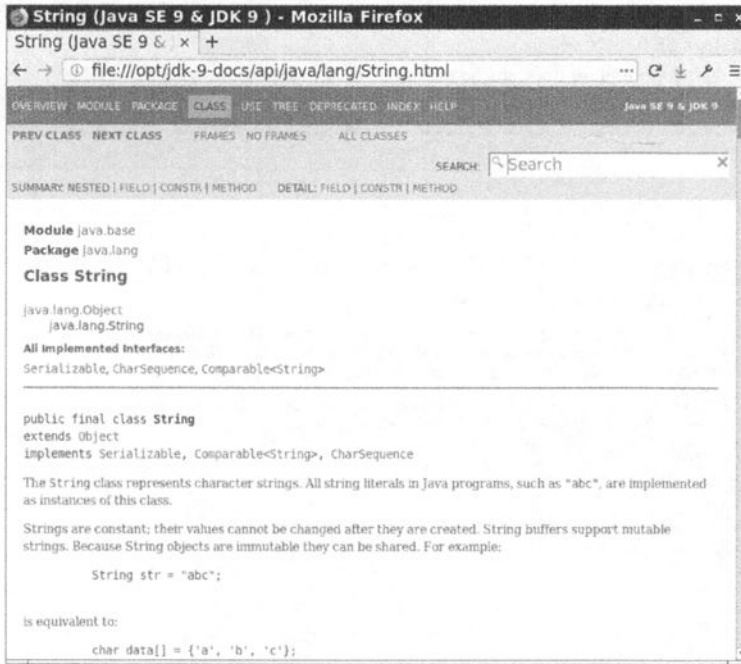


Рис. 3.3. Описание класса String

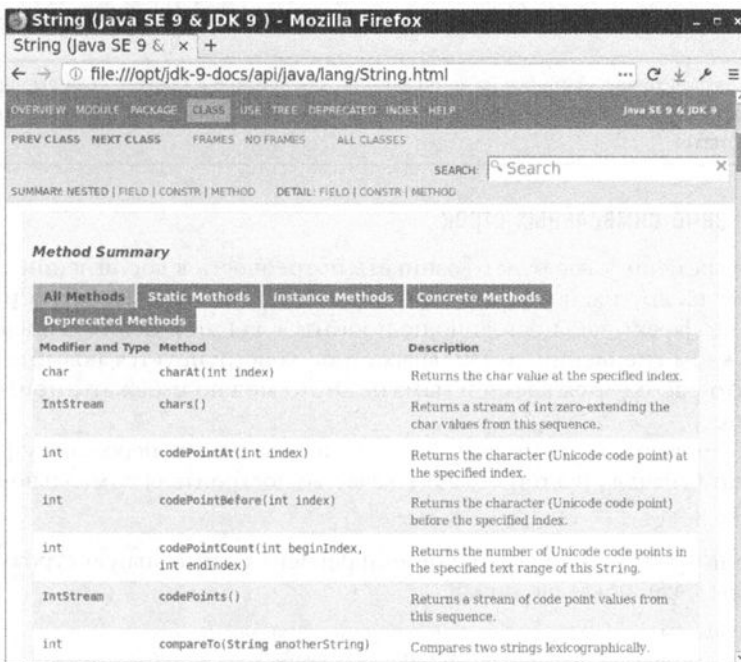


Рис. 3.4. Перечень методов из класса String

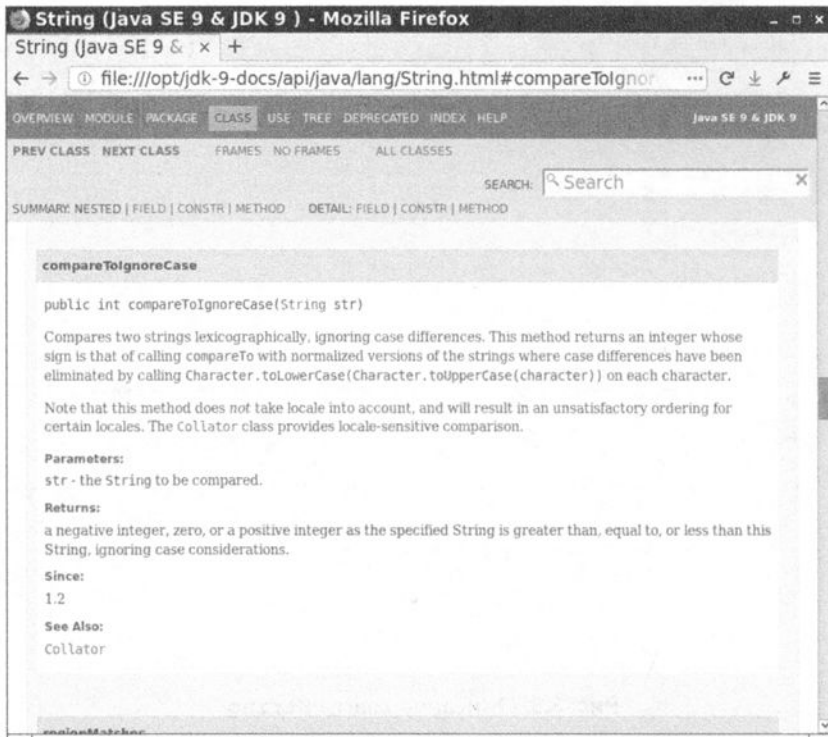


Рис. 3.5. Подробное описание метода `compareToIgnoreCase()` из класса `String`



СОВЕТ. Загрузите документацию на комплект JDK, как описано в главе 3, если вы еще не сделали этого. Сразу же сделайте в своем браузере закладку на страницу документации `jdk-9-docs/index.html`.

3.6.9. Построение символьных строк

Время от времени у вас будет возникать потребность в составлении одних символьных строк из других, более коротких строк, вводимых с клавиатуры или из файла. Было бы неэффективно постоянно пользоваться для этой цели сцеплением строк. Ведь при каждом сцеплении символьных строк конструируется новый объект типа `String`, на что расходуется время и память. Этого можно избежать, применяя класс `StringBuilder`.

Если требуется создать символьную строку из нескольких небольших фрагментов, сконструируйте сначала пустой объект в качестве построителя символьной строки:

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

Когда же потребуется добавить новый фрагмент в символьную строку, вызовите метод `append()`, как показано ниже.

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

Завершив составление символьной строки, вызовите метод `toString()`. Таким образом, вы получите объект типа `String`, состоящий из последовательности символов, содержащихся в объекте построителя символьных строк:

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



НА ЗАМЕТКУ! Класс `StringBuilder` появился в версии JDK 5.0. Его предшественник, класс `StringBuffer`, менее эффективен, но позволяет добавлять и удалять символы во многих потоках исполнения. Если же редактирование символьной строки происходит полностью в одном потоке исполнения (как это обычно и бывает), то следует, напротив, использовать класс `StringBuilder`. Прикладные программные интерфейсы API обоих классов идентичны.

В приведенном ниже описании прикладного программного интерфейса API перечислены наиболее употребительные конструкторы и методы из класса `StringBuilder`.

`java.lang.StringBuilder` 5.0

- **`StringBuilder()`**
Конструирует пустой объект построителя символьных строк.
- **`int length()`**
Возвращает количество кодовых единиц из объекта построителя символьных строк или буфера.
- **`StringBuilder append(String str)`**
Добавляет строку и возвращает ссылку `this` на текущий объект построителя символьных строк.
- **`StringBuilder append(char c)`**
Добавляет кодовую единицу и возвращает ссылку `this` на текущий объект построителя символьных строк.
- **`StringBuilder appendCodePoint(int cp)`**
Добавляет кодовую точку, преобразуя ее в одну или две кодовые единицы, возвращает ссылку `this` на текущий объект построителя символьных строк.
- **`void setCharAt(int i, int c)`**
Устанавливает символ `c` на позиции `i`-й кодовой единицы.
- **`StringBuilder insert(int offset, String str)`**
Вставляет строку на позиции `offset` и возвращает ссылку `this` на текущий объект построителя символьных строк.
- **`StringBuilder insert(int offset, char c)`**
Вставляет кодовую единицу на позиции `offset` и возвращает ссылку `this` на текущий объект построителя символьных строк.
- **`StringBuilder delete(int startIndex, int endIndex)`**
Удаляет кодовые единицы со смещениями от `startIndex` до `endIndex` - 1 и возвращает ссылку `this` на текущий объект построителя символьных строк.
- **`String toString()`**
Вставляет строку, содержащую те же данные, что и объект построителя символьных строк или буфер.

3.7. Ввод и вывод

Для того чтобы немного “оживить” программы, рассматриваемые здесь в качестве примеров, организуем ввод информации и форматирование выводимых данных. Безусловно, в современных приложениях для ввода данных используются элементы графического интерфейса, но для программирования такого интерфейса требуются приемы и инструментальные средства, которые пока еще не рассматривались. В этой главе преследуется цель — ознакомить вас с основными языковыми средствами Java, поэтому ограничимся лишь консольным вводом-выводом.

3.7.1. Чтение вводимых данных

Как вам должно быть уже известно, информацию можно легко направить в стандартный поток вывода (т.е. в консольное окно), вызвав метод `System.out.println()`. А вот организовать чтение из стандартного потока ввода `System.in` (т.е. с консоли) не так-то просто. Для этого придется создать объект типа `Scanner` и связать его со стандартным потоком ввода `System.in`, как показано ниже.

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

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

Сделав это, можно получить в свое распоряжение многочисленные методы из класса `Scanner`, предназначенные для чтения вводимых данных. Например, метод `nextLine()` осуществляет чтение вводимой строки, как показано ниже.

```
System.out.print("What is your name? ");  
String name = in.nextLine();
```

В данном случае метод `nextLine()` был применен, потому что вводимая строка может содержать пробелы. А для того чтобы прочитать одно слово, отделяемое пробелами, можно сделать следующий вызов:

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

Для чтения целочисленного значения служит приведенный ниже метод `nextInt()`. Нетрудно догадаться, что метод `nextDouble()` осуществляет чтение очередного числового значения в формате с плавающей точкой.

```
System.out.print("How old are you? ");  
int age = in.nextInt();
```

В примере программы, исходный код которой представлен в листинге 3.2, сначала запрашивается имя пользователя и его возраст, а затем выводится сообщение, аналогичное следующему:

```
Hello, Cay. Next year, you'll be 571
```

Исходный код этой программы начинается со следующей строки:

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

Класс `Scanner` относится к пакету `java.util`. Если вы собираетесь использовать в своей программе класс, отсутствующий в базовом пакете `java.lang`, включите в ее исходный код оператор `import`. Более подробно пакеты и оператор `import` будут рассмотрены в главе 4.

¹Здравствуйте, Кей. В следующем году вам будет 57

Листинг 3.2. Исходный код из файла `InputTest/InputTest.java`

```
1 import java.util.*;
2
3 /**
4  * Эта программа демонстрирует консольный ввод
5  * @version 1.10 2004-02-10
6  * @author Cay Horstmann
7  */
8 public class InputTest
9 {
10     public static void main(String[] args)
11     {
12         Scanner in = new Scanner(System.in);
13
14         // получить первую вводимую строку
15         System.out.print("What is your name? ");
16         String name = in.nextLine();
17
18         // получить вторую вводимую строку
19         System.out.print("How old are you? ");
20         int age = in.nextInt();
21
22         // вывести результат на консоль
23         System.out.println("Hello, " + name
24             + ". Next year, you'll be " + (age + 1));
25     }
26 }
```



НА ЗАМЕТКУ! Класс `Scanner` не подходит для ввода паролей с консоли, поскольку такой ввод будет явно виден всякому желающему. В версии Java 6 появился класс `Console`, специально предназначенный для этой цели. Чтобы организовать ввод пароля с консоли, можно воспользоваться следующим фрагментом кода:

```
Console cons = System.console();
String username = cons.readLine("User name: ");
char[] passwd = cons.readPassword("Password: ");
```

Из соображений безопасности пароль возвращается в виде массива символов, а не в виде символьной строки. После обработки пароля следует немедленно перезаписать элементы массива значением заполнителя. (Обработка массивов обсуждается в разделе 3.10.)

Обработка вводимых данных с помощью объекта типа `Console` не так удобна, как с помощью объекта типа `Scanner`, поскольку вводимые данные в этом случае можно читать только построчно. В классе `Console` отсутствуют методы для чтения отдельных слов или чисел.

java.util.Scanner 5.0

- **Scanner(InputStream in)**
Конструирует объект типа `Scanner` на основе заданного потока ввода.
- **String nextLine()**
Читает очередную строку.

java.util.Scanner 5.0 (окончание)

- **String next()**
Читает очередное вводимое слово, отделяемое пробелами.
- **int nextInt()**
- **double nextDouble()**
Читают очередную последовательность символов, представляющую целое число или число с плавающей точкой, выполняя соответствующее преобразование.
- **boolean hasNext()**
Проверяет, присутствует ли еще одно слово в потоке ввода.
- **boolean hasNextInt()**
- **boolean hasNextDouble()**
Проверяют, присутствует ли в потоке ввода последовательность символов, представляющая целое число или число с плавающей точкой.

java.lang.System 1.0

- **static Console console()**
Возвращает объект типа **Console** для взаимодействия с пользователем через консольное окно, а если такое взаимодействие невозможно — пустое значение **null**. Объект типа **Console** доступен в любой программе, запущенной в консольном окне. В противном случае его доступность зависит от конкретной системы.

java.io.Console 6

- **static char[] readPassword(String *prompt*, Object... *args*)**
- **static String readLine(String *prompt*, Object... *args*)**
Отображают приглашение и читают вводимые пользователем данные до тех пор, пока не получают конец вводимой строки. Параметры **args** могут быть использованы для предоставления аргументов форматирования, как поясняется в следующем разделе.

3.7.2. Форматирование выводимых данных

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

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

В ряде случаев это вызывает осложнения. Так, если вывести на экран крупную сумму в долларах и центах, большое количество цифр затруднит ее восприятие. В ранних версиях Java процесс форматирования чисел был сопряжен с определенными

трудностями. К счастью, в версии Java SE 5.0 была реализована в виде метода функция `printf()`, хорошо известная из библиотеки C.

Например, с помощью приведенного ниже оператора можно вывести значение **x** в виде числа, ширина поля которого составляет 8 цифр, а дробная часть равна двум цифрам. (Число цифр дробной части иначе называется *точностью*.)

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

В результате на экран будет выведено семь символов, не считая начальных пробелов.
3333.33

Метод `printf()` позволяет задавать произвольное количество параметров. Ниже приведен пример вызова этого метода с несколькими параметрами.

```
System.out.printf("Hello, %s. Next year, you'll be %d", name, age);
```

Каждый *спецификатор формата*, начинающийся с символа **%**, заменяется соответствующим параметром. А *символ преобразования*, которым завершается спецификатор формата, задает тип форматируемого значения: **f** — число с плавающей точкой; **s** — символьная строка; **d** — десятичное число. Все символы преобразования приведены в табл. 3.5.

Таблица 3.5. Символы преобразования для метода `printf()`

Символ преобразования	Тип	Пример
d	Десятичное целое	159
x	Шестнадцатеричное целое	9f
f	Число с фиксированной или плавающей точкой	15.9
e	Число с плавающей точкой в экспоненциальной форме	1.59e+01
g	Число с плавающей точкой в общем формате (чаще всего используется формат e или f , в зависимости от того, какой из них дает более короткую запись)	—
a	Шестнадцатеричное представление числа с плавающей точкой	0x1.fcddp3
s	Символьная строка	Hello
c	Символ	H
b	Логическое значение	true
h	Хеш-код	42628b2
tx или Tx	Дата и время (T означает обозначение даты и времени прописными буквами)	Устарел, пользуйтесь классами из пакета <code>java.time</code> , как поясняется в главе 6 второго тома настоящего издания
%	Знак процента	%
n	Разделитель строк, зависящий от платформы	—

В спецификаторе формата могут также присутствовать *флаги*, управляющие форматом выходных данных. Назначение всех флагов вкратце описано в табл. 3.6. Например, запятая, указываемая в качестве флага, добавляет разделители групп. Так, в результате приведенного ниже вызова на экран будет выведена строка 3, 333.33.

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

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



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

Таблица 3.6. Флаги для метода `printf()`

Флаг	Назначение	Пример
+	Выводит знак не только для отрицательных, но и для положительных чисел	+3333.33
Пробел	Добавляет пробел перед положительными числами	3333.33
0	Выводит начальные нули	003333.33
-	Выравнивает поле по левому краю	3333.33
(Заключает отрицательные числа в скобки	(3333.33)
,	Задаёт использование разделителя групп	3,333.33
# (для формата <code>f</code>)	Всегда отображает десятичную точку	3,333.
# (для формата <code>x</code> или <code>o</code>)	Добавляет префикс <code>0x</code> или <code>0</code>	0xcafe
\$	Определяет индекс параметра, предназначенного для форматирования. Например, выражение <code>%1\$d %1\$x</code> указывает на то, что первый параметр должен быть сначала выведен в десятичной, а затем в шестнадцатеричной форме	159 9F
<	Задаёт форматирование того же самого значения, которое отформатировано предыдущим спецификатором. Например, выражение <code>%d %<x</code> указывает на то, что одно и то же значение должно быть представлено как в десятичной, так и в шестнадцатеричной форме	159 9F

Для составления отформатированной символьной строки без последующего ее вывода можно вызвать статический метод `String.format()`, как показано ниже.

```
String message = String.format(
    "Hello, %s. Next year, you'll be %d", name, age);
```

Ради полноты изложения рассмотрим вкратце параметры форматирования даты и времени в методе `printf()`. При написании нового кода следует пользоваться методами из пакета `java.time`, описываемого в главе 6 второго тома настоящего издания. Но в унаследованном коде можно еще встретить класс `Date` и связанные с ним параметры форматирования. Последовательность форматирования даты и времени состоит из двух символов и начинается с буквы `t`, после которой следует одна из букв, приведенных в табл. 3.7. Ниже демонстрируется пример такого форматирования.

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

В результате этого вызова выводятся текущая дата и время:

```
Mon Feb 09 18:05:19 PST 2015
```

Как следует из табл. 3.7, некоторые форматы предполагают отображение лишь отдельных составляющих даты — дня или месяца. Было бы неразумно многократно задавать дату лишь для того, чтобы отформатировать различные ее составляющие.

По этой причине в форматирующей строке даты может задаваться *индекс* форматируемого аргумента. Индекс должен следовать сразу после знака % и завершаться знаком \$, как показано ниже.

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

В результате этого вызова будет выведена следующая строка:

```
Due date: February 9, 2015
```

Можно также воспользоваться флагом <, который означает, что форматированию подлежит тот же аргумент, который был отформатирован последним. Так, приведенная ниже строка кода дает точно такой же результат, как и упомянутая выше.

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

Таблица 3.7. Символы для форматирования даты и времени

Символ преобразования	Выходные данные	Пример
c	Полные сведения о дате и времени	Mon Feb 04 18:05:19 PST 2015
F	Дата в формате ISO 8601	2015-02-09
D	Дата в формате, принятом в США (месяц/день/год)	02/09/2015
T	Время в 24-часовом формате	18:05:19
r	Время в 12-часовом формате	06:05:19 pm
R	Время в 24-часовом формате (без секунд)	18:05
Y	Год в виде четырех цифр (с начальными нулями, если требуется)	2015
y	Год в виде двух последних цифр (с начальными нулями, если требуется)	15
C	Год в виде двух первых цифр (с начальными нулями, если требуется)	20
B	Полное название месяца	February
b или h	Сокращенное название месяца	Feb
m	Месяц в виде двух цифр (с начальными нулями, если требуется)	02
d	День в виде двух цифр (с начальными нулями, если требуется)	09
e	День в виде одной или двух цифр (без начальных нулей)	9
A	Полное название дня недели	Monday
a	Сокращенное название дня недели	Mon
j	День года в виде трех цифр в пределах от 001 до 366 (с начальными нулями, если требуется)	069
H	Час в виде двух цифр в пределах от 00 до 23 (с начальными нулями, если требуется)	18
k	Час в виде одной или двух цифр в пределах от 0 до 23 (без 9, 18 начальных нулей)	

Символ преобразования	Выходные данные	Пример
I	Час в виде двух цифр в пределах от 01 до 12 (с начальными нулями, если требуется)	06
l	Час в виде одной или двух цифр в пределах от 01 до 12 (без начальных нулей)	6
M	Минуты в виде двух цифр (с начальными нулями, если требуется)	05
S	Секунды в виде двух цифр (с начальными нулями, если требуется)	19
L	Миллисекунды в виде трех цифр (с начальными нулями, если требуется)	047
N	Наносекунды в виде девяти цифр (с начальными нулями, если требуется)	047000000
P	Метка времени до полудня или после полудня строчными буквами	am или pm
Z	Смещение относительно времени по Гринвичу (GMT) по стандарту RFC 822	-0800
Z	Часовой пояс	PST (Стандартное тихоокеанское время)
s	Количество секунд от начала отсчета времени 1970-01-01 00:00:00 GMT	1078884319
Q	Количество миллисекунд от начала отсчета времени 1970-01-01 00:00:00 GMT	1078884319047



ВНИМАНИЕ! Индексы аргументов начинаются с единицы, а не с нуля. Так, выражение `%1$` задает форматирование первого аргумента. Это сделано для того, чтобы избежать конфликтов с флагом 0.

Итак, мы рассмотрели все особенности применения метода `printf()`. На рис. 3.6 приведена блок-схема, наглядно показывающая синтаксический порядок указания спецификаторов формата.

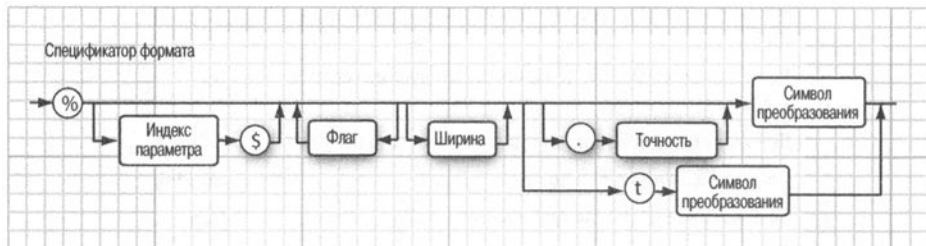


Рис. 3.6. Синтаксический порядок указания спецификаторов формата



НА ЗАМЕТКУ! Форматирование чисел и дат зависит от конкретных региональных настроек. Например, в Германии в качестве разделителя групп цифр в крупных числах принято употреблять точку, но не запятую, а понедельник форматируется как **Montag**. В главе 7 второго тома настоящего издания поясняется, каким образом осуществляется интернационализация приложений на Java.

3.7.3. Файловый ввод и вывод

Чтобы прочитать данные из файла, достаточно сконструировать объект типа **Scanner**:

```
Scanner in = new Scanner(Path.of("myfile.txt"),  
                          StandardCharsets.UTF_8);
```

Если имя файла содержит знаки обратной косой черты, их следует экранировать дополнительными знаками обратной косой черты, как, например, "c:\\mydirectory\\myfile.txt". После этого можно произвести чтение из файла, используя любые упомянутые выше методы из класса **Scanner**.



НА ЗАМЕТКУ! В данном примере указана кодировка символов UTF-8, которая является распространенной, хотя и не универсальной для файлов в Интернете. Для чтения текстовых файлов нужно знать кодировку символов (подробнее об этом — в главе 2 второго тома настоящего издания). Если опустить кодировку символов, то по умолчанию выбирается кодировка, действующая в той системе, где выполняется программа на Java. Но это не самая удачная идея, поскольку программа может действовать по-разному, в зависимости от того, где она выполняется.

А для того чтобы записать данные в файл, достаточно сконструировать объект типа **PrintWriter**, указав в его конструкторе имя файла:

```
PrintWriter out = new PrintWriter("myfile.txt",  
                                   StandardCharsets.UTF_8);
```

Если файл не существует, он создается. Для вывода в файл можно воспользоваться методами **print()**, **println()** и **printf()** точно так же, как это делается для вывода на консоль (или в стандартный поток вывода **System.out**).



ВНИМАНИЕ! Объект типа **Scanner** можно сконструировать со строковым параметром, но в этом случае символьная строка будет интерпретирована как данные, а не как имя файла. Так, если вызвать конструктор следующим образом:

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

объект типа **Scanner** интерпретирует заданное имя файла как отдельные символы 'м', 'й', 'ф' и т.д. Но ведь это совсем не то, что требуется.



НА ЗАМЕТКУ! Когда указывается относительное имя файла, например **"myfile.txt"**, **"mydirectory/myfile.txt"** или **"../myfile.txt"**, поиск файла осуществляется в том каталоге, в котором была запущена виртуальная машина Java. Если запустить программу на выполнение из командной строки следующим образом:

```
java MyProg
```

то начальным окажется текущий каталог командной оболочки. Но если программа запускается на выполнение в IDE, то начальный каталог определяется этой средой. Задать начальный каталог можно, сделав следующий вызов:

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

Если вы испытываете трудности при обнаружении файлов, попробуйте указать абсолютные имена путей вроде **"c:\\mydirectory\\myfile.txt"** или **"/home/me/mydirectory/myfile.txt"**.

Как видите, обращаться к файлам так же легко, как и при консольном вводе и выводе в стандартные потоки **System.in** и **System.out** соответственно. Правда, здесь имеется одна уловка: если вы конструируете объект типа **Scanner** с файлом, который

еще не существует, или объект типа `PrintWriter` с именем файла, который не может быть создан, возникает исключение. Компилятор Java рассматривает подобные исключения как более серьезные, чем, например, исключение при делении на нуль. В главе 7 будут рассмотрены различные способы обработки исключений. А до тех пор достаточно уведомить компилятор о том, что при файловом вводе и выводе может возникнуть исключение типа “файл не найден”. Для этого в объявление метода `main()` вводится предложение `throws`, как показано ниже.

```
public static void main(String[] args) throws IOException
{
    Scanner in = new Scanner(Path.of("myfile.txt"),
                               StandardCharsets.UTF_8);
    . . .
}
```

Теперь вы знаете, как читать и записывать текстовые данные в файлы. Более сложные вопросы файлового ввода-вывода, в том числе применение различных кодировок символов, обработка двоичных данных, чтение каталогов и запись архивных файлов, рассматриваются в главе 2 второго тома настоящего издания.



НА ЗАМЕТКУ! Запуская программу из командной строки, можно воспользоваться синтаксисом перенаправления ввода-вывода из командной оболочки, чтобы направить любой файл в стандартные потоки ввода-вывода `System.in` и `System.out`, как показано ниже.

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

В этом случае вам не придется обрабатывать исключение типа `IOException`.

`java.util.Scanner` 5.0

- `Scanner(Path p, String encoding)`

Конструирует объект типа `Scanner`, который читает данные из файла по указанному пути, используя заданную кодировку символов.

- `Scanner(String data)`

Конструирует объект типа `Scanner`, который читает данные из указанной символьной строки.

`java.io.PrintWriter` 1.1

- `PrintWriter(String fileName)`

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

`java.nio.file.Paths` 7

- `static Path get(String pathname)`

Конструирует объект типа `Path` для ввода данных из файла по указанному пути.

3.8. Управляющая логика

В языке Java, как и в любом другом языке программирования, в качестве логики, управляющей ходом выполнения программы, служат условные операторы и циклы. Рассмотрим сначала условные операторы, а затем перейдем к циклам. И завершим обсуждение управляющей логики довольно громоздким оператором `switch`, который можно применять для проверки многих значений одного выражения.



НА ЗАМЕТКУ C++! Языковые конструкции управляющей логики в Java такие же, как и в C и C++, за исключением двух особенностей. Среди них нет оператора безусловного перехода `goto`, но имеется версия оператора `break` с метками, который можно использовать для выхода из вложенного цикла (в языке C для этого пришлось бы применять оператор `goto`). Кроме того, в Java реализован вариант цикла `for`, не имеющий аналогов в C или C++. Его можно сравнить с циклом `foreach` в C#.

3.8.1. Область видимости блоков

Прежде чем перейти к обсуждению конкретных языковых конструкций управляющей логики, необходимо рассмотреть блоки. Блок состоит из ряда операторов Java, заключенных в фигурные скобки. Блоки определяют область видимости переменных и могут быть вложенными один в другой. Ниже приведен пример одного блока, вложенного в другой блок в методе `main()`.

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

В языке Java нельзя объявлять переменные с одинаковым именем в двух вложенных блоках. Например, приведенный ниже фрагмент кода содержит ошибку и не будет скомпилирован.

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



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

3.8.2. Условные операторы

Условный оператор `if` в Java имеет приведенную ниже форму. Условие должно быть заключено в скобки.

if (условие) оператор

В программах на Java, как и на большинстве других языков программирования, часто приходится выполнять много операторов в зависимости от истинности одного условия. В этом случае применяется *оператор задания блока*, как показано ниже.

```
{  
    Оператор;;  
    Оператор;;  
    ...  
}
```

Рассмотрим в качестве примера следующий фрагмент кода:

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

В этом фрагменте кода все операторы, заключенные в фигурные скобки, будут выполнены при условии, что значение переменной `yourSales` больше значения переменной `target` или равно ему (рис. 3.7).

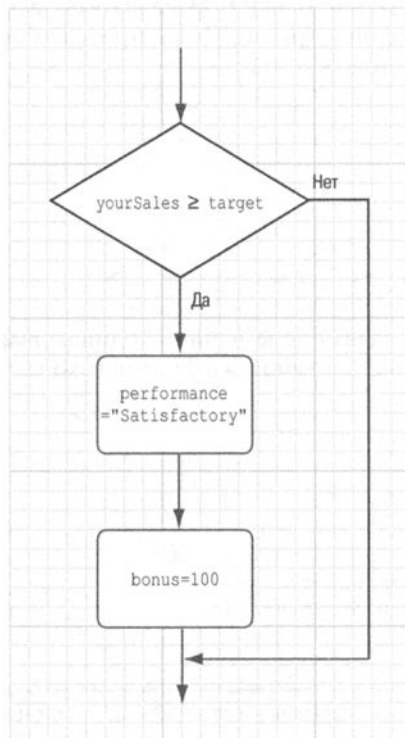


Рис. 3.7. Блок-схема, иллюстрирующая принцип действия условного оператора `if`



НА ЗАМЕТКУ! Блок, иначе называемый *составным оператором*, позволяет включать несколько (простых) операторов в любую языковую конструкцию Java, которая в противном случае допускает лишь один (простой) оператор.

Ниже приведена более общая форма условного оператора `if` в Java. А принцип его действия в данной форме наглядно показан на рис. 3.8 и в приведенном далее примере. `if` (условие) оператор; `else` оператор;

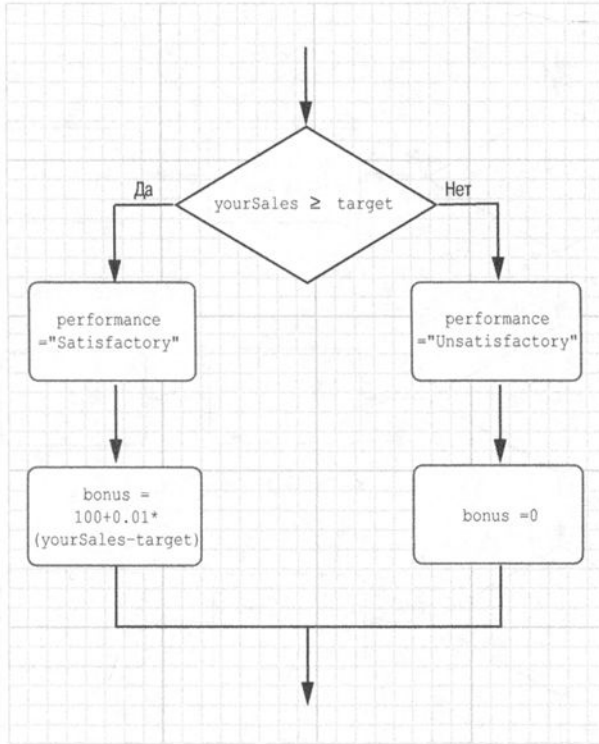


Рис. 3.8. Блок-схема, иллюстрирующая принцип действия условного оператора `if/else`

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

Часть `else` данного оператора не является обязательной и объединяется с ближайшим условным оператором `if`. Таким образом, в следующей строке кода оператор `else` относится ко второму оператору `if`:

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

Разумеется, для повышения удобочитаемости такого кода следует воспользоваться фигурными скобками, как показано ниже.

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

В программах на Java часто встречаются также повторяющиеся условные операторы `if...else if...` (рис. 3.9). Ниже приведен пример применения такой языковой конструкции непосредственно в коде.

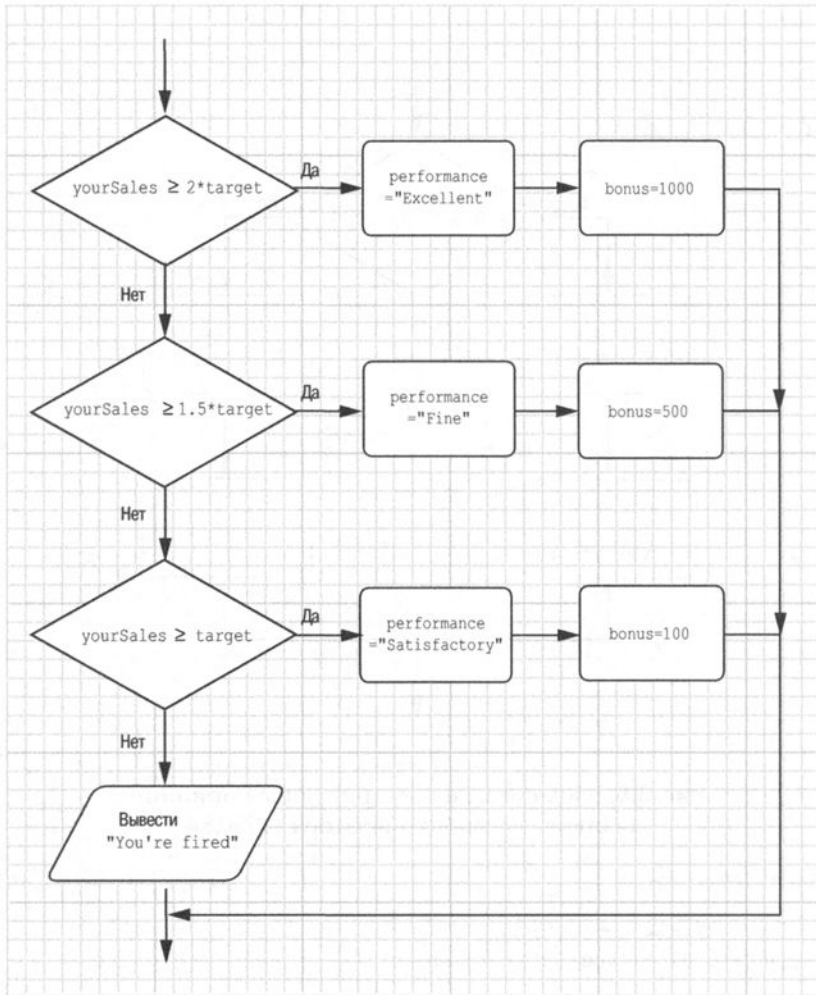


Рис. 3.9. Блок-схема, иллюстрирующая принцип действия условного оператора `if/else if` со множественным ветвлением

```
if (yourSales >= 2 * target)
{
    performance = "Excellent";
    bonus = 1000;
}
else if (yourSales >= 1.5 * target)
{
    performance = "Fine";
    bonus = 500;
}
```

```
else if (yourSales >= target)
{
    performance = "Satisfactory";
    bonus = 100;
}
else
{
    System.out.println("You're fired");
}
```

3.8.3. Неопределенные циклы

Цикл `while` выполняет выражение (или группу операторов, составляющих блок) до тех пор, пока условие истинно, т.е. оно имеет логическое значение `true`. Ниже приведена общая форма объявления этого цикла.

`while` (условие) оператор

Цикл `while` не будет выполнен ни разу, если его условие изначально ложно, т.е. имеет логическое значение `false` (рис. 3.10).

В листинге 3.3 приведен пример программы, подсчитывающей, сколько лет нужно вносить деньги на счет, чтобы накопить определенную сумму на заслуженный отдых. Считается, что каждый год вносится одна и та же сумма и процентная ставка не меняется. В теле цикла из данного примера увеличивается счетчик и обновляется сумма, накопленная на текущий момент. И так происходит до тех пор, пока итоговая сумма не превысит заданную величину:

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

(Не особенно полагайтесь на эту программу при планировании своей пенсии. В ней не учтены такие немаловажные детали, как инфляция и ожидаемая продолжительность вашей жизни.)

Условие цикла `while` проверяется в самом начале. Следовательно, возможна ситуация, когда код, содержащийся в блоке, образующем тело цикла, вообще не будет выполнен. Если же требуется, чтобы блок выполнялся хотя бы один раз, проверку условия следует перенести в конец. Это можно сделать с помощью цикла `do-while`, общая форма объявления которого приведена ниже.

`do` оператор `while` (условие);

Условие проверяется лишь после выполнения оператора, а зачастую блока операторов, в теле данного цикла. Затем цикл повторяется, снова проверяет условие и т.д. Например, программа, исходный код которой приведен в листинге 3.4, вычисляет новый остаток на пенсионном счету работника, а затем запрашивает, не собирается ли он на заслуженный отдых:

```
do
{
    balance += payment;
```



```

double interest = balance * interestRate / 100;
balance += interest;
year++;
// вывести текущий остаток на счету
. . .
// запросить готовность работника выйти на
// пенсию и получить ответ
. . .
}
while (input.equals("N"));

```

Цикл повторяется до тех пор, пока не будет получен положительный ответ "Y" (рис. 3.11). Эта программа служит характерным примером применения циклов, которые нужно выполнить хотя бы один раз, поскольку ее пользователь должен увидеть остаток на своем пенсионном счету, прежде чем решать, хватит ли ему средств на жизнь после выхода на пенсию.

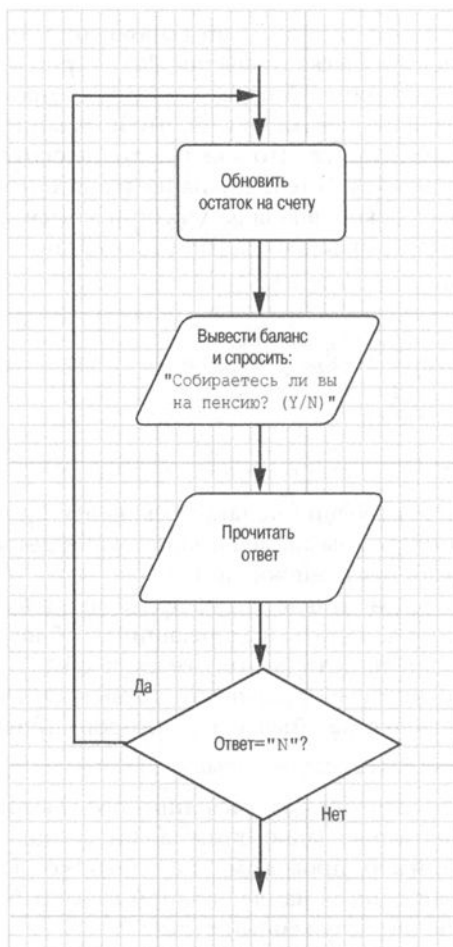
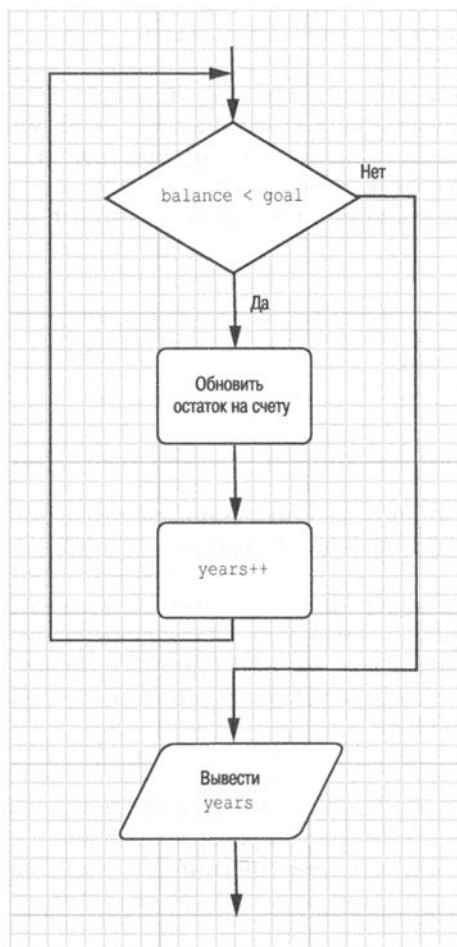


Рис. 3.10. Блок-схема, иллюстрирующая принцип действия оператора цикла while

Рис. 3.11. Блок-схема, поясняющая принцип действия оператора цикла do-while

Листинг 3.3. Исходный код из файла `Retirement/Retirement.java`

```
1  import java.util.*;
2
3  /**
4   * В этой программе демонстрируется применение
5   * цикла <code>while</code>
6   * @version 1.20 2004-02-10
7   * @author Cay Horstmann
8   */
9  public class Retirement
10 {
11     public static void main(String[] args)
12     {
13         // прочитать вводимые данные
14         Scanner in = new Scanner(System.in);
15
16         System.out.print("How much money do you "
17                         + "need to retire? ");
18         double goal = in.nextDouble();
19
20         System.out.print("How much money will you "
21                         + "contribute every year? ");
22         double payment = in.nextDouble();
23
24         System.out.print("Interest rate in %: ");
25         double interestRate = in.nextDouble();
26
27         double balance = 0;
28         int years = 0;
29
30         // обновить остаток на счету, пока не
31         // достигнута заданная сумма
32         while (balance < goal)
33         {
34             // добавить ежегодный взнос и проценты
35             balance += payment;
36             double interest = balance * interestRate / 100;
37             balance += interest;
38             years++;
39         }
40
41         System.out.println("You can retire in "
42                         + years + " years.");
43     }
44 }
```

Листинг 3.4. Исходный код из файла `Retirement2/Retirement2.java`

```
1  import java.util.*;
2
3  /**
4   * В этой программе демонстрируется применение
5   * цикла <code>do-while</code>
```

```
6  * @version 1.20 2004-02-10
7  * @author Cay Horstmann
8  */
9  public class Retirement2
10 {
11     public static void main(String[] args)
12     {
13         Scanner in = new Scanner(System.in);
14
15         System.out.print("How much money will you "
16                         + "contribute every year? ");
17         double payment = in.nextDouble();
18
19         System.out.print("Interest rate in %: ");
20         double interestRate = in.nextDouble();
21
22         double balance = 0;
23         int year = 0;
24
25         String input;
26
27         // обновить остаток на счете, пока работник
28         // не готов выйти на пенсию
29         do
30         {
31             // добавить ежегодный взнос и проценты
32             balance += payment;
33             double interest = balance * interestRate / 100;
34             balance += interest;
35
36             year++;
37
38             // вывести текущий остаток на счету
39             System.out.printf("After year %d,
40                             your balance is %,2f%n", year, balance);
41
42             // запросить готовность работника выйти
43             // на пенсию и получить ответ
44             System.out.print("Ready to retire? (Y/N) ");
45             input = in.next();
46         }
47         while (input.equals("N"));
48     }
49 }
```

3.8.4. Определенные циклы

Цикл `for` является весьма распространенной языковой конструкцией. В нем количество повторений находится под управлением переменной, выполняющей роль счетчика и обновляемой на каждом шаге цикла. В приведенном ниже примере цикла `for` на экран выводятся числа от 1 до 10, а порядок выполнения этого цикла наглядно показан на рис. 3.12.

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

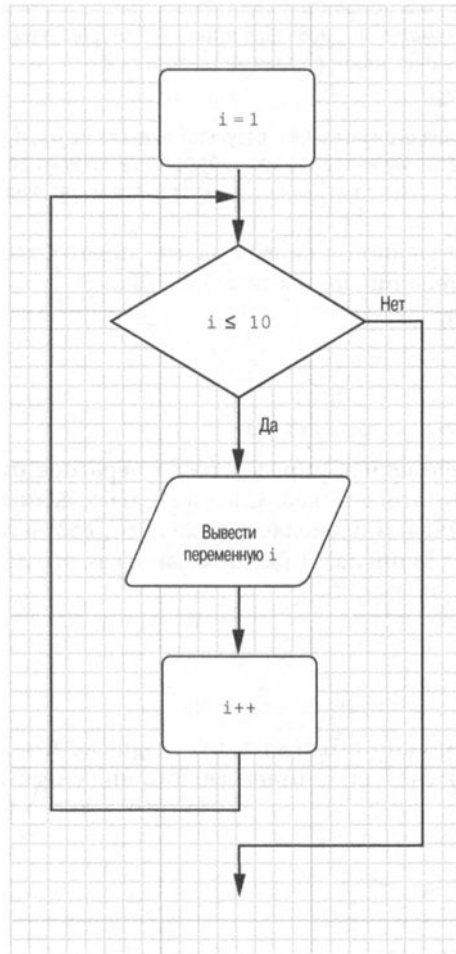


Рис. 3.12. Блок-схема, поясняющая порядок выполнения оператора цикла `for`

В первой части оператора `for` обычно выполняется инициализация счетчика, во второй его части задается условие выполнения тела цикла, а в третьей — порядок обновления счетчика.

Несмотря на то что в Java, как и в C++, отдельными частями оператора цикла `for` могут быть практически любые выражения, существуют неписанные правила, согласно которым все три части оператора цикла `for` должны только инициализировать, проверять и обновлять один и тот же счетчик. Если не придерживаться этих правил, полученный код станет неудобным, а то и вообще непригодным для чтения.

Подобные правила не слишком сковывают инициативу программирующего. Даже если придерживаться их, с помощью оператора `for` можно сделать немало, например, реализовать цикл с обратным отсчетом шагов:

```
for (int i = 10; i > 0; i--)  
    System.out.println("Counting down . . . " + i);  
System.out.println("Blastoff!");
```



ВНИМАНИЕ! Будьте внимательны, проверяя в цикле равенство двух чисел с плавающей точкой. Так, приведенный ниже цикл может вообще не завершиться.

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

Из-за ошибок округления окончательный результат никогда не будет получен. В частности, переменная `x` изменит свое значение с `9.999999999999998` сразу на `10.099999999999998`, потому что для числа `0.1` не существует точного двоичного представления.

При объявлении переменной в первой части оператора `for` ее область действия простирается до конца тела цикла, как показано ниже.

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

В частности, если переменная определена в операторе цикла `for`, ее нельзя использовать за пределами этого цикла. Следовательно, если требуется использовать конечное значение счетчика за пределами цикла `for`, соответствующую переменную следует объявить до начала цикла! Ниже показано, как это делается.

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

С другой стороны, можно объявлять переменные, имеющие одинаковое имя в разных циклах `for`, как следует из приведенного ниже примера кода.

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

Цикл `for` является сокращенным и более удобным вариантом цикла `while`. Например, следующий фрагмент кода:

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

можно переписать так, как показано ниже. И оба фрагмента кода будут равнозначны.

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

Типичный пример применения оператора цикла `for` непосредственно в коде приведен в листинге 3.5. Данная программа вычисляет вероятность выигрыша в лотерее. Так, если нужно угадать 6 номеров из 50, количество возможных вариантов будет равно $(50 \times 49 \times 48 \times 47 \times 46 \times 45) / (1 \times 2 \times 3 \times 4 \times 5 \times 6)$, поэтому шансы на выигрыш равны 1 из 15890700. Желаем удачи! Вообще говоря, если требуется угадать k номеров из n , количество возможных вариантов определяется следующей формулой:

$$\frac{n \times (n-1) \times (n-2) \times \dots \times (n-k+1)}{1 \times 2 \times 3 \times 4 \times \dots \times k}$$

Шансы на выигрыш по этой формуле рассчитываются с помощью следующего цикла `for`:

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



НА ЗАМЕТКУ! В разделе 3.10.3 будет рассмотрен обобщенный цикл `for`, называемый также циклом в стиле `for each`. Эта языковая конструкция была внедрена в версии Java 5.0.

Листинг 3.5. Исходный код из файла `LotteryOdds/LotteryOdds.java`

```
1  import java.util.*;
2  /**
3   * В этой программе демонстрируется применение цикла for
4   * @version 1.20 2004-02-10
5   * @author Cay Horstmann
6   */
7  public class LotteryOdds
8  {
9      public static void main(String[] args)
10     {
11         Scanner in = new Scanner(System.in);
12
13         System.out.print("How many numbers do you "
14             + "need to draw? ");
15         int k = in.nextInt();
16
17         System.out.print("What is the highest number "
18             + "you can draw? ");
19         int n = in.nextInt();
20         /*
21          * вычислить биномиальный коэффициент по формуле:
22          * n*(n-1)*(n-2)*...*(n-k+1)/(1*2*3*...*k)
23          */
24         int lotteryOdds = 1;
25         for (int i = 1; i <= k; i++)
26             lotteryOdds = lotteryOdds * (n - i + 1) / i;
27
28         System.out.println("Your odds are 1 in "
29             + lotteryOdds + ". Good luck!");
30     }
31 }
```

3.8.5. Оператор **switch** для многовариантного выбора

Языковая конструкция `if/else` может оказаться неудобной, если требуется организовать в коде выбор одного из многих вариантов. Для этой цели в Java имеется оператор `switch`, полностью соответствующий одноименному оператору в C и C++. Например, для выбора одного из четырех альтернативных вариантов (рис. 3.13) можно написать следующий код:

```
Scanner in = new Scanner(System.in);
System.out.print("Select an option (1, 2, 3, 4) ");
int choice = in.nextInt();
switch (choice)
{
    case 1:
        . . .
        break;
    case 2:
        . . .
        break;
    case 3:
        . . .
        break;
    case 4:
        . . .
        break;
    default:
        // неверный ввод
        . . .
        break;
}
```

Выполнение начинается с метки ветви `case`, соответствующей значению 1 переменной `choice`, и продолжается до очередного оператора `break` или конца оператора `switch`. Если ни одна из меток ветвей `case` не совпадает со значением переменной, выполняется выражение по метке ветви `default` (если таковое предусмотрено).



ВНИМАНИЕ! Если не ввести оператор **break** в конце ветви **case**, то возможно последовательное выполнение кода по нескольким ветвям **case**. Очевидно, что такая ситуация чревата ошибками, поэтому пользоваться оператором **switch** не рекомендуется. Если же вы предпочитаете пользоваться оператором **switch** в своих программах, скомпилируйте их исходный код с параметром **-Xlint:fallthrough**, как показано ниже.

```
javac -Xlint:fallthrough Test.java
```

В этом случае компилятор выдаст предупреждающее сообщение, если альтернативный выбор не завершается оператором **break**.

А если требуется последовательное выполнение кода по нескольким ветвям **case**, объемлющий метод следует пометить аннотацией **@SuppressWarnings("fallthrough")**. В таком случае никаких предупреждений для данного метода не выдается. (Аннотация служит механизмом для предоставления дополнительных сведений компилятору или инструментальному средству, обрабатывающему файлы с исходным кодом Java или классами. Более подробно аннотации будут рассматриваться в главе 8 второго тома настоящего издания.)

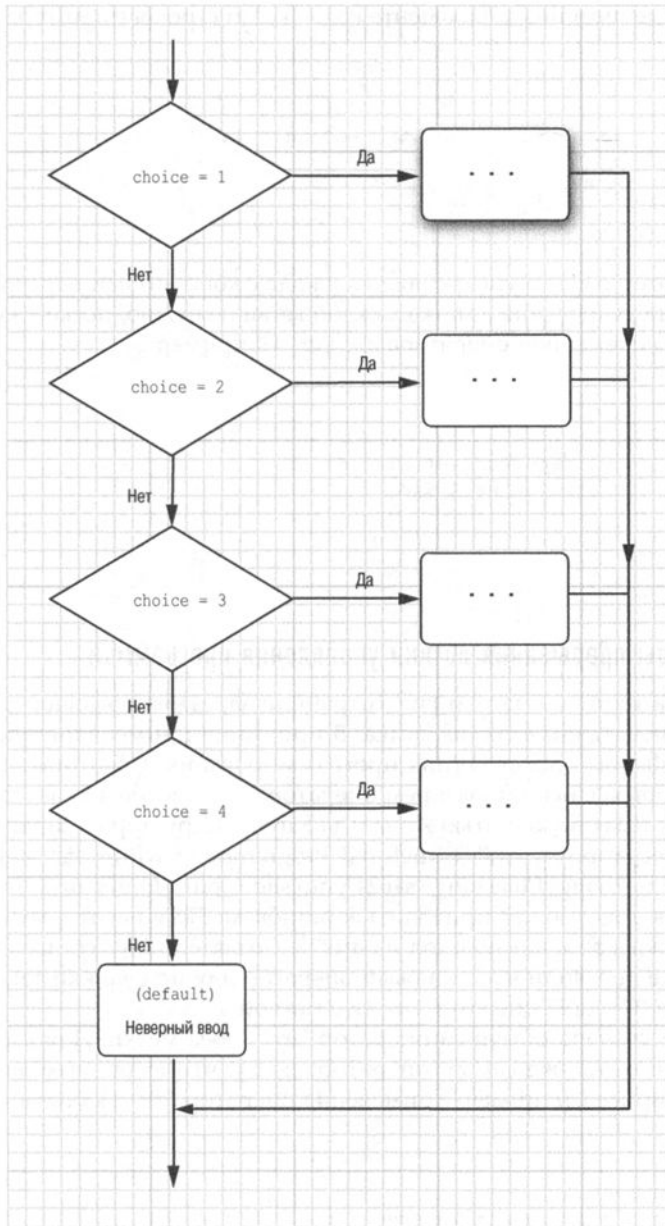


Рис. 3.13. Блок-схема, поясняющая принцип действия оператора `switch`

В качестве метки ветви `case` может быть указано следующее.

- Константное выражение типа `char`, `byte`, `short` или `int`.
- Константа перечислимого типа.
- Строковый литерал, начиная с версии Java 7.

Так, в приведенном ниже фрагменте кода указан строковый литерал в ветви `case`.

```
String input = . . . ;
switch (input.toLowerCase())
{
    case "yes": // допустимо, начиная с версии Java 7
        . . .
        break;
        . . .
}
```

Когда оператор `switch` употребляется в коде с константами перечислимого типа, указывать имя перечисления в метке каждой ветви не нужно, поскольку оно выводится из значения переменной оператора `switch`. Например:

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

3.8.6. Операторы прерывания логики управления программой

Несмотря на то что создатели Java сохранили зарезервированное слово `goto`, они решили не включать его в состав языка. В принципе применение операторов `goto` считается признаком плохого стиля программирования. Некоторые программисты считают, что борьба с использованием оператора `goto` ведется недостаточно активно (см., например, известную статью Дональда Кнута “Структурное программирование с помощью операторов `goto`” (Structured Programming with `goto` statements; <http://people.cs.pitt.edu/~zhangyt/teaching/cs1621/goto.slides.pdf>). Они считают, что применение операторов `goto` может приводить к ошибкам. Но в некоторых случаях нужно выполнять преждевременный выход из цикла. Создатели Java согласились с их аргументами и даже добавили в язык новый оператор для поддержки такого стиля программирования. Им стал оператор `break` с меткой.

Рассмотрим сначала обычный оператор `break` без метки. Для выхода из цикла можно применять тот же самый оператор `break`, что и для выхода из оператора `switch`. Ниже приведен пример применения оператора `break` без метки.

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

В данном фрагменте кода выход из цикла осуществляется при выполнении одного из двух условий: `years > 100` в начале цикла или `balance >= goal` в теле цикла. Разумеется, то же самое значение переменной `years` можно было бы вычислить и без применения оператора `break`, как показано ниже.

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

Однако следует иметь в виду, что проверка условия `balance < goal` в данном варианте кода повторяется дважды. Этого позволяет избежать оператор `break`.

В отличие от C++, в Java поддерживается оператор `break` с *меткой*, обеспечивающий выход из вложенных циклов. С его помощью можно организовать прерывание глубоко вложенных циклов при нарушении логики управления программой. А задавать дополнительные условия для проверки каждого вложенного цикла попросту неудобно.

Ниже приведен пример, демонстрирующий применение оператора `break` с меткой в коде. Однако метка должна предшествовать тому внешнему циклу, из которого требуется выйти. Кроме того, метка должна оканчиваться двоеточием.

```
Scanner in = new Scanner(System.in);
int n;
read_data:
while ( . . . ) // Этот цикл помечен меткой
{
    . . .
    for ( . . . ) // Этот цикл не помечен
    {
        System.out.print("Enter a number >= 0: ");
        n = in.nextInt();
        if (n < 0) // условие для прерывания цикла
            break read_data;
            // прервать цикл
        . . .
    }
}
// этот оператор выполняется сразу же после
// оператора break с меткой
if (n < 0) // поверить наличие недопустимой ситуации
{
    // принять меры против недопустимой ситуации
}
else
{
    // выполнить действия при нормальном ходе
    // выполнения программы
}
```

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



НА ЗАМЕТКУ! Любопытно, что метку можно связать с любым оператором — даже с условным оператором `if` или блоком, как показано ниже.

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

Итак, если вам крайне необходим оператор **goto** для безусловного перехода, поместите блок, из которого нужно немедленно выйти, непосредственно перед тем местом, куда требуется перейти, и примените оператор **break**! Естественно, такой прием не рекомендуется применять в практике программирования на Java. Следует также иметь в виду, что подобным способом можно выйти из блока, но невозможно войти в него.

Существует также оператор **continue**, который, подобно оператору **break**, прерывает нормальный ход выполнения программы. Оператор **continue** передает управление в начало текущего вложенного цикла. Ниже приведен характерный пример применения данного оператора.

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

Если $n < 0$, то оператор **continue** выполняет переход в начало цикла, пропуская оставшуюся часть текущего шага цикла. Если же оператор **continue** применяется в цикле **for**, он передает управление оператору увеличения счетчика цикла. В качестве примера рассмотрим следующий цикл:

```
for (count = 1; count <= 100; count++)
{
    System.out.print("Enter a number, -1 to quit: ");
    n = in.nextInt();
    if (n < 0) continue;
    sum += n; // не выполняется, если n < 0
}
```

Если $n < 0$, оператор **continue** перейдет к оператору **count++**. В языке Java имеется также оператор **continue** с меткой, передающий управление заголовку оператора цикла, помеченного соответствующей меткой.



СОВЕТ. Многие программирующие на Java считают, что операторы **break** и **continue** неоправданно усложняют текст программы. Применять эти операторы совсем не обязательно, поскольку те же самые действия можно реализовать, не прибегая к ним. В данной книге операторы **break** и **continue** не употребляются нигде, кроме примеров кода, приведенных в этом разделе.

3.9. Большие числа

Если для решения задачи недостаточно точности основных типов, чтобы представить целые и вещественные числа, то можно обратиться к классам **BigInteger** и **BigDecimal** из пакета **java.math**. Эти классы предназначены для выполнения

действий с числами, состоящими из произвольного количества цифр. В классах `BigInteger` и `BigDecimal` реализуются арифметические операции произвольной точности соответственно для целых и вещественных чисел.

Для преобразования обычного числа в число с произвольной точностью (называемого также *большим числом*) служит статический метод `valueOf()`:

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

Для представления более длинных чисел служит приведенный ниже конструктор со строковым параметром.

```
BigInteger reallyBig= new BigInteger
("222232244629420445529739893461909967206666939096499764990979600");
```

Имеются также константы `BigInteger.ZERO`, `BigInteger.ONE`, `BigInteger.TEN`, а в версии Java 9 — и константа `BigInteger.TWO`.

К сожалению, над большими числами нельзя выполнять обычные математические операции вроде `+` или `*`. Вместо этого следует применять методы `add()` и `multiply()` из классов соответствующих типов больших чисел, как в приведенном ниже примере кода.

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



НА ЗАМЕТКУ C++! В отличие от C++, перегрузка операций в Java не поддерживается. Поэтому разработчики класса `BigInteger` были лишены возможности переопределить операции `+` и `*` для методов `add()` и `multiply()`, чтобы реализовать в классе `BigInteger` аналоги операций сложения и умножения соответственно. Создатели Java реализовали только перегрузку операции `+` для обозначения сцепления строк. Они решили не перегружать остальные операции и не предоставили такой возможности программирующим на Java в их собственных классах.

В листинге 3.6 приведен видоизмененный вариант программы из листинга 3.5 для подсчета шансов на выигрыш в лотерее. Теперь эта программа может оперировать большими числами. Так, если вам предложат сыграть в лотерею, в которой нужно угадать 60 чисел из 490 возможных, эта программа сообщит, что ваши шансы на выигрыш составляют 1 из 716395843461995557415116222540092933411717612789263493493351013459481104668848. Желаем удачи!

В программе из листинга 3.5 вычислялось следующее выражение:

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

Для обработки больших чисел соответствующая строка кода будет выглядеть следующим образом:

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

Листинг 3.6. Исходный код из файла `BigIntegerTest/BigIntegerTest.java`

```
1 import java.math.*;
2 import java.util.*;
3
4 /**
5  * В этой программе используются большие числа для
6  * оценки шансов на выигрыша в лотерею
```

```
7  * @version 1.20 2004-02-10
8  * @author Cay Horstmann
9  */
10 public class BigIntegerTest
11 {
12     public static void main(String[] args)
13     {
14         Scanner in = new Scanner(System.in);
15
16         System.out.print("How many numbers do you "
17             + "need to draw? ");
18         int k = in.nextInt();
19
20         System.out.print("What is the highest number you "
21             + "can draw? ");
22         int n = in.nextInt();
23
24         /*
25          *  вычислить биномиальный коэффициент по формуле:
26          *   $n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot (n-k+1) / (1 \cdot 2 \cdot 3 \cdot \dots \cdot k)$ 
27          *
28          */
29
30         BigInteger lotteryOdds = BigInteger.valueOf(1);
31
32         for (int i = 1; i <= k; i++)
33             lotteryOdds = lotteryOdds.multiply(BigInteger
34                 .valueOf(n - i + 1))
35                 .divide(BigInteger
36                     .valueOf(i));
37
38         System.out.println("Your odds are 1 in "
39             + lotteryOdds + ". Good luck!");
40     }
41 }
```

java.math.BigInteger 1.1

- **BigInteger subtract(BigInteger other)**
- **BigInteger multiply(BigInteger other)**
- **BigInteger divide(BigInteger other)**
- **BigInteger mod(BigInteger other)**

Возвращают сумму, разность, произведение, частное и остаток от деления, полученные в результате выполнения соответствующих операций над текущим большим числом и значением параметра *other*.

- **BigInteger sqrt()** 9

Возвращает квадратный корень данного числа типа **BigInteger**.

java.math.BigInteger 1.1 (Окончание)

- **BigInteger subtract(BigInteger other)**
- **BigInteger multiply(BigInteger other)**
- **BigInteger divide(BigInteger other)**
- **BigInteger mod(BigInteger other)**

Возвращают сумму, разность, произведение, частное и остаток от деления, полученные в результате выполнения соответствующих операций над текущим большим числом и значением параметра *other*.

- **BigInteger sqrt() 9**

Возвращает квадратный корень данного числа типа **BigInteger**.

- **int compareTo(BigInteger other)**

Возвращает 0, если текущее большое число равно значению параметра *other*, отрицательное число, если это большое число меньше значения параметра *other*, а иначе — положительное число.

- **static BigInteger valueOf(long x)**

Возвращает большое число, равное значению параметра *x*.

java.math.BigDecimal 1.1

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

Возвращают сумму, разность, произведение и частное от деления, полученные в результате соответствующих операций над текущим большим числом и значением параметра *other*. Чтобы вычислить частное от деления, следует указать режим округления. Так, режим **roundingMode**. **ROUND_HALF_UP** означает округление в сторону уменьшения для цифр 0–4 и в сторону увеличения для цифр 5–9. Для обычных вычислений этого оказывается достаточно. Другие режимы округления описаны в документации.

- **int compareTo(BigDecimal other)**

Возвращает 0, если текущее число типа **BigDecimal** равно значению параметра *other*, отрицательное число, если это число меньше значения параметра *other*, а иначе — положительное число.

- **static BigDecimal valueOf(long x)**

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

Возвращают большое десятичное число, значение которого равно значению параметра *x* или $x / 10^{\text{scale}}$.

3.10. Массивы

Массивы служат для хранения последовательностей однотипных значений. В следующих разделах будет показано, как обращаться с массивами в Java.

3.10.1. Объявление массивов

Массив — это структура данных, в которой хранятся величины одинакового типа. Доступ к отдельному элементу массива осуществляется по целочисленному *индексу*. Так, если *a* — массив целых чисел, то *a[i]* — *i*-е целое число в массиве. Массив объявляется следующим образом: сначала указывается тип массива, т.е. тип элементов, содержащихся в нем, затем следует пара пустых квадратных скобок, а после них — имя переменной. Ниже приведено объявление массива, состоящего из целых чисел.

```
int[] a;
```

Но этот оператор лишь объявляет переменную *a*, не инициализируя ее. Чтобы создать массив, нужно выполнить операцию *new*, как показано ниже.

```
int[] a = new int[100]; // или var a = new int[100];
```

В этой строке кода создается массив, состоящий из 100 целых чисел. Длина массива не обязательно должна быть постоянной. Так, операция *new int[n]* создает массив длиной *n*.

Как только массив будет создан, изменить его длину нельзя, хотя это и можно, конечно, сделать, изменив отдельный элемент массива. Если приходится часто изменять длину массива во время выполнения прикладной программы, то лучше воспользоваться *списочным массивом*, как поясняется в главе 5.



НА ЗАМЕТКУ! Объявить массив можно двумя способами:

```
int[] a;
```

или

```
int a[];
```

Большинство программирующих на Java пользуются первым способом, поскольку в этом случае тип более явно отделяется от имени переменной.

В языке Java имеется синтаксическая конструкция для одновременного создания массива и его инициализации. Пример такой синтаксической конструкции приведен ниже.

```
int[] smallPrimes = { 2, 3, 5, 7, 11, 13};
```

Обратите внимание на то, что в данном случае операция *new* не требуется. Последнее инициализируемое значение допускается завершать запятой, что может быть удобно для последующего расширения массива, как показано ниже.

```
String[] authors = {  
    "James Gosling",  
    "Bill Joy",  
    "Guy Steele",  
    // ввести здесь дополнительные имена,  
    // завершая каждое из них запятой  
};
```

Можно даже инициализировать массив, не имеющий имени или называемый иначе *анонимным*:

```
new int[] {16, 19, 23, 29, 31, 37}
```

В этом выражении выделяется память для нового массива, а его элементы заполняются числами, указанными в фигурных скобках. При этом подсчитывается их

количество и соответственно определяется размер массива. Такую синтаксическую конструкцию удобно применять для повторной инициализации массива без необходимости создавать новую переменную. Например, выражение

```
smallPrimes = new int{ 17, 19, 23, 29, 31, 37 };
```

является сокращенной записью приведенного ниже фрагмента кода.

```
int[] anonymous = { 17, 19, 23, 29, 31, 37 };
smallPrimes = anonymous;
```



НА ЗАМЕТКУ! При необходимости можно создать массив нулевого размера. Такой массив может оказаться полезным при написании метода, возвращающего массив, который оказывается в некоторых случаях пустым. Массив нулевой длины объявляется следующим образом:

```
new тип_элементов[0]
или
new тип_элементов[] {}
```

Следует, однако, иметь в виду, что массив нулевой длины не равнозначен массиву с пустыми значениями `null`.

3.10.2. Доступ к элементам массива

Элементы сформированного выше массива нумеруются от 0 до 99, а не от 1 до 100. После создания массива его можно заполнить конкретными значениями. Это, в частности, можно делать в цикле:

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

При создании массива чисел все его элементы инициализируются нулями. Массивы значений типа `boolean` инициализируются логическим значением `false`, а массивы объектов — пустым значением `null`, указывающим на то, что массив пока еще не содержит ни одного объекта. Для начинающих это может показаться неожиданным. Так, в приведенной ниже строке кода создается массив из десяти символьных строк, причем все они нулевые, т.е. имеют пустое значение `null`.

```
String[] names = new String[10];
```

Если же требуется создать массив из пустых символьных строк, его придется специально заполнить пустыми строками, как показано ниже.

```
for (int i = 0; i < 10; i++) names[i] = "";
```



ВНИМАНИЕ! Если, создав массив, состоящий из 100 элементов, вы попытаетесь обратиться к элементу `a[100]` (или любому другому элементу, индекс которого выходит за пределы от 0 до 99), выполнение программы прервется, поскольку будет сгенерировано исключение в связи с выходом индекса массива за допустимые пределы.

Для определения количества элементов в массиве служит свойство `имя_массива.length`. Например, в следующем фрагменте кода свойство `length` используется для вывода элементов массива:

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


3.10.3. Цикл в стиле `for each`

В языке Java имеется эффективная разновидность цикла, позволяющая перебирать все элементы массива (а также любого другого набора данных), не применяя счетчик. Эта усовершенствованная разновидность цикла `for` записывается следующим образом:

`for` (переменная : коллекция) оператор

При выполнении этого цикла его переменной последовательно присваивается каждый элемент заданной коллекции, после чего выполняется указанный оператор (или блок). В качестве коллекции может быть задан массив или экземпляр класса, реализующего интерфейс `Iterable`, например `ArrayList`. Списочные массивы типа `ArrayList` будут обсуждаться в главе 5, а интерфейс `Iterable` — в главе 9.

В приведенном ниже примере кода рассматриваемый здесь цикл организуется для вывода каждого элемента массива `a` в отдельной строке.

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

Действие этого цикла можно кратко описать как обработку каждого элемента из массива `a`. Создатели Java рассматривали возможность применения ключевых слов `foreach` и `in` для обозначения данного типа цикла. Но такой тип цикла появился намного позже основных языковых средств Java, и введение нового ключевого слова привело бы к необходимости изменять исходный код некоторых уже существовавших приложений, содержащих переменные или методы с подобными именами, как, например, `System.in`.

Безусловно, действия, выполняемые с помощью этой разновидности цикла, можно произвести и средствами традиционного цикла `for`:

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

Но при использовании цикла в стиле `for each` запись получается более краткой, поскольку отпадает необходимость в начальном выражении и условии завершения цикла. Да и вероятность ошибок при написании кода уменьшается.



НА ЗАМЕТКУ! Переменная цикла в стиле `for each` перебирает элементы массива, а не значения индекса.

Несмотря на появление усовершенствованного цикла в стиле `for each`, упрощающего во многих случаях составление программ, традиционный цикл `for` совсем не устарел. Без него нельзя обойтись, например, в тех случаях, когда требуется обработать не всю коллекцию, а лишь ее часть, или тогда, когда счетчик явно используется в теле цикла.



СОВЕТ. Еще более простой способ вывести все значения из массива состоит в использовании метода `toString()` из класса `Arrays`. Так, в результате вызова `Arrays.toString(a)` будет возвращена символьная строка, содержащая все элементы массива, заключенные в квадратные скобки и разделенные запятыми, например: `"[2, 3, 5, 7, 11, 13]"`. Следовательно, чтобы вывести массив, достаточно сделать вызов `System.out.println(Arrays.toString(a)) ;`.

3.10.4. Копирование массивов

При необходимости одну переменную массива можно скопировать в другую, но в этом случае обе переменные будут ссылаться на один и тот же массив:

```
int[] luckyNumbers = smallPrimes;  
luckyNumbers[5] = 12; // теперь элемент smallPrimes[5] также равен 12
```

Результат копирования переменной массива приведен на рис. 3.14.

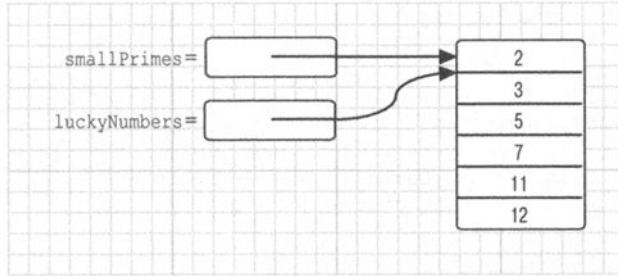


Рис. 3.14. Копирование переменной массива

Если требуется скопировать все элементы одного массива в другой, следует вызвать метод `copyTo()` из класса `Arrays`, как показано ниже.

```
int[] copiedLuckyNumbers = Arrays.copyOf(luckyNumbers, luckyNumbers.length);
```

В качестве второго параметра метода `copyTo()` указывается длина нового массива. Обычно этот метод применяется для увеличения размера массива следующим образом:

```
luckyNumbers = Arrays.copyOf(luckyNumbers, 2 * luckyNumbers.length);
```

Дополнительные элементы заполняются нулями, если массив содержит числа, или логическими значениями `false`, если это значения типа `boolean`. С другой стороны, если длина нового массива меньше длины исходного массива, то копируются только начальные элементы.



НА ЗАМЕТКУ C++! Массив в Java значительно отличается от массива в C++, располагающегося в стеке. Но переменную массива можно условно сравнить с указателем на динамически созданный массив. Таким образом, следующее выражение в Java:

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

можно сравнить с таким выражением в C++:

```
int* a = new int[100]; // C++
```

но оно существенно отличается от выражения

```
int a[100]; // C++
```

При выполнении операции `[]` в Java по умолчанию проверяются границы массива. Кроме того, в Java не поддерживается арифметика указателей. В частности, нельзя увеличить указатель на массив `a`, чтобы обратиться к следующему элементу этого массива.

3.10.5. Параметры командной строки

В каждом из рассмотренных ранее примеров программ на Java присутствовал метод `main()` с параметром `String[] args`. Этот параметр означает, что метод `main()` получает массив, элементами которого являются параметры, указанные в командной строке. Рассмотрим в качестве примера следующую программу:

```
public class Message
{
    public static void main(String[] args)
    {
        if (args[0].equals("-h"))
            System.out.print("Hello,");
        else if (args[0].equals("-g"))
            System.out.print("Goodbye,");
        // вывести остальные параметры командной строки
        for (int i = 1; i < args.length; i++)
            System.out.print(" " + args[i]);
        System.out.println("!");
    }
}
```

Если вызвать данную программу из командной строки следующим образом:

```
java Message -g cruel world
```

то массив `args` будет состоять из таких элементов:

```
args[0]: "-g"
args[1]: "cruel"
args[2]: "world"
```

А в результате выполнения данной программы будет выведено следующее сообщение:

```
Goodbye, cruel world!2
```



НА ЗАМЕТКУ C++! При запуске программы на Java ее имя не сохраняется в массиве `args`, передаваемом методу `main()`. Так, после запуска программы `Message` по команде `java Message -h world` из командной строки элемент массива `args[0]` будет содержать параметр `"-h"`, а не имя программы `"Message"` или команду `"java"`.

3.10.6. Сортировка массивов

Если требуется упорядочить массив чисел, для этого достаточно вызвать метод `sort()` из класса `Arrays`:

```
int[] a = new int[10000];
...
Arrays.sort(a);
```

В этом методе используется усовершенствованный вариант алгоритма быстрой сортировки, которая считается наиболее эффективной для большинства наборов данных. Класс `Arrays` содержит ряд удобных методов, предназначенных для работы с массивами. Эти методы приведены в конце раздела.

²Прощай, жестокий мир!

Программа, исходный код которой представлен в листинге 3.7, создает массив и генерирует случайную комбинацию чисел для лотереи. Так, если нужно угадать 6 чисел из 49, программа может вывести следующее сообщение:

```
Bet the following combination. It'll make you rich!3
```

```
4
7
8
19
30
44
```

Для выбора случайных чисел массив `numbers` сначала заполняется последовательностью чисел 1, 2, ..., *n*, как показано ниже.

```
int[] numbers = new int[n];
for (int i = 0; i < numbers.length; i++)
    numbers[i] = i + 1;
```

Второй массив служит для хранения сгенерированных чисел:

```
int[] result = new int[k];
```

Затем генерируется *k* чисел. Метод `Math.random()` возвращает случайное число с плавающей точкой, находящееся в пределах от 0 (включительно) до 1 (исключительно). Умножение результата на число *n* дает случайное число, находящееся в пределах от 0 до *n*-1, как показано в следующей строке кода:

```
int r = (int) (Math.random() * n);
```

Далее *i*-е число присваивается *i*-му элементу массива. Сначала там будет находиться результат *r*+1, но, как будет показано ниже, содержимое массива `number` будет изменяться после генерирования каждого нового числа.

```
result[i] = numbers[r];
```

Теперь следует убедиться, что ни одно число не повторится, т.е. все номера должны быть разными. Следовательно, нужно сохранить в элементе массива `number[r]` последнее число, содержащееся в массиве, и уменьшить *n* на единицу:

```
numbers[r] = numbers[n - 1];
n--;
```

Обратите внимание на то, что всякий раз при генерировании чисел получается индекс, а не само число. Это индекс массива, содержащего числа, которые еще не были выбраны. После генерирования *k* номеров лотереи сформированный в итоге массив `result` сортируется, чтобы результат выглядел более изящно:

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

Листинг 3.7. Исходный код из файла `LotteryDrawing/LotteryDrawing.java`

```
1 import java.util.*;
2
3 /**
```

³ Попробуйте следующую комбинацию, чтобы разбогатеть!

```
4  * В этой программе демонстрируется обращение с массивами
5  * @version 1.20 2004-02-10
6  * @author Cay Horstmann
7  */
8  public class LotteryDrawing
9  {
10     public static void main(String[] args)
11     {
12         Scanner in = new Scanner(System.in);
13
14         System.out.print("How many numbers do you "
15             + "need to draw? ");
16         int k = in.nextInt();
17
18         System.out.print("What is the highest number you "
19             + "can draw? ");
20         int n = in.nextInt();
21
22         // заполнить массив числами 1 2 3 . . . n
23         int[] numbers = new int[n];
24         for (int i = 0; i < numbers.length; i++)
25             numbers[i] = i + 1;
26
27         // выбрать k номеров и ввести их во второй массив
28         int[] result = new int[k];
29         for (int i = 0; i < result.length; i++)
30         {
31             // получить случайный индекс
32             // в пределах от 0 до n - 1
33             int r = (int) (Math.random() * n);
34
35             // выбрать элемент из произвольного места
36             result[i] = numbers[r];
37
38             // переместить последний элемент
39             // в произвольное место
40             numbers[r] = numbers[n - 1];
41             n--;
42         }
43
44         // вывести отсортированный массив
45         Arrays.sort(result);
46         System.out.println("Bet the following "
47             + "combination. It'll make you rich!");
48         for (int r : result)
49             System.out.println(r);
50     }
51 }
```

java.util.Arrays 1.2

- **static String toString(~~xxx~~[] a)** 5.0

Возвращает строку с элементами массива **a**, заключенную в квадратные скобки и разделенную запятыми. В этом и последующих методах типом ~~xxx~~ элемента массива может быть **int**, **long**, **short**, **char**, **byte**, **boolean**, **float** или **double**.

java.util.Arrays 1.2 (окончание)

- **static type copyOf(*xxx[]* a, int length) 6**
- **static type copyOf(*xxx[]* a, int start, int end) 6**

Возвращают массив такого же типа, что и заданный массив *a*, длиной *length* или *end - start* и заполненный значениями из массива *a*. Если значение параметра *end* оказывается меньше значения свойства *a.length*, то получаемый результат заполняется нулевыми значениями или же логическими значениями **false**.

- **static void sort(*xxx[]* a) 6**
Сортирует массив, используя алгоритм быстрой сортировки.
- **static int binarySearch(*xxx[]* a, *xxx* v) 6**
- **static int binarySearch(*xxx[]* a, int start, int end, *xxx* v) 6**

Используют алгоритм двоичного поиска для нахождения указанного значения *v*. При удачном исходе возвращается индекс найденного элемента. В противном случае возвращается отрицательное значение индекса *r*; а значение индекса $-r - 1$ указывает на место, куда должен быть вставлен искомый элемент, чтобы сохранился порядок сортировки.

- **static void fill(*xxx[]* a, *xxx* v)**
Устанавливает указанное значение *v* во всех элементах заданного массива *a*.
- **static boolean equals(*xxx[]* a, *xxx* b)**
Возвращает логическое значение **true**, если сравниваемые массивы имеют равную длину и совпадают все их элементы по индексу.

3.10.7. Многомерные массивы

Для доступа к элементам многомерного массива применяется несколько индексов. Такие массивы служат для хранения таблиц и более сложных упорядоченных структур данных. Если вы не собираетесь пользоваться многомерными массивами в своей практической деятельности, можете смело пропустить этот раздел.

Допустим, требуется составить таблицу чисел, показывающих, как возрастут первоначальные капиталовложения на сумму 10 тысяч долларов при разных процентных ставках, если прибыль ежегодно выплачивается и снова вкладывается в дело. Пример таких числовых данных приведен в табл. 3.8.

Таблица 3.8. Рост капиталовложений при разных процентных ставках

10%	11%	12%	13%	14%	15%
10000.00	10000.00	10000.00	10000.00	10000.00	10000.00
11000.00	11100.00	11200.00	11300.00	11400.00	11500.00
12100.00	12321.00	12544.00	12769.00	12996.00	13225.00
13310.00	13676.31	14049.28	14428.97	14815.44	15208.75
14641.00	15180.70	15735.19	16304.74	16889.60	17490.06
16105.10	16850.58	17623.42	18424.35	19254.15	20113.57
17715.61	18704.15	19738.23	20819.52	21949.73	23130.61
19487.17	20761.60	22106.81	23526.05	25022.69	26600.20
21435.89	23045.38	24759.63	26584.44	28525.86	30590.23
23579.48	25580.37	27730.79	30040.42	32519.49	35178.76

Очевидно, что эти данные лучше всего хранить в двумерном массиве (или матрице), например, под названием `balances`. Объявить двумерный массив в Java нетрудно. Это можно, в частности, сделать следующим образом:

```
double[][] balances;
```

Массивом нельзя пользоваться до тех пор, пока он не инициализирован с помощью операции `new`. В данном случае инициализация двумерного массива осуществляется следующим образом:

```
balances = new double[NYEARS][NRATES];
```

А в других случаях, когда элементы массива известны заранее, можно воспользоваться сокращенной записью для его инициализации, не прибегая к операции `new`. Ниже приведен соответствующий пример.

```
int[][] magicSquare =  
{  
    {16, 3, 2, 13},  
    {5, 10, 11, 8},  
    {9, 6, 7, 12},  
    {4, 15, 14, 1}  
};
```

После инициализации массива к его отдельным элементам можно обращаться, используя две пары квадратных скобок, например `balances[i][j]`.

В качестве примера рассмотрим программу, сохраняющую процентные ставки в одномерном массиве `interest`, а результаты подсчета остатка на счете ежегодно по каждой процентной ставке — в двумерном массиве `balances`. Сначала первая строка массива инициализируется исходной суммой следующим образом:

```
for (int j = 0; j < balances[0].length; j++)  
    balances[0][j] = 10000;
```

Затем подсчитывается содержимое остальных строк, как показано ниже.

```
for (int i = 1; i < balances.length; i++)  
{  
    for (int j = 0; j < balances[i].length; j++)  
    {  
        double oldBalance = balances[i - 1][j];  
        double interest = . . .;  
        balances[i][j] = oldBalance + interest;  
    }  
}
```

Исходный код данной программы полностью приведен в листинге 3.8.



НА ЗАМЕТКУ! Цикл в стиле `for each` не обеспечивает автоматического перебора элементов двумерного массива. Он лишь перебирает строки, которые, в свою очередь, являются одномерными массивами. Так, для обработки всех элементов двумерного массива `a` требуются два следующих цикла:

```
for (double[] row : a)  
    for (double value : row)  
        обработать значения value
```



СОВЕТ. Чтобы вывести на скорую руку список элементов двумерного массива, достаточно сделать вызов `System.out.println(Arrays.deepToString(a))`; . Вывод будет отформатирован следующим образом:

```
[[16, 3, 2, 13], [5, 10, 11, 8], [9, 6, 7, 12], [4, 15, 14, 1]]
```

Листинг 3.8. Исходный код из файла `CompoundInterest/CompoundInterest.java`

```
1  /**
2   * В этой программе демонстрируется сохранение табличных
3   * данных в двумерном массиве
4   * @version 1.40 2004-02-10
5   * @author Cay Horstmann
6   */
7  public class CompoundInterest
8  {
9      public static void main(String[] args)
10     {
11         final double STARTRATE = 10;
12         final int NRATES = 6;
13         final int NYEARS = 10;
14
15         // установить процентные ставки 10 . . . 15%
16         double[] interestRate = new double[NRATES];
17         for (int j = 0; j < interestRate.length; j++)
18             interestRate[j] = (STARTRATE + j) / 100.0;
19
20         double[][] balances = new double[NYEARS][NRATES];
21
22         // установить исходные остатки на счету равными 10000
23         for (int j = 0; j < balances[0].length; j++)
24             balances[0][j] = 10000;
25
26         // рассчитать проценты на следующие годы
27         for (int i = 1; i < balances.length; i++)
28         {
29             for (int j = 0; j < balances[i].length; j++)
30             {
31                 // получить остатки на счету за прошлый год
32                 double oldBalance = balances[i - 1][j];
33
34                 // рассчитать проценты
35                 double interest = oldBalance * interestRate[j];
36
37                 // рассчитать остатки на счету в текущем году
38                 balances[i][j] = oldBalance + interest;
39             }
40         }
41
42         // вывести один ряд процентных ставок
43         for (int j = 0; j < interestRate.length; j++)
44             System.out.printf("%9.0f%%",
45                               100 * interestRate[j]);
46
47         System.out.println();
48         // вывести таблицу остатков на счету
49         for (double[] row : balances)
50         {
51             // вывести строку таблицы
52             for (double b : row)
53                 System.out.printf("%10.2f", b);
54
55             System.out.println();
```



```
56    }  
57  }  
58 }
```

3.10.8. Неровные массивы

Все рассмотренные ранее языковые конструкции мало чем отличались от аналогичных конструкций в других языках программирования. Но механизм массивов в Java имеет особенность, предоставляющую совершенно новые возможности. В этом языке вообще нет многомерных массивов, а только одномерные. Многомерные массивы — это искусственно создаваемые “массивы массивов”.

Например, массив `balances` из предыдущего примера программы фактически представляет собой массив, состоящий из 10 элементов, каждый из которых является массивом из 6 элементов, содержащих числа с плавающей точкой (рис. 3.15).

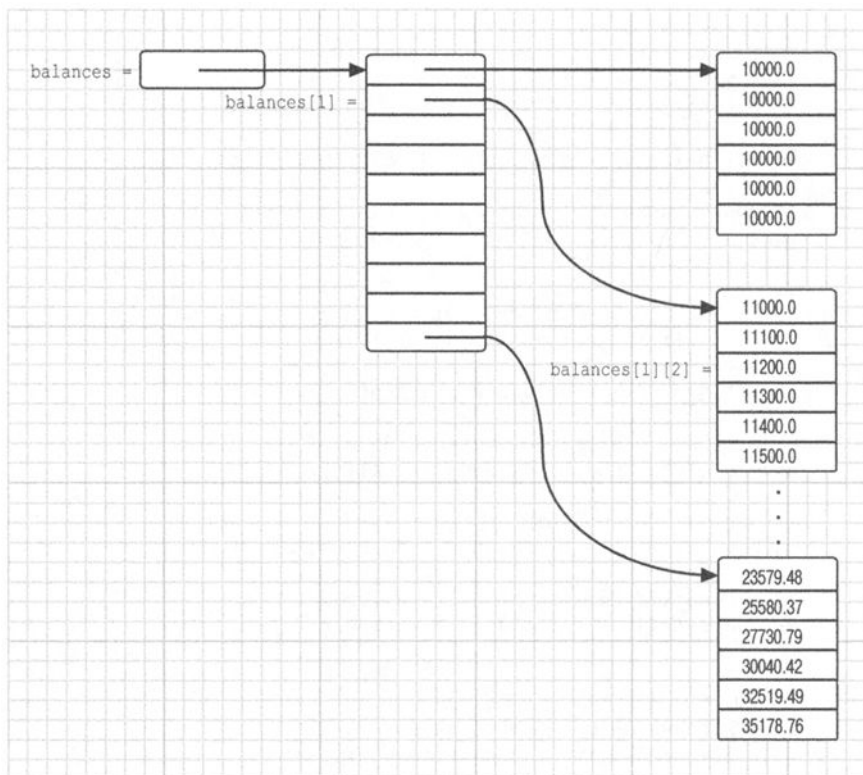


Рис. 3.15. Двумерный массив

Выражение `balance[i]` определяет i -й подмассив, т.е. i -ю строку таблицы. Эта строка сама представляет собой массив, а выражение `balance[i][j]` относится к его j -му элементу. Строки массива доступны из программы, поэтому их, как показано ниже, можно легко переставлять!

```
double[] temp = balances[i];  
balances[i] = balances[i + 1];  
balances[i + 1] = temp;
```

Кроме того, в Java легко формируются неровные, “рванные”, массивы, т.е. такие массивы, у которых разные строки имеют разную длину. Рассмотрим типичный пример, создав массив, в котором элемент, стоящий на пересечении i -й строки и j -го столбца, равен количеству вариантов выбора j чисел из i .

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
```

Число j не может превышать число i , из-за чего получается треугольная матрица. В i -й строке этой матрицы находится $i+1$ элемент. (Допускается выбрать и 0 элементов, но сделать это можно лишь одним способом.) Чтобы создать неровный массив, следует сначала разместить в памяти массив, хранящий его строки:

```
int[][] odds = new int[NMAX+1][];
```

Затем в памяти размещаются сами строки:

```
for (int n=0; n<=NMAX; n++)
    odds[n] = new int[n+1];
```

Теперь в памяти размещен весь массив, а следовательно, к его элементам можно обращаться, как обычно, но при условии, что индексы не выходят за допустимые пределы. В приведенном ниже фрагменте кода показано, как это делается.

```
for (int n=0; n<odds.length; n++)
    for (int k=0; k<odds[n].length; k++)
    {
        // рассчитать варианты выигрыша в лотерею
        ...
        odds[n][k] = lotteryOdds;
    }
```

Исходный код программы, реализующей рассматриваемый здесь неровный треугольный массив, приведен в листинге 3.9.

Листинг 3.9. Исходный код из файла `LotteryArray/LotteryArray.java`

```
1 /**
2  * В этой программе демонстрируется применение
3  * треугольного массива
4  * @version 1.20 2004-02-10
5  * @author Cay Horstmann
6  */
7 public class LotteryArray
8 {
9     public static void main(String[] args)
10     {
11         final int NMAX = 10;
12
13         // выделить память под треугольный массив
14
15         int[][] odds = new int[NMAX + 1][];
```

```

17     for (int n = 0; n <= NMAX; n++)
18         odds[n] = new int[n + 1];
19
20     // заполнить треугольный массив
21
22     for (int n = 0; n < odds.length; n++)
23         for (int k = 0; k < odds[n].length; k++)
24             {
25                 /*
26                  *   вычислить биномиальный коэффициент:
27                  *   n*(n-1)*(n-2)*...*(n-k+1)/(1*2*3*...*k)
28                  */
29                 int lotteryOdds = 1;
30                 for (int i = 1; i <= k; i++)
31                     lotteryOdds = lotteryOdds * (n - i + 1) / i;
32
33                 odds[n][k] = lotteryOdds;
34             }
35
36     // вывести треугольный массив
37     for (int[] row : odds)
38     {
39         for (int odd : row)
40             System.out.printf("%4d", odd);
41             System.out.println();
42     }
43 }
44 }

```



НА ЗАМЕТКУ C++! В языке Java следующее объявление:

```
double[][] balance = new double[10][6]; // Java
```

не равнозначно такому объявлению:

```
double balance[10][6]; // C++
```

и даже не соответствует объявлению

```
double (*balance)[6] = new double[10][6]; // C++
```

Вместо этого в памяти размещается массив, состоящий из десяти указателей. Средствами C++ это можно выразить следующим образом:

```
double** balance = new double*[10]; // C++
```

Затем каждый элемент в массиве указателей заполняется массивом, состоящим из 6 чисел, как показано ниже.

```
for (i = 0; i < 10; i++)
    balance[i] = new double[6];
```

Правда, этот цикл выполняется автоматически при объявлении массива с помощью операции **new double[10][6]**. Если же требуется неровный массив, то массивы его строк следует разместить в оперативной памяти по отдельности.

Итак, вы ознакомились со всеми основными языковыми конструкциями Java. А следующая глава посвящена особенностям объектно-ориентированного программирования на Java.

Объекты и классы

В этой главе...

- ▶ Введение в объектно-ориентированное программирование
- ▶ Применение predefined классов
- ▶ Определение собственных классов
- ▶ Статические поля и методы
- ▶ Параметры методов
- ▶ Конструирование объектов
- ▶ Пакеты
- ▶ Архивные JAR-файлы
- ▶ Путь к классам
- ▶ Комментарии и документирование
- ▶ Рекомендации по разработке классов

В этой главе рассматриваются следующие вопросы.

- Введение в объектно-ориентированное программирование.
- Создание объектов классов из стандартной библиотеки Java.
- Создание собственных классов.

Если вы недостаточно хорошо ориентируетесь в вопросах объектно-ориентированного программирования, внимательно прочитайте эту главу. Для объектно-ориентированного программирования требуется совершенно иной образ мышления по сравнению с подходом, типичным для процедурных языков. Освоить новые принципы создания программ не всегда просто, но сделать это необходимо. Для овладения языком Java нужно хорошо знать основные понятия объектно-ориентированного программирования.

Тем, у кого имеется достаточный опыт программирования на C++, материал этой и предыдущей глав покажется хорошо знакомым. Но у Java и C++ имеются

существенные отличия, поэтому последний раздел этой главы следует прочесть очень внимательно. Примечания к C++ помогут вам плавно перейти от C++ к Java.

4.1. Введение в ООП

Объектно-ориентированное программирование (ООП) в настоящее время стало доминирующей методикой программирования, вытеснив структурные или процедурные подходы, разработанные в 1970-х годах. Java — это полностью объектно-ориентированный язык, и для продуктивного программирования на нем необходимо знать основные принципы ООП.

Объектно-ориентированная программа состоит из объектов, каждый из которых обладает определенными функциональными возможностями, предоставляемыми в распоряжение пользователей, а также скрытой реализацией. Одни объекты для своих программ вы можете взять в готовом виде из библиотеки, другие вам придется спроектировать самостоятельно. Строить ли свои объекты или приобретать готовые — зависит от вашего бюджета или времени. Но, как правило, до тех пор, пока объекты удовлетворяют вашим требованиям, вам не нужно особенно беспокоиться о том, каким образом реализованы их функциональные возможности.

Традиционное структурное программирование заключается в разработке набора процедур (или алгоритмов) для решения поставленной задачи. Определив эти процедуры, программист должен найти подходящий способ хранения данных. Вот почему создатель языка Pascal Никлаус Вирт (Niklaus Wirth) назвал свою известную книгу по программированию *Algorithms + Data Structures = Programs* (Алгоритмы + Структуры данных = Программы; издательство Prentice Hall, 1975 г). Обратите внимание на то, что в названии этой книги алгоритмы стоят на первом месте, а структуры данных — на втором. Это отражает образ мышления программистов того времени. Сначала они решали, как манипулировать данными, а затем — какую структуру применить для организации этих данных, чтобы с ними было легче работать. Подход ООП в корне изменил ситуацию, поставив на первое место данные и лишь на второе — алгоритмы, предназначенные для их обработки.

Для решения небольших задач процедурный подход оказывается вполне пригодным. Но объекты более приспособлены для решения более крупных задач. Рассмотрим в качестве примера небольшой веб-браузер. Его реализация может потребовать 2000 процедур, каждая из которых манипулирует набором глобальных данных. В стиле ООП та же самая программа может быть составлена всего из 100 классов, в каждом из которых в среднем определено по 20 методов (рис. 4.1). Такая структура программы гораздо удобнее для программирования, и в ней легче находить ошибки. Допустим, данные некоторого объекта находятся в неверном состоянии. Очевидно, что намного легче найти причину неполадок среди 20 методов, имеющих доступ к данным, чем среди 2000 процедур.

4.1.1. Классы

Для дальнейшей работы вам нужно усвоить основные понятия и терминологию ООП. Наиболее важным понятием является класс, применение которого уже демонстрировалось в примерах программ из предыдущих глав. Класс — это шаблон или образец, по которому будет создан объект. Обычно класс сравнивают с формой для выпечки печенья, а объект — это само печенье. Конструирование объекта на основе некоторого класса называется получением экземпляра этого класса.

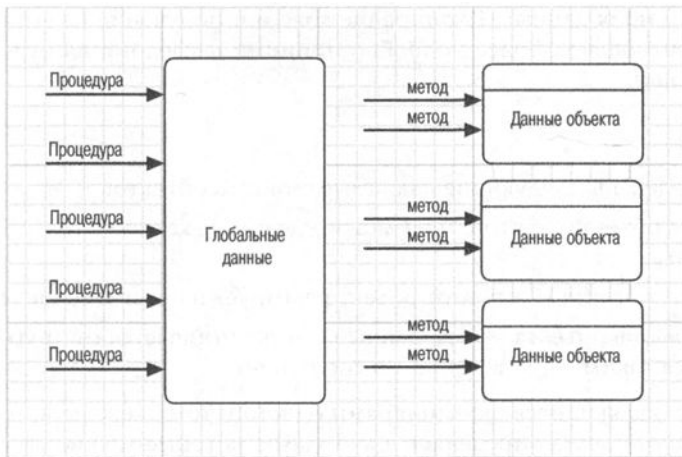


Рис. 4.1. Сравнение процедурного и объектно-ориентированного программирования

Как следует из примеров программ в предыдущих главах, весь код, написанный на Java, находится в классах. Стандартная библиотека Java содержит несколько тысяч классов, предназначенных для решения самых разных задач, например, для построения пользовательского интерфейса, календарей, установления сетевых соединений и т.д. Несмотря на это, программирующие на Java продолжают создавать свои собственные классы, чтобы формировать объекты, характерные для разрабатываемого приложения, а также приспособлять классы из стандартной библиотеки под свои нужды.

Инкапсуляция (иногда называемая *сокрытием информации*) — это ключевое понятие для работы с объектами. Формально инкапсуляцией считается обычное объединение данных и операций над ними в одном пакете и сокрытие данных от других объектов. Данные в объекте называются *полями экземпляра*, а функции и процедуры, выполняющие операции над данными, — его *методами*. В конкретном объекте, т.е. в экземпляре класса, поля экземпляра имеют определенные значения. Множество этих значений называется текущим *состоянием* объекта. Вызов любого метода для объекта может изменить его состояние.

Следует еще раз подчеркнуть, что основной принцип инкапсуляции заключается в запрещении прямого доступа к полям экземпляра данного класса из других классов. Программы должны взаимодействовать с данными объекта только через методы этого объекта. Инкапсуляция обеспечивает внутреннее поведение объектов, что имеет решающее значение для повторного их использования и надежности работы программ. Это означает, что в классе можно полностью изменить способ хранения данных. Но поскольку для манипулирования данными используются одни и те же методы, то об этом ничего не известно, да и не особенно важно другим объектам.

Еще один принцип ООП облегчает разработку собственных классов в Java: один класс можно построить на основе других классов. В этом случае говорят, что новый класс *расширяет* тот класс, на основе которого он создан. Язык Java, по существу, создан на основе “глобального суперкласса”, называемого *Object*. Все остальные объекты расширяют его. В следующей главе мы рассмотрим этот вопрос более подробно.

Если класс разрабатывается на основе уже существующего, то новый класс содержит все свойства и методы расширяемого класса. Кроме того, в него добавляются

новые методы и поля данных. Расширение класса и получение на его основе нового называется *наследованием*. Более подробно принцип наследования будет рассмотрен в следующей главе.

4.1.2. Объекты

В ООП определены следующие ключевые свойства объектов.

- *Поведение* объекта — что с ним можно делать и какие методы к нему можно применять.
- *Состояние* объекта — как этот объект реагирует на применение методов.
- *Идентичность* объекта — чем данный объект отличается от других, характеризующихся таким же поведением и состоянием.

Все объекты, являющиеся экземплярами одного и того же класса, ведут себя одинаково. *Поведение* объекта определяется методами, которые можно вызвать. Каждый объект сохраняет информацию о своем *состоянии*. Со временем состояние объекта может измениться, но спонтанно это произойти не может. Состояние объекта может изменяться только в результате вызовов методов. (Если состояние объекта изменилось вследствие иных причин, значит, принцип инкапсуляции не соблюден.)

Состояние объекта не полностью описывает его, поскольку каждый объект имеет свою собственную *идентичность*. Например, в системе обработки заказов два заказа могут отличаться друг от друга, даже если они относятся к одним и тем же товарам. Заметим, что индивидуальные объекты, представляющие собой экземпляры класса, *всегда* отличаются своей идентичностью и, как правило, — своим состоянием.

Эти основные свойства объектов могут оказывать взаимное влияние. Например, состояние объекта может оказывать влияние на его поведение. (Если заказ выполнен или оплачен, объект может отказаться вызвать метод, требующий добавить или удалить товар. И наоборот, если заказ пуст, т.е. ни одна единица товара не была заказана, он не может быть выполнен.)

4.1.3. Идентификация классов

В традиционной процедурной программе выполнение начинается сверху, т.е. с функции `main()`. При проектировании объектно-ориентированной системы понятия “верха” как такового не существует, поэтому начинающие осваивать ООП часто интересуются, с чего же следует начинать. Сначала нужно идентифицировать классы, а затем добавить методы в каждый класс.

Простое эмпирическое правило для идентификации классов состоит в том, чтобы выделить для них имена существительные при анализе проблемной области. С другой стороны, методы соответствуют глаголам, обозначающим действие. Например, при описании системы обработки заказов используются следующие имена существительные.

- Товар
- Заказ
- Адрес доставки
- Оплата
- Счет

Этим именам соответствуют классы *Item*, *Order* и т.д.

Далее выбираются глаголы. Изделия *вводятся* в заказы. Заказы *выполняются* или *отменяются*. Оплата заказа *осуществляется*. Используя эти глаголы, можно определить объект, выполняющий такие действия. Так, если поступил новый заказ, ответственность за его обработку должен нести объект *Order* (Заказ), поскольку именно в нем содержится информация о способе хранения и сортировке заказываемых товаров. Следовательно, в классе *Order* должен существовать метод *add()* — добавить получающий объект *Item* (Товар) в качестве параметра.

Разумеется, упомянутое выше правило выбора имен существительных и глаголов является не более чем рекомендацией. И только опыт может помочь программисту решить, какие именно существительные и глаголы следует выбрать при создании класса и его методов.

4.1.4. Отношения между классами

Между классами существуют три общих вида отношений.

- Зависимость (“использует — что-то”)
- Агрегирование (“содержит — что-то”)
- Наследование (“является — чем-то”)

Отношение *зависимости* наиболее очевидное и распространенное. Например, в классе *Order* используется класс *Account*, поскольку объекты типа *Order* должны иметь доступ к объектам типа *Account*, чтобы проверить кредитоспособность заказчика. Но класс *Item* не зависит от класса *Account*, потому что объекты типа *Item* вообще не интересуют состояние счета заказчика. Следовательно, один класс зависит от другого класса, если его методы выполняют какие-нибудь действия над экземплярами этого класса.

Старайтесь свести к минимуму количество взаимозависимых классов. Если класс *A* не знает о существовании класса *B*, то он тем более ничего не знает о любых изменениях в нем! (Это означает, что любые изменения в классе *B* не повлияют на поведение объектов класса *A*.)

Отношение *агрегирования* понять нетрудно, потому что оно конкретно. Например, объект типа *Order* может содержать объекты типа *Item*. Агрегирование означает, что объект класса *A* содержит объекты класса *B*.



НА ЗАМЕТКУ! Некоторые специалисты не признают понятие агрегирования и предпочитают использовать более общее отношение ассоциации или связи между классами. С точки зрения моделирования это разумно. Но для программистов гораздо удобнее отношение, при котором один объект содержит другой. Пользоваться понятием агрегирования удобнее по еще одной причине: его обозначение проще для восприятия, чем обозначение отношения ассоциации (табл. 4.1).

Наследование выражает отношение между конкретным и более общим классом. Например, класс *RushOrder* наследует от класса *Order*. Специализированный класс *RushOrder* содержит особые методы для обработки приоритетов и разные методы для вычисления стоимости доставки товаров, в то время как другие его методы, например, для заказа товаров и выписывания счетов, унаследованы от класса *Order*. Вообще говоря, если класс *A* расширяет класс *B*, то класс *A* наследует методы класса *B* и, кроме них, имеет дополнительные возможности. (Более подробно наследование рассматривается в следующей главе.)

Многие программисты пользуются средствами UML (Unified Modeling Language — унифицированный язык моделирования) для составления диаграмм классов, описывающих отношения между классами. Пример такой диаграммы приведен на рис. 4.2, где классы обозначены прямоугольниками, а отношения между ними — различными стрелками. В табл. 4.1 приведены основные обозначения, принятые в языке UML.

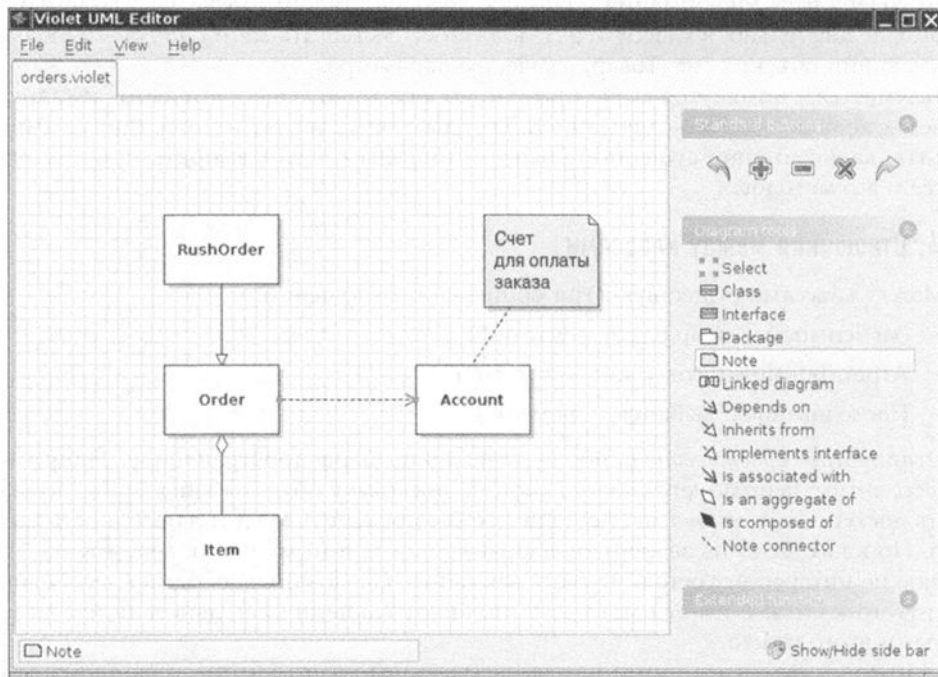


Рис. 4.2. Диаграмма классов

Таблица 4.1. Обозначение отношений между классами в UML

Отношение	Обозначение в UML
Наследование	—————▷
Реализация интерфейса	- - - - -▷
Зависимость	- - - - -➔
Агрегирование	◊—————
Связь	—————
Направленная связь	—————➔

4.2. Применение предопределенных классов

В языке Java ничего нельзя сделать без классов, поэтому мы вкратце обсудили в предыдущих разделах, как действуют некоторые из них. Но в Java имеются также классы, к которым не совсем подходят приведенные выше рассуждения. Характерным тому примером служит класс `Math`. Как упоминалось в предыдущей главе, методы из класса `Math`, например метод `Math.random()`, можно вызывать, вообще ничего

не зная об их реализации. Для обращения к методу достаточно знать его имя и параметры (если они предусмотрены). Это признак инкапсуляции, который, безусловно, справедлив для всех классов. Но в классе `Math` отсутствуют данные; он инкапсулирует только функциональные возможности, не требуя ни данных, ни их сокрытия. Это означает, что можно и не заботиться о создании объектов и инициализации их полей, поскольку в классе `Math` ничего подобного нет!

В следующем разделе будет рассмотрен класс `Date`, на примере которого будет показано, как создаются экземпляры и вызываются методы из класса.

4.2.1. Объекты и объектные переменные

Чтобы работать с объектами, их нужно сначала создать и задать их исходное состояние. Затем к этим объектам можно применять методы. Для создания новых экземпляров в Java служат *конструкторы* — специальные методы, предназначенные для создания и инициализации экземпляра класса. В качестве примера можно привести класс `Date`, входящий в состав стандартной библиотеки Java. С помощью объектов этого класса можно описать текущий или любой другой момент времени, например "December 31, 1999, 23:59:59 GMT".



НА ЗАМЕТКУ! У вас может возникнуть вопрос: почему для представления даты и времени применяются классы, а не встроенные типы данных, как в ряде других языков программирования? Такой подход применяется, например, в Visual Basic, где дата задается в формате `#6/1/1995#`. На первый взгляд это удобно — программист может использовать встроенный тип и не заботиться о классах. Но не кажущееся ли такое удобство? В одних странах даты записываются в формате месяц/день/год, а в других — год/месяц/день. Могут ли создатели языка предусмотреть все возможные варианты? Даже если это и удастся сделать, соответствующие средства будут слишком сложны, причем программисты будут вынуждены применять их. Использование классов позволяет переложить ответственность за решение этих проблем с создателей языка на разработчиков библиотек. Если системный класс не годится, то разработчики всегда могут написать свой собственный класс. В качестве аргумента в пользу такого подхода отметим, что библиотека Java для дат довольно запутана и уже переделывалась дважды.

Имя конструктора всегда совпадает с именем класса. Следовательно, конструктор класса `Date` называется `Date()`. Чтобы создать объект типа `Date`, конструктор следует объединить с операцией `new`, как показано ниже, где создается новый объект, который инициализируется текущими датой и временем.

```
new Date()
```

При желании объект можно передать методу, как показано ниже.

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

И наоборот, можно вызвать метод для вновь созданного объекта. Среди методов класса `Date` имеется метод `toString()`, позволяющий представить дату в виде символьной строки. Он вызывается следующим образом:

```
String = new Date().toString();
```

В этих двух примерах созданный объект использовался только один раз. Но, как правило, объектом приходится пользоваться неоднократно. Чтобы это стало возможным, необходимо связать объект с некоторым идентификатором, иными словами, присвоить объект переменной, как показано ниже.

```
Date birthday = new Date();
```

На рис. 4.3 наглядно демонстрируется, каким образом переменная `birthday` ссылается на вновь созданный объект.

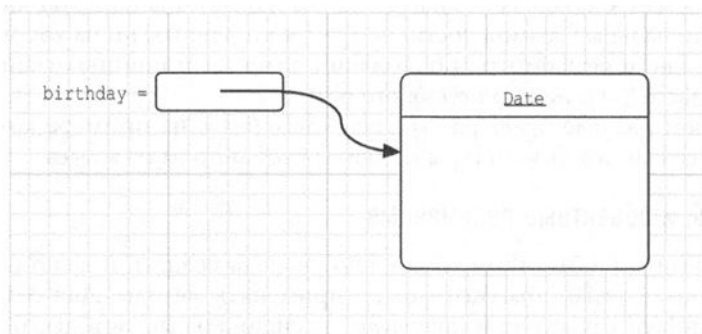


Рис. 4.3. Создание нового объекта

У объектов и объектных переменных имеется существенное отличие. Например, в приведенной ниже строке кода определяется объектная переменная `deadline`, которая может ссылаться на объекты типа `Date`.

```
Date deadline; // переменная deadline не ссылается ни на один из объектов
```

Важно понимать, что на данном этапе сама переменная `deadline` объектом не является и даже не ссылается ни на один из объектов. Поэтому ни один из методов класса `Date` пока еще нельзя вызывать по этой переменной. Попытка сделать это приведет к появлению сообщения об ошибке:

```
s = deadline.toString(); // вызывать метод еще рано!
```

Сначала нужно инициализировать переменную `deadline`. Для этого имеются две возможности. Прежде всего, переменную можно инициализировать вновь созданным объектом:

```
deadline = new Date();
```

Кроме того, переменной можно присвоить ссылку на существующий объект:

```
deadline = birthday;
```

Теперь переменные `deadline` и `birthday` ссылаются на *один и тот же* объект (рис. 4.4).

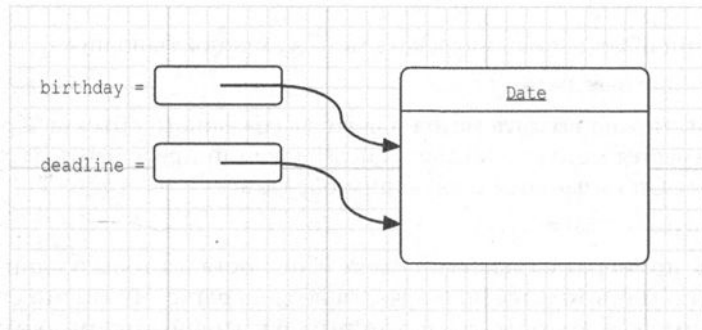


Рис. 4.4. Объектные переменные, ссылающиеся на один и тот же объект

Важно понять, что объектная переменная фактически не содержит никакого объекта. Она лишь *ссылается* на него. Значение любой объектной переменной в Java представляет собой ссылку на объект, размещенный в другом месте. Операция `new` также возвращает ссылку. Например, приведенная ниже строка кода состоит из двух частей: в операции `new Date()` создается объект типа `Date`, а переменной `deadline` присваивается ссылка на вновь созданный объект.

```
Date deadline = new Date();
```

Объектной переменной можно явно присвоить пустое значение `null`, чтобы указать на то, что она пока не ссылается ни на один из объектов:

```
deadline = null;  
...  
if (deadline != null)  
    System.out.println(deadline);
```

Если попытаться вызвать метод по ссылке на переменную с пустым значением `null`, то при выполнении программы возникнет ошибка, как показано ниже.

```
birthday = null;  
String s = birthday.toString(); // Ошибка при выполнении!
```

Более подробно обработка пустых значений `null` рассматривается далее, в разделе 4.3.6.



НА ЗАМЕТКУ C++! Многие ошибочно полагают, что объектные переменные в Java похожи на ссылки в C++. Но в C++ пустые ссылки не допускаются и не присваиваются. Объектные переменные в Java следует считать аналогами указателей на объекты. Например, следующая строка кода Java:

```
Date birthday; // Java  
почти эквивалентна такой строке кода C++:  
Date* birthday; // C++
```

Такая аналогия все расставляет на свои места. Разумеется, указатель `Date*` не инициализируется до тех пор, пока не будет выполнена операция `new`. Синтаксис подобных выражений в C++ и Java почти совпадает.

```
Date* birthday = new Date(); // C++
```

При копировании одной переменной в другую в обеих переменных оказывается ссылка на один и тот же объект. Указатель `NULL` в C++ служит эквивалентом пустой ссылки `null` в Java.

Все объекты в Java располагаются в динамической области памяти, иначе называемой "кучей". Если объект содержит другую объектную переменную, она представляет собой всего лишь указатель на другой объект, расположенный в этой области памяти.

В языке C++ указатели доставляют немало хлопот, поскольку часто приводят к ошибкам. Очень легко создать неверный указатель или потерять управление памятью. А в Java подобные сложности вообще не возникают. Если вы пользуетесь неинициализированным указателем, то исполняющая система обязательно сообщит об ошибке, а не продолжит выполнение некорректной программы, выдавая случайные результаты. Вам не нужно беспокоиться об управлении памятью, поскольку механизм сборки "мусора" выполняет все необходимые операции с памятью автоматически.

В языке C++ большое внимание уделяется автоматическому копированию объектов с помощью копирующих конструкторов и операций присваивания. Например, копией связанного списка является новый связанный список, который, имея старое содержимое, содержит совершенно другие связи. Иными словами, копирование объектов осуществляется так же, как и копирование встроенных типов. А в Java для получения полной копии объекта служит метод `clone()`.

4.2.2. Класс `LocalDate` из библиотеки Java

В предыдущих примерах использовался класс `Date`, входящий в состав стандартной библиотеки Java. Экземпляр класса `Date` находится в состоянии, которое отражает конкретный момент времени.

Пользуясь классом `Date`, совсем не обязательно знать формат даты. Тем не менее время в этом классе представлено количеством миллисекунд (положительным или отрицательным), отсчитанным от фиксированного момента времени, так называемого начала эпохи, т.е. от момента времени 00:00:00 UTC, 1 января 1970 г. Сокращение UTC означает Universal Coordinated Time (Универсальное скоординированное время) — научный стандарт времени. Стандарт UTC применяется наряду с более известным стандартом GMT (Greenwich Mean Time — среднее время по Гринвичу).

Но класс `Date` не очень удобен для манипулирования датами. Разработчики библиотеки Java посчитали, что представление даты, например "December 31, 1999, 23:59:59", является совершенно произвольным и должно зависеть от календаря. Данное конкретное представление подчиняется григорианскому календарю — самому распространенному в мире. Но тот же самый момент времени совершенно иначе представляется в китайском или еврейском лунном календаре, не говоря уже о календаре, которым будут пользоваться потенциальные потребители с Марса.



НА ЗАМЕТКУ! Вся история человечества сопровождалась созданием календарей — систем именования различных моментов времени. Как правило, основой для календарей служил солнечный или лунный цикл. Если вас интересуют подобные вопросы, обратитесь за справкой, например, к книге Наума Дершовица (Nachum Dershowitz) и Эдварда М. Рейнгольда (Edward M. Reingold) *Calendrical Calculations* (издательство Cambridge University Press, 3rd ed., 2007 г.). Там вы найдете исчерпывающие сведения о календаре французской революции, календаре Майя и других экзотических системах отсчета времени.

Разработчики библиотеки Java решили отделить вопросы, связанные с отслеживанием моментов времени, от вопросов их представления. Таким образом, стандартная библиотека Java содержит два отдельных класса: класс `Date`, представляющий момент времени, и класс `LocalDate`, выражающий даты в привычном календарном представлении. В версии Java 8 внедрено немало других классов для манипулирования различными характеристиками даты и времени, как поясняется в главе 6 второго тома настоящего издания.

Отделение измерения времени от календарей является грамотным решением, вполне отвечающим принципам ООП. Для построения объектов класса `LocalDate` нужно пользоваться не его конструктором, а статическими *фабричными методами*, автоматически вызывающими соответствующие конструкторы. Так, в выражении

```
LocalDate.now()
```

создается новый объект, представляющий дату построения этого объекта.

Безусловно, построенный объект желательно сохранить в объектной переменной, как показано ниже.

```
LocalDate newYearsEve = LocalDate.of(1999, 12, 31);
```

Имея в своем распоряжении объект типа `LocalDate`, можно определить год, месяц и день с помощью методов `getYear()`, `getMonthValue()` и `getDayOfMonth()`, как демонстрируется в следующем примере кода:

```
int year = newYearsEve.getYear(); // 1999 г.  
int month = newYearsEve.getMonthValue(); // 12-й месяц  
int day = newYearsEve.getDayOfMonth(); // 31-е число
```

На первый взгляд, в этом нет никакого смысла, поскольку те же самые значения даты использовались для построения объекта. Но иногда можно воспользоваться датой, вычисленной иным способом, чтобы вызвать упомянутые выше методы и получить дополнительные сведения об этой дате. Например, с помощью метода `plusDays()` можно получить новый объект типа `LocalDate`, отстоящий во времени на заданное количество дней от объекта, к которому этот метод применяется:

```
LocalDate aThousandDaysLater = newYearsEve.plusDays(1000);  
year = aThousandDaysLater.getYear(); // 2002 г.  
month = aThousandDaysLater.getMonthValue(); // 9-й месяц  
day = aThousandDaysLater.getDayOfMonth(); // 26-е число
```

В классе `LocalDate` инкапсулированы поля экземпляра для хранения заданной даты. Не заглянув в исходный код этого класса, нельзя узнать, каким образом в нем представлена дата. Но ведь назначение такой инкапсуляции в том и состоит, что пользователю класса `LocalDate` вообще не нужно об этом знать. Ему важнее знать о тех методах, которые доступны в этом классе.



НА ЗАМЕТКУ! На самом деле в классе `Date` имеются такие методы, как `getDay()`, `getMonth()` и `getYear()`, но пользоваться ими без крайней необходимости не рекомендуется. Метод объявляется как не рекомендованный к применению, когда разработчики библиотеки решают, что его не стоит больше применять в новых программах.

Эти методы были частью класса `Date` еще до того, как разработчики библиотеки Java поняли, что классы, реализующие разные календари, разумнее было бы отделить друг от друга. Внедрив такие классы еще в версии Java 1.1, они пометили методы из класса `Date` как не рекомендованные к применению. Вы вольны и дальше пользоваться ими в своих программах, получая при этом предупреждения от компилятора, но лучше вообще отказаться от их применения, поскольку они могут быть удалены из последующих версий библиотеки.



СОВЕТ! В комплекте JDK предоставляется утилита `jdeprscan`, позволяющая проверить, употребляются ли в прикладном коде не рекомендованные к применению функциональные средства из прикладного интерфейса Java API. Инструкции по применению этой утилиты см. по адресу <https://docs.oracle.com/javase/9/tools/jdeprscan.htm#BEGIN>.

4.2.3. Модифицирующие методы и методы доступа

Рассмотрим еще раз следующий вызов метода `plusDays()`, упоминавшегося в предыдущем разделе:

```
LocalDate aThousandDaysLater = newYearsEve.plusDays(1000);
```

Что произойдет с объектом `newYearsEve` после этого вызова? Будет ли он отодвинут во времени на 1000 дней назад? Оказывается, нет. В результате вызова метода `plusDays()` получается новый объект типа `LocalDate`, который затем присваивается переменной `aThousandDaysLater`, а исходный объект остается без изменения. В таком случае говорят, что метод `plusDays()` не модифицирует объект, для которого он вызывается. (Аналогичным образом действует метод `toUpperCase()` из класса `String`, упоминавшийся в главе 3. Когда этот метод вызывается для символьной строки, которая остается без изменения, в конечном счете возвращается новая строка символов в верхнем регистре.)

В более ранней версии стандартной библиотеки Java имелся другой класс `GregorianCalendar`, предназначенный для обращения с календарями. Ниже показано, как добавить тысячу дней к дате, представленной этим классом.

```
GregorianCalendar someDay =
    new GregorianCalendar(1999, 11, 31);
    // В этом классе месяцы нумеруются от 0 до 11
someDay.add(Calendar.DAY_OF_MONTH, 1000);
```

В отличие от метода `LocalDate.plusDays()`, метод `GregorianCalendar.add()` является *модифицирующим*. После его вызова состояние объекта `someDay` изменяется. Ниже показано, как определить это новое состояние. Переменная была названа `someDay`, а не `newYearsEve` потому, что она больше не содержит канун нового года после вызова модифицирующего метода.

```
year = someDay.get(Calendar.YEAR); // 2002 г.
month = someDay.get(Calendar.MONTH) + 1; // 9-й месяц
day = someDay.get(Calendar.DAY_OF_MONTH); // 26-е число
```

А методы, получающие только доступ к объектам, не модифицируя их, называют *методами доступа*. К их числу относятся, например, методы `LocalDate.getYear()` и `GregorianCalendar.get()`.



НА ЗАМЕТКУ C++! Для обозначения метода доступа в C++ служит суффикс `const`. Метод, не объявленный с помощью ключевого слова `const`, считается модифицирующим. Но в Java отсутствуют специальные синтаксические конструкции, позволяющие отличать модифицирующие методы от методов доступа.

И в завершение этого раздела рассмотрим пример программы, в которой демонстрируется применение класса `LocalDate`. Эта программа выводит на экран календарь текущего месяца в следующем формате:

```
Sun Mon Tue Wed Thu Fri Sat
                1
 2   3   4   5   6   7   8
 9  10  11  12  13  14  15
16  17  18  19  20  21  22
23  24  25  26* 27  28  29
30
```

Текущий день помечен в календаре звездочкой. Как видите, в данной программе должно быть известно, как вычисляется длина месяца и текущий день недели. Рассмотрим основные стадии выполнения данной программы. Сначала в ней создается объект типа `LocalDate`, инициализированный текущей датой:

```
LocalDate date = LocalDate.now();
```

Затем определяется текущий день и месяц:

```
int month = date.getMonthValue();
int today = date.getDayOfMonth();
```

После этого переменной `date` присваивается первый день месяца и из этой даты получается день недели.

```
date = date.minusDays(today - 1); // задать 1-й день месяца
DayOfWeek weekday = date.getDayOfWeek();
int value = weekday.getValue();
    // 1 = понедельник, ... 7 = воскресенье
```

Сначала переменной `weekday` присваивается объект типа `DayOfWeek`, а затем для этого объекта вызывается метод `getValue()`, чтобы получить числовое значение дня недели. Это числовое значение соответствует международному соглашению о том, что воскресенье приходится на конец недели. Следовательно, понедельнику соответствует возвращаемое данным методом числовое значение 1, вторнику — 2 и так далее до воскресенья, которому соответствует числовое значение 7.

Обратите внимание на то, что первая строка после заголовка календаря выводится с отступом, чтобы первый день месяца выпадал на соответствующий день недели. Ниже приведен фрагмент кода для вывода заголовка и отступа в первой строке календаря.

```
System.out.println("Mon Tue Wed Thu Fri Sat Sun");
for (int i = 1; i < value; i++)
    System.out.print(" ");
```

Теперь все готово для вывода самого календаря. С этой целью организуется цикл, где дата в переменной `date` перебирается по всем дням месяца. На каждом шаге цикла выводится числовое значение даты. Если дата в переменной `date` приходится на текущий день месяца, этот день помечается знаком *. Затем дата в переменной `date` устанавливается на следующий день, как показано ниже. Когда в цикле достигается начало новой недели, выводится знак перевода строки.

```
while (date.getMonthValue() == month)
{
    System.out.printf("%3d", date.getDayOfMonth());
    if (date.getDayOfMonth() == today)
        System.out.print("*");
    else
        System.out.print(" ");
    date = date.plusDays(1);
    if (date.getDayOfWeek().getValue() == 1)
        System.out.println();
}
```

Когда же следует остановиться? Заранее неизвестно, сколько в месяце дней: 31, 30, 29 или 28. Поэтому цикл продолжается до тех пор, пока дата в переменной `date` остается в пределах текущего месяца. Исходный код данной программы полностью приведен в листинге 4.1.

Как видите, класс `LocalDate` позволяет легко создавать программы для работы с календарем, выполняя такие сложные действия, как отслеживание дней недели и учет продолжительности месяцев. Программисту не нужно ничего знать, каким образом в классе `LocalDate` вычисляются месяцы и дни недели. Ему достаточно пользоваться *интерфейсом* данного класса, включая методы `plusDays()` и `getDayOfWeek()`. Основное назначение рассмотренной здесь программы — показать, как пользоваться *интерфейсом* класса для решения сложных задач, не вдаваясь в подробности реализации.

Листинг 4.1. Исходный код из файла `CalendarTest/CalendarTest.java`

```
1 import java.time.*;
2
3 /**
4  * @version 1.5 2015-05-08
5  * @author Cay Horstmann
6  */
```



```
7
8 public class CalendarTest
9 {
10     public static void main(String[] args)
11     {
12         LocalDate date = LocalDate.now();
13         int month = date.getMonthValue();
14         int today = date.getDayOfMonth();
15
16         date = date.minusDays(today - 1);
17         // задать 1-й день месяца
18         DayOfWeek weekday = date.getDayOfWeek();
19         int value = weekday.getValue();
20         // 1 = понедельник, ... 7 = воскресенье
21         System.out.println("Mon Tue Wed Thu Fri Sat Sun");
22         for (int i = 1; i < value; i++)
23             System.out.print(" ");
24         while (date.getMonthValue() == month)
25         {
26             System.out.printf("%3d", date.getDayOfMonth());
27             if (date.getDayOfMonth() == today)
28                 System.out.print("*");
29             else
30                 System.out.print(" ");
31             date = date.plusDays(1);
32             if (date.getDayOfWeek().getValue() == 1)
33                 System.out.println();
34         }
35         if (date.getDayOfWeek().getValue() != 1)
36             System.out.println();
37     }
38 }
```

java.util.LocalDate 8

- **static `LocalTime now()`**
Строит объект, представляющий текущую дату.
- **static `LocalTime of(int year, int month, int day)`**
Строит объект, представляющий заданную дату.
- **`int getYear()`**
- **`int getMonthValue()`**
- **`int getDayOfMonth()`**
Получают год, месяц и день из текущей даты.
- **`DayOfWeek getDayOfWeek()`**
Получает день недели из текущей даты в виде экземпляра класса **DayOfWeek**. Для получения дня недели в пределах от 1 (понедельник) до 7 (воскресенье) следует вызвать метод **getValue()**.
- **`LocalDate plusDays(int n)`**
- **`LocalDate minusDays(int n)`**
Выдают дату на *n* дней после или до текущей даты соответственно.

4.3. Определение собственных классов

В примерах кода из главы 3 уже предпринималась попытка создавать простые классы. Но все они состояли из единственного метода `main()`. Теперь настало время показать, как создаются “рабочие” классы для более сложных приложений. Как правило, в этих классах метод `main()` отсутствует. Вместо этого они содержат другие методы и поля. Чтобы написать полностью завершенную программу, необходимо объединить несколько классов, один из которых содержит метод `main()`.

4.3.1. Класс `Employee`

Простейшая форма определения класса в Java выглядит следующим образом:

```
class ИмяКласса
{
    поле_1
    поле_2

    конструктор_1
    конструктор_2
    ...
    метод_1
    метод_2
    ...
}
```

Рассмотрим следующую, весьма упрощенную версию класса `Employee`, который можно использовать для составления платежной ведомости:

```
class Employee
{
    // поля экземпляра
    private String name;
    private double salary;
    private Date hireDay;

    // конструктор
    public Employee(String n, double s, int year,
                    int month, int day)
    {
        name = n;
        salary = s;
        hireDay = LocalDate.of(year, month, day);
    }

    // метод
    public String getName()
    {
        return name;
    }

    // другие методы
    . . .
}
```

Более подробно реализация этого класса будет проанализирована в последующих разделах, а сейчас рассмотрим код, приведенный в листинге 4.2 и демонстрирующий практическое применение класса `Employee`.

Листинг 4.2. Исходный код из файла `EmployeeTest/EmployeeTest.java`

```
1  import java.time.*;
2
3  /**
4   * В этой программе проверяется класс Employee
5   * @version 1.13 2018-04-10
6   * @author Cay Horstmann
7   */
8  public class EmployeeTest
9  {
10     public static void main(String[] args)
11     {
12         // заполнить массив staff тремя
13         // объектами типа Employee
14         Employee[] staff = new Employee[3];
15
16         staff[0] = new Employee("Carl Cracker", 75000,
17                                1987, 12, 15);
18         staff[1] = new Employee("Harry Hacker", 50000,
19                                1989, 10, 1);
20         staff[2] = new Employee("Tony Tester", 40000,
21                                1990, 3, 15);
22
23         // поднять всем работникам зарплату на 5%
24         for (Employee e : staff)
25             e.raiseSalary(5);
26
27         // вывести данные обо всех объектах типа Employee
28         for (Employee e : staff)
29             System.out.println("name=" + e.getName() + ",
30                                salary=" + e.getSalary() + ",
31                                hireDay=" + e.getHireDay());
32     }
33 }
34
35 class Employee
36 {
37     private String name;
38     private double salary;
39     private LocalDate hireDay;
40
41     public Employee(String n, double s, int year,
42                     int month, int day)
43     {
44         name = n;
45         salary = s;
46         hireDay = LocalDate.of(year, month, day);
47     }
48
49     public String getName()
```

```
50 {
51     return name;
52 }
53
54 public double getSalary()
55 {
56     return salary;
57 }
58
59 public LocalDate getHireDay()
60 {
61     return hireDay;
62 }
63
64 public void raiseSalary(double byPercent)
65 {
66     double raise = salary * byPercent / 100;
67     salary += raise;
68 }
69 }
```

В данной программе сначала создается массив типа `Employee`, в который заносятся три объекта работников:

```
Employee[] staff = new Employee[3];
staff[0] = new Employee("Carl Cracker", . . .);
staff[1] = new Employee("Harry Hacker", . . .);
staff[2] = new Employee("Tony Tester", . . .);
```

Затем вызывается метод `raiseSalary()` из класса `Employee`, чтобы поднять зарплату каждого работника на 5%:

```
for (Employee e : staff)
    e.raiseSalary(5);
```

И, наконец, с помощью методов `getName()`, `getSalary()` и `getHireDay()` выводятся данные о каждом работнике:

```
for (Employee e : staff)
    System.out.println("name=" + e.getName()
        + ",salary=" + e.getSalary()
        + ",hireDay=" + e.getHireDay());
```

Следует заметить, что данный пример программы состоит из двух классов, `Employee` и `EmployeeTest`, причем последний объявлен открытым с модификатором доступа `public`. Метод `main()` с описанными выше операторами содержится в классе `EmployeeTest`. Исходный код данной программы содержится в файле `EmployeeTest.java`, поскольку его имя должно совпадать с именем открытого класса. В исходном файле может быть только один класс, объявленный как `public`, а также любое количество классов, в объявлении которых данное ключевое слово отсутствует.

При компиляции исходного кода данной программы создаются два файла классов: `EmployeeTest.class` и `Employee.class`. Затем начинается выполнение программы, для чего интерпретатору байт-кода указывается имя класса, содержащего основной метод `main()` данной программы:

```
java EmployeeTest
```

Интерпретатор начинает обрабатывать метод `main()` из класса `EmployeeTest`. В результате выполнения кода создаются три новых объекта типа `Employee` и отображается их состояние.

4.3.2. Использование нескольких исходных файлов

Рассмотренная выше программа из листинга 4.2 состоит из двух классов в одном исходном файле. Многие программирующие на Java предпочитают размещать каждый класс в отдельном файле. Например, класс `Employee` можно разместить в файле `Employee.java`, а класс `EmployeeTest` — в файле `EmployeeTest.java`.

Имеются разные способы компиляции программы, код которой содержится в двух исходных файлах. Например, при вызове компилятора можно использовать шаблонный символ подстановки следующим образом:

```
javac Employee*.java
```

В результате все исходные файлы, имена которых совпадают с указанным шаблоном, будут скомпилированы в файлы классов. С другой стороны, можно также ограничиться командой

```
javac EmployeeTest.java
```

Как ни странно, файл `Employee.java` также будет скомпилирован. Обнаружив, что в файле `EmployeeTest.java` используется класс `Employee`, компилятор Java станет искать файл `Employee.class`. Если компилятор его не найдет, то автоматически будет скомпилирован файл `Employee.java`. Более того, если файл `Employee.java` создан позже, чем существующий файл `Employee.class`, компилятор языка Java *автоматически* выполнит повторную компиляцию и создаст исходный файл данного класса.



НА ЗАМЕТКУ! Если вы знакомы с утилитой `make`, доступной в Unix, или родственными ей утилитами в других операционных системах (например, `nmake` в Windows), то такое поведение компилятора не станет для вас неожиданностью. Дело в том, что в компиляторе Java реализованы функциональные возможности этой утилиты.

4.3.3. Анализ класса `Employee`

Проанализируем класс `Employee`, начав с его методов. Изучая исходный код, не трудно заметить, что в классе `Employee` реализованы один конструктор и четыре метода, перечисляемые ниже.

```
public Employee(String n, double s, int year, int month, int day)
public String getName()
public double getSalary()
public Date getHireDay()
public void raiseSalary(double byPercent)
```

Все методы этого класса объявлены как `public`, т.е. обращение к ним может осуществляться из любого класса. (Существуют четыре возможных уровня доступа. Все они рассматриваются в этой и следующей главах.)

Далее следует заметить, что в классе имеются три поля экземпляра для хранения данных, обрабатываемых в объекте типа `Employee`:

```
private String name;  
private double salary;  
private Date hireDay;
```

Ключевое слово `private` означает, что к данным полям имеют доступ только методы самого класса `Employee`. Ни один внешний метод не может читать или записывать данные в эти поля.



НА ЗАМЕТКУ! Поля экземпляра могут быть объявлены как `public`, но делать этого не следует. Ведь в этом случае любые компоненты программы (классы и методы) могут обратиться к открытым полям и видоизменить их содержимое, и, как показывает опыт, всегда найдется какой-нибудь код, который непременно воспользуется этими правами доступа в самый неподходящий момент. Поэтому настоятельно рекомендуем всегда закрывать доступ к полям экземпляра с помощью ключевого слова `private`.

И, наконец, следует обратить внимание на то, что два из трех полей экземпляра сами являются объектами. В частности, поля `name` и `hireDay` являются ссылками на экземпляры классов `String` и `Date`. В ООП это довольно распространенное явление: одни классы часто содержат поля с экземплярами других классов.

4.3.4. Первые действия с конструкторами

Рассмотрим следующий конструктор класса `Employee`:

```
public Employee(String n, double s, int year, int month, int day)  
{  
    name = n;  
    salary = s;  
    GregorianCalendar calendar =  
        new GregorianCalendar(year, month - 1, day);  
    hireDay = calendar.getTime();  
}
```

Как видите, имя конструктора совпадает с именем класса. Этот конструктор выполняется при создании объекта типа `Employee`, заполняя поля экземпляра заданными значениями. Например, при создании экземпляра класса `Employee` с помощью оператора `new`

```
new Employee("James Bond", 100000, 1950, 1, 1);
```

поля экземпляра заполняются такими значениями:

```
name = "James Bond";  
salary = 100000;  
hireDay = LocalDate.of(1950, 1, 1); // 1 января 1950 г.
```

У конструкторов имеется существенное отличие от других методов: конструктор можно вызывать только в сочетании с операцией `new`. Конструктор нельзя применить к существующему объекту, чтобы изменить информацию в его полях. Например, приведенный ниже вызов приведет к ошибке во время компиляции.

```
james.Employee("James Bond", 250000, 1950, 1, 1); // ОШИБКА!
```

Мы еще вернемся в этой главе к конструкторам. А до тех пор запомните следующее.

- Имя конструктора совпадает с именем класса.
- Класс может иметь несколько конструкторов.

- Конструктор может иметь один или несколько параметров или же вообще их не иметь.
- Конструктор не возвращает никакого значения.
- Конструктор всегда вызывается с операцией `new`.



НА ЗАМЕТКУ C++! Конструкторы в Java и C++ действуют одинаково. Но учтите, что все объекты в Java размещаются в динамической памяти и конструкторы вызываются только вместе с операцией `new`. Те, у кого имеется опыт программирования на C++, часто допускают следующую ошибку:

```
Employee number007("James Bond", 10000, 1950, 1, 1)  
    // допустимо в C++, но не в Java!
```

Это выражение в C++ допустимо, а в Java — нет.



ВНИМАНИЕ! Будьте осмотрительны, чтобы не присваивать локальным переменным такие же имена, как и полям экземпляра. Например, приведенный ниже конструктор не сможет установить зарплату работника.

```
public Employee(String n, double s, ...)
{
    String name = n; // ОШИБКА!
    double salary = s; // ОШИБКА!
}
```

В конструкторе объявляются *локальные* переменные `name` и `salary`. Доступ к этим переменным возможен только внутри конструктора. Они *скрывают* поля экземпляра с аналогичными именами. Некоторые программисты могут написать такой код автоматически. Подобные ошибки очень трудно обнаружить. Поэтому нужно быть внимательными, чтобы не присваивать переменным имена полей экземпляра.

4.3.5. Объявление локальных переменных с помощью ключевого слова `var`

Начиная с версии Java 10, локальные переменные можно объявлять с помощью ключевого слова `var` вместо того, чтобы указывать их тип, при условии, что этот тип может быть выведен из первоначального значения. Например, вместо объявления

```
Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
```

достаточно написать объявляемую переменную так:

```
var harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
```

И это очень удобно, поскольку избавляет от необходимости повторять имя типа `Employee`.

В последующих далее примерах кода локальные переменные будут объявляться с помощью ключевого слова `var` в тех случаях, когда их тип очевиден из правой части выражения даже тем, кто вообще незнаком с прикладным интерфейсом Java API. Но ключевое слово `var` не будет употребляться для объявления локальных переменных таких числовых типов, как `int`, `long` или `double`, чтобы были ясны отличия между значениями `0`, `0L` и `0.0`. Как только вы приобретете больше опыта работы с прикладным интерфейсом Java API, у вас возникнет потребность чаще пользоваться ключевым словом `var`.

Следует, однако, иметь в виду, что ключевое слово `var` можно употреблять в локальных переменных, объявляемых *только* в теле методов. А типы параметров и полей следует всегда объявлять явным образом.

4.3.6. Обработка пустых ссылок на объекты

Как было показано в разделе 4.2.1, в объектной переменной хранится ссылка на объект или специальное значение `null`, обозначающее отсутствие объекта. На первый взгляд, это удобный механизм в тех особых случаях, когда Ф.И.О. или дата найма работника неизвестны. Но обращаться с пустыми значениями `null` следует очень аккуратно.

Так, если применить метод к пустому значению `null`, возникнет исключение типа `NullPointerException`:

```
LocalDate birthday = null;
String s = birthday.toString();
// возникает исключение типа NullPointerException
```

Это весьма серьезная ошибка, подобная исключению, возникающему в связи с выходом индекса за границы массива. Если не перехватить подобное исключение в прикладной программе, она завершится преждевременно. Как правило, в прикладных программах подобные исключения не перехватываются, но их создатели полагаются, прежде всего, на свое умение не допустить ничего подобного.

Определяя класс, целесообразно уточнить те его поля, которые могут содержать пустые значения `null`. Так, в рассматриваемом здесь примере поля `name` и `hireDay` не должны содержать пустое значение `null`, тогда как о поле `salary` можно не особенно беспокоиться, поскольку оно относится к примитивному типу и вообще не может принимать пустое значение `null`.

Поле `hireDay` гарантированно не может содержать пустое значение `null`, поскольку оно инициализируется новым объектом типа `LocalDate`. А вот поле `name` может принимать пустое значение `null`, если конструктор его класса вызывается с пустым значением `null` аргумента `n`.

Разрешить подобное затrudнение можно двумя способами. Простейший состоит в том, чтобы преобразовать пустое значение `null` аргумента в непустое:

```
if (n == null) name = "unknown";
else name = n;
```

Начиная с версии Java 9, в класс `Objects` для этой цели был внедрен удобный метод `requireNonNullElse()`, вызываемый как показано ниже.

```
public Employee(String n, double s, int year, int month, int day)
{
    name = Objects.requireNonNullElse(n, "unknown");
    . . .
}
```

Более “жесткий” способ состоит в том, чтобы просто отвергнуть аргумент с пустым значением `null`:

```
public Employee(String n, double s, int year, int month, int day)
{
    Objects.requireNonNull(n, "The name cannot be null");
```



```
name = n;  
...  
}
```

Если попытаться построить объект типа `Employee` с пустым значением `null` в поле `name`, то возникнет исключение типа `NullPointerException`. На первый взгляд, пользы от такого способа мало, однако у него имеются следующие преимущества.

1. Возникшее исключение сообщает о возникшем затруднении.
2. В сообщении о возникшем исключении ясно указывается место появления затруднения. В противном случае исключение типа `NullPointerException` возникало бы где угодно, и проследить обратно от него ошибочный аргумент конструктора было бы нелегко.



НА ЗАМЕТКУ! Всякий раз, когда вы принимаете ссылку на объект в качестве параметра конструктора, задайтесь вопросом: действительно ли вы намереваетесь смоделировать значения, которые могут присутствовать или отсутствовать. Если у вас нет такого намерения, выберите "жесткий" способ обработки пустых ссылок на объекты.

4.3.7. Явные и неявные параметры

Методы объекта имеют доступ ко всем его полям. Рассмотрим следующий метод:

```
public void raiseSalary(double byPercent)  
{  
    double raise = salary * byPercent / 100;  
    salary += raise;  
}
```

В этом методе устанавливается новое значение в поле `salary` того объекта, для которого вызывается этот метод. Например, следующий вызов данного метода:

```
number007.raiseSalary(5);
```

приведет к увеличению на 5% значения в поле `number007.salary` объекта `number007`. Строго говоря, вызов данного метода приводит к выполнению следующих двух операторов:

```
double raise = number007.salary * 5 / 100;  
number007.salary += raise;
```

У метода `raiseSalary()` имеются два параметра. Первый, называемый *неявным*, представляет собой ссылку на объект типа `Employee`, который указывается перед именем метода. Второй параметр называется *явным* и указывается как число в скобках после имени данного метода.

Нетрудно заметить, что явные параметры перечисляются в объявлении метода, например `double byPercent`. Неявный параметр в объявлении метода не приводится. В каждом методе ключевое слово `this` обозначает неявный параметр. По желанию метод `raiseSalary()` можно было бы переписать следующим образом:

```
public void raiseSalary(double byPercent)  
{  
    double raise = this.salary * byPercent / 100;  
    this.salary += raise;  
}
```

Некоторые предпочитают именно такой стиль программирования, поскольку в нем более отчетливо различаются поля экземпляра и локальные переменные.



НА ЗАМЕТКУ C++! Методы обычно определяются в C++ за пределами класса, как показано ниже.

```
void Employee::raiseSalary(double byPercent)
    // В C++, но не в Java
{
    ...
}
```

Если определить метод в классе, он автоматически станет встраиваемым.

```
class Employee
{
    ...
    int getName() { return name; }
    // встраиваемая функция в C++
}
```

А в Java все методы определяются в пределах класса, но это не делает их встраиваемыми. Виртуальная машина Java анализирует, как часто производится обращение к методу, и принимает решение, должен ли метод быть встраиваемым. Динамический компилятор находит краткие, частые вызовы методов, которые не являются переопределенными, и оптимизирует их соответствующим образом.

4.3.8. Преимущества инкапсуляции

Рассмотрим очень простые методы `getName()`, `getSalary()` и `getHireDay()` из класса `Employee`. Их исходный код приведен ниже.

```
public String getName()
{
    return name;
}

public double getSalary()
{
    return salary;
}

public Date getHireDay()
{
    return hireDay;
}
```

Они служат характерным примером *методов доступа*. А поскольку они лишь возвращают значения полей экземпляра, то иногда их еще называют *методами доступа к полям*. Но не проще ли было сделать поля `name`, `salary` и `hireDay` открытыми для доступа (т.е. объявить их как `public`) и не создавать отдельные методы доступа к ним?

Дело в том, что поле `name` доступно только для чтения. После того как значение этого поля будет установлено конструктором, ни один метод не сможет его изменить. А это дает гарантию, что данные, хранящиеся в этом поле, не будут искажены.

Поле `salary` доступно не только для чтения, но и для записи, но изменить значение в нем способен только метод `raiseSalary()`. И если окажется, что в поле

записано неверное значение, то отладить нужно будет только один метод. Если бы поле `salary` было открытым, причина ошибки могла бы находиться где угодно.

Иногда требуется иметь возможность читать и видоизменять содержимое поля. Для этого придется реализовать в составе класса следующие три компонента.

- Закрытое (`private`) поле данных.
- Открытый (`public`) метод доступа.
- Открытый (`public`) модифицирующий метод.

Конечно, сделать это намного труднее, чем просто объявить открытым единственное поле данных. Но такой подход дает немалые преимущества. Во-первых, внутреннюю реализацию класса можно изменять совершенно независимо от других классов. Допустим, что имя и фамилия работника хранятся отдельно:

```
String firstName;  
String lastName;
```

Тогда в методе `getName()` возвращаемое значение должно быть сформировано следующим образом:

```
firstName + " " + lastName
```

И такое изменение оказывается совершенно незаметным для остальной части программы. Разумеется, методы доступа и модифицирующие методы должны быть переработаны, чтобы учесть новое представление данных. Но это дает еще одно преимущество: модифицирующие методы могут выполнять проверку ошибок, тогда как при непосредственном присваивании открытому полю некоторого значения ошибки не выявляются. Например, в методе `setSalary()` можно проверить, не стала ли зарплата отрицательной величиной.



ВНИМАНИЕ! Будьте осмотрительны при создании методов доступа, возвращающих ссылки на изменяемый объект. Создавая класс `Employee`, мы нарушили это правило в предыдущих изданиях книги: метод `getHireDay()` возвращает объект класса `Date`, как показано ниже.

```
class Employee  
{  
    private Date hireDay;  
    ...  
    public Date getHire();  
    {  
        return hireDay; // Неудачно!  
    }  
    ...  
}
```

В отличие от класса `LocalDate`, где отсутствуют модифицирующие методы, в классе `Date` имеется модифицирующий метод `setTime()` для установки времени в миллисекундах. Но из-за того, что объекты типа `Date` оказываются изменяемыми, нарушается принцип инкапсуляции! Рассмотрим следующий пример неверного кода:

```
Employee harry = ...;  
Date d = harry.getHireDay();  
double tenYearsInMilliSeconds =  
    10 * 365.25 * 24 * 60 * 60 * 1000;  
d.setTime(d.getTime() - (long) tenYearsInMilliSeconds);  
// значение в объекте изменено
```

Причина ошибки в этом коде проста: обе ссылки, `d` и `harry.hireDay`, делаются на один и тот же объект (рис. 4.5). В результате применения модифицирующих методов к объекту `d` автоматически изменяется открытое состояние объекта работника типа **Employee**!

Чтобы вернуть ссылку на изменяемый объект, его нужно сначала клонировать. Клон — это точная копия объекта, находящаяся в другом месте памяти. Подробнее о клонировании речь пойдет в главе 6. Ниже приведен исправленный код.

```
class Employee
{
    . . .
    public Date getHireDay()
    {
        return hireDay.clone();
    }
    . . .
}
```

В качестве эмпирического правила пользуйтесь методом `clone()`, если вам нужно скопировать изменяемое поле данных.

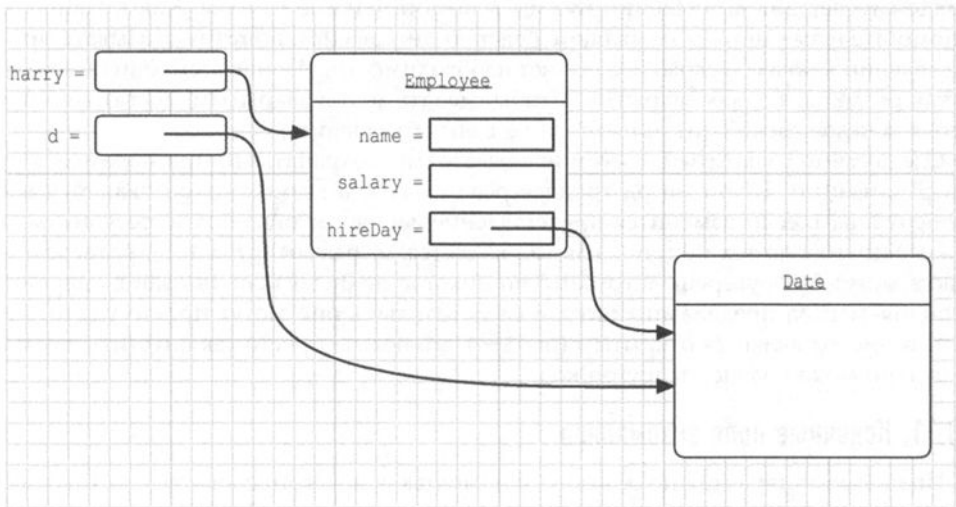


Рис. 4.5. Возврат ссылки на изменяемое поле данных

4.3.9. Привилегии доступа к данным в классе

Как вам должно быть уже известно, метод имеет доступ к закрытым данным того объекта, для которого он вызывается. Но он также может обращаться к закрытым данным *всех* объектов своего класса! Рассмотрим в качестве примера метод `equals()`, сравнивающий два объекта типа `Employee`:

```
class Employee
{
    . . .
    public boolean equals(Employee other)
    {
```

```
    return name.equals(other.name);  
}  
}
```

Типичный вызов этого метода выглядит следующим образом:

```
if (harry.equals(boss)) ...
```

Этот метод имеет доступ к закрытым полям объекта `harry`, что и не удивительно. Но он также имеет доступ к полям объекта `boss`. И это вполне объяснимо, поскольку `boss` — объект типа `Employee`, а методы, принадлежащие классу `Employee`, могут обращаться к закрытым полям *любого* объекта этого класса.



НА ЗАМЕТКУ C++! В языке C++ действует такое же правило. Метод имеет доступ к переменным и функциям любого объекта своего класса.

4.3.10. Закрытые методы

При реализации класса все поля данных делаются закрытыми, поскольку предоставлять к ним доступ из других классов весьма рискованно. А как поступить с методами? Для взаимодействия с другими объектами требуются открытые методы. Но в ряде случаев для вычислений нужны вспомогательные методы. Как правило, эти вспомогательные методы не являются частью интерфейса, поэтому указывать при их объявлении ключевое слово `public` нет необходимости. И чаще всего они объявляются как `private`, т.е. как закрытые. Чтобы сделать метод закрытым, достаточно изменить ключевое слово `public` на `private` в его объявлении.

Сделав метод закрытым, совсем не обязательно сохранять его при переходе к другой реализации. Такой метод труднее реализовать, а возможно, он окажется вообще ненужным, если изменится представление данных, что, в общем, несущественно. Важнее другое: до тех пор, пока метод является закрытым (`private`), разработчики класса могут быть уверены в том, что он никогда не будет использован в операциях, выполняемых за пределами класса, а следовательно, они могут просто удалить его. Если же метод является открытым (`public`), его нельзя просто так опустить, поскольку от него может зависеть другой код.

4.3.11. Конечные поля экземпляра

Поля экземпляра можно объявить с помощью ключевого слова `final`. Такое поле должно инициализироваться при создании объекта, т.е. необходимо гарантировать, что значение поля будет установлено по завершении каждого конструктора. После этого его значение изменить уже нельзя. Например, поле `name` из класса `Employee` можно объявить конечным (т.е. неизменяемым), поскольку после создания объекта оно уже не изменяется, а метода `setName()` для этого не существует.

```
class Employee  
{  
    ...  
    private final String name;  
}
```

Модификатор `final` удобно применять при объявлении полей простых типов или полей, типы которых задаются *конечными классами*. Конечным называется такой класс, методы которого не позволяют изменить состояние объекта. Например,

конечным является класс `String`. Если класс допускает изменения, то ключевое слово `final` может стать источником недоразумений. Рассмотрим следующее поле:

```
private final StringBuilder evaluations;
```

которое инициализируется в конструкторе класса `Employee` таким образом:

```
evaluations = new StringBuilder();
```

Ключевое слово `final` означает, что ссылка на объект, хранящаяся в переменной `evaluations`, вообще не будет делаться на другой объект типа `StringBuilder`. Но в то же время объект может быть изменен следующим образом:

```
public void giveGoldStar()
{
    evaluations.append(LocalDate.now() + ": Gold star!\n");
}
```

4.4. Статические поля и методы

При объявлении метода `main()` во всех рассматривавшихся до сих пор примерах программ использовался модификатор `static`. Рассмотрим назначение этого модификатора доступа.

4.4.1. Статические поля

Поле с модификатором доступа `static` существует в одном экземпляре для всего класса. Но если поле не статическое, то каждый объект содержит его копию. Допустим, требуется присвоить однозначный идентификационный номер каждому работнику. Для этого достаточно ввести в класс `Employee` поле `id` и статическое поле `nextId`, как показано ниже.

```
class Employee
{
    ...
    private int id;

    private static int nextId = 1;
}
```

Теперь у каждого объекта типа `Employee` имеется свое поле `id`, а также поле `nextId`, которое одновременно принадлежит всем экземплярам данного класса. Иными словами, если существует тысяча объектов типа `Employee`, то в них есть тысяча полей `id`: по одному на каждый объект. В то же время существует только один экземпляр статического поля `nextId`. Даже если не создано ни одного объекта типа `Employee`, статическое поле `nextId` все равно существует. Оно принадлежит классу, а не конкретному объекту.



НА ЗАМЕТКУ! В большинстве объектно-ориентированных языков статические поля называются *полями класса*. Термин *статический* унаследован как малозначащий пережиток от языка C++.

Реализуем следующий простой метод:

```
public void setId()
{
```

```
id = nextId;
nextId++;
}
```

Допустим, требуется задать идентификационный номер объекта `harry` следующим образом:

```
harry.setId();
```

Затем устанавливается текущее значение в поле `id` объекта `harry`, а значение статического поля `nextId` увеличивается на единицу, как показано ниже.

```
harry.id = Employee.nextId;
Employee.nextId++;
```

4.4.2. Статические константы

Статические переменные используются довольно редко. В то же время статические константы применяются намного чаще. Например, статическая константа, задающая число π , определяется в классе `Math` следующим образом:

```
public class Math
{
    ...
    public static final double PI = 3.14159265358979323846;
    ...
}
```

Обратиться к этой константе в программе можно с помощью выражения `Math.PI`. Если бы ключевое слово `static` было пропущено, константа `PI` была бы обычным полем экземпляра класса `Math`. Это означает, что для доступа к такой константе нужно было бы создать объект типа `Math`, причем каждый такой объект имел бы свою копию константы `PI`.

Еще одной часто употребляемой является статическая константа `System.out`. Она объявляется в классе `System` следующим образом:

```
public class System
{
    ...
    public static final PrintStream out = ...;
    ...
}
```

Как уже упоминалось не раз, делать поля открытыми в коде не рекомендуется, поскольку любой объект сможет изменить их значения. Но открытыми константами (т.е. полями, объявленными с ключевым словом `final`) можно пользоваться смело. Так, если поле `out` объявлено как `final`, ему нельзя присвоить другой поток вывода:

```
System.out = new PrintStream(...); // ОШИБКА: поле out изменить нельзя!
```



НА ЗАМЕТКУ! Анализируя исходный код класса `System`, можно обнаружить в нем метод `setOut()`, позволяющий присвоить полю `System.out` другой поток вывода. Как же этот метод может изменить переменную, объявленную как `final`? Дело в том, что метод `setOut()` является платформенно-ориентированным, т.е. он реализован средствами конкретной платформы, а не языка Java. Платформенно-ориентированные методы способны обходить механизмы контроля, предусмотренные в Java. Это весьма необычный обходной прием, который ни в коем случае не следует применять в своих программах.

4.4.3. Статические методы

Статическими называют методы, которые не оперируют объектами. Например, метод `pow()` из класса `Math` является статическим. При вызове метода `Math.pow(x, a)` вычисляется степень числа x^a . При выполнении этого метода не используется ни один из экземпляров класса `Math`. Иными словами, у него нет неявного параметра `this`. Это означает, что в статических методах не используется текущий объект по ссылке `this`. (А в нестатических методах неявный параметр `this` ссылается на текущий объект; см. выше раздел 4.3.7).

Статическому методу из класса `Employee` недоступно поле экземпляра `id`, поскольку он не оперирует объектом. Но статические методы имеют доступ к статическим полям класса. Ниже приведен пример статического метода.

```
public static int getNextId()
{
    return nextId; // вернуть статическое поле
}
```

Чтобы вызвать этот метод, нужно указать имя класса следующим образом:

```
int n = Employee.getNextId();
```

Можно ли пропустить ключевое слово `static` при обращении к этому методу? Можно, но тогда для его вызова потребуется ссылка на объект типа `Employee`.



НА ЗАМЕТКУ! Для вызова статического метода можно использовать и объекты. Так, если `harry` — это объект типа `Employee`, то вместо вызова `Employee.getNextId()` можно сделать вызов `harry.getNextId()`. Но такое обозначение усложняет восприятие программы, поскольку для вычисления результата метод `getNextId()` не обращается к объекту `harry`. Поэтому для вызова статических методов рекомендуется использовать имена их классов, а не объекты.

Статические методы следует применять в двух случаях.

- Когда методу не требуется доступ к данным о состоянии объекта, поскольку все необходимые параметры задаются явно (например, в методе `Math.pow()`).
- Когда методу требуется доступ лишь к статическим полям класса (например, при вызове метода `Employee.getNextId()`).



НА ЗАМЕТКУ C++! Статические поля и методы в Java и C++, по существу, отличаются только синтаксически. Для доступа к статическому полю или методу, находящемуся вне области действия, в C++ можно воспользоваться операцией `::`, например `Math::PI`.

Любопытно происхождение термина *статический*. Сначала ключевое слово `static` было внедрено в C для обозначения локальных переменных, которые не уничтожались при выходе из блока. В этом контексте термин *статический* имеет смысл: переменная продолжает существовать после выхода из блока, а также при повторном входе в него. Затем термин *статический* приобрел в C второе значение для глобальных переменных и функций, к которым нельзя получить доступ из других файлов. Ключевое слово `static` было просто использовано повторно, чтобы не вводить новое. И, наконец, в C++ это ключевое слово было применено в третий раз, получив совершенно новую интерпретацию. Оно обозначает переменные и методы, принадлежащие классу, но ни одному из объектов этого класса. Именно этот смысл ключевое слово `static` имеет и в Java.

4.4.4. Фабричные методы

Рассмотрим еще одно применение статических методов. В таких классах, как, например, `LocalDate` и `NumberFormat`, для построения объектов применяются статические фабричные методы. Ранее в этой главе уже демонстрировалось применение фабричных методов `LocalDate.now()` и `LocalDate.of()`. А в следующем примере кода показано, как в классе `NumberFormat` получают форматирующие объекты для разных стилей вывода результатов:

```
NumberFormat currencyFormatter = NumberFormat.getCurrencyInstance();
NumberFormat percentFormatter = NumberFormat.getPercentInstance();
double x = 0.1;
System.out.println(currencyFormatter.format(x)); // выводит $0.10
System.out.println(percentFormatter.format(x)); // выводит 10%
```

Почему бы не использовать для этой цели конструктор? На то есть две причины.

- Конструктору нельзя присвоить произвольное имя — его имя всегда должно совпадать с именем класса. Так, в классе `NumberFormat` имеет смысл применять разные имена для разных типов форматирования.
- При использовании конструктора тип объекта фиксирован. Если же применяются фабричные методы, они возвращают объект типа `DecimalFormat`, наследующий свойства из класса `NumberFormat`. (Подробнее вопросы наследования будут обсуждаться в главе 5.)

4.4.5. Метод `main()`

Отметим, что статические методы можно вызывать, не имея ни одного объекта данного класса. Например, для того чтобы вызвать метод `Math.pow()`, объекты типа `Math` не нужны. По той же причине метод `main()` объявляется как статический:

```
public class Application
{
    public static void main(String[] args)
    {
        // здесь создаются объекты
    }
}
```

Метод `main()` не оперирует никакими объектами. На самом деле при запуске программы еще нет никаких объектов. Статический метод `main()` выполняется и конструирует объекты, необходимые программе.



СОВЕТ. Каждый класс может содержать метод `main()`. С его помощью удобно организовать модульное тестирование классов. Например, метод `main()` можно добавить в класс `Employee` следующим образом:

```
class Employee
{
    public Employee(String n, double s, int year,
                    int month, int day)
    {
        name = n;
        salary = s;
        hireDay = LocalDate.of(year, month, day);
    }
}
```

```

    }
    . . .
    public static void main(String[] args) // модульный тест
    {
        var e = new Employee ("Romeo", 50000, 2003, 3, 31);
        e.raiseSalary(10);
        System.out.println(e.getName() + " " + e.getSalary());
    }
    . . .
}

```

Если требуется протестировать только класс **Employee**, то для этого достаточно ввести команду `java Employee`. А если класс **Employee** является частью крупного приложения, то последнее можно запустить на выполнение по команде `java Application`. В этом случае метод `main()` из класса **Employee** вообще не будет выполнен.

Программа, приведенная в листинге 4.3, содержит простую версию класса **Employee** со статическим полем `nextId` и статическим методом `getNextId()`. В этой программе массив заполняется тремя объектами типа **Employee**, а затем выводятся данные о работниках. И, наконец, для демонстрации статического метода на экран выводится очередной доступный идентификационный номер.

Следует, однако, иметь в виду, что в классе **Employee** имеется также статический метод `main()` для модульного тестирования. Попробуйте выполнить метод `main()` по командам `java Employee` и `java StaticTest`.

Листинг 4.3. Исходный код из файла **StaticTest/StaticTest.java**

```

1  /**
2   * В этой программе демонстрируются статические методы
3   * @version 1.02 2008-04-10
4   * @author Cay Horstmann
5   */
6  public class StaticTest
7  {
8      public static void main(String[] args)
9      {
10         // заполнить массив staff тремя объектами
11         // типа Employee
12         Employee[] staff = new Employee[3];
13
14         staff[0] = new Employee("Tom", 40000);
15         staff[1] = new Employee("Dick", 60000);
16         staff[2] = new Employee("Harry", 65000);
17
18         // вывести данные обо всех объектах типа Employee
19         for (Employee e : staff)
20         {
21             e.setId();
22             System.out.println("name=" + e.getName() + ",id="
23                               + e.getId() + ",salary=" + e.getSalary());
24         }

```

```
25
26     int n = Employee.getNextId(); // вызвать
27                                     // статический метод
28     System.out.println("Next available id=" + n);
29 }
30 }
31
32 class Employee
33 {
34     private static int nextId = 1;
35
36     private String name;
37     private double salary;
38     private int id;
39
40     public Employee(String n, double s)
41     {
42         name = n;
43         salary = s;
44         id = 0;
45     }
46     public String getName()
47     {
48         return name;
49     }
50
51     public double getSalary()
52     {
53         return salary;
54     }
55
56     public int getId()
57     {
58         return id;
59     }
60
61     public void setId()
62     {
63         id = nextId; // установить следующий
64                     // доступный идентификатор
65         nextId++;
66     }
67
68     public static int getNextId()
69     {
70         return nextId; // вернуть статическое поле
71     }
72
73     public static void main(String[] args)
74         // выполнить модульный тест
75     {
76         var e = new Employee("Harry", 50000);
77         System.out.println(e.getName() + " " + e.getSalary());
78     }
79 }
```

java.util.Objects 7

- `static <T> void requireNonNull(T obj)`
- `static <T> void requireNonNull(T obj, String message)`
- `static <T> void requireNonNull(T obj, Supplier<String> messageSupplier)` 8
Если параметр *obj* принимает пустое значение `null`, эти методы генерируют исключение типа `NullPointerException` без стандартного или заданного сообщения об ошибке. (В главе 6 поясняется, как получить значение по требованию с помощью поставщика, а в главе 8 — синтаксис обобщенного типа `<T>`.)
- `static <T> T requireNonNullElse(T obj, T defaultObj)`
- `static <T> T requireNonNullElseGet(T obj, Supplier<T> defaultSupplier)`
Возвращают объект, задаваемый параметром *obj*, если этот параметр не принимает пустое значение `null`, а иначе — объект, выбираемый по умолчанию.

4.5. Параметры методов

Рассмотрим термины, которые употребляются для описания способа передачи параметров методам (или функциям) в языках программирования. Термин *вызов по значению* означает, что метод получает значение, переданное ему из вызывающей части программы. *Вызов по ссылке* означает, что метод получает из вызывающей части программы *местоположение* переменной. Таким образом, метод может *модифицировать* (т.е. видоизменить) значение переменной, передаваемой по ссылке, но не переменной, передаваемой по значению. Фраза “вызов по...” относится к стандартной компьютерной терминологии, описывающей способ передачи параметров в различных языках программирования, а не только в Java. (На самом деле существует еще и третий способ передачи параметров — *вызов по имени*, представляющий в основном исторический интерес, поскольку он был применен в языке Algol, который относится к числу самых старых языков программирования высокого уровня.)

В языке Java всегда используется *только* вызов по значению. Это означает, что метод получает копии значений всех своих параметров. По этой причине метод не может видоизменить содержимое ни одной из переменных, передаваемых ему в качестве параметров.

Рассмотрим для примера следующий вызов:

```
double percent = 10;
harry.raiseSalary(percent);
```

Каким бы образом ни был реализован метод, после его вызова значение переменной `percent` все равно останется равным 10.

Проанализируем эту ситуацию подробнее. Допустим, в методе предпринимается попытка утроить значение параметра, как показано ниже.

```
public static void tripleValue(double x); // не сработает!
{
    x = 3 * x;
}
```

Если вызвать этот метод следующим образом:

```
double percent = 10;
tripleValue(percent);
```

такой прием не работает. После вызова метода значение переменной `percent` по-прежнему остается равным 10. В данном случае происходит следующее.

1. Переменная `x` инициализируется копией значения параметра `percent` (т.е. числом 10).
2. Значение переменной `x` утраивается, и теперь оно равно 30. Но значение переменной `percent` по-прежнему остается равным 10 (рис. 4.6).
3. Метод завершает свою работу, и его переменный параметр `x` больше не используется.

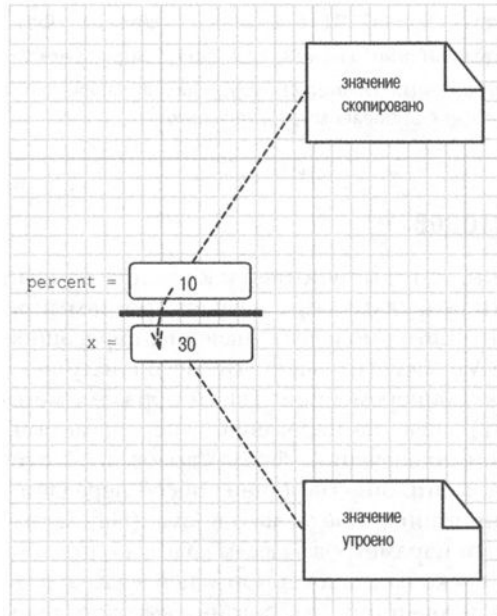


Рис. 4.6. Видоизменение значения в вызываемом методе не оказывает никакого влияния на параметр, передаваемый из вызывающей части программы

Но существуют два следующих типа параметров методов.

- Примитивные типы (т.е. числовые и логические значения).
- Ссылки на объекты.

Как было показано выше, методы не могут видоизменить параметры примитивных типов. Совсем иначе дело обстоит с объектами. Нетрудно реализовать метод, утраивающий зарплату работников, следующим образом:

```
public static void tripleSalary(Employee x) // работает!  
{  
    x.raiseSalary(200);  
}
```

При выполнении следующего фрагмента кода происходят перечисленные ниже действия.

```
harry = new Employee(...);  
tripleSalary(harry);
```

4. Переменная **x** инициализируется копией значения переменной **harry**, т.е. ссылкой на объект.
5. Метод `raiseSalary()` применяется к объекту по этой ссылке. В частности, объект типа `Employee`, доступный по ссылкам **x** и **harry**, получает сумму зарплаты работников, увеличенную на 200%.
6. Метод завершает свою работу, и его параметр **x** больше не используется. Разумеется, переменная **harry** продолжает ссылаться на объект, где зарплата увеличена втрое (рис. 4.7).

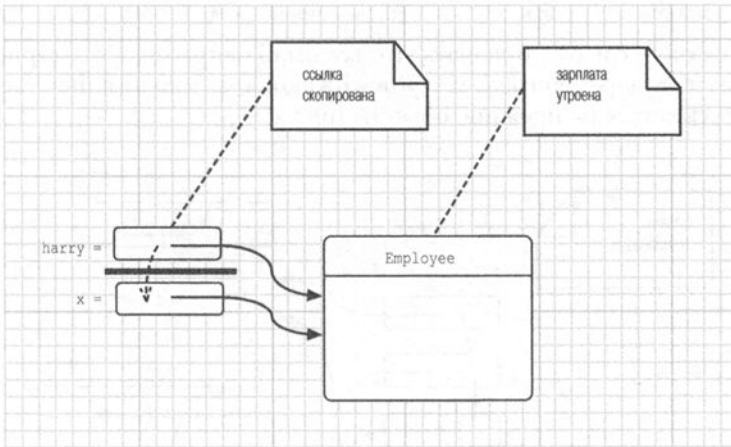


Рис. 4.7. Если параметр ссылается на объект, последний может быть видоизменен

Как видите, реализовать метод, изменяющий состояние объекта, передаваемого как параметр, совсем не трудно. В действительности такие изменения вносятся очень часто по следующей простой причине: метод получает копию ссылки на объект, поэтому копия и оригинал ссылки указывают на один и тот же объект.

Во многих языках программирования (в частности, C++ и Pascal) предусмотрены два способа передачи параметров: вызов по значению и вызов по ссылке. Некоторые программисты (и, к сожалению, даже авторы некоторых книг) утверждают, что в Java при передаче объектов используется вызов по ссылке. Но это совсем не так. Для того чтобы развеять это бытующее заблуждение, обратимся к конкретному примеру. Ниже приведен метод, выполняющий обмен двух объектов типа `Employee`.

```
public static void swap(Employee x, Employee y) // не сработает!  
{  
    Employee temp = x;  
    x = y;  
    y = temp;  
}
```

Если бы в Java для передачи объектов в качестве параметров использовался вызов по ссылке, этот метод действовал бы следующим образом:

```
var a = new Employee("Alice", . . .);  
var b = new Employee("Bob", . . .);  
swap(a, b); // ссылается ли теперь переменная a на Bob,  
            // а переменная b – на Alice?
```

Но на самом деле этот метод не меняет местами ссылки на объекты, хранящиеся в переменных `a` и `b`. Сначала параметры `x` и `y` метода `swap()` инициализируются копиями этих ссылок, а затем эти копии меняются местами в данном методе, как показано ниже.

```
// исходно переменная x ссылается на Alice, а переменная y – на Bob  
Employee temp = x;  
x = y;  
y = temp;  
// теперь переменная x ссылается на Bob, а переменная y – на Alice
```

В конечном счете следует признать, что все было напрасно. По завершении работы данного метода переменные `x` и `y` уничтожаются, а исходные переменные `a` и `b` продолжают ссылаться на прежние объекты (рис. 4.8).

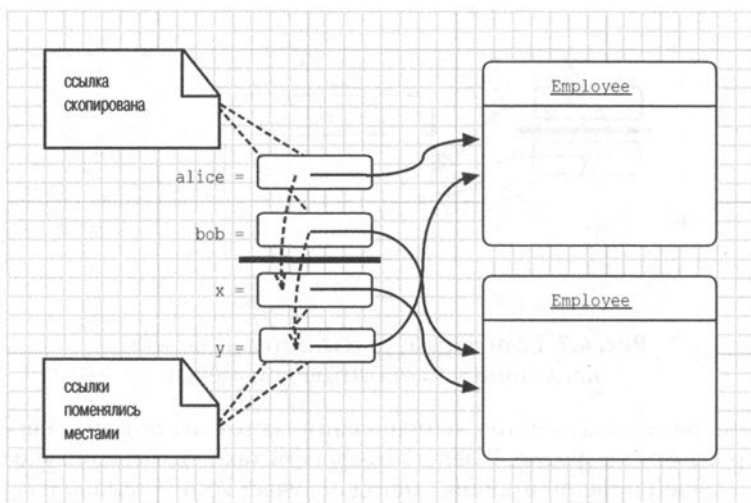


Рис. 4.8. Перестановка ссылок в вызываемом методе не имеет никаких последствий для вызывающей части программы

Таким образом, в Java для передачи объектов не применяется вызов по ссылке. Вместо этого ссылки на объекты передаются *по значению*. Ниже поясняется, что может и чего не может делать метод со своими параметрами.

- Не может изменять параметры примитивных типов (т.е. числовые и логические значения).
- Может изменять *состояние* объекта, передаваемого в качестве параметра.
- Не может делать в своих параметрах ссылки на новые объекты.

Все эти положения демонстрируются в примере программы из листинга 4.4. Сначала в ней предпринимается безуспешная попытка устроить значение числового параметра.

```
Testing tripleValue:
Before: percent=10.0
End of method: x=30.0
After: percent=10.01
```

Затем в программе успешно утраивается зарплата работника.

```
Testing tripleSalary:
Before: salary=50000.0
End of method: salary=150000.0
After: salary=150000.0
```

После выполнения метода состояние объекта, на который ссылается переменная `harry`, изменяется. Ничего невероятного здесь нет, поскольку в данном методе состояние объекта было модифицировано по копии ссылки на него. И, наконец, в программе демонстрируется безрезультатность работы метода `swap()`.

```
Testing swap:
Before: a=Alice
Before: b=Bob
End of method: x=Bob
End of method: y=Alice
After: a=Alice
After: b=Bob
```

Как видите, переменные параметры `x` и `y` меняются местами, но исходные переменные `a` и `b` остаются без изменения.



НА ЗАМЕТКУ C++! В языке C++ применяется как вызов по значению, так и вызов по ссылке. Например, можно без особого труда реализовать методы `void tripleValue(double& x)` или `void swap(Employee& x, Employee& y)`, которые видоизменяют свои ссылочные параметры.

Листинг 4.4. Исходный код из файла `ParamTest/ParamTest.java`

```
1  /**
2   * В этой программе демонстрируется передача
3   * параметров в Java
4   * @version 1.01 2018-04-10
5   * @author Cay Horstmann
6   */
7  public class ParamTest
8  {
9      public static void main(String[] args)
10     {
11         /*
12          * Тест 1: методы не могут видоизменять
13          * числовые параметры
14          */
15         System.out.println("Testing tripleValue:");
16         double percent = 10;
17         System.out.println("Before: percent=" + percent);
```

¹Тестирование метода `tripleValue()`
До выполнения: `percent = 10.0`
В конце метода: `x=30.0`
После выполнения: `percent=10.0`


```
18     tripleValue(percent);
19     System.out.println("After: percent=" + percent);
20
21     /*
22      * Тест 2: методы могут изменять состояние объектов,
23      * передаваемых в качестве параметров
24      */
25     System.out.println("\nTesting tripleSalary:");
26     var harry = new Employee("Harry", 50000);
27     System.out.println("Before: salary="
28         + harry.getSalary());
29     tripleSalary(harry);
30     System.out.println("After: salary="
31         + harry.getSalary());
32
33     /*
34      * Тест 3: методы не могут присоединять новые
35      * объекты к объектным параметрам
36      */
37     System.out.println("\nTesting swap:");
38     var a = new Employee("Alice", 70000);
39     var b = new Employee("Bob", 60000);
40     System.out.println("Before: a=" + a.getName());
41     System.out.println("Before: b=" + b.getName());
42     swap(a, b);
43     System.out.println("After: a=" + a.getName());
44     System.out.println("After: b=" + b.getName());
45 }
46
47 public static void tripleValue(double x)
48     // не сработает!
49 {
50     x = 3 * x;
51     System.out.println("End of method: x=" + x);
52 }
53
54 public static void tripleSalary(Employee x)
55     // сработает!
56 {
57     x.raiseSalary(200);
58     System.out.println("End of method: salary="
59         + x.getSalary());
60 }
61
62 public static void swap(Employee x, Employee y)
63 {
64     Employee temp = x;
65     x = y;
66     y = temp;
67     System.out.println("End of method: x="
68         + x.getName());
69     System.out.println("End of method: y="
70         + y.getName());
71 }
72 }
73
```

```
74 class Employee // упрощенный класс Employee
75 {
76     private String name;
77     private double salary;
78
79     public Employee(String n, double s)
80     {
81         name = n;
82         salary = s;
83     }
84
85     public String getName()
86     {
87         return name;
88     }
89
90     public double getSalary()
91     {
92         return salary;
93     }
94
95     public void raiseSalary(double byPercent)
96     {
97         double raise = salary * byPercent / 100;
98         salary += raise;
99     }
100 }
```

4.6. Конструирование объектов

Ранее было показано, как писать простые конструкторы, определяющие начальные состояния объектов. Но конструирование объектов — очень важная операция, поэтому в Java предусмотрены самые разные механизмы написания конструкторов. Все эти механизмы рассматриваются ниже.

4.6.1. Перегрузка

У некоторых классов имеется не один конструктор. Например, пустой объект типа `StringBuilder` можно сконструировать (или, проще говоря, построить) следующим образом:

```
StringBuilder messages = new StringBuilder();
```

С другой стороны, исходную символьную строку можно указать таким образом:

```
StringBuilder todoList = new StringBuilder("To do:\n");
```

Оба способа конструирования объектов носят общее название *перегрузки*. О перегрузке говорят в том случае, если у нескольких методов (в данном случае нескольких конструкторов) имеются одинаковые имена, но разные параметры. Компилятор должен сам решить, какой метод вызвать, сравнивая типы параметров, определяемых при объявлении методов, с типами значений, указанных при вызове методов. Если ни один из методов не соответствует вызову или же если одному вызову одновременно соответствует несколько вариантов, возникает ошибка компиляции. (Этот процесс называется *разрешением перегрузки*.)



НА ЗАМЕТКУ! В языке Java можно перегрузить любой метод, а не только конструкторы. Следовательно, для того чтобы полностью описать метод, нужно указать его имя и типы параметров. Подобное написание называется *сигнатурой* метода. Например, в классе `String` имеются четыре открытых метода под названием `indexOf()`, которые имеют следующие сигнатуры:

```
indexOf(int)
indexOf(int, int)
indexOf(String)
indexOf(String, int)
```

Тип возвращаемого методом значения не входит в его сигнатуру. Следовательно, нельзя создать два метода, имеющих одинаковые имена и типы параметров и отличающихся лишь типом возвращаемого значения.

4.6.2. Инициализация полей по умолчанию

Если значение поля в конструкторе явно не задано, то ему автоматически присваивается значение по умолчанию: числам — нули; логическим переменным — логическое значение `false`; ссылкам на объект — пустое значение `null`. Но полагаться на действия по умолчанию не следует. Если поля инициализируются неявно, программа становится менее понятной.



НА ЗАМЕТКУ! Между полями и локальными переменными имеется существенное отличие. Локальные переменные всегда должны явно инициализироваться в методе. Но если поле не инициализируется в классе явно, то ему автоматически присваивается значение, задаваемое по умолчанию (`0`, `false` или `null`).

Рассмотрим в качестве примера класс `Employee`. Допустим, в конструкторе не задана инициализация значений некоторых полей. По умолчанию поле `salary` должно инициализироваться нулем, а поля `name` и `hireDay` — пустым значением `null`. Но при вызове метода `getName()` или `getHireDay()` пустая ссылка `null` может оказаться совершенно нежелательной, как показано ниже.

```
Date h = harry.getHireDay();
calendar.setTime(h); // Если h = null, генерируется исключение
```

4.6.3. Конструктор без аргументов

Многие классы содержат конструктор без аргументов, создающий объект, состояние которого устанавливается соответствующим образом по умолчанию. В качестве примера ниже приведен конструктор без аргументов для класса `Employee`.

```
public Employee()
{
    name = "";
    salary = 0;
    hireDay = new Date();
}
```

Если в классе совсем не определены конструкторы, то автоматически создается конструктор без аргументов. В этом конструкторе *всем* полям экземпляра присваиваются их значения, предусмотренные по умолчанию. Так, все числовые значения, содержащиеся в полях экземпляра, окажутся равными нулю, логические переменные — `false`, объектные переменные — `null`.

Если же в классе есть хотя бы один конструктор и явно не определен конструктор без аргументов, то создавать объекты, не предоставляя аргументы, нельзя. Например, у класса `Employee` из листинга 4.2 имеется один следующий конструктор:

```
Employee(String name, double salary, int y, int m, int d)
```

В этой версии данного класса нельзя создать объект, поля которого принимали бы значения по умолчанию. Иными словами, следующий вызов приведет к ошибке:

```
e = new Employee();
```



ВНИМАНИЕ! Следует иметь в виду, что конструктор без аргументов вызывается только в том случае, если в классе не определены другие конструкторы. Если же в классе имеется хотя бы один конструктор с параметрами и требуется создать экземпляр класса с помощью приведенного ниже выражения, следует явно определить конструктор без аргументов.

```
new ИмяКласса()
```

Разумеется, если значения по умолчанию во всех полях вполне устраивают, можно создать следующий конструктор без аргументов:

```
public ИмяКласса()  
{  
}
```

4.6.4. Явная инициализация полей

Конструкторы можно перегружать в классе, как и любые другие методы, а следовательно, задать начальное состояние полей его экземпляров можно несколькими способами. Каждое поле экземпляра следует всегда снабжать осмысленными значениями независимо от вызова конструктора.

В определении класса имеется возможность присвоить каждому полю соответствующее значение, как показано ниже.

```
class Employee  
{  
    ...  
    private String name = "";  
}
```

Это присваивание выполняется до вызова конструктора. Такой подход оказывается особенно полезным в тех случаях, когда требуется, чтобы поле имело конкретное значение независимо от вызова конструктора класса. При инициализации поля совсем не обязательно использовать константу. Ниже приведен пример, в котором поле инициализируется с помощью вызова метода.

```
class Employee  
{  
    private static int nextId;  
    private int id = assignId();  
    ...  
    private static int assignId()  
    {  
        int r = nextId;  
        nextId++;  
        return r;  
    }  
}
```

```

    }
    . . .
}

```



НА ЗАМЕТКУ C++! В языке C++ нельзя инициализировать поля экземпляра непосредственно в описании класса. Значения всех полей должны задаваться в конструкторе. Но в C++ имеется синтаксическая конструкция, называемая *списком инициализации*:

```

Employee::Employee(String n, double s, int y,
                    int m, int d) // C++
{
    : name(n),
      salary(s),
      hireDay(y, m, d)
}

```

Эта специальная синтаксическая конструкция служит в C++ для вызова конструкторов полей. А в Java поступать подобным образом нет никакой необходимости, поскольку объекты не могут содержать подобъекты, но разрешается иметь только ссылки на них.

4.6.5. Имена параметров

Создавая даже элементарный конструктор (а большинство из них таковыми и являются), трудно выбрать подходящие имена для его параметров. Обычно в качестве имен параметров служат отдельные буквы, как показано ниже.

```

public Employee(String n, double s)
{
    name = n;
    salary = s;
}

```

Но недостаток такого подхода заключается в том, что, читая программу, невозможно понять, что же означают параметры *n* и *s*. Некоторые программисты добавляют к осмысленным именам параметров префикс "a".

```

public Employee(String aName, double aSalary)
{
    name = aName;
    salary = aSalary
}

```

Такой код вполне понятен. Любой читающий его может сразу определить, в чем заключается смысл параметра. Имеется еще один широко распространенный прием. Чтобы воспользоваться им, следует знать, что параметры *скрывают* поля экземпляра с такими же именами. Так, если вызвать метод с параметром *salary*, то ссылка *salary* будет делаться на параметр, а не на поле экземпляра. Доступ к полю экземпляра осуществляется с помощью выражения *this.salary*. Напомним, что ключевое слово *this* обозначает неявный параметр, т.е. конструируемый объект, как демонстрируется в следующем примере:

```

public Employee(String name, double salary)
{
    this.name = name;
    this.salary = salary;
}

```



НА ЗАМЕТКУ C++! В языке C++ к именам полей экземпляра обычно добавляются префиксы, обозначаемые знаком подчеркивания или буквой (нередко для этой цели служит буква **m** или **x**). Например, поле, в котором хранится сумма зарплаты, может называться `_salary`, `mSalary` или `xSalary`. Программирующие на Java, как правило, так не поступают.

4.6.6. Вызов одного конструктора из другого

Ключевым словом `this` обозначается неявный параметр метода. Но у этого слова имеется еще одно назначение. Если *первый оператор* конструктора имеет вид `this(...)`, то вызывается другой конструктор этого же класса. Ниже приведен характерный тому пример.

```
public Employee(double s)
{
    // вызвать конструктор Employee(String, double)
    this("Employee " + nextId, s);
    nextId++;
}
```

Если выполняется операция `new Employee(60000)`, то конструктор `Employee(double)` вызывает конструктор `Employee(String, double)`. Применять ключевое слово `this` для вызова другого конструктора очень удобно — нужно лишь один раз написать общий код для конструирования объекта.



НА ЗАМЕТКУ C++! Ссылка `this` в Java сродни указателю `this` в C++. Но в C++ нельзя вызвать один конструктор из другого. Для того чтобы реализовать общий код инициализации объекта в C++, нужно создать отдельный метод.

4.6.7. Блоки инициализации

Ранее мы рассмотрели два способа инициализации поля:

- установка его значения в конструкторе;
- присваивание значения при объявлении.

На самом деле в Java существует еще и третий механизм: использование блока инициализации. Такой блок выполняется всякий раз, когда создается объект данного класса. Рассмотрим следующий пример кода:

```
class Employee
{
    private static int nextId;

    private int id;
    private String name;
    private double salary;

    // блок инициализации
    {
        id = nextId;
        nextId++;
    }

    public Employee(String n, double s)
```

```
{
    name = n;
    salary = s;
}

public Employee()
{
    name = "";
    salary = 0;
}

...
}
```

В этом примере начальное значение поля `id` задается в блоке инициализации объекта, причем неважно, какой именно конструктор используется для создания экземпляра класса. Блок инициализации выполняется первым, а вслед за ним — тело конструктора. Этот механизм совершенно не обязателен и обычно не применяется. Намного чаще применяются более понятные способы задания начальных значений полей.



НА ЗАМЕТКУ! В блоке инициализации допускается обращение к полям, определения которых находятся после данного блока. Несмотря на то что инициализация полей, определяемых после блока, формально допустима, поступать так не рекомендуется во избежание циклических определений. Конкретные правила изложены в разделе 8.3.2.3 спецификации Java (<http://docs.oracle.com/javase/specs>). Эти правила достаточно сложны, и учесть их в реализации компилятора крайне трудно. Так, в ранних версиях компилятора они были реализованы не без ошибок. Поэтому в исходном коде блоки инициализации рекомендуется размещать после определений полей.

При таком многообразии способов инициализации полей данных довольно трудно отследить все возможные пути процесса конструирования объектов. Поэтому рассмотрим подробнее те действия, которые происходят при вызове конструктора.

1. Если в первой строке кода одного конструктора вызывается другой конструктор, то этот последний конструктор выполняется с предоставляемыми аргументами.
2. В противном случае:
 - все поля инициализируются значениями, предусмотренными по умолчанию (0, false или null);
 - инициализаторы всех полей и блоки инициализации выполняются в порядке их следования в объявлении класса.
3. Выполняется тело конструктора.

Естественно, что код, отвечающий за инициализацию полей, нужно организовать так, чтобы в нем было легко разобраться. Например, было бы странным, если бы вызов конструкторов класса зависел от порядка объявления полей. Такой подход чреват ошибками.

Инициализировать статическое поле следует, задавая его начальное значение или используя статический блок инициализации. Первый механизм уже рассматривался ранее, а его пример приведен ниже.

```
static int nextId = 1;
```

Если для инициализации статических полей класса требуется сложный код, то удобнее использовать статический блок инициализации. Для этого следует разместить код в блоке и пометить его ключевым словом `static`. Допустим, идентификационные номера работников должны начинаться со случайного числа, не превышающего 10000. Соответствующий блок инициализации будет выглядеть следующим образом:

```
// Статический блок инициализации
static
{
    var generator = new Random();
    nextId = generator.nextInt(10000);
}
```

Статическая инициализация выполняется в том случае, если класс загружается впервые. Аналогично полям экземпляра, статические поля принимают значения 0, `false` или `null`, если не задать другие значения явным образом. Все операторы, задающие начальные значения статических полей, а также статические блоки инициализации выполняются в порядке их перечисления в объявлении класса.



НА ЗАМЕТКУ! Оказывается, что до версии JDK 6 элементарную программу, выводящую символьную строку `"Hello, World"`, можно было написать и без метода `main()`, как показано ниже.

```
public class Hello
{
    static
    {
        System.out.println("Hello, World!");
    }
}
```

При выполнении команды `java Hello` загружался класс `Hello`, статический блок инициализации выводил строку `"Hello, World!"` и лишь затем появлялось сообщение о том, что метод `main()` не определен. Но, начиная с версии Java 7, сначала выполняется проверка наличия в программе метода `main()`.

В программе, приведенной в листинге 4.5, наглядно демонстрируются многие языковые средства Java, обсуждавшиеся в этом разделе, включая следующие.

- Перегрузка конструкторов.
- Вызов одного конструктора из другого по ссылке `this(...)`.
- Применение конструктора без аргументов.
- Применение блока инициализации.
- Применение статической инициализации.
- Инициализация полей экземпляра.

Листинг 4.5. Исходный код из файла `ConstructorTest/ConstructorTest.java`

```
1 import java.util.*;
2
3 /**
4  * В этой программе демонстрируется
5  * конструирование объектов
```



```
6  * @version 1.02 2018-04-10
7  * @author Cay Horstmann
8  */
9  public class ConstructorTest
10 {
11     public static void main(String[] args)
12     {
13         // заполнить массив staff тремя
14         // объектами типа Employee
15         var staff = new Employee[3];
16
17         staff[0] = new Employee("Harry", 40000);
18         staff[1] = new Employee(60000);
19         staff[2] = new Employee();
20
21         // вывести данные обо всех объектах типа Employee
22         for (Employee e : staff)
23             System.out.println("name=" + e.getName() + ",id="
24                                 + e.getId() + ",salary="
25                                 + e.getSalary());
26     }
27 }
28
29 class Employee
30 {
31     private static int nextId;
32
33     private int id;
34     private String name = ""; // инициализация
35                               // поля экземпляра
36     private double salary;
37
38     // статический блок инициализации
39     static
40     {
41         var generator = new Random();
42         // задать произвольное число 0-999 в поле nextId
43         nextId = generator.nextInt(10000);
44     }
45
46     // блок инициализации объекта
47     {
48         id = nextId;
49         nextId++;
50     }
51
52     // три перегружаемых конструктора
53     public Employee(String n, double s)
54     {
55         name = n;
56         salary = s;
57     }
58
59     public Employee(double s)
60     {
61         // вызвать конструктор Employee(String, double)
62         this("Employee #" + nextId, s);
```

```
63 }
64
65 // конструктор без аргументов
66 public Employee()
67 {
68     // поле name инициализируется пустой строкой "" -
69     // поле salary не устанавливается явно, см. выше;
70     // а инициализируется нулем
71     // поле id инициализируется в блоке инициализации
72 }
73
74 public String getName()
75 {
76     return name;
77 }
78
79 public double getSalary()
80 {
81     return salary;
82 }
83
84 public int getId()
85 {
86     return id;
87 }
88 }
```

java.util.Random 1.0

- **Random()**
Создает новый генератор случайных чисел.
- **int nextInt(int n) 1.2**
Возвращает случайное число в пределах от 0 до $n - 1$.

4.6.8. Уничтожение объектов и метод **finalize()**

В некоторых объектно-ориентированных языках программирования и, в частности, в C++ имеются явные деструкторы, предназначенные для уничтожения объектов. Их основное назначение — освободить память, занятую объектами. А в Java реализован механизм автоматической сборки “мусора”, освобождать память вручную нет никакой необходимости, и поэтому в этом языке деструкторы отсутствуют.

Разумеется, некоторые объекты используют кроме памяти и другие ресурсы, например файлы, или оперируют другими объектами, которые, в свою очередь, обращаются к системным ресурсам. В этом случае очень важно, чтобы занимаемые ресурсы освобождались, когда они больше не нужны.

Если ресурс должен быть освобожден сразу после его использования, в таком случае придется написать соответствующий код самостоятельно. С этой целью следует предоставить метод `close()`, выполняющий необходимые операции по очистке памяти, вызвав его, когда соответствующий объект больше не нужен. В главе 7 будет показано, как осуществляется автоматический вызов такого метода.

Если же можно подождать до окончания срока действия виртуальной машины, то с помощью метода `Runtime.addShutdownHook()` можно ввести “перехватчик завершения”. Начиная с версии Java 9, можно также воспользоваться классом `Cleaner`, чтобы зарегистрировать действие, выполняемое, когда объект больше недоступен, разве что для его очистки из оперативной памяти. Подобные ситуации нередко возникают на практике. Подробнее об этих двух способах освобождения ненужных объектов из оперативной памяти см. в документации на прикладной интерфейс Java API.



ВНИМАНИЕ! Ни в коем случае не пользуйтесь методом `finalize()` для освобождения ненужных объектов из оперативной памяти. Этот метод предназначался раньше для вызова непосредственно перед удалением объекта сборщиком “мусора”. Но дело в том, что вы не можете знать заранее, когда именно следует вызывать данный метод. Кроме того, он больше не рекомендован к применению в прикладных программах на Java.

4.7. Пакеты

Язык Java позволяет объединять классы в наборы, называемые *пакетами*. Пакеты облегчают организацию работы и позволяют отделить классы, созданные одним разработчиком, от классов, разработанных другими. В последующих разделах поясняется, как создавать пакеты и пользоваться ими.

4.7.1. Именованние пакетов

Пакеты служат в основном для обеспечения однозначности имен классов. Допустим, двух программистов осенила блестящая идея создать класс `Employee`. Если оба класса будут находиться в разных пакетах, конфликта имен не возникнет. Чтобы обеспечить абсолютную однозначность имени пакета, рекомендуется использовать доменное имя компании в Интернете, записанное в обратном порядке (оно по определению единственное в своем роде). В составе пакета можно создавать подпакеты и использовать их в разных проектах. Рассмотрим в качестве примера домен `horstmann.com`, зарегистрированный автором данной книги. Записав это имя в обратном порядке, можно использовать его как название пакета — `com.horstmann`. К этому имени можно затем присоединить наименование проекта, например `com.horstmann.corejava`. Если в дальнейшем вести класс `Employee` в данный пакет, то его полностью уточненное имя окажется следующим: `com.horstmann.corejava.Employee`.



НА ЗАМЕТКУ! С точки зрения компилятора между вложенными пакетами отсутствует какая-либо связь. Например, пакеты `java.util` и `java.util.jar` вообще не связаны друг с другом. Каждый из них представляет собой независимое собрание классов.

4.7.2. Импорт классов

В классе могут использоваться все классы из собственного пакета и все *открытые* классы из других пакетов. Доступ к классам из других пакетов можно получить двумя способами. Во-первых, можно указать *полностью уточненное имя*, т.е. имя пакета и класса:

```
java.time.LocalDate today = java.time.LocalDate.now();
```

Очевидно, что такой способ не совсем удобен. Второй, более простой и распространенный способ предусматривает применение ключевого слова `import`. В этом случае имя пакета указывать перед именем класса необязательно.

Импортировать можно как один конкретный класс, так и пакет в целом. Операторы `import` следует разместить в начале исходного файла (после всех операторов `package`). Например, все классы из пакета `java.time` можно импортировать следующим образом:

```
import java.time.*;
```

После этого имя пакета не указывается, как показано ниже.

```
LocalDate today = LocalDate.now();
```

Отдельный класс можно также импортировать из пакета следующим образом:

```
import java.time.LocalDate;
```

Проще импортировать все классы, например, из пакета `java.time.*`. На объем кода это не оказывает никакого влияния. Но если указать импортируемый класс явным образом, то читающему исходный код программы станет ясно, какие именно классы будут в ней использоваться.



СОВЕТ. Работая в IDE Eclipse, можно выбрать команду меню `Source⇒Organize Imports` (Исходный код⇒Организовать импорт). В итоге выражения типа `import java.util.*;` будут автоматически преобразованы в последовательности строк, предназначенных для импорта отдельных классов:

```
import java.util.ArrayList;
import java.util.Date;
```

Такая возможность очень удобна при написании кода.

Следует, однако, иметь в виду, что оператор `import` со звездочкой можно применять для импорта только одного пакета. Но нельзя использовать оператор `import java.*` или `import java.*.*`, чтобы импортировать все пакеты, имена которых содержат префикс `java`. В большинстве случаев импортируется весь пакет, независимо от его размера. Единственный случай, когда следует обратить особое внимание на пакет, возникает при конфликте имен. Например, оба пакета, `java.util` и `java.sql`, содержат класс `Date`. Допустим, разрабатывается программа, в которой оба эти пакета импортируются следующим образом:

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

Если теперь попытаться воспользоваться классом `Date`, то возникнет ошибка компиляции, как показано ниже.

```
Date today; // ОШИБКА: неясно, какой именно выбрать пакет:
              // java.util.Date или java.sql.Date?
```

Компилятор не в состоянии определить, какой именно класс `Date` требуется в программе. Разрешить это затруднение можно, добавив уточняющий оператор `import` следующим образом:

```
import java.util.*;
import java.sql.*;
import java.util.Date;
```

А что, если на самом деле нужны оба класса `Date`? В таком случае нужно указать полное имя пакета перед именем каждого класса:

```
var deadline = new java.util.Date();
var today = new java.sql.Date(...);
```

Обнаружение классов в пакетах входит в обязанности компилятора. В байт-кодах, находящихся в файлах классов, всегда содержатся полные имена пакетов.



НА ЗАМЕТКУ C++! Программирующие на C++ считают, что оператор `import` является аналогом директивы `#include`. Но у них нет ничего общего. В языке C++ директиву `#include` приходится применять в объявлениях внешних ресурсов потому, что компилятор C++ не просматривает файлы, кроме компилируемого, а также файлы, указанные в самой директиве `#include`. А компилятор Java просматривает содержимое всех файлов, при условии, если известно, где их искать. В языке Java можно и не применять механизм импорта, явно называя все пакеты, например `java.util.Date`. А в C++ обойтись без директивы `#include` нельзя.

Единственное преимущество оператора `import` заключается в его удобстве. Он позволяет использовать более короткие имена классов, не указывая полное имя пакета. Например, после оператора `import java.util.*` (или `import java.util.Date`) к классу `java.util.Date` можно обращаться по имени `Date`.

Аналогичный механизм работы с пакетами в C++ реализован в виде директивы `namespace`. Операторы `package` и `import` в Java можно считать аналогами директив `namespace` и `using` в C++.

4.7.3. Статический импорт

Имеется форма оператора `import`, позволяющая импортировать не только классы, но и статические методы и поля. Допустим, в начале исходного файла введена следующая строка кода:

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

Это позволит использовать статические методы и поля, определенные в классе `System`, не указывая имени класса:

```
out.println("Goodbye, World!"); // вместо System.out
exit(0); // вместо System.exit
```

Статические методы или поля можно также импортировать явным образом:

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

Но в практике программирования на Java такие выражения, как `System.out` или `System.exit`, обычно не сокращаются. Ведь в этом случае исходный код станет более трудным для восприятия. С другой стороны, следующая строка кода:

```
sqrtpow(x, 2) + pow(y, 2))
```

выглядит более ясной, чем такая строка:

```
Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2))
```

4.7.4. Ввод классов в пакеты

Чтобы ввести класс в пакет, следует указать имя пакета в начале исходного файла *перед* определением класса. Например, исходный файл `Employee.java` из листинга 4.7 начинается следующими строками кода:

```
package com.horstmann.corejava;
```

```
public class Employee
{
    ...
}
```

Если оператор `package` в исходном файле не указан, то классы, описанные в этом файле, вводятся в *безымянный пакет*. У безымянного пакета нет имени. Все рассмотренные до сих пор классы принадлежали безымянному пакету.

Пакеты следует размещать в подкаталоге, путь к которому соответствует полному имени пакета. Например, все файлы классов из пакета `com.horstmann.corejava` должны находиться в подкаталоге `com/horstmann/corejava` (или `com\horstmann\corejava` в Windows). Компилятор размещает файлы классов в той же самой структуре каталогов.

Программы из листингов 4.6 и 4.7 распределены по двум пакетам: класс `PackageTest` принадлежит пакету по умолчанию, а класс `Employee` — пакету `com.horstmann.corejava`. Следовательно, файл `Employee.class` должен находиться в подкаталоге `com/horstmann/corejava`. Иными словами, структура каталогов должна выглядеть следующим образом:

```
.(базовый каталог)
|__ PackageTest.java
|__ PackageTest.class
|__ com/
|   |__ horstmann/
|       |__ corejava/
|           |__ Employee.java
|           |__ Employee.class
```

Чтобы скомпилировать программу из листинга 4.6, перейдите в каталог, содержащий файл `PackageTest.java`, и выполните следующую команду:

```
javac Package.java
```

Компилятор автоматически найдет файл `com/horstmann/corejava/Employee.java` и скомпилирует его.

Рассмотрим более практический пример, в котором пакет по умолчанию не используется. Вместо этого классы распределены по разным пакетам (`com.horstmann.corejava` и `com.mycompany`), как показано ниже.

```
.(базовый каталог)
|__ com/
|   |__ horstmann/
|       |__ corejava/
|           |__ Employee.java
|           |__ Employee.class
|   |__ mycompany/
|       |__ PayrollApp.java
|       |__ PayrollApp.class
```

И в этом случае классы следует компилировать и запускать из *базового каталога*, т.е. того, в котором содержится подкаталог `com`:

```
javac com/mycompany/PayrollApp.java
java com.mycompany.PayrollApp
```

Не следует, однако, забывать, что компилятор работает с *файлами* (при указании имени файла задаются путь и расширение `.java`). А интерпретатор Java оперирует *классами* (имя каждого класса указывается в пакете через точку).



СОВЕТ. Начиная со следующей главы в исходном коде приводимых примеров программ будут использоваться пакеты. Это даст возможность создавать проекты в IDE по отдельным главам, а не по разделам.



ВНИМАНИЕ! Компилятор не проверяет структуру каталогов. Допустим, исходный файл начинается со следующей директивы:

package com.шусомрану;

Этот файл можно скомпилировать, даже если он не находится в каталоге `com/шусомрану`. Исходный файл будет скомпилирован без ошибок, если он не зависит от других пакетов. Но при попытке выполнить скомпилированную программу виртуальная машина не найдет нужные классы, если пакеты не соответствуют указанным каталогам, а все файлы классов не размещены в нужном месте.

Листинг 4.6. Исходный код из файла `PackageTest/PackageTest.java`

```
1 import com.horstmann.corejava.*;
2 // В этом пакете определен класс Employee
3
4 import static java.lang.System.*;
5 /**
6  * В этой программе демонстрируется применение пакетов
7  * @version 1.11 2004-02-19
8  * @author Cay Horstmann
9  */
10 public class PackageTest
11 {
12     public static void main(String[] args)
13     {
14         // здесь не нужно указывать полное имя
15         // класса com.horstmann.corejava.Employee
16         // поскольку используется оператор import
17         Employee harry = new Employee("Harry Hacker",
18                                     50000, 1989, 10, 1);
19
20         harry.raiseSalary(5);
21
22         // здесь не нужно указывать полное имя System.out,
23         // поскольку используется оператор static import
24         out.println("name=" + harry.getName() + ", salary="
25                  + harry.getSalary());
26     }
27 }
```

Листинг 4.7. Исходный код из файла `PackageTest/com/horstmann/corejava/Employee.java`

```
1 package com.horstmann.corejava;
2
3 // классы из этого файла входят в указанный пакет
4
5 import java.time.*;
6
7 // операторы import следуют после оператора package
8
9 /**
10  * @version 1.11 2015-05-08
11  * @author Cay Horstmann
```

```
12 */
13 public class Employee
14 {
15     private String name;
16     private double salary;
17     private LocalDate hireDay;
18
19     public Employee(
20         String name, double salary, int year,
21         int month, int day)
22     {
23         this.name = name;
24         this.salary = salary;
25         hireDay = LocalDate.of(year, month, day);
26     }
27
28     public String getName()
29     {
30         return name;
31     }
32
33     public double getSalary()
34     {
35         return salary;
36     }
37
38     public LocalDate getHireDay()
39     {
40         return hireDay;
41     }
42
43     public void raiseSalary(double byPercent)
44     {
45         double raise = salary * byPercent / 100;
46         salary += raise;
47     }
48 }
```

4.7.5. Область видимости пакетов

В приведенных ранее примерах кода уже встречались модификаторы доступа `public` и `private`. Открытые компоненты, помеченные ключевым словом `public`, могут использоваться любым классом. А закрытые компоненты, в объявлении которых указано ключевое слово `private`, могут использоваться только тем классом, в котором они были определены. Если же ни один из модификаторов доступа не указан, то компонент программы (класс, метод или переменная) доступен всем методам в том же самом *пакете*.

Обратимся снова к примеру программы из листинга 4.2. Класс `Employee` не определен в ней как открытый. Следовательно, любой другой класс из того же самого пакета (в данном случае — пакета по умолчанию), например класс `EmployeeTest`, может получить к нему доступ. Для классов такой подход следует признать вполне разумным. Но для переменных подобный способ доступа не годится. Переменные должны быть явно обозначены как `private`, иначе их область видимости будет

по умолчанию расширена до пределов пакета, что, безусловно, нарушает принцип инкапсуляции. В процессе работы над программой разработчики часто забывают указать ключевое слово `private`. Ниже приведен пример из класса `Window`, принадлежащего пакету `java.awt`, который входит в состав комплекта JDK.

```
public class Window extends Container
{
    String warningString;
    . . .
}
```

Обратите внимание на то, что у переменной `warningString` отсутствует модификатор доступа `private`! Это означает, что методы всех классов из пакета `java.awt` могут обращаться к ней и изменить ее значение, например, присвоить ей строку "Trust me!" (Доверьтесь мне!). Фактически все методы, имеющие доступ к переменной `warningString`, принадлежат классу `Window`, поэтому эту переменную можно было бы смело объявить как закрытую. Вероятнее всего, те, кто писал этот код, торопились и просто забыли указать ключевое слово `private`. (Не станем упоминать имени автора этого кода, чтобы не искать виноватого, — вы сами можете заглянуть в исходный код.) Но факт тот, что больше двадцати лет спустя переменная `warningString` по-прежнему остается незакрытой. Помимо этого, в данный класс были со временем введены новые поля, и половина из них вообще не объявлена закрытыми.

Это может вызвать затруднения. По умолчанию пакеты не являются закрытыми, а это означает, что всякий может добавлять в пакеты свои классы. Разумеется, злонамеренные или невежественные программисты могут написать код, модифицирующий переменные, область видимости которых ограничивается пакетом. Например, в ранних версиях Java можно было легко проникнуть в другой класс пакета `java.awt`. Для этого достаточно было начать определение нового класса со следующей строки:

```
package java.awt;
```

а затем разместить полученный файл класса в подкаталоге `java/awt`. В итоге содержимое пакета `java.awt` становилось открытым для доступа. Подобным ловким способом можно было изменить строку предупреждения, отображаемую на экране (рис. 4.9).

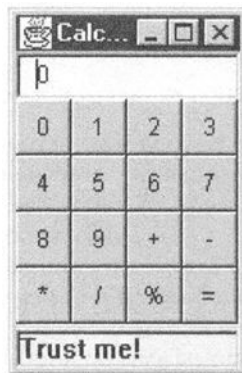


Рис. 4.9. Изменение строки предупреждения в окне апплета

В версии 1.2 создатели JDK запретили загрузку классов, определенных пользователем, если их имя начиналось с "java. ". Разумеется, эта защита не распространяется

на ваши собственные классы. Вместо этого вы можете воспользоваться другим механизмом, который называется *герметизацией пакета* и ограничивает доступ к пакету. Произведя герметизацию пакета, вы запрещаете добавлять в него классы. В главе 9 второго тома настоящего издания будет показано, как создать архивный JAR-файл, содержащий герметичные пакеты.

4.7.6. Путь к классам

Как вам должно быть уже известно, классы хранятся в подкаталогах файловой системы. Путь к классу должен совпадать с именем пакета. Кроме того, можно воспользоваться утилитой `jar`, чтобы разместить классы в архивном файле формата JAR (архиве Java; в дальнейшем — просто JAR-файле). В одном архивном файле многие файлы классов и подкаталогов находятся в сжатом виде, что позволяет экономить память и сокращать время доступа к ним. Когда вы пользуетесь библиотекой от независимых разработчиков, то обычно получаете в свое распоряжение один или несколько JAR-файлов. О том, как создавать архивные JAR-файлы собственными силами, речь пойдет в главе 11.



СОВЕТ. Для организации файлов и подкаталогов в архивных JAR-файлах используется формат ZIP. Манипулировать архивными JAR-файлами можно с помощью любой утилиты, поддерживающей формат ZIP.

Чтобы обеспечить совместный доступ программ к классам, выполните следующие действия.

1. Разместите файлы классов в одном или нескольких специальных каталогах, например `/home/user/classdir`. Следует иметь в виду, что этот каталог является *базовым* по отношению к дереву пакета. Если потребуется добавить класс `com.horstmann.corejava.Employee`, то файл класса следует разместить в подкаталоге `/home/user/classdir/com/horstmann/corejava`.
2. Разместите все JAR-файлы в одном каталоге, например `/home/user/archives`.
3. Задайте путь к классу. *Путь к классу* — это совокупность всех базовых каталогов, которые могут содержать файлы классов.

В Unix составляющие пути к классам отделяются друг от друга двоеточиями:

```
/home/user/classdir:./home/user/archives/archive.jar
```

А в Windows они разделяются точками с запятой:

```
c:\clasdir;.;c:\archives\archive.jar
```

В обоих случаях точка обозначает текущий каталог.

Путь к классу содержит:

- имя базового каталога `/home/user/classdir` или `c:\classes`;
- обозначение текущего каталога `(.)`;
- имя JAR-файла `/home/user/archives/archive.jar` или `c:\archives\archive.jar`.

Начиная с версии Java 6, для обозначения каталога с архивными JAR-файлами можно указывать метасимволы подстановки одним из следующих способов:

```
/home/user/classdir:./home/user/archives/'*'
```

или

```
c:\classdir;.;c:\archives\*
```

В UNIX метасимвол подстановки `*` должен быть экранирован, чтобы исключить его интерпретацию в командной оболочке. Все архивные JAR-файлы (но не файлы с расширением `.class`), находящиеся в каталоге `archives`, включаются в путь к классам.

Файлы из прикладного интерфейса Java API всегда участвуют в поиске классов, поэтому их не стоит включать явным образом в путь к классам.



ВНИМАНИЕ! Компилятор `javac` всегда ищет файлы в текущем каталоге, а загрузчик виртуальной машины `java` обращается к текущему каталогу только в том случае, если в пути к классам указана точка `(.)`. Если путь к классам не указан, никаких осложнений не возникает — по умолчанию в него включается текущий каталог `(.)`. Если же вы задали путь к классам и забыли указать точку, ваша программа будет скомпилирована без ошибок, но выполняться не будет.

В пути к классам перечисляются все каталоги и архивные файлы, которые служат исходными точками для поиска классов. Рассмотрим следующий простой путь к классам:

```
/home/user/classdir:./home/user/archives/archive.jar
```

Допустим, интерпретатор осуществляет поиск файла, содержащего класс `com.horstmann.corejava.Employee`. Сначала он ищет его в файлах классов из прикладного интерфейса Java API. В этих файлах искомый класс отсутствует, поэтому интерпретатор анализирует пути к классам, по которым он осуществляет поиск следующих файлов:

- `/home/user/classdir/com/horstmann/corejava/Employee.class`;
- `com.horstmann/corejava/Employee.class`, начиная с текущего каталога;
- `com.horstmann/corejava/Employee.class` в каталоге `/home/user/archives/archive.jar`.

На поиск файлов компилятор затрачивает больше времени, чем виртуальная машина. Если вы указали класс, не назвав пакета, которому он принадлежит, компилятор сначала должен сам определить, какой именно пакет содержит данный класс. В качестве возможных источников для классов рассматриваются пакеты, указанные в операторах `import`. Допустим, исходный файл содержит приведенные ниже директивы, а также код, в котором происходит обращение к классу `Employee`.

```
import java.util.*;
import com.horstmann.corejava.*;
```

Компилятор попытается найти классы `java.lang.Employee` (поскольку пакет `java.lang` всегда импортируется по умолчанию), `java.util.Employee`, `com.horstmann.corejava.Employee` и `Employee` в текущем пакете. Он ищет каждый из этих файлов во всех каталогах, указанных в пути к классам. Если найдено несколько таких классов, возникает ошибка компиляции. (Классы должны быть определены однозначно, поэтому порядок следования операторов `import` особого значения не имеет.)

Компилятор делает еще один шаг, просматривая исходные файлы, чтобы проверить, были ли они созданы позже, чем файлы классов. Если проверка дает положительный результат, исходный файл автоматически перекомпилируется. Напомним, что из других пакетов можно импортировать лишь открытые классы. Исходный файл

может содержать только один открытый класс, а кроме того, имена файла и класса должны совпадать. Следовательно, компилятор может легко определить, где находятся исходные файлы, содержащие открытые классы. Из текущего пакета можно импортировать также классы, не определенные как открытые (т.е. `public`). Исходные коды классов могут содержаться в файлах с разными именами. При импорте класса из текущего пакета компилятор проверяет *все* исходные файлы, чтобы выяснить, в каком из них определен требуемый класс.

4.7.7. Указание пути к классам

Путь к классам лучше всего указать с помощью параметра `-classpath` (`-cp` или `--class-path`, начиная с версии Java 9) следующим образом:

```
java -classpath /home/user/classdir:.  
      /home/user/archives/archive.jar MyProg
```

или

```
java -classpath c:\classdir:;  
      c:\archives\archive.jar MyProg
```

Вся команда должна быть набрана в одной строке. Лучше всего ввести такую длинную команду в сценарий командной оболочки или командный файл. Применение параметра `-classpath` считается более предпочтительным способом установки путей к классам. Альтернативой этому способу служит установка переменной окружения `CLASSPATH`. А далее все зависит от конкретной командной оболочки. Так, в командной оболочке Bourne Again (bash) для установки путей к классам используется следующая команда:

```
export CLASSPATH=/home/user/classdir:./home/user/archives/archive.jar
```

В командной оболочке C shell для этой цели применяется следующая команда:

```
setenv CLASSPATH /home/user/classdir:./home/user/archives/archive.jar
```

а в командной строке Windows — такая:

```
set CLASSPATH=c:\classdir;.;c:\archives\archive.jar
```

Путь к классам действителен до завершения работы командной оболочки.



ВНИМАНИЕ! Некоторые советуют устанавливать переменную окружения `CLASSPATH` на постоянной основе. В целом это неудачная идея. Сначала разработчики забывают о глобальных установках, а затем удивляются, когда классы не загружаются правильно. Характерным тому примером служит программа установки приложения QuickTime от компании Apple в Windows. Эта программа глобально устанавливает переменную окружения `CLASSPATH` таким образом, чтобы она указывала на нужный ей архивный JAR-файл, не включая в путь текущий каталог. В итоге очень многие программирующие на Java попадают в тупиковую ситуацию, когда их программы сначала компилируются, а затем не запускаются.



ВНИМАНИЕ! В прошлом некоторые рекомендовали вообще обходиться без пути к классам, опустив все архивные JAR-файлы в каталоге `java/lib/ext`. Начиная с версии Java 9, этот механизм устарел, хотя и раньше он всегда считался плохим советом. Ведь очень легко было запутаться, когда давно забытые классы загружались из каталога расширений.



НА ЗАМЕТКУ! Начиная с версии Java 9, классы можно также загружать по пути к модулю. Более подробно модули рассматриваются в главе 9 второго тома настоящего издания.

4.8. Архивные JAR-файлы

Приложение обычно упаковывается для того, чтобы предоставить в распоряжение пользователя единственный файл, а не целую структуру каталогов, заполненную файлами классов. Специально для этой цели был разработан формат архивных файлов Java Archive (JAR). Файл формата JAR (в дальнейшем просто JAR-файл) может содержать, помимо файлов классов, файлы других типов, в том числе файлы изображений и звука. Более того, архивные JAR-файлы уплотняются по широко известному алгоритму сжатия данных в формате ZIP.

4.8.1. Создание JAR-файлов

Для создания JAR-файлов служит утилита `jar`. (При установке JDK по умолчанию утилита `jar` располагается в каталоге `jdk/bin`.) Новый JAR-файл создается с помощью следующего синтаксиса командной строки:

```
jar cvf имяJAR-файла файл_1 файл_2 ...
```

Например:

```
jar cvf CalculatorClasses.jar *.java icon.gif
```

Ниже приведена общая форма команды `jar`.

```
jar параметры файл_1 файл_2 ...
```

В табл. 4.2 перечислены все параметры утилиты `jar`. Они напоминают параметры команды `tar`, хорошо известной пользователям операционной системы Unix.

В архивные JAR-файлы можно упаковывать прикладные программы, программные компоненты, а также библиотеки кода. Так, если в прикладной программе на Java требуется отправить сообщение по электронной почте, для этой цели можно воспользоваться библиотекой, упакованной в файле `javax.mail.jar`.

Таблица 4.2. Параметры утилиты `jar`

Параметр	Описание
<code>c</code>	Создает новый или пустой архив и добавляет в него файлы. Если в качестве имени файла указано имя каталога, утилита <code>jar</code> обрабатывает его рекурсивно
<code>C</code>	Временно изменяет каталог. Например, следующая команда направляет файлы в подкаталог <code>classes</code> : <code>jar cvf JARFileName.jar -C classes *.class</code>
<code>e</code>	Создает точку входа в манифест (см. далее раздел 4.8.3)
<code>f</code>	Задает имя JAR-файла в качестве второго параметра командной строки. Если этот параметр пропущен, то утилита <code>jar</code> выводит результат в стандартный поток вывода (при создании JAR-файла) или вводит его из стандартного потока ввода (при извлечении или просмотре содержимого JAR-файла)
<code>i</code>	Создает индексный файл (для ускорения поиска в крупных архивах)

Окончание табл. 4.2

Параметр	Описание
m	Добавляет в JAR-файл манифест, представляющий собой описание содержимого архива и его происхождения. Манифест создается по умолчанию для каждого архива, но для подробного описания содержимого JAR-файла можно создать свой собственный манифест
M	Отменяет создание манифеста
t	Отображает содержание архива
u	Обновляет существующий JAR-файл
v	Выводит подробные сведения об архиве
x	Извлекает файлы. Если указано несколько имен файлов, извлекаются только они. В противном случае извлекаются все файлы
0	Сохраняет данные в архиве, не упаковывая их в формате ZIP

4.8.2. Файл манифеста

Помимо файлов классов, изображений и прочих ресурсов, каждый архивный JAR-файл содержит также *файл манифеста*, описывающий особые характеристики данного архива. Файл манифеста называется `MANIFEST.MF` и находится в специальном подкаталоге `META-INF`. Минимально допустимый манифест не особенно интересен, как показано ниже.

```
Manifest-Version: 1.0
```

Сложные манифесты содержат намного больше элементов описания, группируемых по разделам. Первый раздел манифеста называется *главным* и относится ко всему JAR-файлу. Остальные элементы описания относятся к отдельным файлам, пакетам или URL. Эти элементы должны начинаться со слова `Name`. Разделы отделяются друг от друга пустыми строками, как показано в приведенном ниже примере.

```
Manifest-Version: 1.0
```

строки описания данного архива

```
Name: Woozle.class
```

строки описания данного архива

```
Name: com/мусcompany/мурpkg/
```

строки описания данного архива

Чтобы отредактировать этот манифест, достаточно ввести в него нужные строки, сохранив их в обычном текстовом файле, а затем выполнить следующую команду:

```
jar cfm имяJAR-файла имяфайлаМанифеста
```

Например, для того чтобы создать новый JAR-файл с манифестом, нужно вызвать утилиту `jar` из командной строки следующим образом:

```
jar cfm MyArchive.jar manifest.mf
com/мусcompany/мурpkg/*.class
```

Чтобы добавить в манифест существующего JAR-файла новые строки, достаточно ввести и сохранить их в текстовом файле, а затем выполнить следующую команду:

```
jar ufm MyArchive.jar manifest-additions.mf
```



НА ЗАМЕТКУ! Подробнее о форматах архивных JAR-файлов, а также файлов манифестов можно узнать по адресу <https://docs.oracle.com/javase/10/docs/specs/jar/jar.html>.

4.8.3. Исполняемые JAR-файлы

С помощью параметра **e** утилиты **jar** можно указать *точку входа* в прикладную программу, т.е. класс, который обычно указывается при запуске программы по команде **java**:

```
jar cvfe MyProgram.jar com.mycompany.mypkg.MainAppClass добавляемые файлы
```

С другой стороны, в манифесте можно указать *главный класс* прикладной программы, включая оператор в следующей форме:

```
Main-Class: com.mycompany.mypkg.MainAppClass
```

Только не добавляйте расширение **.class** к имени главного класса. Независимо от способа указания точки входа в прикладную программу ее пользователи могут запустить ее на выполнение, введя следующую команду:

```
java -jar MyProgram.jar
```



ВНИМАНИЕ! Последняя строка в манифесте должна оканчиваться символом перевода строки. В противном случае манифест не будет прочитан правильно. Весьма распространена ошибка, когда создается текстовый файл с единственной строкой **Main-Class**, но без ограничителя строки.

В зависимости от конфигурации операционной системы приложения можно запускать двойным щелчком на пиктограмме JAR-файла. Ниже описываются особенности запуска приложений из JAR-файлов в разных операционных системах.

- В Windows установщик исполняющей системы Java образует сопоставление файлов с расширением **.jar** для запуска подобных файлов по команде **javaw -jar**. (В отличие от команды **java**, команда **javaw** не открывает окно командной оболочки.)
- В Mac OS X операционная система распознает расширение файла **.jar** и выполняет программу на Java после двойного щелчка на JAR-файле.

Но программа на Java, находящаяся в архивном JAR-файле, имеет тот же вид, что и собственное приложение операционной системы. В Windows можно использовать утилиты-оболочки сторонних производителей, превращающие JAR-файлы в исполняемые файлы Windows. Такая оболочка представляет собой программу для Windows с известным расширением **.exe**, которая запускает виртуальную машину Java (JVM) или же сообщает пользователю, что делать, если эта машина не найдена. Для запуска прикладных программ из JAR-файлов имеется немало коммерческих и свободно доступных программных продуктов вроде Launch4J (<http://launch4j.sourceforge.net>) и IzPack (<http://izpack.org>).

4.8.4. Многоверсионные архивные JAR-файлы

С внедрением модулей и строгой инкапсуляции пакетов некоторые доступные ранее внутренние прикладные интерфейсы API становятся недоступными. Например, в библиотеке JavaFX 8 имелся внутренний класс `com.sun.javafx.css.CssParser`.

Если вы пользовались раньше этим классом для синтаксического анализа стилевых таблиц в своей прикладной программе, то обнаружите, что она больше не компилируется. В качестве выхода из этого затруднения достаточно перейти к классу `javafx.css.CssParser`, доступному в версии Java 9. Но теперь возникает другое затруднение, поскольку придется распространять другие приложения для версий Java 8 и 9 или принять специальные меры, включая загрузку классов или рефлексии.

Для разрешения подобных затруднений в версии Java 9 были внедрены *многоверсионные архивные JAR-файлы*, которые могут содержать файлы классов для разных версий Java. Ради обратной совместимости дополнительные файлы классов размещаются в каталоге `META-INF/versions`, как показано ниже.

```
Application.class
BuildingBlocks.class
Util.class
META-INF
|__MANIFEST.MF (со строкой Multi-Release: true)
|__versions
|__9
|__|__Application.class
|__|__BuildingBlocks.class
|__10
|__|__BuildingBlocks.class
```

Допустим, в классе `Application` применяется класс `CssParser`. В таком случае файл устаревшего класса `Application.class` может быть скомпилирован для применения класса `com.sun.javafx.css.CssParser`, а в версии Java 9 — класса `javafx.css.CssParser`. В версии Java 8 ничего неизвестно о каталоге `META-INF/versions`, и поэтому будут загружены устаревшие версии классов. А когда архивный JAR-файл читается в версии Java 9, то вместо устаревших будут использоваться новые версии классов.

Чтобы ввести привязанные к версии файлы классов, достаточно указать параметр **--release**, как показано ниже

```
jar uf MyProgram.jar --release 9 Application.class
```

А для того чтобы создать многоверсионный архивный JAR-файл, достаточно указать параметр **-C** и переход к отдельному каталогу с файлами классов для каждой версии:

```
jar cf MyProgram.jar -C bin/8 . --release 9 -C bin/9 Application.class
```

Когда прикладная программа компилируется для разных версий, выходной каталог указывается с помощью параметров **--release** и **-d**, как представлено ниже. Начиная с версии Java 9, параметр **-d** позволяет создать каталог, если таковой отсутствует.

```
javac -d bin/8 --release 8 . . .
```

Параметр **--release** также является новым для версии Java 9. А в прежних версиях Java вместо него приходилось пользоваться параметрами **-source**, **-target** и **-bootclasspath**. Теперь же в состав JDK входят символичные файлы для двух предыдущих версий прикладного интерфейса API. Так, прикладную программу можно скомпилировать в версии Java 9, указав символ версии 9, 8 или 7 после параметра **--release** в командной строке.

Многоверсионные архивные JAR-файлы не предназначены для разных версий прикладной программы или библиотеки. Открытый для всех классов прикладной интерфейс API должен быть одинаковым для разных версий. Единственное назначение многоверсионных архивных JAR-файлов — разрешить нормальное взаимодействие конкретной версии прикладной программы или библиотеки с разными версиями комплекта JDK. Если внедрить новые или изменить прежние функциональные средства в прикладном интерфейсе API, то вместо многоверсионного архивного JAR-файла придется предоставить новую версию обычного архивного JAR-файла.



НА ЗАМЕТКУ! Такие инструментальные средства, как утилита `javap`, не модернизированы для обработки многоверсионных архивных JAR-файлов. Так, если выполнить следующую команду:

```
javap -classpath MyProgram.jar Application.class
```

в конечном счете получится базовая версия класса с таким же, как предполагается, открытым прикладным интерфейсом API, как и у новой версии. А если требуется проверить новую версию, придется выполнить команду

```
javap -classpath MyProgram.jar\!/META-INF/versions/9/Application.class
```

4.8.5. Примечание к параметрам командной строки

По традиции параметры командной строки в комплекте Java Development Kit обозначаются одним знаком дефиса и несколькими буквами, как показано ниже.

```
java -jar . . .
javac -Xlint:unchecked -classpath . . .
```

Исключение из этого правила сделано для команды `jar`, после которой параметры можно указывать без дефиса в формате упоминавшейся ранее команды `tar`, как демонстрируется в следующем примере:

```
jar cvf . . .
```

Начиная с версии Java 9, инструментальные средства командной строки переведены на более распространенный формат, где многобуквенные имена параметров предваряются двумя дефисами, а однобуквенные имена параметров — одним дефисом. Например, команда `ls` может быть вызвана из Linux с параметром режима “удобочитаемости” одним из следующих способов:

```
ls --human-readable
```

или

```
ls -h
```

В версии Java 9 следует использовать обозначение `--version` вместо `-version` и `--class-path` вместо `-class-path` для указания соответствующих параметров в командной строке. И как будет показано в главе 9 второго тома настоящего издания, у параметра `--module-path` имеется сокращенное обозначение `-p`.

Более подробные рекомендации относительно параметров командной строки можно найти в спецификации JEP 293 запроса на усовершенствование по адресу <http://openjdk.java.net/jeps/293>. В частности, авторы этого документа предлагают также стандартизировать аргументы, задаваемые для параметров командной строки. Так, многобуквенные имена параметров с двумя дефисами могут отделяться от своих аргументов пробелом или знаком `=`, как показано ниже.

```
javac --class-path /home/user/classdir . . .
```

или

```
javac --class-path=/home/user/classdir . . .
```

Аргументы однобуквенных параметров могут отделяться пробелом или указываться сразу же после своего параметра:

```
javac -p moduledir . . .
```

или

```
javac -pmoduledir . . .
```



ВНИМАНИЕ! Последний вариант неработоспособен и вообще кажется неудачным. Зачем навлекать конфликты с унаследованными параметрами, если каталог модулей может называться `arameters` или `rocessor`?

Однобуквенные параметры без аргументов могут быть сгруппированы вместе, как показано ниже.

```
jar -cvf MyProgram.jar -e mypackage.MyProgram */*.class
```



ВНИМАНИЕ! В настоящее время такой способ пока еще не действует и может вызвать путаницу. Допустим, команда `javac` вызывается с параметром `-c`. Означает ли команда `javac -cp` то же самое, что и команда `javac -c -p`, или же приоритет в ней отдается унаследованному параметру `-cp`?

Такие правила указания параметров в командной строке вызвали путаницу, которая, как можно надеяться, будет со временем устранена. Как бы нам ни хотелось отойти от архаичного обозначения параметров команды `jar`, лучше всего, по-видимому, подождать до тех пор, пока эти правила не устоятся. Но если вы все же желаете идти в ногу со временем, то можете без опаски пользоваться длинными обозначениями параметров команды `jar`, как показано ниже.

```
jar --create --verbose --file ИмяJAR-файла файл1 файл2. . .
```

Однобуквенные обозначения параметров также пригодны, если только не группировать их вместе, как в следующем примере:

```
jar -c -v -f ИмяJAR-файла файл1 файл2. . .
```

4.9. Документирующие комментарии

В состав JDK входит полезное инструментальное средство — утилита `javadoc`, составляющая документацию в формате HTML из исходных файлов. По существу, интерактивная документация на прикладной программный интерфейс API является результатом применения утилиты `javadoc` к исходному коду стандартной библиотеки Java.

Добавив в исходный код комментарии, начинающиеся с последовательности знаков `/**`, нетрудно составить документацию, имеющую профессиональный вид. Это очень удобный способ, поскольку он позволяет совместно хранить как код, так и документацию к нему. Если же поместить документацию в отдельный файл, то со временем она перестанет соответствовать исходному коду. В то же время документацию нетрудно обновить, повторно запустив на выполнение утилиту `javadoc`, поскольку комментарии являются неотъемлемой частью исходного файла.

4.9.1. Вставка комментариев

Утилита `javadoc` извлекает сведения о следующих компонентах программы.

- Модули.
- Пакеты.
- Классы и интерфейсы, объявленные как `public`.
- Методы, объявленные как `public` или `protected`.
- Поля, объявленные как `public` или `protected`.

Защищенные компоненты программы, для объявления которых используется ключевое слово `protected`, будут рассмотрены в главе 5, интерфейсы — в главе 6, а модули — в главе 9 второго тома настоящего издания. Разрабатывая программу, можно (и даже нужно) комментировать каждый из перечисленных выше компонентов. Комментарии размещаются непосредственно перед тем компонентом, к которому они относятся. Комментарии начинаются знаками `/**` и оканчиваются знаками `*/`. Комментарии вида `/** ... */` содержат произвольный текст, после которого следует дескриптор. Дескриптор начинается со знака `@`, например `@author` или `@param`.

Первое предложение в тексте комментариев должно быть кратким описанием. Утилита `javadoc` автоматически формирует страницы, состоящие из кратких описаний. В самом тексте можно использовать элементы HTML-разметки, например, `...` — для выделения текста курсивом, `<code>...</code>` — для форматирования текста моноширинным шрифтом, `...` — для выделения текста полужирным, `` — для обозначения маркированных списков и даже `` — для вставки рисунков. Следует, однако, избегать применения заголовков (`<h1>` — `<h6>`) и горизонтальных линий (`<hr>`), поскольку они могут помешать нормальному форматированию документа.



НА ЗАМЕТКУ! Если комментарии содержат ссылки на другие файлы, например, рисунки (диаграммы или изображения компонентов пользовательского интерфейса), разместите эти файлы в каталоге `doc-files`, содержащем исходный файл. Утилита `javadoc` скопирует эти каталоги вместе с находящимися в них файлами из исходного каталога в данный каталог, выделяемый для документации. Поэтому в своих ссылках вы должны непременно указать каталог `doc-files`, например ``.

4.9.2. Комментарии к классам

Комментарии к классу должны размещаться *после* операторов `import`, непосредственно перед определением класса. Ниже приведен пример подобных комментариев.

```
/**
 * Объект класса Card имитирует игральную карту,
 * например даму червей. Карта имеет масть и ранг
 * (1=туз, 2...10, 11=валет, 12=дама, 13=король).
 */
public class Card
{
    ...
}
```



НА ЗАМЕТКУ! Начинать каждую строку документирующего комментария звездочкой нет никакой нужды. Например, следующий комментарий вполне корректен:

```
/**
    Объект класса <code>Card</code> имитирует игральную карту,
    например, даму червей. Карта имеет масть и ранг
    ( 1=туз, 2...10, 11=валет, 12=дама, 13=король).
*/
```

Но в большинстве IDE звездочки в начале строки документирующего комментария устанавливаются автоматически и перестраиваются при изменении расположения переносов строк.

4.9.3. Комментарии к методам

Комментарии должны непосредственно предшествовать методу, который они описывают. Кроме дескрипторов общего назначения, можно использовать перечисленные ниже специальные дескрипторы.

- **@param** *описание переменной*

Добавляет в описание метода раздел параметров. Раздел параметров можно развернуть на несколько строк. Кроме того, можно использовать элементы HTML-разметки. Все дескрипторы @param, относящиеся к одному методу, должны быть сгруппированы.

- **@return** *описание*

Добавляет в описание метода раздел возвращаемого значения. Этот раздел также может занимать несколько строк и допускает форматирование с помощью дескрипторов HTML-разметки.

- **@throws** *описание класса*

Указывает на то, что метод способен генерировать исключение. Исключения будут рассмотрены в главе 7.

Рассмотрим следующий пример комментариев к методу:

```
/**
 * Увеличивает зарплату работников
 * @param Переменная byPercent содержит величину
 * в процентах, на которую повышается зарплата
 * (например, 10 = 10%).
 * @return Величина, на которую повышается зарплата
 */
public double raiseSalary(double byPercent)
{
    double raise = salary * byPercent / 100;
    salary += raise;
    return raise;
}
```

4.9.4. Комментарии к полям

Документировать нужно лишь открытые поля. Они, как правило, являются статическими константами. Ниже приведен пример комментария к полю.

```
/**
 * Масть черви
 */
public static final int HEARTS = 1;
```

4.9.5. Комментарии общего характера

В комментариях к классам можно использовать следующие дескрипторы.

- **@author** *имя*

Создает раздел автора программы. В комментариях может быть несколько таких дескрипторов — по одному на каждого автора.

- **since** *текст*

Создает раздел начальной точки отсчета. Здесь *текст* означает описание версии программы, в которой впервые был внедрен данный компонент. Например, @since version 1.7.1.

- **@version** *текст*

Создает раздел версии программы. В данном случае *текст* означает произвольное описание текущей версии программы.

С помощью дескрипторов @see и @link можно указывать гипертекстовые ссылки на соответствующие внешние документы или на отдельную часть того же документа, сформированного с помощью утилиты **javadoc**.

Дескриптор @see *ссылка* добавляет ссылку в раздел “См. также”. Этот дескриптор можно употреблять в комментариях к классам и методам. Здесь *ссылка* означает одну из следующих конструкций:

```
пакет.класс#метка_компонента
<a href="...">метка</a>
"текст"
```

Первый вариант чаще всего встречается в документирующих комментариях. Здесь нужно указать имя класса, метода или переменной, а утилита **javadoc** вставит в документацию соответствующую гипертекстовую ссылку. Например, в приведенной ниже строке кода создается ссылка на метод `raiseSalary(double)` из класса `com.horstmann.corejava.Employee`. Если опустить имя пакета или же имя пакета и класса, то комментарии к данному компоненту будут размещены в текущем пакете или классе.

```
@see com.horstmann.corejava.Employee#raiseSalary(double)
```

Обратите внимание на то, что для отделения имени класса от имени метода или переменной служит знак #, а не точка. Компилятор **javac** корректно обрабатывает точки, выступающие в роли разделителей имен пакетов, подпакетов, классов, внутренних классов, методов и переменных. Но утилита **javadoc** не настолько развита логически, поэтому возникает потребность в специальном синтаксисе.

Если после дескриптора @see следует знак <, то необходимо указать гипертекстовую ссылку. Ссылаться можно на любой веб-адрес в формате URL, как показано ниже.

```
@see <a href= "www.horstmann.com/corejava.html">
    Web-страница книги Core Java</a>
```

В каждом из рассматриваемых здесь вариантов составления гипертекстовых ссылок можно указать необязательную *метку*, которая играет роль точки привязки гипертекстовой ссылки. Если не указать метку, точкой привязки гипертекстовой ссылки будет считаться имя программы или URL.

Если после дескриптора @see следует знак ", то текст отображается в разделе "См. также":

```
@see "Core Java volume 2"
```

Для комментирования одного и того же элемента можно использовать несколько дескрипторов @see, но их следует сгруппировать.

По желанию в любом комментарии можно разместить гипертекстовые ссылки на другие классы или методы. Для этого в любом месте документации достаточно ввести следующий специальный дескриптор:

```
{link пакет.класс#метка_элемента}
```

Описание функционального средства в этом дескрипторе подчиняется тем же правилам, что и для дескриптора @see.

И, наконец, начиная с версии Java 9, можно пользоваться дескриптором {@index элемент} для ввода указанного элемента в поле поиска.

4.9.6. Комментарии к пакетам

Комментарии к классам, методам и переменным, как правило, размещаются непосредственно в исходных файлах и выделяются знаками `/** ... */`. Но для формирования комментариев к *пакетам* следует добавить отдельный файл в каталог каждого пакета. Для этого имеются два варианта выбора.

1. Предоставить HTML-файл под именем `package-info.java`. Этот файл должен содержать первоначальные комментарии, выделяемые знаками `/**` и `*/` в формате утилиты **javadoc** и дополняемые оператором **package**. Никакого другого кода или комментариев в этом файле быть не должно.
2. Предоставить файл под именем `package.html`. Весь текст, содержащийся между дескрипторами `<body>` и `</body>`, извлекается утилитой **javadoc**.

4.9.7. Извлечение комментариев

Допустим, `docDirectory` — это имя каталога, в котором должны храниться HTML-файлы. Для извлечения комментариев в каталог выполните описанные ниже действия.

1. Перейдите в каталог с исходными файлами, которые подлежат документированию. Если документированию подлежат вложенные пакеты, например `com.horstmann.corejava`, перейдите в каталог, содержащий подкаталог `com`. (Именно в этом каталоге должен храниться файл `overview.html`.)
2. Выполните следующую команду, чтобы извлечь документирующие комментарии в указанный каталог из одного пакета:

```
javadoc -d docDirectory имя_пакета
```

3. А для того чтобы извлечь документирующие комментарии в указанный каталог из нескольких пакетов, выполните такую команду:

```
javadoc -d docDirectory имя_пакета_1 имя_пакета_2 ...
```

4. Если файлы находятся в пакете по умолчанию, вместо приведенных выше команд выполните команду

```
javadoc -d docDirectory *.java
```

Если опустить параметр `-d docDirectory`, HTML-файлы документации будут извлечены в текущий каталог, что вызовет путаницу. Поэтому делать этого не рекомендуется.

При вызове утилиты `javadoc` можно указывать различные параметры. Например, чтобы включить в документацию дескрипторы `@author` и `@version`, можно выбрать параметры `-author` и `-version` (по умолчанию они опускаются). Параметр `-link` позволяет включать в документацию ссылки на стандартные классы. Например, приведенная ниже команда автоматически создаст ссылку на документацию, находящуюся на веб-сайте компании Oracle.

```
javadoc -link http://docs.oracle.com/javase/9/docs/api *.java
```

Если же выбрать параметр `-linksource`, то каждый исходный файл будет преобразован в формат HTML (без цветового кодирования, но с номерами строк), а имя каждого класса и метода превратится в гиперссылку на исходный код. Кроме того, все исходные файлы можно снабдить обзорными комментариями. Они размещаются в файле `overview.html`, расположенном в родительском каталоге со всеми исходными файлами. Так, если выполнить команду

```
javadoc -overview имя_файла
```

то весь текст, находящийся между дескрипторами `<body>` и `</body>`, будет извлечен утилитой `javadoc`. Содержимое указанного файла выводится на экран, когда пользователь выбирает вариант Overview (Обзор) на панели навигации. Другие параметры описаны в документации на утилиту `javadoc`, доступной по адресу <https://docs.oracle.com/javase/9/javadoc/javadoc.htm>.

4.10. Рекомендации по разработке классов

В завершение этой главы приведем некоторые рекомендации, которые помогут вам в разработке классов в стиле ООП.

1. *Всегда храните данные в переменных, объявленных как **private**.*

Первое и главное требование: всеми средствами избегайте нарушения инкапсуляции. Иногда приходится писать методы доступа к полю или модифицирующие методы, но предоставлять доступ к полям не следует. Как показывает горький опыт, способ представления данных может изменяться, но порядок их использования изменяется намного реже. Если данные закрыты, их представление не влияет на использующий их класс, что упрощает выявление ошибок.

2. *Всегда инициализируйте данные.*

В языке Java локальные переменные не инициализируются, но поля в объектах инициализируются. Не полагайтесь на действия по умолчанию, инициализируйте переменные явным образом с помощью конструкторов.

3. *Не употребляйте в классе слишком много простых типов.*

Несколько связанных между собой полей простых типов следует объединять в новый класс. Такие классы проще для понимания, а кроме того, их легче

видоизменить. Например, следующие четыре поля из класса `Customer` нужно объединить в новый класс `Address`:

```
private String street;  
private String city;  
private String state;  
private int zip;
```

Представленный таким образом адрес легче изменить, как, например, в том случае, если требуется указать интернациональные адреса.

4. *Не для всех полей нужно создавать методы доступа и модификации.*

Очевидно, что при выполнении программы расчета зарплаты требуется получать сведения о зарплате работника, а кроме того, ее приходится время от времени изменять. Но вряд ли придется менять дату его приема на работу после того, как объект сконструирован. Иными словами, существуют поля, которые после создания объекта совсем не изменяются. К их числу относится, в частности, массив сокращенных названий штатов США в классе `Address`.

5. *Разбивайте на части слишком крупные классы.*

Это, конечно, слишком общая рекомендация: то, что кажется “слишком крупным” одному программисту, представляется нормальным другому. Но если есть очевидная возможность разделить один сложный класс на два класса попроще, то воспользуйтесь ею. (Опасайтесь, однако, другой крайности. Вряд ли оправданы десять классов, в каждом из которых имеется только один метод.) Ниже приведен пример неудачного составления класса.

```
public class CardDeck // неудачная конструкция  
{  
    private int[] value;  
    private int[] suit;  
  
    public CardDeck() { . . . }  
    public void shuffle() { . . . }  
    public int getTopValue() { . . . }  
    public int getTopSuit() { . . . }  
    public void draw() { . . . }  
}
```

На самом деле в этом классе реализованы два разных понятия: во-первых, *карточный стол* с методами `shuffle()` (тасование) и `draw()` (раздача) и, во-вторых, *игральная карта* с методами для проверки ранга и масти карты. Разумнее было бы ввести класс `Card`, представляющий отдельную игральную карту. В итоге имелись бы два класса, каждый из которых отвечал бы за свое:

```
public class CardDeck  
{  
    private Card[] cards;  
    public CardDeck() { . . . }  
    public void shuffle() { . . . }  
    public Card getTop() { . . . }  
    public void draw() { . . . }  
}
```

```
public class Card
```



```
{
    private int value;
    private int suit;

    public Card(int aValue, int aSuit) { . . . }
    public int getValue() { . . . }
    public int getSuit() { . . . }
}
```

6. *Выбирайте для классов и методов осмысленные имена, ясно указывающие на их назначение.*

Классы, как и переменные, следует называть именами, отражающими их назначение. (В стандартной библиотеке имеются примеры, где это правило нарушается. Например, класс `Date` описывает время, а не дату.)

Удобно принять следующие условные обозначения: имя класса должно быть именем существительным (`Order`) или именем существительным, которому предшествует имя прилагательное (`RushOrder`) или деепричастие (`BillingAddress`). Как правило, методы доступа должны начинаться словом `get`, представленным строчными буквами (`getSalary`), а модифицирующие методы — словом `set`, также представленным строчными буквами (`setSalary`).

7. *Отдавайте предпочтение неизменяемым классам.*

Класс `LocalDate` и прочие классы из пакета `java.time` являются неизменяемыми. Это означает, что они не содержат методы, способные видоизменить (т.е. модифицировать) состояние объекта. Вместо модификации объектов такие методы, как, например, `plusDays()`, возвращают новые объекты с видоизмененным состоянием.

Трудность модификации состоит в том, что она может происходить параллельно, когда в нескольких потоках исполнения предпринимается одновременная попытка обновить объект. Результаты такого обновления непредсказуемы. Если же классы являются неизменяемыми, то их объекты можно благополучно разделять среди нескольких потоков исполнения.

Таким образом, классы рекомендуется делать неизменяемыми при всякой удобной возможности. Это особенно просто сделать с классами, представляющими значения вроде символьной строки или момента времени. А в результате вычислений просто получаются новые значения, а не обновляются уже существующие.

Разумеется, не все классы должны быть неизменяемыми. Было бы, например, нелепо, если бы метод `raiseSalary()` возвращал новый объект типа `Employee`, когда поднимается зарплата работника.

В этой главе были рассмотрены основы создания объектов и классов, которые делают Java объектным языком. Но для того чтобы быть действительно объектно-ориентированным, язык программирования должен также поддерживать наследование и полиморфизм. О реализации этих принципов ООП в Java речь пойдет в следующей главе.

Наследование

В этой главе...

- ▶ Классы, суперклассы и подклассы
- ▶ Глобальный суперкласс `Object`
- ▶ Обобщенные списочные массивы
- ▶ Объектные оболочки и автоупаковка
- ▶ Методы с переменным числом параметров
- ▶ Классы перечислений
- ▶ Рефлексия
- ▶ Рекомендации по применению наследования

В главе 4 были рассмотрены классы и объекты, данная глава посвящена наследованию — фундаментальному принципу объектно-ориентированного программирования (ООП). Принцип наследования состоит в том, что новые классы можно создавать из уже существующих. При наследовании методы и поля существующего класса используются повторно (наследуются) вновь создаваемым классом, причем для адаптации нового класса к новым условиям в него добавляют дополнительные поля и методы. Этот прием играет в Java весьма важную роль.

В этой главе будет также рассмотрен механизм рефлексии, позволяющий исследовать свойства классов в ходе выполнения программы. *Рефлексия* — эффективный, но очень сложный механизм. А поскольку рефлексия больше интересует разработчиков инструментальных средств, чем прикладных программ, то при первом чтении можно ограничиться лишь беглым просмотром той части главы, которая посвящена данному механизму.

5.1. Классы, суперклассы и подклассы

Вернемся к примеру класса `Employee`, рассмотренному в предыдущей главе. Допустим, вы работаете в организации, где работа руководящего состава учитывается иначе, чем работа остальных работников. Разумеется, руководители во многих отношениях являются обычными наемными работниками. Руководящим и обычным работникам выплачивается заработная плата, но первые за свои достижения получают еще и премии. В таком случае для расчета зарплаты следует применять наследование. Почему? Потому что нужно определить новый класс `Manager`, в который придется ввести новые функциональные возможности. Но в этот класс можно перенести кое-что из того, что уже запрограммировано в классе `Employee`, сохранив все поля, определенные в исходном классе. Говоря более абстрактно, между классами `Manager` и `Employee` существует вполне очевидное отношение “является”: каждый руководитель *является* работником. Именно это отношение и служит явным признаком наследования.



НА ЗАМЕТКУ! В этой главе рассматривается классический пример рядовых и руководящих работников, но к этому примеру следует относиться критически. На практике рядовой работник может стать руководителем, поэтому придется предусмотреть и такой случай, когда рядовой работник может выполнять роль руководителя, а не просто отнести руководителей к подклассу, производному от всех работников. Но в данном примере предполагается, что корпоративная среда наполнена в основном следующими категориями трудящихся: теми, кто так и останутся навсегда рядовыми работниками, а также теми, кто всегда были руководителями.

5.1.1. Определение подклассов

Ниже показано, каким образом определяется класс `Manager`, производный от класса `Employee`. Для обозначения наследования в Java служит ключевое слово `extends`.

```
class Manager extends Employee
{
    Дополнительные методы и поля
}
```



НА ЗАМЕТКУ C++! Механизмы наследования в Java и C++ сходны. В языке Java вместо знака : для обозначения наследования служит ключевое слово `extends`. Любое наследование в Java является открытым, т.е. в этом языке нет аналога закрытому и защищенному наследованию, допускаемому в C++.

Ключевое слово `extends` означает, что на основе существующего класса создается новый класс. Существующий класс называется *суперклассом*, *базовым* или *родительским*, а вновь создаваемый — *подклассом*, *производным* или *порожденным*. В среде программирующих на Java наиболее широко распространены термины *суперкласс* и *подкласс*, хотя некоторые из них предпочитают пользоваться терминами *родительский* и *порожденный класс*, более тесно связанными с понятием наследования.

Класс `Employee` является суперклассом. Это не означает, что он имеет превосходство над своим подклассом или обладает более широкими функциональными возможностями. На самом деле все *наоборот*: функциональные возможности подкласса шире, чем у суперкласса. Как станет ясно в дальнейшем, класс `Manager` инкапсулирует больше данных и содержит больше методов, чем его суперкласс `Employee`.



НА ЗАМЕТКУ! Префиксы *супер-* и *под-* заимствованы в программировании из теории множеств. Множество всех работников содержит в себе множество всех руководителей. В этом случае говорят, что множество работников является *супермножеством* по отношению к множеству руководителей. Иначе говоря, множество всех руководителей является *подмножеством* для множества всех работников.

Класс `Manager` содержит новое поле, в котором хранится величина премии, а также новый метод, позволяющий задавать эту величину:

```
class Manager extends Employee
{
    private double bonus;
    . . .
    public void setBonus(double b)
    {
        bonus = b;
    }
}
```

В этих методах и полях нет ничего особенного. Имея объект типа `Manager`, можно просто вызывать для него метод `setBonus()` следующим образом:

```
Manager boss = ...;
boss.setBonus(5000);
```

Разумеется, для объекта типа `Employee` вызвать метод `setBonus()` *нельзя*, поскольку его нет среди методов, определенных в классе `Employee`. Но в то же время методы `getName()` и `getHireDay()` можно вызывать для объектов типа `Manager`, поскольку они наследуются от суперкласса `Employee`, хотя и не определены в классе `Manager`. Поля `name`, `salary` и `hireDay` также наследуются от суперкласса. Таким образом, у каждого объекта типа `Manager` имеются четыре поля: `name`, `salary`, `hireDay` и `bonus`.

Определяя подкласс посредством расширения суперкласса, достаточно указать лишь отличия между подклассом и суперклассом. Разрабатывая классы, следует размещать общие методы в суперклассе, а специальные — в подклассе. Такое выделение общих функциональных возможностей в отдельном суперклассе широко распространено в ООП.

5.1.2. Переопределение методов

Некоторые методы суперкласса не подходят для подкласса `Manager`. В частности, метод `getSalary()` должен возвращать сумму основной зарплаты и премии. Следовательно, нужно *переопределить* метод, т.е. реализовать новый метод, замещающий соответствующий метод из суперкласса, как показано ниже.

```
class Manager extends Employee
{
    . . .
    public double getSalary()
    {
        . . .
    }
    . . .
}
```

Как же реализовать такой метод? На первый взгляд, сделать это очень просто — нужно лишь вернуть сумму полей `salary` и `bonus` следующим образом:

```
public double getSalary()
{
    return salary + bonus; // не сработает!
}
```

Но оказывается, что такой способ не годится. Метод `getSalary()` из класса `Manager` не имеет доступа к закрытым полям суперкласса. Иными словами, метод `getSalary()` из класса `Manager` не может непосредственно обратиться к полю `salary`, несмотря на то, что у каждого объекта типа `Manager` имеется поле с таким же именем. Только методы из класса `Employee` имеют доступ к закрытым полям суперкласса. Если же методом из класса `Manager` требуется доступ к закрытым полям, они должны сделать то же, что и любой другой метод: использовать открытый интерфейс (в данном случае метод `getSalary()` из класса `Employee`).

Итак, сделаем еще одну попытку. Вместо непосредственного обращения к полю `salary` попробуем вызвать метод `getSalary()`, как показано ниже.

```
public double getSalary()
{
    double baseSalary = getSalary(); // по-прежнему не сработает!
    return baseSalary + bonus;
}
```

Дело в том, что метод `getSalary()` вызывает сам себя, поскольку в классе `Manager` имеется одноименный метод (именно его мы и пытаемся реализовать). В итоге возникает бесконечная цепочка вызовов одного и того же метода, что приводит к аварийному завершению программы.

Нужно найти какой-то способ явно указать, что требуется вызвать метод `getSalary()` из суперкласса `Employee`, а не из текущего класса. Для этой цели служит специальное ключевое слово `super`. В приведенной ниже строке кода метод `getSalary()` вызывается именно из класса `Employee`.

```
super.getSalary()
```

Вот как выглядит правильный вариант метода `getSalary()` для класса `Manager`:

```
public double getSalary()
{
    double baseSalary = super.getSalary();
    return baseSalary + bonus;
}
```



НА ЗАМЕТКУ! Некоторые считают ключевое слово `super` аналогом ссылки `this`. Но эта аналогия не совсем точна — ключевое слово `super` не означает ссылку на объект. Например, по нему нельзя присвоить значение другой объектной переменной. Это слово лишь сообщает компилятору, что нужно вызвать метод из суперкласса.

Как видите, в подкласс можно *вводить* поля, а также *вводить* и *переопределять* методы из суперкласса. И в результате наследования ни одно поле или метод не удаляется из класса.



НА ЗАМЕТКУ C++! Для вызова метода из суперкласса в Java служит ключевое слово **super**, а в C++ для этого можно указать имя суперкласса вместе с операцией **::**. Например, метод **getSalary()** из класса **Manager** в C++ можно было бы вызвать следующим образом: **Employee::getSalary()** вместо **super.getSalary()**.

5.1.3. Конструкторы подклассов

И в завершение рассматриваемого здесь примера снабдим класс **Manager** конструктором:

```
public Manager(String n, double s, int year, int month, int day)
{
    super(n, s, year, month, day);
    bonus = 0;
}
```

Здесь ключевое слово **super** имеет уже другой смысл. Приведенное ниже выражение означает вызов конструктора суперкласса **Employee** с параметрами **n**, **s**, **year**, **month** и **day**.

```
super(n, s, year, month, day);
```

Конструктор класса **Manager** не имеет доступа к закрытым полям класса **Employee**, поэтому он должен инициализировать их, вызывая другой конструктор с помощью ключевого слова **super**. Вызов, содержащий обращение **super**, должен быть первым оператором в конструкторе подкласса.

Если конструктор подкласса не вызывает явно ни одного из конструкторов суперкласса, то из этого суперкласса автоматически вызывается конструктор без аргументов. Если же в суперклассе отсутствует конструктор без аргументов, а конструктор подкласса не вызывает явно другой конструктор из суперкласса, то компилятор Java выдаст сообщение об ошибке.



НА ЗАМЕТКУ! Напомним, что ключевое слово **this** применяется в следующих двух случаях: для указания ссылки на неявный параметр и для вызова другого конструктора того же класса. Аналогично ключевое слово **super** используется в следующих двух случаях: для вызова метода и конструктора из суперкласса. При вызове конструкторов ключевые слова **this** и **super** имеют почти одинаковый смысл. Вызов конструктора должен быть первым оператором в вызывающем конструкторе. Параметры такого конструктора передаются вызывающему конструктору того же класса (**this**) или конструктору суперкласса (**super**).



НА ЗАМЕТКУ C++! Вызов конструктора **super** в C++ не применяется. Вместо этого для создания суперкласса служит список инициализации. В коде C++ конструктор класса **Manager** выглядел бы следующим образом:

```
Manager::Manager(String n, double s, int year, int month, int day) // C++
: Employee(n, s, year, month, day)
{
    bonus = 0;
}
```

После переопределения метода **getSalary()** для объектов типа **Manager** всем руководящим работникам дополнительно к зарплате будет начислена премия.

Обратимся к конкретному примеру, создав объект типа `Manager` и установив величину премии, как показано ниже.

```
Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);  
boss.setBonus(5000);
```

Затем образуем массив, в котором должны храниться три объекта типа `Employee`:

```
Employee[] staff = new Employee[3];
```

Заполним его объектами типа `Manager` и `Employee`:

```
staff[0] = boss;  
staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);  
staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
```

И, наконец, выведем зарплату каждого работника:

```
for (Employee e : staff)  
    System.out.println(e.getName() + " " + e.getSalary());
```

В результате выполнения приведенного выше цикла на экран выводятся следующие строки:

```
Carl Cracker 85000.0  
Harry Hacker 50000.0  
Tommy Tester 40000.0
```

Из элементов массива `staff[1]` и `staff[2]` выводятся основные зарплаты обычных работников, поскольку они представлены объектами типа `Employee`. Но объект из элемента массива `staff[0]` относится к классу `Manager`, поэтому в методе `getSalary()` из этого класса к основной зарплате прибавляется премия. Обратите особое внимание на следующий вызов:

```
e.getSalary()
```

Здесь вызывается именно тот метод `getSalary()`, который необходим в данном случае. Обратите также внимание на то, что объявленным типом переменной `e` является `Employee`, но фактическим типом объекта, на который может ссылаться переменная `e`, может быть как `Employee`, так и `Manager`.

Когда переменная `e` ссылается на объект типа `Employee`, при обращении `e.getSalary()` вызывается метод `getSalary()` из класса `Employee`. Но если переменная `e` ссылается на объект типа `Manager`, то вызывается метод `getSalary()` из класса `Manager`. Виртуальной машине известен фактический тип объекта, на который ссылается переменная `e`, поэтому при вызове данного метода никаких затруднений не возникает.

Способность переменной (например, `e`) ссылаться на объекты, имеющие разные фактические типы, называется *полиморфизмом*. Автоматический выбор нужного метода во время выполнения программы называется *динамическим связыванием*. Оба эти понятия будут подробнее обсуждаться далее в главе.



НА ЗАМЕТКУ C++! В языке Java нет необходимости объявлять метод как виртуальный. Динамическое связывание выполняется по умолчанию. Если же не требуется, чтобы метод был виртуальным, его достаточно объявить с ключевым словом `final` (оно будет рассматриваться далее в этой главе).

В листинге 5.1 представлен пример кода, демонстрирующий отличия в расчете заработной платы для объектов типа `Employee` (листинг 5.2) и `Manager` (листинг 5.3).

Листинг 5.1. Исходный код из файла `inheritance/ManagerTest.java`

```
1 package inheritance;
2
3 /**
4  * В этой программе демонстрируется наследование
5  * @version 1.21 2004-02-21
6  * @author Cay Horstmann
7  */
8 public class ManagerTest
9 {
10     public static void main(String[] args)
11     {
12         // построить объект типа Manager
13         Manager boss = new Manager("Carl Cracker",
14                                     80000, 1987, 12, 15);
15         boss.setBonus(5000);
16
17         Employee[] staff = new Employee[3];
18
19         // заполнить массив staff объектами
20         // типа Manager и Employee
21         staff[0] = boss;
22         staff[1] = new Employee("Harry Hacker", 50000,
23                                 1989, 10, 1);
24         staff[2] = new Employee("Tommy Tester", 40000,
25                                 1990, 3, 15);
26
27         // вывести данные обо всех объектах типа Employee
28         for (Employee e : staff)
29             System.out.println("name=" + e.getName()
30                                + ",salary=" + e.getSalary());
31     }
32 }
```

Листинг 5.2. Исходный код из файла `inheritance/Employee.java`

```
1 package inheritance;
2
3 import java.time.*;
4
5 public class Employee
6 {
7     private String name;
8     private double salary;
9     private LocalDate hireDay;
10
11     public Employee(
12         String name, double salary, int year,
13         int month, int day)
14     {
15         this.name = name;
16         this.salary = salary;
17         hireDay = LocalDate.of(year, month, day);
```



```
18 }
19
20 public String getName()
21 {
22     return name;
23 }
24
25 public double getSalary()
26 {
27     return salary;
28 }
29
30 public LocalDate getHireDay()
31 {
32     return hireDay;
33 }
34
35 public void raiseSalary(double byPercent)
36 {
37     double raise = salary * byPercent / 100;
38     salary += raise;
39 }
40 }
```

Листинг 5.3. Исходный код из файла `inheritance/Manager.java`

```
1 package inheritance;
2
3 public class Manager extends Employee
4 {
5     private double bonus;
6     /**
7      * @param n Имя работника
8      * @param s Зарплата
9      * @param year Год приема на работу
10     * @param month Месяц приема на работу
11     * @param day День приема на работу
12     */
13     public Manager(String n, double s, int year,
14                     int month, int day)
15     {
16         super(n, s, year, month, day);
17         bonus = 0;
18     }
19
20     public double getSalary()
21     {
22         double baseSalary = super.getSalary();
23         return baseSalary + bonus;
24     }
25
26     public void setBonus(double b)
27     {
28         bonus = b;
29     }
30 }
```

```
29 }  
30 }
```

5.1.4. Иерархии наследования

Наследование не обязательно ограничивается одним уровнем классов. Например, на основе класса `Manager` можно создать подкласс `Executive`. Совокупность всех классов, производных от общего суперкласса, называется *иерархией наследования*. Условно иерархия наследования показана на рис. 5.1. Путь от конкретного класса к его потомкам в иерархии называется *цепочкой наследования*.

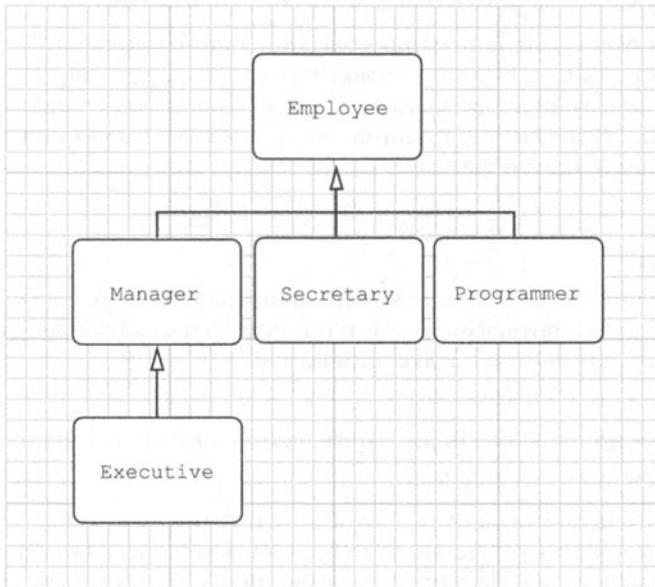


Рис. 5.1. Иерархия наследования для класса `Employee`

Обычно для класса существует несколько цепочек наследования. На основе класса `Employee` для работников можно, например, создать подкласс `Programmer` для программистов или класс `Secretary` для секретарей, причем они будут совершенно независимыми как от класса `Manager` для руководителей, так и друг от друга. Процесс формирования подклассов можно продолжать как угодно долго.



НА ЗАМЕТКУ C++! В языке Java множественное наследование не поддерживается. Задачи, для которых в других языках применяется множественное наследование, в Java решаются с помощью механизма интерфейсов (интерфейсы будут рассматриваться в начале следующей главы).

5.1.5. Полиморфизм

Существует простое правило, позволяющее определить, стоит ли в конкретной ситуации применять наследование или нет. Если между объектами существует отношение “является”, то каждый объект подкласса является объектом суперкласса. Например, каждый руководитель является работником. Следовательно, имеет смысл

сделать класс `Manager` подклассом, производным от класса `Employee`. Естественно, обратное утверждение неверно — не каждый работник является руководителем.

Отношение “является” можно выявить и по-другому, используя *принцип подстановки*, который состоит в том, что объект подкласса можно использовать вместо любого объекта суперкласса. Например, объект подкласса можно присвоить переменной суперкласса следующим образом:

```
Employee e;  
e = new Employee(...);  
    // предполагается объект класса Employee  
e = new Manager(...);  
    // допускается использовать также объект класса Manager
```

В языке Java объектные переменные являются *полиморфными*. Переменная типа `Employee` может ссылаться как на объект класса `Employee`, так и на объект любого подкласса, производного от класса `Employee` (например, `Manager`, `Executive`, `Secretary` и т.п.). Применение принципа *полиморфизма* было продемонстрировано в листинге 5.1 следующим образом:

```
Manager boss = new manager(...);  
Employee[] staff = new Employee[3];  
staff[0] = boss;
```

Здесь переменные `staff[0]` и `boss` ссылаются на один и тот же объект. Но переменная `staff[0]` рассматривается компилятором только как объект типа `Employee`. Это означает, что допускается следующий вызов:

```
boss.setBonus(5000); // Допустимо!
```

В то же время приведенное ниже выражение составлено неверно.

```
staff[0].setBonus(5000); // ОШИБКА!
```

Дело в том, что переменная `staff[0]` объявлена как объект типа `Employee`, а метод `setBonus()` в этом классе отсутствует. Но присвоить ссылку на объект суперкласса переменной подкласса нельзя. Например, следующий оператор считается недопустимым:

```
Manager m = staff[i]; // ОШИБКА!
```

Причина очевидна: не все работники являются руководителями. Если бы это присваивание произошло и переменная `m` могла бы ссылаться на объект типа `Employee`, который не представляет руководителей, то впоследствии оказался бы возможным вызов метода `m.setBonus(...)`, что привело бы к ошибке при выполнении программы.



ВНИМАНИЕ! В языке Java массив ссылок на объекты подкласса можно преобразовать в массив ссылок на объекты суперкласса. Для этого приводить типы явным образом не требуется. В качестве примера рассмотрим следующий массив ссылок на объекты типа **Manager**:

```
Manager[] managers = new Manager[10];
```

Вполне допустимо преобразовать его в массив **Employee[]**, как показано ниже.

```
Employee[] staff = managers; // Допустимо!
```

В самом деле, каждый руководитель является работником, а объект типа **Manager** содержит все поля и методы, присутствующие в объекте типа **Employee**. Но тот факт, что переменные **managers** и **staff** ссылаются на один и тот же массив, может привести к неприятным последствиям. Рассмотрим следующее выражение:

```
staff[0] = new Employee("Harry Hacker", ...);
```

Компилятор вполне допускает подобное присваивание. Но `staff[0]` и `manager[0]` — это одна и та же ссылка, поэтому ситуация выглядит так, как будто мы пытаемся присвоить ссылку на объект, описывающий рядового работника, переменной, соответствующей объекту руководителя. В результате формально становится возможным вызов `managers[0].setBonus(1000)`, который повлечет за собой обращение к несуществующему методу и разрушение содержимого памяти.

Чтобы предотвратить подобную ситуацию, в созданном массиве запоминается тип элементов и отслеживается допустимость хранящихся ссылок. Так, если массив создан с помощью выражения `new Manager[10]`, то он рассматривается как массив объектов типа `Manager`, и попытка записать в него ссылку на объект типа `Employee` приведет к исключению `ArrayStoreException`, возникающему при нарушении порядка сохранения данных в массиве.

5.1.6. Представление о вызовах методов

Важно понять, каким образом вызов метода применяется к объекту. Допустим, делается вызов `x.f(args)` и неявный параметр `x` объявлен как объект класса `C`. В таком случае происходит следующее.

1. Сначала компилятор проверяет объявленный тип объекта, а также имя метода. Следует иметь в виду, что может существовать несколько методов под именем `f`, имеющих разные типы параметров, например, методы `f(int)` и `f(String)`. Компилятор перечисляет все методы под именем `f` в классе `C` и все доступные методы под именем `f` в суперклассах, производных от класса `C`. (Закрытые методы в суперклассе недоступны.) Итак, компилятору известны все возможные претенденты на вызов метода под именем `f`.
2. Затем компилятор определяет типы параметров, указанных при вызове метода. Если среди всех методов под именем `f` имеется только один метод, типы параметров которого совпадают с указанными, происходит его вызов. Этот процесс называется *разрешением перегрузки*. Например, при вызове `x.f("Hello")` компилятор выберет метод `f(String)`, а не `f(int)`. Но ситуация может осложниться вследствие преобразования типов (`int` — в `double`, `Manager` — в `Employee` и т.д.). Если компилятор не находит ни одного метода с подходящим набором параметров или в результате преобразования типов оказывается несколько методов, соответствующих данному вызову, выдается сообщение об ошибке. В конечном счете компилятору становятся известны типы параметров и имя метода, который должен быть вызван.



НА ЗАМЕТКУ! Напомним, что имя метода и список типов его параметров образуют сигнатуру метода. Например, методы `f(int)` и `f(String)` имеют одинаковые имена, но разные сигнатуры. Если в подклассе определен метод, сигнатура которого совпадает с сигнатурой некоторого метода из суперкласса, то метод подкласса переопределяет его, замещая собой.

Возвращаемый тип не относится к сигнатуре метода. Но при переопределении метода необходимо сохранить совместимость возвращаемых типов. В подклассе возвращаемый тип может быть заменен на подтип исходного типа. Допустим, метод `getBuddy()` определен в классе `Employee` следующим образом:

```
public Employee getBuddy() { ... }
```

Как известно, у начальников не принято заводить приятельские отношения с подчиненными. Для того чтобы отразить этот факт, в подклассе `Manager` метод `getBuddy()` переопределяется следующим образом:

```
public Manager getBuddy() { ... } // Изменение возвращаемого типа
// допустимо!
```

В таком случае говорят, что для методов `getBuddy()` определены ковариантные возвращаемые типы.

3. Если метод является закрытым (`private`), статическим (`static`), конечным (`final`) или конструктором, компилятору точно известно, как его вызвать. (Модификатор доступа `final` описывается в следующем разделе.) Такой процесс называется *статическим связыванием*. В противном случае вызываемый метод определяется по фактическому типу неявного параметра, а во время выполнения программы происходит *динамическое связывание*. В данном примере компилятор сформировал бы вызов метода `f(String)` путем динамического связывания.
4. Если при выполнении программы для вызова метода используется динамическое связывание, виртуальная машина должна вызвать версию метода, соответствующую фактическому типу объекта, на который ссылается переменная `x`. Допустим, объект имеет *фактический* тип `D` подкласса, производного от класса `C`. Если в классе `D` определен метод `f(String)`, то вызывается именно он. В противном случае поиск вызываемого метода `f(String)` осуществляется в суперклассе и т.д.

На поиск вызываемого метода уходит слишком много времени, поэтому виртуальная машина заранее создает для каждого класса *таблицу методов*, в которой перечисляются сигнатуры всех методов и фактически вызываемые методы. При вызове метода виртуальная машина просто просматривает таблицу методов. В данном примере виртуальная машина проверяет таблицу методов класса `D` и обнаруживает вызываемый метод `f(String)`. Такими методами могут быть `D.f(String)` или `X.f(String)`, если `X` — некоторый суперкласс для класса `D`. С этим связана одна любопытная особенность. Если вызывается метод `super.f(param)`, то компилятор просматривает таблицу методов суперкласса, на который указывает неявный параметр `super`.

Рассмотрим подробнее вызов метода `e.getSalary()` из листинга 5.1. Переменная `e` объявлена с типом `Employee`. В классе этого типа имеется только один метод `getSalary()`, у которого нет параметров. Следовательно, в данном случае можно не беспокоиться о разрешении перегрузки.

При объявлении метода `getSalary()` не указывались ключевые слова `private`, `static` или `final`, поэтому он связывается динамически. Виртуальная машина создает таблицу методов из классов `Employee` и `Manager`. Из этой таблицы для класса `Employee` следует, что все методы определены в самом классе, как показано ниже.

```
Employee:
getName() -> Employee.getName()
getSalary() -> Employee.getSalary()
getHireDay() -> Employee.getHireDay()
raiseSalary(double) -> Employee.raiseSalary(double)
```

На самом деле это не совсем так. Как станет ясно в дальнейшем, у класса `Employee` имеется суперкласс `Object`, от которого он наследует большое количество методов. Но мы не будем пока что принимать их во внимание.

Таблица методов из класса `Manager` имеет несколько иной вид. В этом классе три метода наследуются, один метод переопределяется и еще один — добавляется:

`Manager`:

```
getName() -> Employee.getName()
getSalary() -> Manager.getSalary()
getHireDay() -> Employee.getHireDay()
raiseSalary(double) -> Employee.raiseSalary(double)
setBonus(double) -> Manager.setBonus(double)
```

Во время выполнения программы вызов метода `e.getSalary()` разрешается следующим образом.

1. Сначала виртуальная машина загружает таблицу методов, соответствующую фактическому типу переменной `e`. Это может быть таблица методов из класса `Employee`, `Manager` или другого подкласса, производного от класса `Employee`.
2. Затем виртуальная машина определяет класс, в котором определен метод `getSalary()` с соответствующей сигнатурой. В итоге вызываемый метод становится известным.
3. И, наконец, виртуальная машина вызывает этот метод.

Динамическое связывание обладает одной важной особенностью: позволяет изменять программы без перекомпиляции их исходного кода. Это делает программы динамически *расширяемыми*. Допустим, в программу добавлен новый класс `Executive`, а переменная `e` может ссылаться на объект этого класса. Код, содержащий вызов метода `e.getSalary()`, заново компилировать не нужно. Если переменная `e` ссылается на объект типа `Executive`, то автоматически вызывается метод `Executive.getSalary()`.



ВНИМАНИЕ! При переопределении область действия метода из подкласса должна быть не меньше области действия метода из суперкласса. Так, если метод из суперкласса был объявлен как `public`, то и метод из подкласса должен быть объявлен как `public`. Программисты часто ошибаются, забывая указать модификатор доступа `public` при объявлении метода в подклассе. В подобных случаях компилятор сообщает, что привилегии доступа к данным ограничены.

5.1.7. Предотвращение наследования: конечные классы и методы

Иногда наследование оказывается нежелательным. Классы, которые нельзя расширить, называются *конечными*. Для указания на это в определении класса используется модификатор доступа `final`. Допустим, требуется предотвратить создание подклассов, производных от класса `Executive`. В таком случае класс `Executive` определяется следующим образом:

```
final class Executive extends Manager
{
    ...
}
```

Отдельный метод класса также может быть конечным. Такой метод не может быть переопределен в подклассах. (Все методы конечного класса автоматически являются конечными.) Ниже приведен пример объявления конечного метода.

```
class Employee
{
```

```
...
public final String getName()
{
    return name;
}
...
```



НА ЗАМЕТКУ! Напомним, что с помощью модификатора доступа **final** могут быть также описаны поля, являющиеся константами. После создания объекта значение такого поля нельзя изменить. Но если класс объявлен как **final**, то конечными автоматически становятся только его методы, но не поля.

Существует единственный аргумент в пользу указания ключевого слова **final** при объявлении метода или класса: гарантия неизменности семантики в подклассе. Так, методы `getTime()` и `setTime()` являются конечными в классе `Calendar`. Поступая подобным образом, разработчики данного класса берут на себя ответственность за корректность преобразования содержимого объекта типа `Date` в состояние календаря. В подклассах невозможно изменить принцип преобразования. В качестве другого примера можно привести конечный класс `String`. Создавать подклассы, производные от этого класса, запрещено. Следовательно, если имеется переменная типа `String`, то можно не сомневаться, что она ссылается именно на символьную строку, а не на что-нибудь другое.

Некоторые программисты считают, что ключевое слово **final** следует применять при объявлении всех методов. Исключением из этого правила являются только те случаи, когда имеются веские основания для применения принципа полиморфизма. Действительно, в C++ и C# для применения принципа полиморфизма в методах необходимо принимать специальные меры. Возможно, данное правило и слишком жесткое, но несомненно одно: при составлении иерархии классов следует серьезно подумать о целесообразности применения конечных классов и методов.

На заре развития Java некоторые программисты пытались использовать ключевое слово **final** для того, чтобы исключить издержки, связанные с динамическим связыванием. Если метод не переопределяется и невелик, компилятор применяет процедуру оптимизации, которая состоит в непосредственном *встраивании* кода. Например, вызов метода `e.getName()` заменяется доступом к полю `e.name`. Такое усовершенствование вполне целесообразно, поскольку ветвление программы несовместимо с упреждающей загрузкой команд, применяемой в процессорах. Но, если метод `getName()` будет переопределен, компилятор не сможет непосредственно встроить его код. Ведь ему неизвестно, какой из методов должен быть вызван при выполнении программы.

Правда, компонент виртуальной машины, называемый *динамическим компилятором*, может выполнять оптимизацию более эффективно, чем обыкновенный компилятор. Ему точно известно, какие именно классы расширяют данный класс, и он может проверить, действительно ли метод переопределяется. Если же метод невелик, часто вызывается и не переопределен, то динамический компилятор выполнит непосредственное встраивание его кода. Но что произойдет, если виртуальная машина загрузит другой подкласс, который переопределяет встраиваемый метод? Тогда оптимизатор должен отменить встраивание кода. В подобных случаях выполнение программы замедляется, хотя это происходит редко.

5.1.8. Приведение типов

В главе 3 был рассмотрен процесс принудительного преобразования одного типа в другой, называемый *приведением типов*. Для этой цели в Java предусмотрена специальная запись. Например, при выполнении следующего фрагмента кода значение переменной `x` преобразуется в целочисленное отбрасыванием дробной части:

```
double x = 3.405;  
int nx = (int) x;
```

И как иногда возникает потребность в преобразовании значения с плавающей точкой в целочисленное, так и ссылку на объект требуется порой привести к типу другого класса. Для такого приведения типов служат те же самые синтаксические конструкции, что и для числовых выражений. С этой целью имя нужного класса следует заключить в скобки и поставить перед той ссылкой на объект, которую требуется привести к искомому типу. Ниже приведен соответствующий пример.

```
Manager boss = (Manager) staff[0];
```

Для такого приведения типов существует только одна причина: необходимость использовать все функциональные возможности объекта после того, как его фактический тип был на время забыт. Например, в классе `ManagerTest` массив `staff` содержит объекты типа `Employee`. Этот тип выбран потому, что в некоторых элементах данного массива хранятся данные о рядовых работниках. А для того чтобы получить доступ ко всем новым полям из класса `Manager`, скорее всего, придется привести некоторые элементы массива `staff` к типу `Manager`. (В примере кода, рассмотренном в начале этой главы, были приняты специальные меры, чтобы избежать приведения типов. В частности, переменная `boss` была инициализирована объектом типа `Manager`, перед тем как разместить ее в массиве. Чтобы задать величину премии руководящего работника, необходимо знать правильный тип соответствующего объекта.)

Как известно, у каждой объектной переменной в Java имеется свой тип. Тип объектной переменной определяет разновидность объекта, на который ссылается эта переменная, а также ее функциональные возможности. Например, переменная `staff[1]` ссылается на объект типа `Employee`, поэтому она может ссылаться и на объект типа `Manager`.

В процессе работы компилятор проверяет, не обещаете ли вы слишком много, сохраняя значение в переменной. Так, если вы присваиваете переменной суперкласса ссылку на объект подкласса, то обещаете *меньше* положенного, и компилятор просто разрешает вам сделать это. А если вы присваиваете объект суперкласса переменной подкласса, то обещаете *больше* положенного и поэтому должны подтвердить свои обещания, указав в скобках имя класса для приведения типов. Таким образом, виртуальная машина получает возможность контролировать ваши действия при выполнении программы.

А что произойдет, если попытаться осуществить приведение типов вниз по цепочке наследования и попробовать обмануть компилятор в отношении содержимого объекта, как в представленной ниже строке кода?

```
Manager boss = (Manager) staff[1]; // ОШИБКА!
```

При выполнении программы система обнаружит несоответствие и сгенерирует исключение типа `ClassCastException`. Если его не перехватить, нормальное выполнение программы будет прервано. Таким образом, перед приведением типов следует проверить его корректность с помощью операции `instanceof`:


```
if (staff[1] instanceof Manager)
{
    boss = (Manager) staff[1];
    . . .
}
```

И, наконец, компилятор не позволит выполнить некорректное приведение типов, если для этого нет никаких оснований. Например, наличие приведенной ниже строки в исходном коде программы приведет к ошибке во время компиляции, поскольку класс `String` не является подклассом, производным от класса `Employee`.

```
Date c = (String) staff[1];
```

Таким образом, можно сформулировать следующие основные правила приведения типов при наследовании.

- Приведение типов можно выполнять только в иерархии наследования.
- Чтобы проверить корректность приведения суперкласса к подклассу, следует выполнить операцию `instanceof`.



НА ЗАМЕТКУ! Если в приведенном ниже выражении переменная **x** содержит пустое значение **null**, исключение не будет сгенерировано и возвратится логическое значение **false**.

```
x instanceof C
```

И это вполне логично. Ведь пустая ссылка типа **null** не указывает ни на один из объектов, а следовательно, она не указывает ни на один из объектов типа **C**.

На самом деле приведение типов при наследовании — не самое лучшее решение. В данном примере выполнять преобразование объекта типа `Employee` в объект типа `Manager` совсем не обязательно. Метод `getSalary()` вполне способен оперировать объектами обоих типов, поскольку при динамическом связывании правильный метод автоматически определяется благодаря принципу полиморфизма.

Приведение типов целесообразно лишь в том случае, когда для объектов, представляющих руководителей, требуется вызвать особый метод, имеющийся только в классе `Manager`, например метод `setBonus()`. Если же по какой-нибудь причине потребуется вызвать метод `setBonus()` для объекта типа `Employee`, следует задать себе вопрос: не свидетельствует ли это о недостатках суперкласса? Возможно, имеет смысл пересмотреть структуру суперкласса и добавить в него метод `setBonus()`. Не забывайте, что для преждевременного завершения программы достаточно единственного неперехваченного исключения типа `ClassCastException`. А в целом при наследовании лучше свести к минимуму приведение типов и выполнение операции `instanceof`.



НА ЗАМЕТКУ C++! В языке Java для приведения типов служит устаревший синтаксис языка C, но он действует подобно безопасной операции **dynamic_cast** динамического приведения типов в C++. Например, следующие строки кода, написанные на разных языках, почти равнозначны:

```
Manager boss = (Manager)staff[1]; // Java
Manager* boss = dynamic_cast<Manager*>(staff[1]); // C++
```

Но у них имеется одно важное отличие. Если приведение типов завершается неудачно, то вместо пустого объекта генерируется исключение. В этом смысле приведение типов в Java напоминает приведение ссылок в C++. И это весьма существенный недостаток языка Java. Ведь в C++ контроль и преобразование типов можно выполнить в одной операции следующим образом:

```
Manager* boss = dynamic_cast<Manager*>(staff[1]); // C++  
if (boss != NULL) ...
```

А в Java для этой цели приходится сочетать операцию `instanceof` с приведением типов, как показано ниже.

```
if (staff[1] instanceof Manager) // Java  
{  
    Manager boss = (Manager)staff[1];  
    ...  
}
```

5.1.9. Абстрактные классы

Чем дальше вверх по иерархии наследования, тем более универсальными и абстрактными становятся классы. В некотором смысле родительские классы, находящиеся на верхней ступени иерархии, становятся настолько абстрактными, что их рассматривают как основу для разработки других классов, а не как классы, позволяющие создавать конкретные объекты. Например, работник — это человек, а человек может быть и студентом. Поэтому расширим иерархию, в которую входит класс `Employee`, добавив в нее классы `Person` и `Student`. Отношения наследования между всеми этими классами показаны на рис. 5.2.

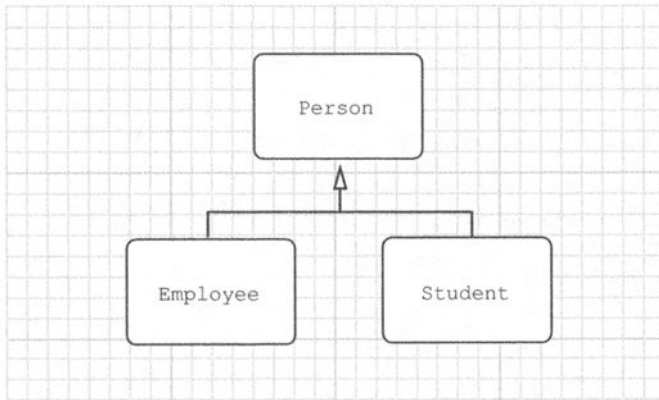


Рис. 5.2. Блок-схема, демонстрирующая иерархию наследования для класса **Person**

А зачем вообще столь высокий уровень абстракции? Существуют определенные свойства, характерные для каждого человека, например имя. Ведь у работников вообще и у студентов в частности имеются свои имена, и поэтому при внедрении общего суперкласса придется перенести метод `getName()` на более высокий уровень в иерархии наследования.

А теперь введем новый метод `getDescription()`, предназначенный для составления краткой характеристики человека, например:

an employee with a salary of \$50,000.00
a student majoring in computer science¹

Для классов `Employee` и `Student` такой метод реализуется довольно просто. Но какие сведения о человеке следует разместить в классе `Person`? Ведь в нем ничего нет, кроме имени. Разумеется, можно было бы реализовать метод `Person.getDescription()`, возвращающий пустую строку. Но есть способ получше. От реализации этого метода в классе `Person` можно вообще отказаться, если воспользоваться ключевым словом `abstract`, как показано ниже.

```
public abstract String getDescription(); // реализация не требуется
```

Для большей ясности класс, содержащий один или несколько абстрактных методов, можно объявить абстрактным следующим образом:

```
public abstract class Person
{
    . . .
    public abstract String getDescription();
}
```

Помимо абстрактных методов, абстрактные классы могут содержать конкретные поля и методы. Например, в классе `Person` хранится имя человека и содержится конкретный метод, возвращающий это имя:

```
public abstract class Person
{
    private String name;

    public Person(String name)
    {
        this.name = name;
    }

    public abstract String getDescription();

    public String getName()
    {
        return name;
    }
}
```



СОВЕТ. Некоторые программисты не осознают, что абстрактные классы могут содержать конкретные методы. Общие поля и методы (будь то абстрактные или конкретные) следует всегда перемещать в суперкласс, каким бы он ни был: абстрактным или конкретным.

Абстрактные методы представляют собой прототипы методов, реализованных в подклассах. Расширяя абстрактный класс, можно оставить некоторые или все абстрактные методы неопределенными. При этом подкласс также станет абстрактным. Если даже определить все методы, то и тогда подкласс не перестанет быть абстрактным.

Определим, например, класс `Student`, расширяющий абстрактный класс `Person` и реализующий метод `getDescription()`. Ни один из методов в классе `Student` не

¹ работник с годовой зарплатой 50 тыс. долларов США
студент, изучающий вычислительную технику

является абстрактным, поэтому нет никакой необходимости объявлять сам класс абстрактным. Впрочем, класс может быть объявлен абстрактным, даже если он и не содержит ни одного абстрактного метода.

Создать экземпляры абстрактного класса нельзя. Например, приведенное ниже выражение ошибочно. Но можно создать объекты конкретного подкласса.

```
new Person("Vince Vu");
```

Однако для абстрактных классов *можно* создавать объектные переменные, но они должны ссылаться на объект неабстрактного класса. Рассмотрим следующую строку кода:

```
Person p = new Student("Vince Vu", "Economics");
```

где `p` — переменная абстрактного типа `Person`, ссылающаяся на экземпляр неабстрактного подкласса `Student`.



НА ЗАМЕТКУ C++! В языке C++ абстрактный метод называется чистой виртуальной функцией. Его обозначение оканчивается символами `= 0`, как показано ниже.

```
class Person // C++
{
    public:
        virtual string getDescription() = 0;
        ...
};
```

Класс в C++ является абстрактным, если он содержит хотя бы одну чистую виртуальную функцию. В языке C++ отсутствует специальное ключевое слово для обозначения абстрактных классов.

Определим конкретный подкласс `Student`, расширяющий абстрактный класс `Person`, следующим образом:

```
public class Student extends Person
{
    private String major;

    public Student(String name, String major)
    {
        super(name);
        this.major = major;
    }

    public String getDescription()
    {
        return "a student majoring in " + major;
    }
}
```

В этом подклассе определяется метод `getDescription()`. Все методы из класса `Student` являются конкретными, а следовательно, класс больше не является абстрактным.

В примере программы из листинга 5.4 определяются один абстрактный суперкласс `Person` (из листинга 5.5) и два конкретных подкласса, `Employee` (из листинга 5.6) и `Student` (из листинга 5.7). Сначала в этой программе массив типа `Person` заполняется ссылками на экземпляры классов `Employee` и `Student`:

Листинг 5.5. Исходный код из файла **abstractClasses/Person.java**

```
1 package abstractClasses;
2
3 public abstract class Person
4 {
5     public abstract String getDescription();
6     private String name;
7
8     public Person(String name)
9     {
10         this.name = name;
11     }
12
13     public String getName()
14     {
15         return name;
16     }
17 }
```

Листинг 5.6. Исходный код из файла **abstractClasses/Employee.java**

```
1 package abstractClasses;
2
3 import java.time.*;
4
5 public class Employee extends Person
6 {
7     private double salary;
8     private LocalDate hireDay;
9
10    public Employee(
11        String name, double salary, int year,
12        int month, int day)
13    {
14        super(name);
15        this.salary = salary;
16        hireDay = LocalDate.of(year, month, day);
17    }
18
19    public double getSalary()
20    {
21        return salary;
22    }
23
24    public LocalDate getHireDay()
25    {
26        return hireDay;
27    }
28
29    public String getDescription()
30    {
31        return String.format(
32            "an employee with a salary of $%.2f", salary);
33    }
34 }
```

```
33  }
34
35  public void raiseSalary(double byPercent)
36  {
37      double raise = salary * byPercent / 100;
38      salary += raise;
39  }
40 }
```

Листинг 5.7. Исходный код из файла `abstractClasses/Student.java`

```
1  package abstractClasses;
2
3  public class Student extends Person
4  {
5      private String major;
6
7      /**
8       * @param name Имя студента
9       * @param major Специализация студента
10     */
11     public Student(String name, String major)
12     {
13         // передать строку name конструктору суперкласса
14         // в качестве его параметра
15         super(name);
16         this.major = major;
17     }
18
19     public String getDescription()
20     {
21         return "a student majoring in " + major;
22     }
23 }
```

5.1.10. Защищенный доступ

Как известно, поля в классе желательно объявлять как `private`, а некоторые методы — как `public`. Любые закрытые (т.е. `private`) компоненты программы невидимы из других классов. Это утверждение справедливо и для подклассов: у подкласса отсутствует доступ к закрытым полям суперкласса.

Но иногда приходится ограничивать доступ к некоторому методу и открывать его лишь для подклассов. Реже возникает потребность предоставлять методам из подкласса доступ к полям в суперклассе. В таком случае компонент класса объявляется защищенным с помощью модификатора доступа, обозначаемого ключевым словом `protected`. Так, если поле `hireDay` объявлено в суперклассе `Employee` защищенным, а не закрытым, методы из подкласса `Manager` смогут обращаться к нему непосредственно.

В языке Java защищенное поле доступно в любом классе из того же самого пакета. А теперь допустим, что в другом пакете имеется подкласс `Administrator`. Методы этого класса могут обращаться *только* к полю `hireDay` объектов типа `Administrator`, но не объектов типа `Employee`. Такое ограничение введено в качестве

предупредительной меры против злоупотреблений механизмом защищенного доступа, не позволяющим создавать подклассы лишь для того, чтобы получить доступ к защищенным полям.

На практике пользоваться защищенными полями следует очень аккуратно. Допустим, созданный вами класс, где имеются защищенные поля, используется другими разработчиками. Без вашего ведома другие могут создавать подклассы, производные от вашего класса, тем самым получая доступ к защищенным полям. В таком случае вы уже не сможете изменить реализацию своего класса, не уведомив об этом других заинтересованных лиц. Но это противоречит самому духу ООП, поощряющему инкапсуляцию данных.

Применение защищенных методов более оправданно. Метод можно объявить в классе защищенным, чтобы ограничить его применение. Это означает, что в методах подклассов, предшественники которых известны изначально, можно вызвать защищенный метод, а методы других классов — нельзя. В качестве примера можно привести защищенный метод `clone()` из класса `Object`, более подробно рассматриваемый в главе 6.



НА ЗАМЕТКУ C++! В языке Java защищенные члены класса доступны из всех подклассов, а также из других классов того же самого пакета. Этот язык Java отличается от C++, где ключевое слово `protected` имеет несколько иной смысл. Таким образом, в Java ограничения на доступ к защищенным элементам менее строги, чем в C++.

Итак, в Java предоставляются следующие четыре модификатора доступа, определяющие границы области видимости компонентов программы.

1. Модификатор доступа `private` — ограничивает область видимости пределами класса.
2. Модификатор доступа `public` — не ограничивает область видимости.
3. Модификатор доступа `protected` — ограничивает область видимости пределами пакета и всеми подклассами.
4. Модификатор доступа *не требуется* — область видимости ограничивается пределами пакета (к сожалению) по умолчанию.

5.2. Глобальный суперкласс Object

Класс `Object` является исходным предшественником всех остальных классов, поэтому каждый класс в Java расширяет класс `Object`. Но явно отражать этот факт, как в приведенной ниже строке кода, совсем не обязательно.

```
public class Employee extends Object
```

Если суперкласс явно не указан, им считается класс `Object`. А поскольку *каждый* класс в Java расширяет класс `Object`, то очень важно знать, какими средствами обладает сам класс `Object`. В этой главе дается беглый обзор самых основных из них, прочие функциональные средства класса `Object` будут рассмотрены в других главах, а остальные сведения о них можно найти в соответствующей документации. (Некоторые методы из класса `Object` целесообразно рассматривать лишь вместе со служебными средствами параллелизма, которые обсуждаются в главе 12.)

5.2.1. Переменные типа `Object`

Переменную типа `Object` можно использовать в качестве ссылки на объект любого типа следующим образом:

```
Object obj = new Employee("Harry Hacker", 35000);
```

Разумеется, переменная этого класса полезна лишь как средство для хранения значений произвольного типа. Чтобы сделать с этим значением что-то конкретное, нужно знать его исходный тип, а затем выполнить приведение типов, как показано ниже. В языке Java объектами *не* являются только *примитивные* типы: числа, символы и логические значения.

```
Employee e = (Employee) obj;
```

Все массивы относятся к типам объектов тех классов, которые расширяют класс `Object`, независимо от того, содержатся ли в элементах массива объекты или примитивные типы:

```
Employee[] staff = new Employee[10];  
obj = staff; // Допустимо!  
obj = new int[10]; // Допустимо!
```



НА ЗАМЕТКУ C++! В языке C++ аналогичного глобального базового класса нет, хотя любой указатель можно, конечно, преобразовать в указатель типа `void*`.

5.2.2. Метод `equals()`

В методе `equals()` из класса `Object` проверяется, равны ли два объекта. А поскольку метод `equals()` реализован в классе `Object`, то в нем определяется только следующее: ссылаются ли переменные на один и тот же объект. В качестве проверки по умолчанию эти действия вполне оправданы: всякий объект равен самому себе. Для некоторых классов большего и не требуется. Например, вряд ли кому-то потребуется анализировать два объекта типа `PrintStream` и выяснять, отличаются ли они чем-нибудь. Но в ряде случаев равными должны считаться объекты одного типа, находящиеся в одном и том же состоянии.

Рассмотрим в качестве примера объекты, описывающие работников. Очевидно, что они одинаковы, если совпадают имена, размеры заработной платы и даты их приема на работу. (Строго говоря, в настоящих системах учета данных о работниках более оправдано сравнение идентификационных номеров. А здесь лишь демонстрируются принципы реализации метода `equals()` в прикладном коде, как показано ниже.)

```
class Employee  
{  
    public boolean equals(Object otherObject)  
    {  
        // быстро проверить объекты на идентичность  
        if (this == otherObject) return true;  
  
        // вернуть логическое значение false,  
        // если явный параметр имеет пустое значение null  
        if (otherObject == null) return false;  
  
        // если классы не совпадают, они не равны  
        if (getClass() != otherObject.getClass())  
            return false;
```

```
// Теперь известно, что объект otherObject
// относится к типу Employee и не является пустым
Employee other = (Employee) otherObject;

// проверить, хранятся ли в полях объектов
// одинаковые значения
return name.equals(other.name)
    && salary == other.salary
    && hireDay.equals(other.hireDay);
}
}
```

Метод `getClass()` возвращает класс объекта (подробно он будет обсуждаться далее в главе). Для того чтобы объекты можно было считать равными, они как минимум должны быть объектами одного и того же класса.



СОВЕТ. Для того чтобы в поле `name` или `hireDay` не оказалось пустого значения `null`, воспользуйтесь методом `Objects.equals()`. В результате вызова метода `Objects.equals(a, b)` возвращается логическое значение `true`, если оба его аргумента имеют пустое значение `null`; логическое значение `false`, если только один из аргументов имеет пустое значение `null`; а иначе делается вызов `a.equals(b)`. С учетом этого последний оператор в теле приведенного выше метода `Employee.equals()` приобретает следующий вид:

```
return Objects.equals(name, other.name)
    && salary == other.salary
    && Object.equals(hireDay, other.hireDay);
```

Определяя метод `equals()` для подкласса, сначала следует вызывать одноименный метод из суперкласса. Если проверка даст отрицательный результат, объекты нельзя считать равными. Если же поля суперкласса совпадают, можно приступить к сравнению полей подкласса, как показано ниже.

```
class Manager extends Employee
{
    . . .
    public boolean equals(Object otherObject)
    {
        if (!super.equals(otherObject)) return false;
        // При вызове метода super.equals() проверяется,
        // принадлежит ли объект otherObject
        // тому же самому классу
        Manager other = (Manager) otherObject;
        return bonus == other.bonus;
    }
}
```

5.2.3. Проверка объектов на равенство и наследование

Каким образом должен действовать метод `equals()`, если неявные и явные его параметры не принадлежат одному и тому же классу? Это спорный вопрос. В предыдущем примере из метода `equals()` возвращалось логическое значение `false`, если классы не совпадали полностью. Но многие программисты пользуются следующей проверкой:

```
if (!(otherObject instanceof Employee)) return false;
```

При этом остается вероятность, что объект `otherObject` принадлежит подклассу. Поэтому такой подход может стать причиной непредвиденных осложнений. Спецификация Java требует, чтобы метод `equals()` обладал следующими характеристиками.

1. *Рефлексивность*. При вызове `x.equals(x)` по любой ненулевой ссылке `x` должно возвращаться логическое значение `true`.
2. *Симметричность*. При вызове `x.equals(y)` по любым ссылкам `x` и `y` должно возвращаться логическое значение `true` тогда и только тогда, когда при вызове `y.equals(x)` возвращается логическое значение `true`.
3. *Транзитивность*. Если при вызовах `x.equals(y)` и `y.equals(z)` по любым ссылкам `x`, `y` и `z` возвращается логическое значение `true`, то и при вызове `x.equals(z)` возвращается логическое значение `true`.
4. *Согласованность*. Если объекты, на которые делаются ссылки `x` и `y`, не изменяются, то при повторном вызове `x.equals(y)` должно возвращаться то же самое значение.
5. При вызове `x.equals(null)` по любой непустой ссылке `x` должно возвращаться логическое значение `false`.

Обоснованность приведенных выше правил не вызывает сомнений. Так, совершенно очевидно, что результаты проверки не должны зависеть от того, делается ли в программе вызов `x.equals(y)` или `y.equals(x)`. Но применение правила симметрии имеет свои особенности, если явный и неявный параметры принадлежат разным классам. Рассмотрим следующий вызов:

```
e.equals(m)
```

где объект `e` принадлежит классу `Employee`, а объект `m` — классу `Manager`, причем каждый из них содержит одинаковые имена, зарплату и дату приема на работу. Если при вызове `e.equals(m)` выполняется проверка с помощью операции `instanceof`, то возвращается логическое значение `true`. Но это означает, что и при обратном вызове `m.equals(e)` также должно возвращаться логическое значение `true`, поскольку правило симметричности не позволяет возвращать логическое значение `false` или генерировать исключение.

В итоге класс `Manager` попадает в затруднительное положение. Его метод `equals()` должен сравнивать объект данного класса с любым объектом типа `Employee` без учета данных, позволяющих отличить руководящего работника от рядового! И в этом случае операция `instanceof` выглядит менее привлекательно.

Некоторые специалисты считают, что проверка с помощью метода `getClass()` некорректна, поскольку в этом случае нарушается принцип подстановки. В подтверждение они приводят метод `equals()` из класса `AbstractSet`, который проверяет, содержат ли два множества одинаковые элементы и расположены ли они в одинаковом порядке. Класс `AbstractSet` выступает в роли суперкласса для классов `TreeSet` и `HashSet`, которые не являются абстрактными. В этих классах применяются различные алгоритмы для обращения к элементам множества. Но на практике требуется иметь возможность сравнивать любые два множества, независимо от того, как они реализованы.

Следует, однако, признать, что рассматриваемый здесь пример слишком специфичен. В данном случае имело бы смысл объявить метод `AbstractSet.equals()` конечным, чтобы нельзя было изменить семантику проверки множеств на равенство. (На самом деле при объявлении метода ключевое слово `final` не указано. Это дает

возможность подклассам реализовать более эффективный алгоритм проверки на равенственность.)

Таким образом, возникают два варианта.

- Если проверка на равенство реализована в подклассе, правило симметричности требует использовать метод `getClass()`.
- Если же проверка на равенство производится средствами суперкласса, можно выполнить операцию `instanceof`. И тогда возможна ситуация, когда два объекта разных классов будут признаны равными.

В примере с рядовыми и руководящими работниками два объекта считаются равными, если их поля совпадают. Так, если имеются два объекта типа `Manager` с одинаковыми именами, заработной платой и датой приема на работу, но с отличающимися величинами премии, такие объекты следует признать разными. А для этого требуется проверка с помощью метода `getClass()`.

Но допустим, что для проверки на равенство используется идентификационный номер работника. Такая проверка имеет смысл для всех подклассов. В этом случае можно выполнить операцию `instanceof` и объявить метод `Employee.equals()` как `final`.



НА ЗАМЕТКУ! В стандартной библиотеке Java содержится более 150 реализаций метода `equals()`. В одних из них применяются в различных сочетаниях операции `instanceof`, вызовы метода `getClass()` и фрагменты кода, предназначенные для обработки исключения типа `ClassCastException`, а в других не выполняется практически никаких действий. Обратитесь за справкой к документации на класс `java.sql.Timestamp`, где указывается на некоторые непреодолимые трудности реализации. В частности, класс `Timestamp` наследует от класса `java.util.Date`, где в методе `equals()` организуется проверка с помощью операции `instanceof`, но переопределить этот метод симметрично и точно не представляется возможным.

Ниже приведены рекомендации для создания приближающегося к идеалу метода `equals()`.

1. Присвойте явному параметру имя `otherObject`. Впоследствии его тип нужно будет привести к типу другой переменной под названием `other`.
2. Проверьте, одинаковы ли ссылки `this` и `otheObject`, следующим образом:

```
if (this == otherObject) return true;
```
3. Это выражение составлено лишь в целях оптимизации проверки. Ведь намного быстрее проверить одинаковость ссылок, чем сравнивать поля объектов.
4. Выясните, является ли ссылка `otherObject` пустой (`null`), как показано ниже. Если она оказывается пустой, следует вернуть логическое значение `false`. Эту проверку нужно сделать обязательно.

```
if (otherObject == null) return false;
```
5. Сравните классы `this` и `otheObject`. Если семантика проверки может измениться в подклассе, воспользуйтесь методом `getClass()` следующим образом:

```
if (getClass() != otherObject.getClass()) return false;
```
6. Если одна и та же семантика остается справедливой для *всех* подклассов, проведите проверку с помощью операции `instanceof` следующим образом:

```
if (!(otherObject instanceof ИмяКласса)) return false;
```

7. Приведите тип объекта `otherObject` к типу переменной требуемого класса:

```
ИмяКласса other = (ИмяКласса)otherObject;
```

8. Сравните все поля, как показано ниже. Для полей примитивных типов служит операция `==`, а для объектных полей — метод `Objects.equals()`. Если все поля двух объектов совпадают, возвращается логическое значение `true`, а иначе — логическое значение `false`.

```
return поле1 == other.поле1
    && поле2.equals(other.поле2)
    && ...;
```

9. Если вы переопределяете в подклассе метод `equals()`, в него следует включить вызов `super.equals(other)`.



СОВЕТ. Если имеются поля типа массива, для проверки на равенство соответствующих элементов массива можно воспользоваться статическим методом `Arrays.equals()`.



ВНИМАНИЕ! Реализуя метод `equals()`, многие программисты допускают типичную ошибку. Сможете ли вы сами выяснить, какая ошибка возникнет при выполнении следующего фрагмента кода?

```
public class Employee
{
    public boolean equals(Employee other)
    {
        return other != null
            && getClass() == other.getClass()
            && Objects.equals(name, other.name)
            && salary == other.salary
            && Objects.equals(hireDay, other.hireDay);
    }
    . . .
}
```

В этом методе тип явного параметра определяется как `Employee`. В итоге переопределяется не метод `equals()` из класса `Object`, а совершенно посторонний метод. Чтобы застраховаться от подобной ошибки, следует специально пометить аннотацией `@Override` метод, который переопределяет соответствующий метод из суперкласса, как показано ниже.

```
@Override public boolean equals(Object other)
```

Если при этом будет случайно определен новый метод, компилятор возвратит сообщение об ошибке. Допустим, в классе `Employee` имеется такая строка кода:

```
@Override public boolean equals(Employee other)
```

Этот метод не переопределяет ни один из методов суперкласса `Object`, поэтому будет обнаружена ошибка.

`java.util.Arrays 1.2`

- `static boolean equals(XXX[] a, XXX[] b)` 5.0

Возвращает логическое значение `true`, если сравниваемые массивы имеют одинаковую длину и одинаковые элементы на соответствующих позициях. Сравниваемые массивы могут содержать элементы типа `Object`, `int`, `long`, `short`, `char`, `byte`, `boolean`, `float` или `double`.

java.util.Objects 7

- **static boolean equals(Object a, Object b)**

Возвращает логическое значение **true**, если оба параметра, **a** и **b**, имеют пустое значение **null**; логическое значение **false**, если один из них имеет пустое значение **null**; а иначе — результат вызова **a.equals(b)**.

5.2.4. Метод hashCode()

Хеш-код — это целое число, генерируемое на основе конкретного объекта. Хеш-код можно рассматривать как некоторый шифр: если **x** и **y** — разные объекты, то с большой степенью вероятности должны различаться результаты вызовов **x.hashCode()** и **y.hashCode()**. В табл. 5.1 приведено несколько примеров хеш-кодов, полученных в результате вызова метода **hashCode()** из класса **String**.

Таблица 5.1. Хеш-коды, получаемые с помощью метода **hashCode()**

Символьная строка	Хеш-код
Hello	69609650
Harry	69496448
Hacker	-2141031506

Для вычисления хеш-кода в классе **String** применяется следующий алгоритм:

```
int hash = 0;
for (int i = 0; i < length(); i++)
    hash = 31 * hash + charAt(i);
```

Метод **hashCode()** определен в классе **Object**. Поэтому у каждого объекта имеет-ся хеш-код, определяемый по умолчанию. Этот хеш-код вычисляется по адресу памяти, занимаемой объектом. Рассмотрим следующий пример кода:

```
String s = "Ok";
StringBuilder sb = new StringBuilder(s);
System.out.println(s.hashCode() + " " + sb.hashCode());
String t = new String("Ok");
StringBuilder tb = new StringBuilder(t);
System.out.println(t.hashCode() + " " + tb.hashCode());
```

Результаты выполнения этого фрагмента кода приведены в табл. 5.2.

Таблица 5.2. Хеш-коды для объектов типа **String** и **StringBuffer**

Объект	Хеш-код	Объект	Хеш-код
s	2556	t	2556
sb	20526976	tb	20527144

Обратите внимание на то, что символьным строкам **s** и **t** соответствуют одинаковые хеш-коды, поскольку они вычисляются на основе *содержимого* строкового объекта. А у строкопостроителей **sb** и **tb** хеш-коды отличаются. Дело в том, что в классе **StringBuilder** метод **hashCode()** не определен, и поэтому из класса **Object** вызывается исходный метод **hashCode()**, где хеш-код определяется по адресу памяти, занимаемой объектом.

Если переопределяется метод `equals()`, то следует переопределить и метод `hashCode()` для объектов, которые пользователи могут вставлять в хеш-таблицу. (Подробнее хеш-таблицы будут обсуждаться в главе 9.)

Метод `hashCode()` должен возвращать целочисленное значение, которое может быть отрицательным. Чтобы хеш-коды разных объектов отличались, достаточно объединить хеш-коды полей экземпляра. Ниже приведен пример реализации метода `hashCode()` в классе `Employee`.

```
class Employee
{
    public int hashCode()
    {
        return 7 * name.hashCode()
            + 11 * new Double(salary).hashCode()
            + 13 * hireDay.hashCode();
    }
    . . .
}
```

Но то же самое можно сделать лучше. Во-первых, можно воспользоваться методом `Objects.hashCode()`, безопасно обрабатывающим пустые значения. В частности, он возвращает нуль, если его аргумент имеет пустое значение `null`, а иначе — результат вызова метода `hashCode()` для заданного аргумента. Чтобы не создавать объект `Double`, можно также вызвать статический метод `Double.hashCode()`, как показано ниже.

```
public int hashCode()
{
    return 7 * Objects.hashCode(name)
        + 11 * new Double(salary).hashCode()
        + 13 * Objects.hashCode(hireDay);
}
```

И во-вторых, можно вызвать метод `Objects.hash()`, если требуется объединить несколько хеш-значений, что еще лучше. В этом случае метод `Objects.hashCode()` будет вызван для каждого аргумента с целью объединить получаемые в итоге хеш-значения. В таком случае метод `Employee.hashCode()` реализуется очень просто:

```
public int hashCode()
{
    return Objects.hash(name, salary, hireDay);
}
```

Методы `equals()` и `hashCode()` должны быть совместимы: если в результате вызова `x.equals(y)` возвращается логическое значение `true`, то и результаты вызовов `x.hashCode()` и `y.hashCode()` также должны совпадать. Так, если в методе `Employee.equals()` сравниваются идентификационные номера работников, то при вычислении хеш-кода методу `hashCode()` также потребуются идентификационные номера, но не имя работника и не адрес памяти, занимаемой соответствующим объектом.



СОВЕТ. Если имеются поля типа массива, для вычисления хеш-кода, состоящего из хеш-кодов элементов массива, можно воспользоваться методом `Arrays.hashCode()`.

java.lang.Object 1.0

- **int hashCode()**

Возвращает хеш-код объекта. Хеш-код представляет собой положительное или отрицательное целое число. Для равнозначных объектов должны возвращаться одинаковые хеш-коды.

java.lang.Objects 7

- **static int hash(Object... *объекты*)**

Возвращает хеш-код, состоящий из хеш-кодов всех предоставляемых объектов.

- **static int hashCode(Object *a*)**

Возвращает нуль, если параметр *a* имеет пустое значение *null*, а иначе — делает вызов *a.hashCode()*.

java.lang. (Integer|Long|Short|Byte|Double|Float|Character|Boolean) 1.0

- **static int hashCode((int|long|short|byte|double|float|char|boolean) *value*) 8**

Возвращает хеш-код заданного значения.

java.util.Arrays 1.2

- **static int hashCode(*type[] a*) 5.0**

Вычисляет хеш-код массива *a*, который может содержать элементы типа *Object*, *int*, *long*, *short*, *char*, *byte*, *boolean*, *float* или *double*.

5.2.5. Метод toString()

Еще одним важным в классе *Object* является метод *toString()*, возвращающий значение объекта в виде символьной строки. В качестве примера можно привести метод *toString()* из класса *Point*, который возвращает символьную строку, подобную приведенной ниже.

```
java.awt.Point[x=10,y=20]
```

Большая часть, но не все реализации метода *toString()* возвращают символьную строку, состоящую из имени класса, после которого следуют значения его полей в квадратных скобках. Ниже приведен пример реализации метода *toString()* в классе *Employee*.

```
public String toString()
{
    return "Employee[name=" + name
        + ",salary=" + salary
```



```

    + ",hireDay=" + hireDay
    + "];";
}

```

На самом деле этот метод можно усовершенствовать. Вместо жесткого кодирования имени класса в методе `toString()` достаточно вызвать метод `getClass().getName()` и получить символьную строку, содержащую имя класса:

```

public String toString()
{
    return getClass().getName()
        + "[name=" + name
        + ",salary=" + salary
        + ",hireDay=" + hireDay
        + "];";
}

```

Такой метод `toString()` пригоден для подклассов. Разумеется, при создании подкласса следует определить собственный метод `toString()` и добавить поля подкласса. Так, если в суперклассе делается вызов `getClass().getName()`, то в подклассе просто вызывается метод `super.toString()`. Ниже приведен пример реализации метода `toString()` в классе `Manager`.

```

class Manager extends Employee
{
    . . .
    public String toString()
    {
        return super.toString()
            + "[bonus=" + bonus
            + "];";
    }
}

```

Теперь состояние объекта типа `Manager` выводится следующим образом:

```
Manager[name=...,salary=...,hireDay=...][bonus=...]
```

Метод `toString()` универсален. Имеется следующее веское основание для его реализации в каждом классе: если объект объединяется с символьной строкой с помощью операции `+`, то компилятор Java автоматически вызывает метод `toString()`, чтобы получить строковое представление этого объекта:

```

var p = new Point(10, 20);
String message = "The current position is " + p;
// метод p.toString() вызывается автоматически

```



СОВЕТ. Вместо вызова `x.toString()` можно воспользоваться выражением `" " + x`. Сцепление пустой символьной строки со строковым представлением в переменной `x` равнозначно вызову метода `x.toString()`. Такое выражение будет корректным, даже если переменная `x` относится к одному из примитивных типов.

Если `x` — произвольный объект и в программе имеется следующая строка кода:

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

то из метода `println()` будет вызван метод `x.toString()` и выведена символьная строка результата. Метод `toString()`, определенный в классе `Object`, выводит имя класса и адрес объекта. Рассмотрим следующий вызов:

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

После выполнения метода `println()` отображается такая строка:

```
java.io.PrintStream@2f6684
```

Как видите, разработчики класса `PrintStream` не позаботились о переопределении метода `toString()`.



ВНИМАНИЕ! Как ни досадно, но массивы наследуют метод `toString()` от класса `Object`, в результате чего тип массива выводится в архаичном формате. Например, при выполнении приведенного ниже фрагмента кода получается символьная строка "[I@1a46e30", где префикс `[I` означает массив целых чисел.

```
int[] luckyNumbers = { 2, 3, 5, 7, 11, 13 };
String s = "" + luckyNumbers;
```

В качестве выхода из этого неприятного положения можно вызвать статический метод `Arrays.toString()`. Так, при выполнении следующего фрагмента кода будет получена символьная строка "[2, 3, 5, 7, 11, 13]":

```
String s = Arrays.toString(luckyNumbers);
```

А для корректного вывода многомерных массивов следует вызывать метод `Arrays.deepToString()`.

Метод `toString()` отлично подходит для регистрации и протоколирования. Он определен во многих классах из стандартной библиотеки Java, что позволяет получать полезные сведения о состоянии объекта. Так, например, для регистрации сообщений можно применить следующее выражение:

```
System.out.println("Current position = " + position);
```

Как будет показано в главе 7, существует более совершенное решение, состоящее в том, чтобы воспользоваться объектом класса `Logger` и сделать следующий вызов:

```
Logger.global.info("Current position = " + position);
```



СОВЕТ. Настоятельно рекомендуется переопределять метод `toString()` в каждом создаваемом вами классе. Это будет полезно как вам, так и тем, кто пользуется плодами ваших трудов.

В примере программы из листинга 5.8 демонстрируется применение методов `equals()`, `hashCode()` и `toString()`, реализованных в классах `Employee` (из листинга 5.9) и `Manager` (из листинга 5.10).

Листинг 5.8. Исходный код из файла `equals/EqualsTest.java`

```
1 package equals;
2
3 /**
4  * В этой программе демонстрируется
5  * применение метода equals()
6  * @version 1.12 2012-01-26
```

```
7  * @author Cay Horstmann
8  */
9  public class EqualsTest
10 {
11     public static void main(String[] args)
12     {
13         Employee alicel = new Employee("Alice Adams",
14                                         75000, 1987, 12, 15);
15         Employee alicel2 = alicel;
16         Employee alicel3 = new Employee("Alice Adams",
17                                         75000, 1987, 12, 15);
18         Employee bob = new Employee("Bob Brandson",
19                                     50000, 1989, 10, 1);
20
21         System.out.println("alicel == alicel2: "
22                             + (alicel == alicel2));
23
24         System.out.println("alicel == alicel3: "
25                             + (alicel == alicel3));
26
27         System.out.println("alicel.equals(alicel3): "
28                             + alicel.equals(alicel3));
29
30         System.out.println("alicel.equals(bob): "
31                             + alicel.equals(bob));
32
33         System.out.println("bob.toString(): " + bob);
34
35         Manager carl = new Manager("Carl Cracker",
36                                    80000, 1987, 12, 15);
37         Manager boss = new Manager("Carl Cracker",
38                                    80000, 1987, 12, 15);
39         boss.setBonus(5000);
40         System.out.println("boss.toString(): " + boss);
41         System.out.println("carl.equals(boss): "
42                             + carl.equals(boss));
43         System.out.println("alicel.hashCode(): "
44                             + alicel.hashCode());
45         System.out.println("alicel3.hashCode(): "
46                             + alicel3.hashCode());
47         System.out.println("bob.hashCode(): "
48                             + bob.hashCode());
49         System.out.println("carl.hashCode(): "
50                             + carl.hashCode());
51     }
52 }
```

Листинг 5.9. Исходный код из файла `equals/Employee.java`

```
1  package equals;
2
3  import java.time.*;
4  import java.util.Objects;
5
6  public class Employee
```

```
7 {
8     private String name;
9     private double salary;
10    private LocalDate hireDay;
11
12    public Employee(String name, double salary,
13                    int year, int month, int day)
14    {
15        this.name = name;
16        this.salary = salary;
17        hireDay = LocalDate.of(year, month, day);
18    }
19
20    public String getName()
21    {
22        return name;
23    }
24
25    public double getSalary()
26    {
27        return salary;
28    }
29
30    public LocalDate getHireDay()
31    {
32        return hireDay;
33    }
34
35    public void raiseSalary(double byPercent)
36    {
37        double raise = salary * byPercent / 100;
38        salary += raise;
39    }
40
41    public boolean equals(Object otherObject)
42    {
43        // быстро проверить объекты на идентичность
44        if (this == otherObject) return true;
45
46        // если явный параметр имеет пустое значение null,
47        // должно быть возвращено логическое значение false
48        if (otherObject == null)
49            return false;
50
51        // если классы не совпадают, они не равны
52        if (getClass() != otherObject.getClass())
53            return false;
54
55        // теперь известно, что otherObject - это
56        // непустой объект типа Employee
57        var other = (Employee) otherObject;
58
59        // проверить, содержат ли поля одинаковые значения
60        return Objects.equals(name, other.name)
61            && salary == other.salary
62            && Objects.equals(hireDay, other.hireDay);
```

```

63     }
64
65     public int hashCode()
66     {
67         return Objects.hash(name, salary, hireDay);
68     }
69
70     public String toString()
71     {
72         return getClass().getName() + "[name=" + name
73             + ",salary=" + salary + ",hireDay="
74             + hireDay + "]";
75     }
76 }

```

Листинг 5.10. Исходный код из файла `equals/Manager.java`

```

1 package equals;
2
3 public class Manager extends Employee
4 {
5     private double bonus;
6
7     public Manager(String name, double salary,
8                     int year, int month, int day)
9     {
10         super(name, salary, year, month, day);
11         bonus = 0;
12     }
13
14     public double getSalary()
15     {
16         double baseSalary = super.getSalary();
17         return baseSalary + bonus;
18     }
19
20     public void setBonus(double bonus)
21     {
22         this.bonus = bonus;
23     }
24
25     public boolean equals(Object otherObject)
26     {
27         if (!super.equals(otherObject)) return false;
28         var other = (Manager) otherObject;
29         // В методе super.equals() проверяется,
30         // принадлежат ли объекты, доступные по
31         // ссылкам this и other, одному и тому же классу
32         return bonus == other.bonus;
33     }
34
35     public int hashCode()
36     {
37         return java.util.Objects.hash(super.hashCode(),
38                                         bonus);

```

```
39  }
40
41  public String toString()
41  {
42      return super.toString() + "[bonus=" + bonus + "];";
43  }
44 }
```

java.lang.Object 1.0

- **Class getClass()**
Возвращает класс объекта, содержащий сведения об объекте. Как будет показано далее в этой главе, в Java поддерживается динамическое представление классов, инкапсулированное в классе **Class**.
- **boolean equals(Object otherObject)**
Сравнивает два объекта и возвращает логическое значение **true**, если объекты занимают одну и ту же область памяти, а иначе — логическое значение **false**. Этот метод следует переопределить при создании собственных классов.
- **String toString()**
Возвращает символьную строку, представляющую значение объекта. Этот метод следует переопределить при создании собственных классов.

java.lang.Class 1.0

- **String getName()**
Возвращает имя класса.
- **Class getSuperclass()**
Возвращает имя суперкласса данного класса в виде объекта типа **Class**.

5.3. Обобщенные списочные массивы

В ряде языков программирования (например, C и C++) размер всех массивов должен задаваться еще на стадии компиляции программы. Это ограничение существенно затрудняет работу программиста. Сколько работников требуется в отделе? Очевидно, не больше 100 человек. А что, если где-то есть отдел, насчитывающий 150 работников, или же если отдел невелик и в нем работают только 10 человек? Ведь в этом случае 90 элементов массива выделяется на 10 работников, т.е. он используется только на 10%!

В языке Java ситуация намного лучше — размер массива можно задавать во время выполнения, как показано ниже.

```
int actualSize = ...;
var staff = new Employee[actualSize];
```

Разумеется, этот код не решает проблему динамической модификации массивов во время выполнения. Задав размер массива, его затем нелегко изменить. Это

затруднение проще всего разрешить, используя списочные массивы. Для их создания применяются экземпляры класса `ArrayList`. Этот класс действует подобно обычному массиву, но может динамически изменять его размеры по мере добавления новых элементов или удаления существующих, не требуя написания дополнительного кода.

Класс `ArrayList` является *обобщенным с параметром типа*. Тип элемента массива заключается в угловые скобки и добавляется к имени класса: `ArrayList<Employee>`. Подробнее о создании собственных обобщенных классов речь пойдет в главе 8, но пользоваться объектами типа `ArrayList` можно, и не зная все эти технические подробности. В последующих разделах поясняется, как обращаться со списочными массивами.

5.3.1. Объявление списочных массивов

В приведенной ниже строке кода создается списочный массив, предназначенный для хранения объектов типа `Employee`.

```
ArrayList<Employee> staff = new ArrayList<Employee>();
```

Начиная с версии Java 10, целесообразно пользоваться ключевым словом `var` во избежание дублирования имени класса:

```
var staff = new ArrayList<Employee>();
```

Если не пользоваться ключевым словом `var`, то в правой части выражения можно опустить параметр типа, как демонстрируется в следующем примере:

```
ArrayList<Employee> staff = new ArrayList<>();
```

Это так называемый *ромбовидный* оператор, потому что угловые скобки, `<>`, напоминают ромб. Синтаксис ромбовидного оператора используется вместе с операцией `new`. Компилятор проверяет, что именно происходит с новым значением. Если оно присваивается переменной, передается методу или возвращается из метода, то компилятор сначала проверяет обобщенный тип переменной, параметра или метода, а затем заключает этот тип в угловые скобки. В данном примере новое значение `new ArrayList<>()` присваивается переменной типа `ArrayList<Employee>`. Следовательно, тип `Employee` становится обобщенным.



ВНИМАНИЕ! Если списочный массив типа `ArrayList` объявляется с помощью ключевого слова `var`, то пользоваться ромбовидным синтаксисом не следует. Так, следующее объявление:

```
var elements = new ArrayList<>();
```

даст в итоге списочный массив типа `ArrayList<Object>`.



НА ЗАМЕТКУ! До версии Java 5.0 обобщенные классы отсутствовали. Вместо них применялся единственный класс `ArrayList`, который позволял хранить объекты типа `Object`. Если вы работаете со старыми версиями Java, не добавляйте к имени `ArrayList` суффикс в угловых скобках `<...>`, пользуясь и дальше именем `ArrayList` без суффикса `<...>`. Тип `ArrayList` можно рассматривать как базовый, или так называемый “сырой” тип со стертým параметром типа.



НА ЗАМЕТКУ! В еще более старых версиях Java для создания массивов, размеры которых динамически изменялись, программисты пользовались классом `Vector`. Но класс `ArrayList` эффективнее. С его появлением отпала необходимость пользоваться классом `Vector`.

Для добавления новых элементов в списочный массив служит метод `add()`. Ниже приведен фрагмент кода, используемый для заполнения такого массива объектами типа `Employee`.

```
staff.add(new Employee("Harry Hacker", . . .));  
staff.add(new Employee("Tony Tester", . . .));
```

Списочный массив управляет внутренним массивом ссылок на объекты. В конечном итоге элементы массива могут оказаться исчерпанными. И здесь на помощь приходят специальные средства списочного массива. Так, если метод `add()` вызывается при заполненном внутреннем массиве, из списочного массива автоматически создается массив большего размера, куда копируются все объекты.

Если заранее известно, сколько элементов требуется хранить в массиве, то перед заполнением списочного массива достаточно вызвать метод `ensureCapacity()` следующим образом:

```
staff.ensureCapacity(100);
```

При вызове этого метода выделяется память для внутреннего массива, состоящего из 100 объектов. Затем можно вызвать метод `add()`, который уже не будет испытывать затруднений при перераспределении памяти. Первоначальную емкость списочного массива можно передать конструктору класса `ArrayList` в качестве параметра:

```
ArrayList<Employee> staff = new ArrayList<Employee>(100);
```



ВНИМАНИЕ! Выделение памяти для списочного и обычного массивов происходит по-разному. Так, приведенные ниже выражения не равнозначны.

```
new ArrayList<Employee>(100) // емкость списочного массива равна 100  
new Employee[100] // размер обычного массива равен 100
```

Между емкостью списочного массива и размером обычного массива имеется существенное отличие. Если выделить память для массива из 100 элементов, она будет зарезервирована для дальнейшего использования именно такого количества элементов массива. В то же время емкость списочного массива — это всего лишь возможная величина. В ходе работы она может быть увеличена (за счет выделения дополнительной памяти), но сразу после создания списочный массив не содержит ни одного элемента.

Метод `size()` возвращает фактическое количество элементов в списочном массиве. Например, в результате следующего вызова возвращается текущее количество элементов в массиве `staff`:

```
staff.size()
```

Равнозначное выражение для определения размера обычного массива `a` выглядит таким образом:

```
a.length
```

Если вы уверены, что списочный массив будет иметь постоянный размер, можете вызвать метод `trimToSize()`, который устанавливает размер блока памяти таким образом, чтобы он точно соответствовал количеству хранимых элементов. Система сборки “мусора” предотвращает неэффективное использование памяти, освобождая ее излишки.

Если добавить новые элементы в списочный массив после усечения его размера методом `trimToSize()`, блок памяти будет перемещен, на что потребуются

дополнительное время. Поэтому вызывать данный метод следует лишь в том случае, когда точно известно, что дополнительные элементы в списочный массив вводиться не будут.



НА ЗАМЕТКУ C++! Класс `ArrayList` действует аналогично шаблону `vector` в C++. В частности, класс `ArrayList` и шаблон `vector` относятся к обобщенным типам. Но в шаблоне `vector` происходит перегрузка операции `[]`, что упрощает доступ к элементам массива. А в Java перегрузка операций не предусмотрена, поэтому методы следует вызывать явным образом. Кроме того, шаблон `vector` в C++ передается по значению. Если `a` и `b` — два вектора, то выражение `a = b` приведет к созданию нового вектора, длина которого равна `b`. Все элементы вектора `b` будут скопированы в вектор `a`. В результате выполнения того же самого выражения в Java переменные `a` и `b` будут ссылаться на один и тот же списочный массив.

`java.util.ArrayList<T>` 1.2

- **`ArrayList<T>()`**
Конструирует пустой списочный массив.
- **`ArrayList<T>(int initialCapacity)`**
Конструирует пустой списочный массив заданной емкости.
- **`boolean add(T obj)`**
Добавляет элемент в конец массива. Всегда возвращает логическое значение `true`.
- **`int size()`**
Возвращает количество элементов, хранящихся в списочном массиве. (Количество элементов отличается от емкости массива и не превосходит ее.)
- **`void ensureCapacity(int capacity)`**
Обеспечивает емкость списочного массива, достаточную для хранения заданного количества элементов без изменения внутреннего массива, предназначенного для хранения данных в памяти.
- **`void trimToSize()`**
Сокращает емкость списочного массива до его текущего размера.

5.3.2. Доступ к элементам списочных массивов

К сожалению, ничто не дается бесплатно. За удобство, предоставляемое автоматическим регулированием размера списочного массива, приходится расплачиваться более сложным синтаксисом, который требуется для доступа к его элементам. Дело в том, что класс `ArrayList` не входит в состав Java, а является лишь служебным классом, специально введенным в стандартную библиотеку этого языка.

Вместо удобных квадратных скобок для доступа к элементам списочного массива приходится вызывать методы `get()` и `set()`. Например, для установки *i*-го элемента списочного массива служит следующее выражение:

```
staff.set(i, harry);
```

Это равнозначно приведенному ниже выражению для установки *i*-го элемента обычного массива `a`. Как в обычных, так и в списочных массивах индексы отсчитываются от нуля.

```
a[i] = harry;
```



ВНИМАНИЕ! Не вызывайте метод `list.set(i, x)` до тех пор, пока размер списочного массива больше `i`. Например, следующий код написан неверно:

```
ArrayList<Employee> list = new ArrayList<Employee>(100);
    // емкость списочного массива равна 100,
    // а его размер равен 0
list.set(0, x); // нулевого элемента в списочном
               // массиве пока еще нет
```

Для заполнения списочного массива вызывайте метод `add()` вместо метода `set()`, а последний применяйте только для замены ранее введенного элемента.

Получить элемент списочного массива можно с помощью метода `get()`:

```
Employee e = staff.get(i);
```

Это равнозначно следующему выражению для обращения к массиву `a`:

```
Employee e = a[i];
```



НА ЗАМЕТКУ! Когда обобщенные классы отсутствовали, единственным вариантом возврата значения из метода `get()` базового типа `ArrayList` была ссылка на объект класса `Object`. Очевидно, что в вызывающем методе приходилось выполнять приведение типов следующим образом:

```
Employee e = (Employee) staff.get(i);
```

При использовании класса базового типа `ArrayList` возможны неприятные ситуации, связанные с тем, что его методы `add()` и `set()` допускают передачу параметра любого типа. Так, приведенный ниже вызов воспринимается компилятором как правильный.

```
staff.set(i, "Harry Hacker");
```

Серьезные осложнения могут возникнуть лишь после того, когда вы извлечете объект и попытаетесь привести его к типу `Employee`. А если вы воспользуетесь обобщением `ArrayList<Employee>`, то ошибка будет выявлена уже на стадии компиляции.

Иногда гибкость и удобство доступа к элементам удастся сочетать, пользуясь следующим приемом. Сначала создается список массивов, и в него добавляются все нужные элементы:

```
ArrayList<X> list = new ArrayList<X>();
while (...)
{
    x = ...;
    list.add(x);
}
```

Затем вызывается метод `toArray()` для копирования всех элементов в массив:

```
X[] a = new X[list.size()];
list.toArray(a);
```

Элементы можно добавлять не только в конец списочного массива, но и в его середину:

```
int n = staff.size() / 2;
staff.add(n, e);
```

Элемент по индексу `n` и следующие за ним элементы сдвигаются, чтобы освободить место для нового элемента. Если после вставки элемента новый размер списочного массива превышает его емкость, происходит копирование массива.

Аналогично можно удалить элемент из середины списочного массива следующим образом:

```
Employee e = staff.remove(n);
```

Элементы, следующие после удаленного элемента, сдвигаются влево, а размер списочного массива уменьшается на единицу. Вставка и удаление элементов списочного массива не особенно эффективна. Для массивов, размеры которых невелики, это не имеет особого значения. Но если при обработке больших объемов данных приходится часто вставлять и удалять элементы, попробуйте вместо списочного массива воспользоваться связным списком. Особенности программирования связных списков рассматриваются в главе 9.

Для перебора содержимого списочного массива можно также организовать цикл в стиле `for each` следующим образом:

```
for (Employee e : staff)
    сделать что-нибудь с переменной e
```

Выполнение этого цикла дает такой же результат, как и выполнение цикла

```
for (int i = 0; i < staff.size(); i++)
{
    Employee e = staff.get(i);
    сделать что-нибудь с переменной e
}
```

В листинге 5.11 приведен исходный код видоизмененной версии программы `EmployeeTest` из главы 4. Обычный массив `Employee[]` в ней заменен обобщенным списочным массивом `ArrayList<Employee>`. Обратите внимание на следующие особенности данной версии программы.

- Не нужно задавать размер массива.
- С помощью метода `add()` в массив можно добавлять сколько угодно элементов.
- Вместо свойства `length` для подсчета количества элементов в массиве служит метод `size()`.
- Вместо выражения `a[i]` для доступа к элементу массива вызывается метод `a.get(i)`.

Листинг 5.11. Исходный код из файла `arrayList/ArrayListTest.java`

```
1 package arrayList;
2
3 import java.util.*;
4
5 /**
6  * В этой программе демонстрируется
7  * применение класса ArrayListTest
8  * @version 1.11 2012-01-26
9  * @author Cay Horstmann
10 */
11 public class ArrayListTest
12 {
13     public static void main(String[] args)
14     {
15         // заполнить списочный массив staff тремя
```

```
16 // объектами типа Employee
17 ArrayList<Employee> staff = new ArrayList<>();
18
19 staff.add(new Employee("Carl Cracker", 75000,
20                        1987, 12, 15));
21 staff.add(new Employee("Harry Hacker", 50000,
22                        1989, 10, 1));
23 staff.add(new Employee("Tony Tester",
24                        40000, 1990, 3, 15));
25
26 // поднять всем работникам зарплату на 5%
27 for (Employee e : staff)
28     e.raiseSalary(5);
29
30 // вывести данные обо всех объектах типа Employee
31 for (Employee e : staff)
32     System.out.println("name=" + e.getName()
33                        + ",salary=" + e.getSalary()
34                        + ",hireDay=" + e.getHireDay());
36 }
37 }
```

java.util.ArrayList<T> 1.2

- **void set(int index, T obj)**
Устанавливает значение в элементе списочного массива по указанному индексу, заменяя предыдущее его содержимое.
- **T get(int index)**
Извлекает значение, хранящееся в элементе списочного массива по указанному индексу.
- **void add(int index, T obj)**
Сдвигает существующие элементы списочного массива для вставки нового элемента.
- **T remove(int index)**
Удаляет указанный элемент и сдвигает следующие за ним элементы. Возвращает удаленный элемент.

5.3.3. Совместимость типизированных и базовых списочных массивов

Ради дополнительной безопасности в своем коде следует всегда пользоваться параметрами типа. В этом разделе поясняется, каким образом достигается совместимость с унаследованным кодом, в котором параметры типа не применяются.

Допустим, имеется следующий класс, унаследованный из прежней версии программы:

```
public class EmployeeDB
{
    public void update(ArrayList list) { ... }
    public ArrayList find(String query) { ... }
}
```

В качестве параметра при вызове метода `update()` можно указать типизированный списочный массив без всякого приведения типов:

```
ArrayList<Employee> staff = ...;  
employeeDB.update(staff);
```

В этом случае объект `staff` просто передается методу `update()`.



ВНИМАНИЕ! Несмотря на то что компилятор не обнаружит в приведенном выше фрагменте кода ошибки и даже не выведет предупреждающее сообщение, такой подход нельзя считать полностью безопасным. Метод `update()` может добавлять в списочный массив элементы, типы которых отличаются от `Employee`. Это настораживает, но если хорошенько подумать, то именно такое поведение было характерно для кода до внедрения обобщений в Java. И хотя целостность виртуальной машины Java при этом не нарушается, а безопасность обеспечивается, в то же время теряются преимущества контроля типов на стадии компиляции.

С другой стороны, если попытаться присвоить списочный массив базового типа `ArrayList` типизированному массиву, как показано ниже, то компилятор выдаст соответствующее предупреждение.

```
ArrayList<Employee> result = employeeDB.find(query);  
// выдается предупреждение
```



НА ЗАМЕТКУ! Чтобы увидеть текст предупреждающего сообщения, при вызове компилятора следует указать параметр `-Xlint:unchecked`.

Попытка выполнить приведение типов не исправит ситуацию, как показано ниже. Изменится лишь само предупреждающее сообщение, на этот раз оно уведомит о неверном приведении типов.

```
ArrayList<Employee> result = (ArrayList<Employee>) employeeDB.find(query);  
// на этот раз появится другое предупреждение
```

Это происходит из-за некоторых неудачных ограничений, накладываемых на обобщенные типы в Java. Ради совместимости компилятор преобразует типизированные списочные массивы в объекты базового типа `ArrayList` после того, как будет проверено, соблюдены ли правила контроля типов. В процессе выполнения программы все списочные массивы одинаковы: виртуальная машина не получает никаких данных о параметрах типа. Поэтому приведение типов (`ArrayList`) и (`ArrayList<Employee>`) проходит одинаковую проверку во время выполнения.

В подобных случаях от вас мало что зависит. При видоизменении унаследованного кода вам остается только следить за сообщениями компилятора, довольствуясь тем, что они не уведомляют о серьезных ошибках.

Удовлетворившись результатами компиляции унаследованного кода, можно поместить переменную, принимающую результат приведения типов, аннотацией `@SuppressWarnings("unchecked")`:

```
@SuppressWarnings("unchecked") ArrayList<Employee> result =  
    (ArrayList<Employee>) employeeDB.find(query);  
// выдается еще одно предупреждение
```

5.4. Объектные оболочки и автоупаковка

Иногда переменные примитивных типов вроде `int` приходится преобразовывать в объекты. У всех примитивных типов имеются аналоги в виде классов. Например,

существует класс `Integer`, соответствующий типу `int`. Такие классы принято называть *объектными оболочками*. Они имеют вполне очевидные имена: `Integer`, `Long`, `Float`, `Double`, `Short`, `Byte`, `Character` и `Boolean`. (У первых шести классов имеется общий суперкласс `Number`.) Классы объектных оболочек являются неизменяемыми. Это означает, что нельзя изменить значение, хранящееся в объектной оболочке после ее создания.

Допустим, в списочном массиве требуется хранить целые числа. К сожалению, с помощью параметра типа в угловых скобках нельзя задать примитивный тип, например, выражение `ArrayList<int>` недопустимо. И здесь на помощь приходит класс объектной оболочки `Integer`. В частности, списочный массив, предназначенный для хранения объектов типа `Integer`, можно объявить следующим образом:

```
var list = new ArrayList<Integer>();
```



ВНИМАНИЕ! Применение объектной оболочки типа `ArrayList<Integer>` оказывается менее эффективным, чем массива `int[]`. Причина очевидна: каждое целочисленное значение инкапсулировано внутри объекта, и для его записи или извлечения необходимо предпринимать дополнительные действия. Таким образом, применение объектных оболочек оправдано лишь для небольших коллекций, когда удобство работы программиста важнее эффективности работы самой программы.

Правда, в Java имеется удобное языковое средство, позволяющее добавлять и извлекать элементы из массива. Рассмотрим следующую строку кода:

```
list.add(3);
```

которая автоматически преобразуется в строку кода, приведенную ниже. Подобное автоматическое преобразование называется *автоупаковкой*.

```
list.add(Integer.valueOf(3));
```



НА ЗАМЕТКУ! Для такого преобразования больше подходит обозначение автоматическое заключение в оболочку, но понятие упаковки было заимствовано из языка C#.

С другой стороны, если присвоить объект типа `Integer` переменной типа `int`, целочисленное значение будет автоматически извлечено из объекта, т.е. *распаковано*. Иными словами, компилятор преобразует следующую строку кода:

```
int n = list.get(i);
```

в такую:

```
int n = list.get(i).intValue();
```

Автоматическая упаковка и распаковка примитивных типов может выполняться и при вычислении арифметических выражений. Например, операцию инкремента можно применить к переменной, содержащей ссылку на объект типа `Integer`, как показано ниже.

```
Integer n = 3;  
n++;
```

Компилятор автоматически распакует целочисленное значение из объекта, увеличит его на единицу и снова упакует в объект.

На первый взгляд может показаться, что примитивные типы и их объектные оболочки — одно и то же. Их отличие становится очевидным при выполнении операции

проверки на равенство. Как вам должно быть уже известно, при выполнении операции `==` над объектом проверяется, ссылаются ли сравниваемые переменные на один и тот же адрес памяти, где находится объект. Ниже приведен пример, где, несмотря на равенство целочисленных значений, проверка на равенство, вероятнее всего, даст отрицательный результат.

```
Integer a = 1000;  
Integer b = 1000;  
if (a == b) . . .
```

Но реализация Java *может*, если пожелает, заключить часто встречающиеся значения в оболочки одинаковых объектов, и тогда сравнение даст положительный результат. Хотя такая неоднозначность результатов мало кому нужна. В качестве выхода из этого положения можно воспользоваться методом `equals()` при сравнении объектов-оболочек.



НА ЗАМЕТКУ! Спецификация автоупаковки требует, чтобы значения типа `boolean`, `byte`, `char` меньше 127, а также значения типа `short` и `int` в пределах от -128 до 127 упаковывались в фиксированные объекты. Так, если переменные `a` и `b` из предыдущего примера инициализировать значением 100, их сравнение должно дать положительный результат.

У автоупаковки и распаковки имеются и другие особенности. Прежде всего, при автораспаковке может быть сгенерировано исключение, если ссылки на класс оболочки окажутся пустыми (`null`):

```
NullPointerException:  
Integer n = null;  
System.out.println(2 * n);  
// генерируется исключение типа NullPointerException
```

А если в условном выражении употребляются типы `Integer` и `Double`, то значение типа `Integer` распаковывается, продвигается к типу `Double` и снова упаковывается, как демонстрируется в следующем фрагменте кода:

```
Integer n = 1;  
Double x = 2.0;  
System.out.println(true ? n : x); // выводится значение 1.0
```

И, наконец, следует заметить, что за упаковку и распаковку отвечает не виртуальная машина, а компилятор. Он включает в программу необходимые вызовы, а виртуальная машина лишь выполняет байт-код.

Объектные оболочки числовых значений находят широкое распространение еще и по другой причине. Создатели Java решили, что в составе классов объектных оболочек удобно было бы реализовать методы для преобразования символьных строк в числа. Чтобы преобразовать символьную строку в целое число, можно воспользоваться выражением, подобным приведенному ниже.

Обратите внимание на то, что создавать объект типа `Integer` в этом случае совсем не обязательно, потому что метод `parseInt()` является статическим. И тем не менее класс `Integer` — подходящее для этого место.

Ниже описаны наиболее употребительные методы из класса `Integer`. Аналогичные методы имеются и в других классах объектных оболочек для значений примитивных типов.



ВНИМАНИЕ! Некоторые считают, что с помощью классов объектных оболочек можно реализовать методы, модифицирующие свои числовые параметры. Но это неверно. Как пояснялось в главе 4, на Java нельзя написать метод, увеличивающий целое число, передаваемое ему в качестве параметра, поскольку все параметры в этом языке передаются *только* по значению.

```
public static void triple(int x) // не работает!
{
    x++; // попытка модифицировать локальную переменную
}
```

Но, может быть, это ограничение удастся обойти, используя вместо типа `int` класс `Integer`?

```
public static void triple(Integer x) // все равно не работает!
{
    ...
}
```

Дело в том, что объект типа `Integer` не позволяет изменять содержащиеся в нем данные. Следовательно, изменять числовые параметры, передаваемые методам, с помощью классов объектных оболочек нельзя.

Если все-таки требуется создать метод, изменяющий свои числовые параметры, для этого можно воспользоваться одним из контейнерных типов, определенных в пакете `org.omg.CORBA`. К таким типам относятся `IntHolder`, `BooleanHolder` и др. Каждый такой тип содержит открытое (sic!) поле `value`, через которое можно обращаться к хранящемуся в нем числу, как показано ниже.

```
public static void triple(IntHolder x)
{
    x.value++;
}
```

`java.lang.Integer` 1.0

- `int intValue()`

Возвращает значение из данного объекта типа `Integer` в виде числового значения типа `int` (этот метод переопределяет метод `intValue()` из класса `Number`).

- `static String toString(int i)`

Возвращает новый объект типа `String`, представляющий числовое значение в десятичной форме.

- `static String toString(int i, int radix)`

Возвращает новый объект типа `String`, представляющий число в системе счисления, определяемой параметром `radix`.

- `static int parseInt(String s)`

- `static int parseInt(String s, int radix)`

Возвращают целое значение. Предполагается, что объект типа `String` содержит символьную строку, представляющую целое число в десятичной системе счисления (в первом варианте метода) или же в системе счисления, которая задается параметром `radix` (во втором варианте метода).

- `static Integer valueOf(String s)`

- `static Integer valueOf(String s, int radix)`

Возвращают новый объект типа `Integer`, инициализированный целым значением, которое задается с помощью первого параметра. Предполагается, что объект типа `String` содержит символьную строку, представляющую целое число в десятичной системе счисления (в первом варианте метода) или же в системе счисления, которая задается параметром `radix` (во втором варианте метода).


```
java.text.NumberFormat 1.1
```

- **Number parse(String s)**

Возвращает числовое значение, полученное в результате синтаксического анализа параметра. Предполагается, что объект типа **String** содержит символьную строку, представляющую числовое значение.

5.5. Методы с переменным числом параметров

Теперь в Java имеется возможность создавать методы, позволяющие при разных вызовах задавать различное количество параметров. С одним из таких методов, `printf()`, вы уже знакомы. Ниже представлены два примера обращения к нему.

```
System.out.printf("%d", n);
System.out.printf("%d %s", n, "widgets");
```

В обоих приведенных выше строках кода вызывается один и тот же метод, но в первом случае этому методу передаются два параметра, а во втором — три. Метод `printf()` определяется в общей форме следующим образом:

```
public class PrintStream
{
    public PrintStream printf(String fmt, Object... args)
    { return format(fmt, args); }
}
```

Здесь многоточием (...) обозначается часть кода Java. Оно указывает на то, что в дополнение к параметру `fmt` можно указывать любое количество объектов. По существу, метод `printf()` получает два параметра: форматирующую строку и массив типа `Object[]`, в котором хранятся все остальные параметры. (Если этому методу передаются целочисленные значения или же значения одного из примитивных типов, то они преобразуются в объекты путем автоупаковки.) После этого метод решает непростую задачу разбора форматирующей строки `fmt` и связывания спецификаторов формата со значениями параметров `args[i]`. Иными словами, в методе `printf()` тип параметра `Object...` означает то же самое, что и `Object[]`.

Компилятор при обработке исходного кода выявляет каждое обращение к методу `printf()`, размещает параметры в массиве и, если требуется, выполняет автоупаковку:

```
System.out.printf("%d %s", new Object[] { new Integer(n), "widgets" } );
```

При необходимости можно создать собственные методы с переменным числом параметров, указав любые типы параметров, в том числе и примитивные. Ниже приведен пример простого метода, в котором вычисляется и возвращается максимальное из произвольного количества значений.

```
public static double max(double... values)
{
    double largest = Double.MIN_VALUE;
    for (double v : values) if (v > largest) largest = v;
    return largest;
}
```

Если вызвать метод `max()` следующим образом:

```
double m = max(3.1, 40.4, -5);
```

компилятор передаст ему параметры в виде такого выражения:

```
new double[] { 3.1, 40.4, -5 }
```



НА ЗАМЕТКУ! В качестве последнего параметра метода, число параметров которого может быть переменным, допускается задавать массив следующим образом:

```
System.out.printf("%d %s", new Object[] { new Integer(1), "widgets" } );
```

Таким образом, существующий метод, последний параметр которого является массивом, можно переопределить как метод с переменным числом параметров, не изменяя уже имеющийся код. Подобным образом в версии Java 5 был расширен метод `MessageFormat.format()`. При желании метод `main()` можно даже объявить следующим образом:

```
public static void main(String... args)
```

5.6. Классы перечислений

В главе 3 пояснялось, каким образом в Java определяются перечислимые типы. Ниже приведен характерный пример применения перечислимых типов в коде.

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

Тип, объявленный подобным образом, на самом деле представляет собой класс. Допускается существование всего четырех экземпляров указанного класса перечисления. Другие объекты в таком классе создать нельзя.

Таким образом, для проверки перечислимых значений на равенство совсем не обязательно использовать метод `equals()`. Для этой цели вполне подойдет операция `==`. По желанию в классы перечислимых типов можно добавить конструкторы, методы и поля. Очевидно, что конструкторы могут вызываться только при создании констант перечислимого типа. Ниже приведен соответствующий пример.

```
public enum Size
{
    SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");

    private String abbreviation;

    private Size(String abbreviation)
    { this.abbreviation = abbreviation; }
    public String getAbbreviation() { return abbreviation; }
}
```

Конструктор перечисления всегда открытый, поэтому при его вызове можно опустить модификатор доступа `private`, как показано в предыдущем примере. Объявлять конструктор перечисления как `public` или `protected` считается синтаксической ошибкой.

Все перечислимые типы реализуются с помощью подклассов, производных от класса `Enum`. Они наследуют от этого класса ряд методов. Наиболее часто применяется метод `toString()`, возвращающий имя константы перечислимого типа. Так, при вызове метода `Size.SMALL.toString()` возвращается символьная строка "SMALL". Статический метод `valueOf()` выполняет действия, противоположные

методу `toString()`. Например, в результате выполнения приведенной ниже строки кода переменной `s` будет присвоено значение `Size.SMALL`.

```
Size s = (Size) Enum.valueOf(Size.class, "SMALL");
```

Каждый перечислимый тип содержит статический метод `values()`, который возвращает массив всех перечислимых значений. Так, в результате вызова

```
Size[] values = Size.values();
```

возвращается массив констант перечислимого типа `Size.SMALL`, `Size.MEDIUM`, `Size.LARGE` и `Size.EXTRA_LARGE`.

Метод `ordinal()` возвращает позицию константы в объявлении перечислимого типа, начиная с нуля. Например, в результате вызова `Size.MEDIUM.ordinal()` возвратится значение 1. Оперирование перечислимыми типами демонстрируется в короткой программе, приведенной в листинге 5.12.



НА ЗАМЕТКУ! У класса `Enum` имеется также параметр типа, который был ранее опущен ради простоты. Например, перечислимый тип `Size` на самом деле расширяется до типа `Enum<Size>`, но здесь такая возможность не рассматривалась. В частности, параметр типа используется в методе `compareTo()`. (Метод `compareTo()` будет обсуждаться в главе 6, а параметры типа — в главе 8.)

Листинг 5.12. Исходный код из файла `enums/EnumTest.java`

```
1 package enums;
2
3 import java.util.*;
4
5 /**
6  * В этой программе демонстрируются перечислимые типы
7  * @version 1.0 2004-05-24
8  * @author Cay Horstmann
9  */
10 public class EnumTest
11 {
12     public static void main(String[] args)
13     {
14         var in = new Scanner(System.in);
15         System.out.print("Enter a size: "
16             + "(SMALL, MEDIUM, LARGE, EXTRA_LARGE) ");
17         String input = in.next().toUpperCase();
18         Size size = Enum.valueOf(Size.class, input);
19         System.out.println("size=" + size);
20         System.out.println("abbreviation="
21             + size.getAbbreviation());
22         if (size == Size.EXTRA_LARGE)
23             System.out.println("Good job--you "
24                 + "paid attention to the _.");
25     }
26 }
27
28 enum Size
29 {
30     SMALL("S"), MEDIUM("M"), LARGE("L"),
31     EXTRA_LARGE("XL");
```

```
32 private Size(String abbreviation)
33     { this.abbreviation = abbreviation; }
34 public String getAbbreviation()
35     { return abbreviation; }
36 private String abbreviation;
37 }
```

`java.lang.Enum<E>` 5.0

- **static Enum valueOf(Class enumClass, String name)**
Возвращает константу перечислимого типа указанного класса с заданным именем.
- **String toString()**
Возвращает имя константы перечислимого типа.
- **int ordinal()**
Возвращает позицию данной константы в объявлении перечислимого типа, начиная с нуля.
- **int compareTo(E other)**
Возвращает отрицательное целое значение, если константа перечислимого типа следует перед параметром `other`, 0 — если `this == other`, а иначе — положительное целое значение. Порядок следования констант задается в объявлении перечислимого типа.

5.7. Рефлексия

Библиотека *рефлексии* предоставляет богатый набор инструментальных средств для манипулирования кодом Java в динамическом режиме. С помощью рефлексии в Java можно поддерживать построители пользовательских интерфейсов, объектно-реляционные преобразователи и много других инструментальных средств разработки программного обеспечения, запрашивающих в динамическом режиме возможности классов.

Программа, способная анализировать возможности классов, называется *рефлексивной*. Рефлексия — очень мощный механизм, который можно применять для решения перечисленных ниже задач. И в последующих разделах поясняется, как пользоваться этим механизмом, чтобы:

- анализировать возможности классов в процессе выполнения программы;
- проверять объекты при выполнении программы; например, с помощью рефлексии можно реализовать метод `toString()`, совместимый со всеми классами;
- реализовывать обобщенный код для работы с массивами;
- применять объекты типа `Method`, которые работают аналогично указателям на функции в языках, подобных C++.

Рефлексия — не только эффективный, но и сложный механизм. Ею интересуются в основном разработчики инструментальных средств, тогда как программисты, пишущие обычные прикладные программы, зачастую ею не пользуются. Если вы занимаетесь только написанием прикладных программ и еще не готовы к разработке инструментальных средств, пропустите оставшуюся часть этой главы, а в дальнейшем, если потребуется, вернитесь к ее заключительным разделам.

5.7.1. Класс Class

Во время выполнения программы исполняющая система Java всегда осуществляет динамическую идентификацию типов объектов любых классов. Получаемые в итоге сведения используются виртуальной машиной для выбора подходящего вызываемого метода.

Но получить доступ к этой информации можно иначе, используя специальный класс, который так и называется: `Class`. Метод `getClass()` из класса `Object` возвращает экземпляр типа `Class`, как показано ниже.

```
Employee e;  
...  
Class cl = e.getClass();
```

Подобно тому, как объект типа `Employee` описывает свойства конкретного сотрудника, объект типа `Class` описывает свойства конкретного класса. Вероятно, наиболее употребительным в классе `Class` является метод `getName()`, возвращающий имя класса. Например, при выполнении следующей строки кода:

```
System.out.println(e.getClass().getName() + " " + e.getName());
```

выводится символьная строка

```
Employee Harry Hacker
```

если объект `e` относится к классу `Employee`, или же символьная строка

```
Manager Harry Hacker
```

если объект `e` относится к классу `Manager`.

Если же класс находится в пакете, то имя пакета включается в имя класса следующим образом:

```
var generator = new Random();  
Class cl = generator.getClass();  
String name = cl.getName(); // в переменной name  
// устанавливается строковое значение "java.util.Date"
```

Вызывая статический метод `forName()`, можно также получить объект типа `Class`, соответствующий имени класса в строковом представлении:

```
String className = "java.util.Date";  
Class cl = Class.forName(className);
```

Этот метод можно применять, если имя класса хранится в символьной строке, содержимое которой изменяется во время выполнения программы. Он действует правильно, если переменная `className` содержит имя класса или интерфейса. В противном случае метод `forName()` генерирует проверяемое исключение. Порядок вызова обработчика исключений при обращении с этим методом описан далее, в разделе 5.7.2.



СОВЕТ. При запуске программы на выполнение сначала загружается класс, содержащий метод `main()`. Он загружает все необходимые классы. Каждый из классов, в свою очередь, загружает необходимые ему классы и т.д. Если приложение достаточно крупное, этот процесс может отнять немало времени, и пользователю придется ожидать окончания загрузки. Чтобы вызвать у пользователя иллюзию быстрого запуска, можно воспользоваться следующим приемом. Убедитесь в том, что класс, содержащий метод `main()`, не обращается явно к другим классам. Отобразите в этом классе начальный экран приложения. Затем приступайте к загрузке остальных классов, вызывая метод `Class.forName()`.

Третий способ получения объекта типа `Class` прост и удобен. Если `T` — это некоторый тип, то `T.class` — объект соответствующего класса. Например:

```
Class c11 = Date.class; // если произведен импорт пакета java.util.*;
Class c12 = int.class;
Class c13 = Double[].class;
```

Следует, однако, иметь в виду, что объект типа `Class` фактически описывает *тип*, который не обязательно является классом. Например, тип `int` — это не класс, но, несмотря на это, `int.class` — это объект типа `Class`.



НА ЗАМЕТКУ! Начиная с версии Java 5, класс `Class` является параметризованным. Например, ссылка `Employee.class` соответствует типу `Class<Employee>`. Не будем пока что обсуждать этот вопрос, чтобы не усложнять и без того довольно абстрактные понятия. На практике можно вообще игнорировать параметр типа и пользоваться обычным вариантом класса `Class`. Более подробно данный вопрос обсуждается в главе 8.



ВНИМАНИЕ! Исторически сложилось так, что метод `getName()` возвращает для массивов не совсем обычные имена их типов, как показано ниже.

- При вызове `Double[].class.getName()` возвращается символьная строка `"[Ljava.lang.Double;"`.
- При вызове `int[].class.getName()` возвращается символьная строка `"[I"`.

Виртуальная машина Java поддерживает однозначный объект типа `Class` для каждого типа объектов. Следовательно, для сравнения объектов можно воспользоваться операцией `==`, как показано ниже.

```
if (e.getClass() == Employee.class) . . .
```

Приведенная выше проверка проходит, если `e` — экземпляр типа `Employee`. В отличие от условия `e instanceof Employee`, данная проверка не проходит, если `e` — экземпляр подкласса (например, `Manager`).

Если имеется объект типа `Class`, с его помощью можно получить экземпляр класса. Сначала следует вызвать метод `getConstructor()`, чтобы получить объект типа `Constructor`, а затем метод `newInstance()`, чтобы получить экземпляр:

```
var className = "java.util.Random";
// или любое другое имя класса с конструктором без аргументов
Class cl = Class.forName(className);
Object obj = cl.getConstructor().newInstance();
```

Если у класса отсутствует конструктор без аргументов, метод `getConstructor()` сгенерирует исключение. О том, как вызываются другие конструкторы, речь пойдет в разделе 5.7.7.



НА ЗАМЕТКУ! Имеется не рекомендованный для применения метод `Class.newInstance()`, который также позволяет получить экземпляр класса с конструктором без аргументов. Но если такой конструктор сгенерирует проверяемое исключение, то оно будет сгенерировано снова, хотя на этот раз как непроверяемое. Тем самым нарушается порядок проверки исключений на стадии компиляции. А метод `Constructor.newInstance()` заключает любое исключение, генерируемое конструктором, в оболочку исключения типа `InvocationTargetException`.



НА ЗАМЕТКУ C++! Метод `newInstance()` является аналогом виртуального конструктора в C++. И хотя виртуальные конструкторы отсутствуют в C++ как языковое средство, такое понятие реализуется с помощью специализированных библиотек. Класс `Class` соответствует классу `type_info` в C++, а метод `getClass()` — операции `typeid`. Класс `Class` в Java более универсален, чем его аналог в C++. В частности, класс `type_info` позволяет только извлекать символьную строку с именем типа, но не способен создавать объекты данного типа.

`java.lang.Class 1.0`

- **static Class `forName(String className)`**
Возвращает объект типа `Class`, представляющий класс под названием `className`.
- **Constructor `getConstructor(Class... parameterTypes) 1.1`**
Получает объект, описывающий конструктор с заданными типами параметров. Подробнее о том, как предоставлять типы параметров, см. в разделе 5.7.7.

`java.lang.Class 1.0`

- **Object `newInstance(Object... params)`**
Получает новый экземпляр того класса, в котором объявлен конструктор, передавая конструктору заданные параметры `params`. Подробнее о том, как предоставлять типы параметров, см. в разделе 5.7.7.

`java.lang.Throwable 1.0`

- **void `printStackTrace()`**
Выводит объект типа `Throwable` и результат трассировки стека в стандартный поток вывода ошибок.

5.7.2. Основы обработки исключений

Обработка исключений подробно рассматривается в главе 7, но прежде в примерах кода иногда встречаются методы, при выполнении которых могут возникать исключения. Если во время выполнения программы возникает ошибка, программа генерирует исключение. Это более гибкий процесс, чем простое прекращение выполнения программы, поскольку программист может создавать *обработчики исключений*, которые перехватывают исключения и каким-то образом обрабатывают их.

Если обработчик не предусмотрен, система преждевременно завершает выполнение программы и выводит на экран сообщение о типе исключения. Возможно, вы уже сталкивались с подобными сообщениями об исключениях. Так, если вы пытались использовать пустую ссылку или обращались за границы массива, то возникала исключительная ситуация и появлялось соответствующее сообщение.

Существуют две разновидности исключений: *непроверяемые* и *проверяемые*. При возникновении проверяемого исключения компилятор проверяет, предусмотрен ли для него обработчик. Но большинство исключений являются непроверяемыми.

К ним относится, например, исключительная ситуация, возникающая при обращении по пустой ссылке. Компилятор не проверяет, предусмотрен ли в программе обработчик исключений. Вообще говоря, при написании программ следует избегать подобных ошибок, а не предусматривать их обработку ради перестраховки.

Но не всех ошибок можно избежать. Если, несмотря на все ваши усилия, остается фрагмент кода, при выполнении которого может быть сгенерировано исключение, и вы не предусмотрели его обработку, компилятор будет настаивать на том, чтобы вы предоставили соответствующий обработчик. Например, метод `Class.forName()` может сгенерировать проверяемое исключение. В главе 7 мы рассмотрим ряд методик обработки исключений, а пока покажем, как реализуются простейшие обработчики исключений.

Всякий раз, когда метод содержит оператор, который может сгенерировать проверяемое исключение, в объявлении метода следует указать предложение `throws`:

```
public static void doSomethingWithClass(String name)
    throws ReflectiveOperationException
{
    Class cl = Class.forName(name); // может сгенерировать исключение
    сделать что-нибудь с переменной cl
}
```

В объявлении любого метода, где вызывается данный метод, должно быть также указано предложение `throws`. Если исключение все же возникает, то метод `main()` завершается трассировкой стека. (Подробнее о том, как перехватывать исключения, чтобы не допустить преждевременного завершения прикладной программы, речь пойдет в главе 7.)

Для перехвата проверяемых исключений достаточно предоставить предложение `throws`. Метод, ответственный за возникновение исключения, обнаружить нетрудно. Если в программе вызывается метод, который может сгенерировать исключение, а соответствующий обработчик для него не предусмотрен, компилятор выведет соответствующее сообщение.

5.7.3. Ресурсы

С классами нередко связаны файлы данных, в том числе следующие:

- Файлы изображения и звука.
- Текстовые файлы, содержащие строки сообщений и метки экранных кнопок.

Все эти файлы в Java называются *ресурсами*. Рассмотрим в качестве примера диалоговое окно, в котором выводится сообщение (рис. 5.3).

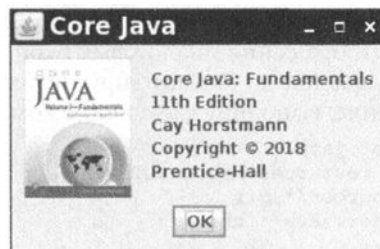


Рис. 5.3. Отображение текстовых и изобразительных ресурсов

Безусловно, название книги и год выпуска, выводимые в диалоговом окне, изменятся в следующем издании книги. Чтобы легко отслеживать такие изменения, текст описания книги следует разместить в отдельном файле, а не встраивать в прикладную программу в виде символической строки. Но где же следует разместить файл `about.txt`? Разумеется, было бы удобно разместить его вместе с остальными файлами прикладной программы, например, в архивном JAR-файле.

Для размещения файлов ресурсов в классе `Class` предоставляются удобные средства. Чтобы воспользоваться ими, необходимо выполнить описанные ниже действия.

1. Получить объект типа `Class`, имеющий ресурс, например `ResourceTest.class`.
2. Некоторые методы (например, метод `getImage()` из класса `ImageIcon`) принимают в качестве параметра `URL`, описывающий местоположение ресурса. В таком случае необходимо сделать следующий вызов:

```
URL url = cl.getResource("about.gif");
```

3. В противном случае следует вызвать метод `getResourceAsStream()`, чтобы получить поток ввода для чтения данных из файла.

Дело в том, что виртуальной машине Java известно, где располагается класс, поэтому она может обнаружить *там же* связанный с ним ресурс. Допустим, класс `ResourceTest` находится в пакете `resources`. В таком случае исходный файл `ResourceTest.class` находится в каталоге `resources`, где обычно размещается и файл изображения с пиктограммой, связанной с данным классом.

Вместо того чтобы размещать файл ресурса в том же каталоге, где и файл класса, можно указать относительный или абсолютный путь к нему:

```
data/about.txt  
/corejava/title.txt
```

Несмотря на то что ресурс загружается автоматически, стандартных методов интерпретации содержимого файла ресурсов не существует. В каждой прикладной программе приходится по-своему интерпретировать данные, находящиеся в подобных файлах.

Кроме того, ресурсы применяются для *интернационализации* прикладных программ. В файлах ресурсов, например, содержатся сообщения и метки на разных языках. Каждый такой файл соответствует отдельному языку. В прикладном интерфейсе API для интернационализации, описываемом в главе 7 второго тома настоящего издания, поддерживается стандартный способ организации и доступа к подобным файлам локализации.

В листинге 5.13 приведен исходный код примера программы, где демонстрируется порядок загрузки ресурсов. (Не обращайтесь пока что особого внимания на фрагмент кода для чтения текста и отображения диалоговых окон, поскольку этот код будет подробно обсуждаться в дальнейшем.) Скомпилируйте, создайте архивный JAR-файл и запустите его на выполнение, выполнив следующие команды:

```
javac resource/ResourceTest.java  
jar cvfe ResourceTest.jar resources.ResourceTest \  
    resources/*.class resources/*.gif  
    resources/data/*.txt corejava/*.txt  
java -jar ResourceTest.jar
```

Переместите полученный в итоге архивный JAR-файл в другой каталог и запустите его снова, чтобы убедиться, что программа прочитает все, что ей требуется, из этого архивного JAR-файла, а не из текущего каталога.

Листинг 5.13. Исходный код из файла `resource/ResourceTest.java`

```
1 package resources;
2
3 import java.io.*;
4 import java.net.*;
5 import java.nio.charset.*;
6 import javax.swing.*;
7
8 /**
9  * @version 1.5 2018-03-15
10  * @author Cay Horstmann
11  */
12 public class ResourceTest
13 {
14     public static void main(String[] args)
15         throws IOException
16     {
17         Class cl = ResourceTest.class;
18         URL aboutURL = cl.getResource("about.gif");
19         var icon = new ImageIcon(aboutURL);
20
21         InputStream stream = cl.getResourceAsStream(
22             "data/about.txt");
23         var about = new String(stream.readAllBytes(),
24             "UTF-8");
25
26         InputStream stream2 = cl.getResourceAsStream(
27             "/corejava/title.txt");
28         var title = new String(stream2.readAllBytes(),
29             StandardCharsets.UTF_8).trim();
30
31         JOptionPane.showMessageDialog(null, about, title,
32             JOptionPane.INFORMATION_MESSAGE, icon);
33     }
34 }
```

java.lang.Class 1.0

- **URL getResource(String name)** 1.1
- **InputStream getResourceAsStream(String name)** 1.1

Находят ресурс там же, где и класс, возвращая его URL или поток ввода, который можно использовать для загрузки. Если ресурс не найден, эти методы возвращают пустое значение `null`, не генерируя исключений и не сообщая об ошибках ввода-вывода.

5.7.4. Анализ функциональных возможностей классов с помощью рефлексии

Ниже приводится краткий обзор наиболее важных характеристик механизма рефлексии, позволяющего анализировать структуру класса.

Три класса, `Field`, `Method` и `Constructor`, из пакета `java.lang.reflect` описывают соответственно поля, методы и конструкторы класса. Все три класса содержат метод `getName()`, возвращающий имя анализируемого класса. В состав класса `Field` входит метод `getType()`, который возвращает объект типа `Class`, описывающий тип

поля. У классов `Method` и `Constructor` имеются методы, определяющие типы параметров, а класс `Method` позволяет также определять тип возвращаемого значения. Все три класса содержат метод `getModifiers()`, возвращающий целое значение, которое соответствует используемым модификаторам доступа, например, `public` или `static`. Для анализа этого числа применяются статические методы класса `Modifiers` из пакета `java.lang.reflect`. В этом классе имеются, в частности, методы `isPublic()`, `isPrivate()` и `isFinal()`, определяющие, является ли анализируемый метод или конструктор открытым, закрытым или конечным. Все, что нужно для этого сделать, — применить соответствующий метод из класса `Modifier` к целому значению, которое возвращается методом `getModifiers()`. Для вывода самих модификаторов служит метод `Modifier.toString()`.

Методы `getFields()`, `getMethods()` и `getConstructors()` из класса `Class` возвращают массивы открытых полей, методов и конструкторов, принадлежащих анализируемому классу. К ним относятся и открытые поля суперклассов. Методы `getDeclaredFields()`, `getDeclaredMethods()` и `getDeclaredConstructors()` из класса `Class` возвращают массивы, состоящие из всех полей, методов и конструкторов, объявленных в классе. К их числу относятся закрытые и защищенные компоненты, но не компоненты суперклассов.

В примере программы из листинга 5.14 показано, как отобразить все сведения о классе. Эта программа предлагает пользователю ввести имя класса, а затем выводит сигнатуры всех методов и конструкторов вместе с именами всех полей класса. Допустим, пользователь ввел следующую команду в режиме командной строки:

`java.lang.Double`

В результате выполнения данной программы на экран будет выведено следующее:

```
public class java.lang.Double extends java.lang.Number
{
    public java.lang.Double(java.lang.String);
    public java.lang.Double(double);

    public int hashCode();
    public int compareTo(java.lang.Object);
    public int compareTo(java.lang.Double);
    public boolean equals(java.lang.Object);
    public java.lang.String toString();
    public static java.lang.String toString(double);
    public static java.lang.Double valueOf(java.lang.String);
    public static boolean isNaN(double);
    public boolean isNaN();
    public static boolean isInfinite(double);
    public boolean isInfinite();
    public byte byteValue();
    public short shortValue();
    public int intValue();
    public long longValue();
    public float floatValue();
    public double doubleValue();
    public static double parseDouble(java.lang.String);
    public static native long doubleToLongBits(double);
    public static native long doubleToRawLongBits(double);
    public static native double longBitsToDouble(long);
```

```
public static final double POSITIVE_INFINITY;
public static final double NEGATIVE_INFINITY;
public static final double NaN;
public static final double MAX_VALUE;
public static final double MIN_VALUE;
public static final java.lang.Class TYPE;
private double value;
private static final long serialVersionUID;
}
```

Эта программа примечательна тем, что способна анализировать любой класс, загружаемый интерпретатором Java, а не только классы, доступные во время компиляции. В следующей главе мы применим ее для анализа внутренних классов, автоматически генерируемых компилятором Java.

Листинг 5.14. Исходный код из файла `reflection/ReflectionTest.java`

```

1 package reflection;
2
3 import java.util.*;
4 import java.lang.reflect.*;
5
6 /**
7  * В этой программе рефлексия применяется для вывода
8  * всех компонентов класса
9  * @version 1.11 2018-03-16
10  * @author Cay Horstmann
11  */
12 public class ReflectionTest
13 {
14     public static void main(String[] args)
15         throws ReflectiveOperationException
16     {
17         // извлечь имя класса из аргументов командной строки или
18         // введенных пользователем данных
19         String name;
20         if (args.length > 0) name = args[0];
21         else
22         {
23             var in = new Scanner(System.in);
24             System.out.println("Enter class name "
25                             + "(e.g. java.util.Date): ");
26             name = in.next();
27         }
28
29         // вывести имя класса и суперкласса (if != Object)
30         Class cl = Class.forName(name);
31         Class supercl = cl.getSuperclass();
32         String modifiers =
33             Modifier.toString(cl.getModifiers());
34         if (modifiers.length() > 0)
35             System.out.print(modifiers + " ");
36         System.out.print("class " + name);
37         if (supercl != null && supercl != Object.class)
38             System.out.print(" extends "
39                             + supercl.getName());
40

```

40

```
41     System.out.print("\n\n");
42     printConstructors(cl);
43     System.out.println();
44     printMethods(cl);
45     System.out.println();
46     printFields(cl);
47     System.out.println("");
48 }
49
50 /**
51  * Выводит все конструкторы класса
52  * @param cl Класс
53  */
54 public static void printConstructors(Class cl)
55 {
56     Constructor[] constructors =
57         cl.getDeclaredConstructors();
58
59     for (Constructor c : constructors)
60     {
61         String name = c.getName();
62         System.out.print(" ");
63         String modifiers = Modifier.toString(
64             c.getModifiers());
65         if (modifiers.length() > 0)
66             System.out.print(modifiers + " ");
67         System.out.print(name + "(");
68
69         // вывести типы параметров
70         Class[] paramTypes = c.getParameterTypes();
71         for (int j = 0; j < paramTypes.length; j++)
72         {
73             if (j > 0) System.out.print(", ");
74             System.out.print(paramTypes[j].getName());
75         }
76         System.out.println(")");
77     }
78 }
79
80 /**
81  * Выводить все методы класса
82  * @param cl Класс
83  */
84 public static void printMethods(Class cl)
85 {
86     Method[] methods = cl.getDeclaredMethods();
87
88     for (Method m : methods)
89     {
90         Class retType = m.getReturnType();
91         String name = m.getName();
92
93         System.out.print(" ");
94         // вывести модификаторы, возвращаемый тип,
95         // а также имя метода
96         String modifiers = Modifier.toString(
97             m.getModifiers());
98         if (modifiers.length() > 0)
```

```

99         System.out.print(modifiers + " ");
100        System.out.print(retType.getName() + " "
101                           + name + "(");
102
103        // вывести типы параметров
104        Class[] paramTypes = m.getParameterTypes();
105        for (int j = 0; j < paramTypes.length; j++)
106        {
107            if (j > 0) System.out.print(", ");
108            System.out.print(paramTypes[j].getName());
109        }
110        System.out.println(")");
111    }
112 }
113
114 /**
115  * Выводит все поля класса
116  * @param cl а Класс
117  */
118 public static void printFields(Class cl)
119 {
120     Field[] fields = cl.getDeclaredFields();
121
122     for (Field f : fields)
123     {
124         Class type = f.getType();
125         String name = f.getName();
126         System.out.print(" ");
127         String modifiers = Modifier.toString(
128                                 f.getModifiers());
129         if (modifiers.length() > 0)
130             System.out.print(modifiers + " ");
131         System.out.println(type.getName() + " "
132                            + name + ";");
133     }
134 }

```

java.lang.Class 1.0

- **Field[] getFields()** 1.1
- **Field[] getDeclaredFields()** 1.1

Метод **getFields()** возвращает массив, который содержит объекты типа **Field**, соответствующие открытым полям анализируемого класса или его суперкласса. А метод **getDeclaredFields()** возвращает массив, содержащий объекты типа **Field**, соответствующие всем полям анализируемого класса. Оба метода возвращают массив нулевой длины, если такие поля отсутствуют или же если объект типа **Class** представляет собой простой тип данных или массив.

- **Method[] getMethods()** 1.1
- **Method[] getDeclaredMethods()** 1.1

Возвращают массив, который содержит объекты типа **Method**, соответствующие только открытым методам, включая унаследованные (метод **getMethods()**), или же всем методам анализируемого класса или интерфейса, за исключением унаследованных (метод **getDeclaredMethods()**).

java.lang.Class 1.0 (окончание)

- **Constructor[] getConstructors () 1.1**

- **Constructor[] getDeclaredConstructors () 1.1**

Возвращают массив, который содержит объекты типа **Constructor**, соответствующие только открытым конструкторам (метод **getConstructors ()**) или же всем конструкторам класса, представленного объектом типа **Class** (метод **getDeclaredMethods ()**).

- **String getPackageName () 9**

Получает имя пакета, содержащего данный тип, имя пакета с элементами данного типа, если он является типом массива, или же имя пакета **"java.lang"**, если данный тип является примитивным.

java.lang.reflect.Field 1.1**java.lang.reflect.Method 1.1****java.lang.reflect.Constructor 1.1**

- **Class getDeclaringClass ()**

- Возвращает объект типа **Class**, соответствующий классу, в котором определен заданный конструктор, метод или поле.

- **Class[] getExceptionTypes ()** [в классах **Constructor** и **Method**]

- Возвращает массив объектов типа **Class**, представляющих типы исключений, генерируемых заданным методом.

- **int getModifiers ()**

- Возвращает целое значение, соответствующее модификатору заданного конструктора, метода или поля. Для анализа возвращаемого значения следует использовать методы из класса **Modifier**.

- **String getName ()**

- Возвращает символьную строку, в которой содержится имя конструктора, метода или поля.

- **Class[] getParameterTypes ()** [в классах **Constructor** и **Method**]

- Возвращает массив объектов типа **Class**, представляющих типы параметров.

- **Class getReturnType ()** [в классе **Method**]

- Возвращает объект тип **Class**, соответствующий возвращаемому типу.

java.lang.reflect.Modifier 1.1

- **static String toString(int modifiers)**

Возвращает символьную строку с модификаторами, соответствующими битам, установленным в целочисленном значении параметра **modifiers**.

- **static boolean isAbstract(int modifiers)**

- **static boolean isFinal(int modifiers)**

- **static boolean isInterface(int modifiers)**

- **static boolean isNative(int modifiers)**

- **static boolean isPrivate(int modifiers)**

```
java.lang.reflect.Modifier 1.1 (окончание)
```

- `static boolean isProtected(int modifiers)`
- `static boolean isPublic(int modifiers)`
- `static boolean isStatic(int modifiers)`
- `static boolean isStrict(int modifiers)`
- `static boolean isSynchronized(int modifiers)`
- `static boolean isVolatile(int modifiers)`

Проверяют разряды целого значения параметра `modifiers`, которые соответствуют модификаторам доступа, указываемым при объявлении методов.

5.7.5. Анализ объектов во время выполнения с помощью рефлексии

Как следует из предыдущего раздела, для определения имен и типов полей любого объекта достаточно выполнить два действия:

- получить соответствующий объект типа `Class`;
- вызвать метод `getDeclaredFields()` для этого объекта.

В этом разделе мы пойдем дальше и попробуем определить содержимое полей данных. Разумеется, если имя и тип объекта известны при написании программы, это не составит особого труда. Но рефлексия позволяет сделать это и для объектов, которые неизвестны на стадии компиляции.

Ключевым в этом процессе является метод `get()` из класса `Field`. Если объект `f` относится к типу `Field` (например, получен в результате вызова метода `getDeclaredFields()`), а объект `obj` относится к тому же самому классу, что и объект `f`, то в результате вызова `f.get(obj)` возвратится объект, значение которого будет текущим значением поля в объекте `obj`. Покажем действие этого механизма на следующем конкретном примере:

```
var harry = new Employee("Harry Hacker", 35000, 10, 1, 1989);
Class cl = harry.getClass();
// объект типа Class, представляющий класс Employee
Field f = cl.getDeclaredField("name");
// поле name из класса Employee
Object v = f.get(harry);
// значение в поле name объекта harry,
// т.е. объект типа String, содержащий
// символьную строку "Harry Hacker"
```

Безусловно, установить можно и те значения, которые удастся получить. Так, в результате вызова `f.set(obj, value)` устанавливается новое значение `value` в поле объекта `obj`, доступном по ссылке `f`.

На первый взгляд, в приведенном выше фрагменте кода нет ничего каверзного, но его выполнение приводит к ошибке. В частности, поле `name` объявлено как `private`, а следовательно, метод `get()` генерирует исключение типа `IllegalAccessException`. Ведь метод `get()` можно применять только к открытым полям. Механизм безопасности Java позволяет определить, какие именно поля содержит объект, но не дает возможности прочитать их содержимое, если эти поля недоступны.

По умолчанию в механизме рефлексии соблюдаются правила доступа, установленные в Java. Но если программа не контролируется диспетчером

защиты, то эти правила можно обойти. Чтобы сделать это, достаточно вызвать метод `setAccessible()` для объектов типа `Field`, `Method` или `Constructor`, например, следующим образом:

```
f.setAccessible(true); // теперь можно сделать вызов f.get(harry);
```

Метод `setAccessible()` относится к классу `AccessibleObject`, являющемуся общим суперклассом для классов `Field`, `Method` и `Constructor`. Он нужен для обеспечения нормальной работы отладчиков, поддержки постоянных хранилищ и выполнения других подобных функций. Мы применим его далее в этой главе для создания обобщенного метода `toString()`.

При обращении к классу `AccessibleObject` генерируется исключение, если доступ не предоставляется. Такой доступ может быть отвергнут модульной системой (см. главу 9 второго тома настоящего издания) или диспетчером безопасности (см. главу 10 второго тома настоящего издания). Диспетчеры безопасности не нашли широкого употребления, но, начиная с версии Java 9, каждая программа содержит модули, поскольку прикладной интерфейс Java API модуляризирован.

В связи с тем что рефлексия применяется во многих библиотеках, в версиях 9 и 10 лишь выдается соответствующее предупреждение при попытке воспользоваться рефлексией для доступа к неоткрытым функциональным средствам в модуле. Так, в примере программы, приведенном в конце этого раздела, анализируется внутренняя структура объектов типа `ArrayList` и `Integer`. При выполнении данной программы на консоль выводится следующее тревожное сообщение:

```
WARNING: An illegal reflective access
operation has occurred
WARNING: Illegal reflective access by
objectAnalyzer.ObjectAnalyzer
(file:/home/cay/books/cj11/code/vlch05/bin/)
to field java.util.ArrayList.serialVersionUID
WARNING: Please consider reporting this
to the maintainers of objectAnalyzer.ObjectAnalyzer
WARNING: Use --illegal-access=warn to enable warnings
of further illegal reflective access operations
WARNING: All illegal access operations will be
denied in a future release2
```

Подобные предупреждения пока еще можно запретить. Для этого пакеты `java.util` и `java.lang` следует сделать “открытыми безымянными модулями” в модуле

²ПРЕДУПРЕЖДЕНИЕ: произошла недопустимая операция
рефлексивного доступа

ПРЕДУПРЕЖДЕНИЕ: недопустимый рефлексивный доступ
из класса `objectAnalyzer.ObjectAnalyzer`
(файл: /home/cay/books/cj11/code/vlch05/bin/)
к полю `java.util.ArrayList.serialVersionUID`

ПРЕДУПРЕЖДЕНИЕ: это сообщение касается тех, кто
сопровождает класс `objectAnalyzer.ObjectAnalyzer`

ПРЕДУПРЕЖДЕНИЕ: воспользуйтесь параметром `--illegal-access=warn`,
чтобы активизировать предупреждения о последующих
недопустимых операциях рефлексивного доступа

ПРЕДУПРЕЖДЕНИЕ: все недопустимые операции доступа будут
отвергнуты в последующем выпуске

java.base. Подробнее об этом речь пойдет в главе 9 второго тома настоящего издания. Ниже показано, как это делается в режиме командной строки.

```
java --add-opens java.base/java.util=ALL-UNNAMED \  
--add-opens java.base/java.lang=ALL-UNNAMED \  
objectAnalyzer.ObjectAnalyzerTest
```

С другой стороны, можно выяснить, каким образом прикладная будет вести себя в последующей версии Java, выполнив приведенную ниже команду. И тогда данная программа просто завершится аварийно исключением типа `IllegalAccessException`.

```
java --illegal-access=deny objectAnalyzer/  
ObjectAnalyzerTest
```



НА ЗАМЕТКУ! В последующих версиях библиотек вполне возможно употребление *переменных дескрипторов* вместо рефлексии для записи и чтения содержимого полей. В частности, переменный дескриптор типа `VarHandle` действует подобно типу `Field`. С его помощью можно записывать и читать содержимое конкретного поля в любом экземпляре отдельного класса. Но для получения переменного дескриптора типа `VarHandle` в библиотечном коде потребуются объект типа `Lookup`, как показано ниже.

```
public Object getFieldValue(Object obj, String fieldName,  
                           Lookup lookup)  
    throws NoSuchFieldException, IllegalAccessException  
{  
    Class<?> cl = obj.getClass();  
    Field field = cl.getDeclaredField(fieldName);  
    VarHandle handle = MethodHandles  
        .privateLookupIn(cl, lookup)  
        .unreflectVarHandle(field);  
    return handle.get(obj);  
}
```

Такой способ оказывается вполне работоспособным, но при условии, что объект типа `Lookup` формируется в модуле, имеющем разрешение на доступ к полю. С этой целью из какого-нибудь метода в модуле просто делается вызов `MethodHandles.lookup()`, который дает в итоге объект, инкапсулирующий права доступа вызывающего кода. Подобным образом один модуль может дать другому модулю разрешение на доступ к закрытым членам анализируемого класса. И в связи с этим возникает следующий практический вопрос: как предоставить подобное разрешение с минимальными хлопотами?

Принимая во внимание упомянутые выше возможности для анализа объектов, рассмотрим все же создание универсального метода `toString()`, пригодного для *любого* класса (см. ниже листинги 5.15 и 5.16). В этом методе сначала вызывается метод `getDeclaredFields()` для получения всех полей, а затем удобный метод `setAccessible()`, делающий все эти поля доступными. Далее определяется имя и значение каждого поля. Каждое получаемое в итоге значение преобразуется в символьную строку путем рекурсивного вызова метода `toString()`.

В универсальном методе `toString()` необходимо разрешить некоторые затруднения. В частности, циклически повторяющиеся ссылки могут привести к бесконечной рекурсии. Следовательно, объект типа `ObjectAnalyzer` должен отслеживать объекты, которые уже были проанализированы. Кроме того, для анализа массивов требуется другой подход, который будет более подробно рассмотрен в следующем разделе.

Универсальным методом `toString()` можно воспользоваться для просмотра содержимого любого объекта. Так, в результате выполнения следующего фрагмента кода:

```
var squares = new ArrayList<>();
for (int i = 1; i <= 5; i++) squares.add(i * i);
System.out.println(new ObjectAnalyzer().toString(squares));
```

выводится такая информация:

```
java.util.ArrayList[elementData=class java.lang.Object[]
{java.lang.Integer[value=1] [],
 java.lang.Integer[value=4] [],
 java.lang.Integer[value=9] [],
 java.lang.Integer[value=16] [],
 java.lang.Integer[value=25] [],
 null,null,null,null,null},size=5][modCount=5] []]
```

Используя такой универсальный метод `toString()`, можно реализовать конкретные методы `toString()` в собственных классах. Это можно сделать, например, следующим образом:

```
public String toString()
{
    return new ObjectAnalyzer().toString(this);
}
```

Предоставление универсального метода `toString()` — без сомнения, удобный и бесхлопотный способ. Но прежде чем приходить в чрезмерный восторг от того, что метод `toString()` не придется реализовывать каждый раз, когда он потребуется, следует помнить, что дни неконтролируемого доступа к внутренней структуре объектов сочтены.

Листинг 5.15. Исходный код из файла `objectAnalyzer/ObjectAnalyzerTest.java`

```
1 package objectAnalyzer;
2
3 import java.util.*;
4
5 /**
6  * В этой программе рефлексия применяется
7  * для слежения за объектами
8  * @version 1.13 2018-03-16
9  * @author Cay Horstmann
10 */
11 public class ObjectAnalyzerTest
12 {
13     public static void main(String[] args)
14         throws ReflectiveOperationException
15     {
16         var squares = new ArrayList<Integer>();
17         for (int i = 1; i <= 5; i++)
18             squares.add(i * i);
19         System.out.println(
20             new ObjectAnalyzer().toString(squares));
21     }
22 }
```

Листинг 5.16. Исходный код из файла `objectAnalyzer/ObjectAnalyzer.java`

```
1 package objectAnalyzer;
2
3 import java.lang.reflect.AccessibleObject;
4 import java.lang.reflect.Array;
5 import java.lang.reflect.Field;
6 import java.lang.reflect.Modifier;
7 import java.util.ArrayList;
8
9 public class ObjectAnalyzer
10 {
11     private ArrayList<Object> visited = new ArrayList<>();
12
13     /**
14      * Преобразует объект в строковое представление
15      * всех перечисляемых полей
16      * @param obj Объект
17      * @return Возвращает строку с именем класса и всеми
18      *         полями объекта, а также их значениями
19      */
20     public String toString(Object obj)
21         throws ReflectiveOperationException
22     {
23         if (obj == null) return "null";
24         if (visited.contains(obj)) return "...";
25         visited.add(obj);
26         Class cl = obj.getClass();
27         if (cl == String.class) return (String) obj;
28         if (cl.isArray())
29         {
30             String r = cl.getComponentType() + "[";
31             for (int i = 0; i < Array.getLength(obj); i++)
32             {
33                 if (i > 0) r += ",";
34                 Object val = Array.get(obj, i);
35                 if (cl.getComponentType().isPrimitive())
36                     r += val;
37                 else r += toString(val);
38             }
39             return r + "]";
40         }
41
42         String r = cl.getName();
43         // проверить поля этого класса и
44         // всех его суперклассов
45         do
46         {
47             r += "[";
48             Field[] fields = cl.getDeclaredFields();
49             AccessibleObject.setAccessible(fields, true);
50             // получить имена и значения всех полей
51             for (Field f : fields)
52             {
53                 if (!Modifier.isStatic(f.getModifiers()))
```

```
54      {
55          if (!r.endsWith("[") ) r += ",";
56          r += f.getName() + "=";
57          Class t = f.getType();
58          Object val = f.get(obj);
59          if (t.isPrimitive()) r += val;
60          else r += toString(val);
61      }
62  }
63  r += "]\n";
64  cl = cl.getSuperclass();
65  }
66  while (cl != null);
67
68  return r;
69  }
70 }
```

java.lang.reflect.AccessibleObject 1.2

- **void setAccessible(boolean flag)**

Устанавливает признак доступности заданного объекта рефлексии. Логическое значение **true** параметра **flag** обозначает, что проверка доступа к компонентам языка Java отменена и закрытые свойства объекта теперь доступны для выборки и установки.

- **void setAccessible(boolean flag)**

- **boolean trySetAccessible() 9**

Первый из этих методов устанавливает признак для данного доступного объекта, а второй возвращает логическое значение **false**, если доступ запрещен.

- **boolean isAccessible()**

Получает значение признака доступности заданного объекта рефлексии.

- **static void setAccessible(AccessibleObject[] array, boolean flag)**

Удобен для установки признака доступности массива объектов.

java.lang.Class 1.1

- **Field getField(String name)**

- **Field[] getFields()**

Возвращают общедоступное поле с указанным именем или массив, содержащий все поля.

- **Field getDeclaredField(String name)**

- **Field[] getDeclaredFields()**

Возвращают объявленное в классе поле с указанным именем или же массив, содержащий все поля.

java.lang.reflect.Field 1.1

- **Object get(Object obj)**
Возвращает значение поля объекта *obj*, описываемого данным объектом типа **Field**.
- **void set(Object obj, Object newValue)**
Устанавливает новое значение в поле объекта *obj*, описываемого данным объектом типа **Field**.

5.7.6. Написание кода универсального массива с помощью рефлексии

Класс `Array` из пакета `java.lang.reflect` позволяет создавать массивы динамически. Этим можно, например, воспользоваться для реализации метода `copyOf()` в классе `Arrays`. Напомним, что этот метод служит для наращивания уже заполненного массива. Ниже показано, каким образом такое наращивание реализуется в коде.

```
var a = new Employee[100];  
...  
// массив заполнен  
a = Arrays.copyOf(a, 2 * a.length);
```

Как написать такой обобщенный метод? В этом нам поможет тот факт, что массив `Employee[]` можно преобразовать в массив `Object[]`. Звучит многообещающе. Итак, сделаем первую попытку создать обобщенный метод следующим образом:

```
public static Object[] badCopyOf(Object[] a, int newLength) // бесполезно  
{  
    var newArray = new Object[newLength];  
    System.arraycopy(a, 0, newArray, 0, Math.min(a.length, newLength));  
    return newArray;  
}
```

Но в этом коде возникает затруднение, связанное с фактическим использованием получаемого в итоге массива. Этот массив содержит *объекты* и относится к типу `Object[]`, поскольку он создан с помощью выражения

```
new Object[newLength]
```

Массив объектов типа `Object[]` не может быть преобразован в массив типа `Employee[]`. При попытке сделать это возникнет исключение типа `ClassCastException`. Как пояснялось ранее, в ходе выполнения программы исполняющая система Java запоминает первоначальный тип элементов массива, т.е. тип, указанный в операции `new`. Массив типа `Employee[]` можно временно преобразовать в массив типа `Object[]` и обратно, но массив, изначально созданный как относящийся к типу `Object[]`, преобразовать в массив типа `Employee[]` нельзя. Чтобы написать подходящий обобщенный метод, нужно каким-то образом создать новый массив, тип которого совпадал бы с типом исходного массива. Для этого потребуются методы класса `Array` из пакета `java.lang.reflect` и особенно метод `newInstance()`, создающий новый массив. Тип элементов массива и требуемая его длина должны передаваться обобщенному методу в качестве параметров следующим образом:

```
Object newArray = Array.newInstance(componentType, newLength);
```

Чтобы сделать это, нужно определить длину и тип элементов нового массива. Длину можно получить с помощью метода `Array.getLength()`. Статический метод `getLength()` из класса `Array` возвращает длину любого массива. А для того чтобы определить тип элементов нового массива, необходимо выполнить следующие действия.

1. Определить, какому именно классу принадлежит объект `a`.
2. Убедиться в том, что он действительно является массивом.
3. Воспользоваться методом `getComponentType()` из класса `Class` (определен лишь для объектов типа `Class`, представляющих массивы), чтобы получить требуемый тип массива.

Почему же метод `getLength()` принадлежит классу `Array`, а метод `getComponentType()` — классу `Class`? Вряд ли это известно кому-либо, кроме разработчиков этих классов. Существующее распределение методов по классам приходится иногда принимать таким, каким оно есть.

Ниже приведен исходный код рассматриваемого здесь обобщенного метода.

```
public static Object goodCopyOf(Object a, int newLength)
{
    Class cl = a.getClass();
    if (!cl.isArray()) return null;
    Class componentType = cl.getComponentType();
    int length = Array.getLength(a);
    Object newArray = Array.newInstance(componentType,
                                       newLength);
    System.arraycopy(a, 0, newArray, 0, Math.min(length,
                                                  newLength));
    return newArray;
}
```

Однако метод `goodCopyOf()` можно применять для наращивания массива любого типа, а не только массива объектов, как показано ниже.

```
int[] a = { 1, 2, 3, 4, 5 };
a = (int[]) goodCopyOf(a, 10);
```

Для этого параметр метода `goodCopyOf()` объявляется как относящийся к типу `Object`, а не как массив объектов (т.е. типа `Object[]`). Массив типа `int[]` можно преобразовать в объект типа `Object`, но не в массив объектов!

В листинге 5.17 демонстрируется применение обоих вариантов метода `CopyOf()`: `badCopyOf()` и `goodCopyOf()`. Но в результате приведения типа значения, возвращаемого методом `badCopyOf()`, возникнет исключение.

Листинг 5.17. Исходный код из файла `arrays/CopyOfTest.java`

```
1 package arrays;
2
3 import java.lang.reflect.*;
4 import java.util.*;
5
6 /**
7  * В этой программе демонстрируется применение рефлексии
8  * для манипулирования массивами
9  * @version 1.2 2012-05-04
```

```
10 * @author Cay Horstmann
11 */
12 public class CopyOfTest
13 {
14     public static void main(String[] args)
15     {
16         int[] a = { 1, 2, 3 };
17         a = (int[]) goodCopyOf(a, 10);
18         System.out.println(Arrays.toString(a));
19
20         String[] b = { "Tom", "Dick", "Harry" };
21         b = (String[]) goodCopyOf(b, 10);
22         System.out.println(Arrays.toString(b));
23
24         System.out.println("The following call will "
25                             + "generate an exception.");
26         b = (String[]) badCopyOf(b, 10);
27     }
28
29     /**
30      * В этом методе предпринимается попытка нарастить
31      * массив путем выделения нового массива и копирования
32      * в него всех прежних элементов
33      * @param Нарастиваемый массив
34      * @param newLength Новая длина массива
35      * @return Возвращаемый наращенный массив, содержащий
36      *         все элементы массива a, но он относится
37      *         к типу Object[], а не к типу массива a
38      */
39     public static Object[] badCopyOf(Object[] a,
40                                     int newLength) // бесполезно
41     {
42         Object[] newArray = new Object[newLength];
43         System.arraycopy(a, 0, newArray, 0,
44                         Math.min(a.length, newLength));
45         return newArray;
46     }
47
48     /**
49      * Этот метод наращивает массив, выделяя новый массив
50      * того же типа и копируя в него все прежние элементы
51      * @param Нарастиваемый массив. Может быть массивом
52      *        объектов или же массивом примитивных типов
53      * @return Возвращаемый наращенный массив, содержащий
54      *         все элементы массива a
55      */
56     public static Object goodCopyOf(Object a,
57                                     int newLength)
58     {
59         Class cl = a.getClass();
60         if (!cl.isArray()) return null;
61         Class componentType = cl.getComponentType();
62         int length = Array.getLength(a);
63         Object newArray = Array.newInstance(componentType,
64                                             newLength);
65         System.arraycopy(a, 0, newArray, 0,
```



```

66             Math.min(length, newLength));
67     return newArray;
68 }
69 }

```

java.lang.reflect.Array 1.1

- **static Object get(Object array, int index)**
- **static xxx getXxx(Object array, int index)**
Возвращают значение элемента указанного массива по заданному индексу. (Символами **xxx** обозначаются примитивные типы **boolean, byte, char, double, float, int, long, short.**)
- **static void set(Object array, int index, Object newValue)**
- **static setXxx(Object array, int index, xxx newValue)**
Устанавливают новое значение в элементе указанного массива по заданному индексу. (Символами **xxx** обозначаются примитивные типы **boolean, byte, char, double, float, int, long, short.**)
- **static int getLength(Object array)**
Возвращает длину указанного массива.
- **static Object newInstance(Class componentType, int length)**
- **static Object newInstance(Class componentType, int[] lengths)**
Возвращают новый массив, состоящий из компонентов указанного типа и имеющий заданную размерность.

5.7.7. Вызов произвольных методов и конструкторов

В языках С и С++ можно выполнить произвольную функцию по указателю на нее. На первый взгляд, в Java не предусмотрены указатели на методы, т.е. в этом языке отсутствует возможность передавать одному методу адрес другого метода, чтобы последний мог затем вызвать первый. Создатели Java заявили, что указатели на методы небезопасны и часто порождают ошибки, а действия, для которых часто применяются такие указатели, удобнее выполнять с помощью интерфейсов и лямбда-выражений, рассматриваемых в следующей главе. Тем не менее механизм рефлексии позволяет вызывать произвольные методы.

Напомним, что поле объекта можно проверить с помощью метода `get()` из класса `Field`. Аналогично класс `Method` содержит метод `invoke()`, позволяющий вызвать метод, заключенный в оболочку текущего объекта этого класса:

```
Object invoke(Object obj, Object... args)
```

Первый параметр этого метода является неявным, а остальные объекты представляют собой явные параметры. Если метод статический, то первый параметр игнорируется и вместо него можно указать пустое значение `null`. Так, если объект `m1` представляет метод `getName()` из класса `Employee`, то можно сделать следующий вызов:

```
String n = (String) m1.invoke(harry);
```

Если возвращаемый тип оказывается примитивным, метод `invoke()` возвратит вместо него тип объекта-оболочки. Допустим, объект `m2` представляет метод `getSalary()` из класса `Employee`. В таком случае возвращаемый объект-оболочка

фактически относится к типу `Double`, поэтому его необходимо привести к примитивному типу `double`. Это нетрудно сделать с помощью автораспаковки следующим образом:

```
double s = (Double) m2.invoke(harry);
```

Как же получить объект типа `Method`? Можно, конечно, вызвать метод `getDeclaredMethods()` и найти искомый объект среди возвращаемого массива объектов типа `Method`. Кроме того, можно вызвать метод `getMethod()` из класса `Class`. Его действие можно сравнить с методом `getField()`, получающим символьную строку с именем поля и возвращающим объект типа `Field`. Но методов с одним и тем же именем может быть несколько, и среди них приходится тщательно выбирать нужный. Именно по этой причине необходимо также предусмотреть массив, содержащий типы параметров искомого метода. Сигнатура метода `getMethod()` выглядит следующим образом:

```
Method getMethod(String name, Class... parameterTypes)
```

В качестве примера ниже показано, как получают указатели на методы `getName()` и `raiseSalary()` из класса `Employee`.

```
Method m1 = Employee.class.getMethod("getName");  
Method m2 = Employee.class.getMethod("raiseSalary", double.class);
```

Аналогичным способом можно вызывать и произвольные конструкторы. Для этого достаточно предоставить типы параметров конструктора вызываемому методу `Class.getConstructor()`, а значения параметров — методу `Constructor.newInstance()`, как показано ниже.

```
Class cl = Random.class; // или любой другой класс  
// с конструктором, принимающим параметр типа long  
Constructor cons = cl.getConstructor(long.class);  
Object obj = cons.newInstance(42L);
```

Итак, выяснив правила применения объектов типа `Method`, продемонстрируем их употребление непосредственно в прикладном коде. В листинге 5.18 приведена программа, выводящая таблицу значений математической функции вроде `Math.sqrt()` или `Math.sin()`. Результат выполнения этой программы выглядит следующим образом:

```
public static native double java.lang.Math.sqrt(double)  
1.0000 | 1.0000  
2.0000 | 1.4142  
3.0000 | 1.7321  
4.0000 | 2.0000  
5.0000 | 2.2361  
6.0000 | 2.4495  
7.0000 | 2.6458  
8.0000 | 2.8284  
9.0000 | 3.0000  
10.0000 | 3.1623
```

Разумеется, код, осуществляющий вывод таблицы на экран, не зависит от конкретной функции:

```
double dx = (to - from) / (n - 1);  
for (double x = from; x <= to; x += dx)
```

```
{  
    double y = (Double) f.invoke(null, x);  
    System.out.printf("%10.4f | %10.4f%n", x, y);  
}
```

где `f` — это объект типа `Method`. А поскольку вызывается статический метод, то в качестве первого параметра методу `invoke()` передается пустое значение `null`. Для вывода таблицы со значениями математической функции `Math.sqrt` служит следующая строка кода:

```
Math.class.getMethod("sqrt", double.class)
```

При вызове метода `getMethod()` ему передается имя метода `sqrt()` из класса `Math` и параметр типа `double`. В листинге 5.18 приведен весь исходный код универсального варианта программы табличного вывода значений функции.

Листинг 5.17. Исходный код из файла `methods/MethodTableTest.java`

```
1 package methods;  
2  
3 import java.lang.reflect.*;  
4  
5 /**  
6  * В этой программе демонстрируется применение  
7  * рефлексии для вызова методов  
8  * @version 1.2 2012-05-04  
9  * @author Cay Horstmann  
10 * /  
11 public class MethodTableTest  
12 {  
13     public static void main(String[] args)  
14         throws Exception  
15     {  
16         // получить указатели на методы square() и sqrt()  
17         Method square = MethodTableTest.class  
18             .getMethod("square", double.class);  
19         Method sqrt = Math.class.getMethod("sqrt",  
20             double.class);  
21  
22         // вывести значения x и y в табличном виде  
23  
24         printTable(1, 10, 10, square);  
25         printTable(1, 10, 10, sqrt);  
26     }  
27  
28     /**  
29     * Возвращает квадрат числа  
30     * @param x Число  
31     * @return x Квадрат числа  
32     * /  
33     public static double square(double x)  
34     {  
35         return x * x;  
36     }  
37  
38     /**  
39     * Выводит в табличном виде значения x и y
```

```
40  * указанного метода
41  * @param Нижняя граница значений x
42  * @param Верхняя граница значений x
43  * @param n Количество строк в таблице
44  * @param f Метод, получающий и возвращающий
45  *           значение типа double
46  */
47  public static void printTable(double from,
48                               double to, int n, Method f)
49  {
50      // вывести сигнатуру метода в заголовке таблицы
51      System.out.println(f);
52
53      double dx = (to - from) / (n - 1);
54
55      for (double x = from; x <= to; x += dx)
56      {
57          try
58          {
59              double y = (Double) f.invoke(null, x);
60              System.out.printf("%10.4f | %10.4f%n", x, y);
61          }
62          catch (Exception e)
63          {
64              e.printStackTrace();
65          }
66      }
67  }
68 }
```

Приведенный выше пример программы наглядно показывает, что с помощью объектов типа `Method` можно делать то же, что и с помощью указателей на функции в С (или делегатов в С#). Как и в С, такой стиль программирования обычно неудобен и часто приводит к ошибкам. Что, если, например, вызвать метод `invoke()` с неверно заданными параметрами? В этом случае метод `invoke()` сгенерирует исключение.

Кроме того, параметры метода `invoke()` и возвращаемое им значение обязательно должны быть типа `Object`. А это влечет за собой приведение типов в соответствующих местах прикладного кода. В итоге компилятор будет лишен возможности тщательно проверить исходный код программы. Следовательно, ошибки в ней проявятся только на стадии тестирования, когда исправить их будет намного труднее. Более того, программа, использующая механизм рефлексии для получения указателей на методы, работает заметно медленнее, чем программа, непосредственно вызывающая эти методы.

По этим причинам пользоваться объектами типа `Method` рекомендуется только в самом крайнем случае. Намного лучше применять интерфейсы, а начиная с версии Java 8, и лямбда-выражения, рассматриваемые в следующей главе. В частности, следуя указаниям разработчиков Java, объекты типа `Method` не рекомендуется применять для организации функций обратного вызова, поскольку для этого вполне подходят интерфейсы, позволяющие создавать программы, которые работают намного быстрее и надежнее.

```
java.lang.reflect.Method 1.1
```

- `public Object invoke(Object implicitParameter, Object[] explicitParameters)`

Вызывает метод, описанный в объекте, передавая ему заданные параметры и возвращая значение, вычисленное этим методом. Для статических методов в качестве неявного параметра передается пустое значение `null`. Прimitives типы следует передавать только в виде объектных оболочек классов. Возвращаемые значения примитивных типов должны извлекаться из объектных оболочек путем автораспаковки.

5.8. Рекомендации по применению наследования

В завершение главы приведем некоторые рекомендации относительно надлежащего применения очень полезного механизма наследования.

1. *Размещайте общие операции и поля в суперклассе.*

Поле `name` было перемещено в класс `Person` именно для того, чтобы не повторять его в классах `Employee` и `Student`.

2. *Старайтесь не пользоваться защищенными полями.*

Некоторые разработчики полагают, что следует на всякий случай объявлять большинство полей защищенными, чтобы подклассы могли обращаться к ним по мере надобности. Но имеются две веские причины, по которым такой механизм не гарантирует достаточной защиты. Во-первых, множество подклассов неограниченно. Всякий может создать подкласс, производный от данного класса, а затем написать программу, получающую непосредственный доступ к защищенным полям его экземпляра, нарушая инкапсуляцию. И во-вторых, в Java к защищенным полям имеют доступ все классы, находящиеся в том же пакете, независимо от того, являются ли они подклассами данного класса или нет. В то же время полезно объявлять защищенными те методы, которые не предназначены для общего употребления и должны быть переопределены в подклассах.

3. *Используйте наследование для моделирования отношений “является”.*

Наследование позволяет экономить время и труд при разработке прикладных программ, но иногда им злоупотребляют. Допустим, требуется создать класс `Contractor`. У работника, нанимаемого по контракту, имеется свое имя и дата заключения договора, но у него нет оклада. У него почасовая оплата, причем он работает не так давно, чтобы повышать оплату его труда. Ниже показано, как можно сделать класс `Contractor` подклассом, производным от класса `Employee`, добавив поле `hourlyWage`.

```
class Contractor extends Employee
{
    private double hourlyWage;
    . . .
}
```

Но это не совсем удачная идея. Ведь в этом случае получается, что каждый работник, нанятый по контракту, имеет и оклад, и почасовую оплату. Если вы попытаетесь реализовать методы для распечатки платежных и налоговых ведомостей, то сразу же проявится недостаток такого подхода. Программа, которую вам придется написать, будет гораздо длиннее той, которую вы могли бы создать, не прибегая к неоправданному наследованию.

Отношение “контрактный работник–постоянный работник” не удовлетворяет критерию “является”. Работники, нанятые по контракту, не являются постоянными и относятся к особой категории работников.

4. *Не пользуйтесь наследованием, если не все методы имеют смысл сделать наследуемыми.*

Допустим, требуется создать класс `Holiday`. Разумеется, праздники — это разновидность календарных дней, а дни можно представить в виде объектов типа `GregorianCalendar`, поэтому наследование можно применить следующим образом:

```
class Holiday extends GregorianCalendar(...)
```

К сожалению, множество праздников оказывается *незамкнутым* при наследовании. Среди открытых методов из класса `GregorianCalendar` имеется метод `add()`, который может превратить праздничные дни в будничные:

```
Holiday christmas;  
christmas.add(Calendar.DAY_OF_MONTH, 12);
```

Следовательно, наследование в данном случае не подходит. Однако подобные затруднения не возникают, если используется класс `LocalDate`. Этот класс является неизменяемым, поэтому в нем отсутствует метод, способный превратить праздничный день в будничный.

5. *Переопределяя метод, не изменяйте его предполагаемое поведение.*

Принцип подстановки распространяется не только на синтаксис, но и на поведение, что важнее. При переопределении метода не следует без особых на то причин изменять его поведение. В этом компилятор вам не поможет, так как он не в состоянии проверить, оправдано ли переопределение метода. Допустим, требуется устранить упомянутый выше недостаток метода `add()` из класса `Holiday`, переопределив этот метод таким образом, чтобы он, например, не выполнял никаких действий или же возвращал следующий праздничный день. Но такое переопределение нарушает принцип подстановки. При выполнении приведенной ниже последовательности операторов пользователь вправе ожидать вполне *определенного* поведения и соответствующего результата, независимо от того, является ли объект `x` экземпляром класса `GregorianCalendar` или `Holiday`.

```
int d1 = x.get(Calendar.DAY_OF_MONTH);  
x.add(Calendar.DAY_OF_MONTH, 1);  
int d2 = x.get(Calendar.DAY_OF_MONTH);  
System.out.println(d2 - d1);
```

Безусловно, здесь есть подвох. Разные пользователи сочтут естественным различное поведение программы. Так, по мнению некоторых, принцип

подстановки требует, чтобы в методе `Manager.equals()` не учитывалась премия в поле `bonus`, поскольку она игнорируется в методе `Employee.equals()`. Подобные споры могут длиться бесконечно и не дать никакого результата. Поэтому, принимая конкретное решение, следует руководствоваться теми целями, для которых создается программа.

6. *Пользуйтесь принципом полиморфизма, а не данными о типе.*

Вспомните о принципе полиморфизма, как только увидите код, имеющий следующий вид:

```
if (x типа 1)
    действие1(x);
else if (x типа 2)
    действие2(x);
```

Имеют ли `действие1` и `действие2` общий характер? Если у них имеется нечто общее, то поместите соответствующие методы в общий суперкласс или интерфейс обоих типов. Тогда можно просто сделать приведенный ниже вызов и выполнить правильное действие с помощью механизма динамического связывания, присущего полиморфизму.

```
x.действие();
```

Код, в котором применяется принцип полиморфизма или реализован интерфейс, намного легче сопровождать и расширять, чем код, изобилующий проверками типов.

7. *Не злоупотребляйте механизмом рефлексии.*

Механизм рефлексии позволяет создавать программы с высоким уровнем абстракции, где поля и методы определяются во время выполнения. Такая возможность чрезвычайно полезна для системного программирования, но для прикладного практически не нужна. Рефлексия — очень хрупкий механизм, поскольку компилятор не может помочь в обнаружении ошибок. Все ошибки проявляются во время выполнения программы и приводят к возникновению исключений.

В этой главе было показано, каким образом в Java поддерживаются основные понятия и принципы ООП: классы, наследование и полиморфизм. А в следующей главе мы затронем две более сложные темы, которые очень важны для эффективного программирования на Java: интерфейсы и лямбда-выражения.

Интерфейсы, лямбда-выражения и внутренние классы

В этой главе...

- ▶ Интерфейсы
- ▶ Лямбда-выражения
- ▶ Внутренние классы
- ▶ Загрузчики служб
- ▶ Прокси-классы

Итак, вы ознакомились с классами и наследованием — основными инструментальными средствами объектно-ориентированного программирования на Java. В этой главе будет представлен ряд усовершенствованных и широко применяемых методик программирования. Несмотря на то что эти методики не совсем очевидны, владеть ими должен всякий, стремящийся профессионально программировать на Java.

Первая из этих методик называется *интерфейсами* и позволяет указывать, что именно должны делать классы, не уточняя, как именно они должны это делать. В классах может быть реализован один или несколько интерфейсов. Если возникает потребность в интерфейсе, применяются объекты этих классов. Рассмотрев интерфейсы, мы перейдем к *лямбда-выражениям*, позволяющим выразить в краткой форме блок кода, который может быть выполнен в дальнейшем. С помощью лямбда-выражений можно изящно и лаконично выразить код, в котором применяются обратные вызовы или переменное поведение.

Далее мы обсудим механизм *внутренних классов*. С технической точки зрения внутренние классы довольно сложны — они определяются в других классах, а их методы имеют доступ к полям окружающего класса. Внутренние классы полезны при разработке коллекций взаимосвязанных классов.

И в завершение главы будут представлены *прокси-классы*, реализующие произвольные интерфейсы. Прокси-классы представляют собой особые конструкции, полезные для создания инструментальных средств системного программирования. При первом чтении книги вы можете благополучно пропустить эту заключительную часть главы.

6.1. Интерфейсы

В последующих разделах поясняется, что собой представляют интерфейсы и как ими пользоваться. Кроме того, будет показано, насколько повышена эффективность интерфейсов в последних версиях Java.

6.1.1. Понятие интерфейса

Интерфейс в Java не является классом и представляет собой ряд *требований*, предъявляемых к классу, который должен соответствовать интерфейсу. Как правило, один разработчик, собирающийся воспользоваться трудами другого разработчика для решения конкретной задачи, заявляет: “Если ваш класс будет соответствовать определенному интерфейсу, я смогу решить свою задачу”. Обратимся к конкретному примеру. Метод `sort()` из класса `Array` позволяет упорядочить массив объектов при одном условии: объекты должны принадлежать классам, реализующим интерфейс `Comparable`.

Этот интерфейс определяется следующим образом:

```
public interface Comparable
{
    int compareTo(Object other);
}
```

Это означает, что любой класс, реализующий интерфейс `Comparable`, должен содержать метод `compareTo()`, получающий параметр типа `Object` и возвращающий целое значение.



НА ЗАМЕТКУ! В версии Java 5 интерфейс `Comparable` стал обобщенным, как показано ниже.

```
public interface Comparable<T>
{
    int compareTo(T other); // этот параметр относится
                          // к обобщенному типу T
}
```

Так, если класс реализует интерфейс `Comparable<Employee>`, он должен содержать следующий метод:

```
int compareTo(Employee other);
```

По желанию можно по-прежнему пользоваться базовым (иначе называемым “сырым”) типом `Comparable`, но тогда придется вручную приводить параметр метода `compareTo()` к требуемому типу. Мы еще вернемся к этому вопросу в дальнейшем, чтобы подробнее разъяснить новые понятия.

Все методы интерфейса автоматически считаются открытыми, поэтому, объявляя метод в интерфейсе, указывать модификатор доступа `public` необязательно. Разумеется, существует и неявное требование: метод `compareTo()` должен на самом деле быть способным сравнивать два объекта и возвращать признак того, что один из них больше другого. В таком случае этот метод должен возвращать отрицательное числовое значение, если объект `x` меньше объекта `y`, нулевое значение — если они равны, а иначе — положительное числовое значение.

Данный конкретный интерфейс имеет единственный метод, а у некоторых интерфейсов может быть больше одного метода. Как станет ясно в дальнейшем, с помощью интерфейсов можно также объявлять константы. Но важнее не это, а то, что интерфейсы *не* могут предоставить. В частности, у них отсутствуют поля экземпляра. До версии Java 8 в интерфейсах нельзя было реализовывать методы, но теперь они могут предоставлять простые методы, как поясняется далее, в разделах 6.1.4 и 6.1.5. Разумеется, в этих методах нельзя ссылаться на поля экземпляра, поскольку они просто отсутствуют в интерфейсах.

Допустим теперь, что требуется воспользоваться методом `sort()` из класса `Array` для сортировки объектов типа `Employee`. В этом случае класс `Employee` должен реализовать интерфейс `Comparable`.

Для того чтобы класс реализовал интерфейс, нужно выполнить следующие действия.

1. Объявить, что класс реализует интерфейс.
2. Определить в классе все методы, указанные в интерфейсе.

Для объявления самой реализации интерфейса в классе служит ключевое слово `implements`:

```
class Employee implements Comparable
```

Разумеется, теперь нужно реализовать метод `compareTo()`, например, для сравнения зарплаты сотрудников. Ниже приведен код, реализующий метод `compareTo()` в классе `Employee`.

```
public int compareTo(Object otherObject)
{
    Employee other = (Employee) otherObject;
    return Double.compare(salary, other.salary);
}
```

В этом коде вызывается статический метод `Double.compare()`, возвращающий отрицательное числовое значение, если первый его аргумент меньше второго; нулевое значение, если аргументы равны; а иначе — положительное числовое значение.



ВНИМАНИЕ! При объявлении метода `compareTo()` в интерфейсе модификатор доступа `public` не указывается, поскольку все методы интерфейса автоматически являются открытыми. Но при реализации интерфейса этот модификатор доступа должен быть непременно указан. В противном случае компилятор предположит, что область видимости этого метода ограничивается пакетом, хотя по умолчанию она не выходит за пределы класса. В итоге компилятор выдаст предупреждение о попытке предоставить более ограниченные права доступа.

Можно принять более изящное решение, реализовав обобщенный интерфейс `Comparable` и снабдив его параметром типа, как показано ниже. Как видите, теперь тип `Object` не приходится приводить к требуемому типу.

```
class Employee implements Comparable<Employee>
{
    public int compareTo(Employee other)
    {
        return Double.compare(salary, other.salary);
    }
    . . .
}
```



СОВЕТ. Метод `compareTo()` из интерфейса `Comparable` возвращает целое значение. Если объекты не равнозначны, возвращается положительное или отрицательное числовое значение. Возможность использовать в качестве признака неравенства любое число, отличное от нуля, может оказаться удобной при сравнении целочисленных полей. Допустим, у каждого работника имеется однозначный идентификационный номер. В таком случае можно вернуть разность идентификационных номеров `id - other.id`. Она будет отрицательной, если идентификационный номер первого работника меньше идентификационного номера второго работника, нулевой, если номера равны, а иначе — положительной. Однако пределы изменения целых значений должны быть достаточно малы, чтобы вычитание не привело к переполнению. Если заранее известно, что идентификационный номер не является отрицательным или его абсолютное значение не превышает величину $(Integer.Max_Value - 1) / 2$, то можно смело применять рассматриваемый здесь способ сравнения.

Разумеется, такой способ вычитания не подходит для чисел с плавающей точкой. Разность `salary - other.salary` может быть округлена до нуля, если размеры зарплат очень близки, но не одинаковы. В результате вызова `Double.compare(x, y)` возвращается значение `-1`, если `x < y`, или значение `1`, если `x > y`.



НА ЗАМЕТКУ! В документации на интерфейс `Comparable` предполагается, что метод `compareTo()` должен быть совместим с методом `equals()`. Это означает, что результат сравнения `x.compareTo(y)` должен быть равен нулю, если к такому же результату приводит сравнение `x.equals(y)`. Этому правилу следует большинство классов из прикладного программного интерфейса Java API, где реализуется интерфейс `Comparable`. А самым примечательным исключением из этого правила служит класс `BigDecimal`. Так, если `x = new BigDecimal("1.0")` и `y = new BigDecimal("1.00")`, то в результате сравнения `x.equals(y)` возвращается логическое значение `false`, поскольку сравниваемые числа имеют разную точность. Но в то же время в результате сравнения `x.compareTo(y)` возвращается нулевое значение. В идеальном случае так не должно быть, но, очевидно, нельзя было решить, какое из этих сравнений важнее.

Теперь вам должно быть ясно, что для сортировки объектов достаточно реализовать в классе метод `compareTo()`. И такой подход вполне оправдан. Ведь должен же существовать какой-то способ воспользоваться методом `sort()` для сравнения объектов. Но почему бы просто не предусмотреть в классе `Employee` метод `compareTo()`, не реализуя интерфейс `Comparable`?

Дело в том, что Java — *строго типизированный* язык программирования. При вызове какого-нибудь метода компилятор должен убедиться, что этот метод действительно существует. В теле метода `sort()` могут находиться операторы, аналогичные следующим:

```
if (a[i].compareTo(a[j]) > 0)
{
    // переставить элементы массивов a[i] и a[j]
    . . .
}
```

Компилятору должно быть известно, что у объекта `a[i]` действительно имеется метод `compareTo()`. Если переменная `a` содержит массив объектов, реализующих интерфейс `Comparable`, то существование такого метода гарантируется, поскольку каждый класс, реализующий данный интерфейс, по определению должен предоставлять метод `compareTo()`.



НА ЗАМЕТКУ! На первый взгляд может показаться, что метод `sort()` из класса `Array` оперирует только массивами типа `Comparable[]` и что компилятор выдаст предупреждение, как только обнаружит вызов метода `sort()` для массива, элементы которого не реализуют данный интерфейс. Увы, это не так. Вместо этого метод `sort()` принимает массивы типа `Object[]` и выполняет приведение типов, как показано ниже.

```
// Такой подход принят в стандартной библиотеке,
// но применять его все же не рекомендуется
if (((Comparable) a[i]).compareTo(a[j]) > 0)
{
    // переставить элементы массивов a[i] и a[j]
    . . .
}
```

Если элемент массива `a[i]` не принадлежит классу, реализующему интерфейс `Comparable`, виртуальная машина сгенерирует исключение.

В листинге 6.1 приведен исходный код программы для сортировки массива, состоящего из объектов класса `Employee` (из листинга 6.2).

Листинг 6.1. Исходный код из файла `interfaces/EmployeeSortTest.java`

```
1 package interfaces;
2
3 import java.util.*;
4
5 /**
6  * В этой программе демонстрируется применение
7  * интерфейса Comparable
8  * @version 1.30 2004-02-27
9  * @author Cay Horstmann
10  */
11 public class EmployeeSortTest
12 {
13     public static void main(String[] args)
14     {
15         var staff = new Employee[3];
16
17         staff[0] = new Employee("Harry Hacker", 35000);
18         staff[1] = new Employee("Carl Cracker", 75000);
19         staff[2] = new Employee("Tony Tester", 38000);
20
21         Arrays.sort(staff);
22
23         // вывести данные обо всех объектах типа Employee
24         for (Employee e : staff)
25             System.out.println("name=" + e.getName()
26                                 + ",salary=" + e.getSalary());
27     }
28 }
```

Листинг 6.2. Исходный код из файла `interfaces/Employee.java`

```
1 package interfaces;
2
3 public class Employee implements Comparable<Employee>
4 {
5     private String name;
6     private double salary;
7
8     public Employee(String name, double salary)
9     {
10         this.name = name;
11         this.salary = salary;
12     }
13
14     public String getName()
15     {
16         return name;
17     }
18
19     public double getSalary()
20     {
21         return salary;
22     }
23
24     public void raiseSalary(double byPercent)
25     {
26         double raise = salary * byPercent / 100;
27         salary += raise;
28     }
29
30     /**
31      * Сравнивает работников по зарплате
32      * @param other Другой объект типа Employee
33      * @return Отрицательное значение, если зарплата
34      *         данного работника меньше, чем у другого
35      *         работника; нулевое значение, если их
36      *         зарплаты одинаковы; а иначе -
37      *         положительное значение
38      */
39     public int compareTo(Employee other)
40     {
41         return Double.compare(salary, other.salary);
42     }
43 }
```

`java.lang.Comparable<T>` 1.0

- `int compareTo(T other)`

Сравнивает текущий объект с заданным объектом *other* и возвращает отрицательное целое значение, если текущий объект меньше, чем объект *other*; нулевое значение, если объекты равны; а иначе — положительное целое значение.

java.util.Arrays 1.2

- **static void sort(Object[] a)**

Сортирует элементы массива **a**. Все элементы массива должны соответствовать классам, реализующим интерфейс **Comparable**, и быть совместимыми.

java.lang.Integer 1.0

- **static int compare(int x, int y) 7**

Возвращает отрицательное целое значение, если $x < y$; нулевое значение — $x = y$; а иначе — положительное целое значение.

java.lang.Double 1.0

- **static int compare(double x, double y) 1.4**

Возвращает отрицательное целое значение, если $x < y$; нулевое значение — $x = y$; а иначе — положительное целое значение.



НА ЗАМЕТКУ! В языке Java имеется следующее стандартное требование: “Автор реализации метода должен гарантировать, что для всех объектов **x** и **y** выполняется условие $\text{sgn}(x.\text{compareTo}(y)) = -\text{sgn}(y.\text{compareTo}(x))$ ”. (Это означает, что если при вызове **y.compareTo(x)** генерируется исключение, то же самое должно происходить и при вызове **x.compareTo(y)**.)” Здесь **sgn** означает знак числа: **sgn(n)** равно **-1**, если **n** — отрицательное число; **0**, если **n = 0**; или **1**, если **n** — положительное число. Иными словами, если поменять местами явный и неявный параметры метода **compareTo()**, знак возвращаемого числового значения (но не обязательно его фактическая величина) также должен измениться на противоположный.

Что касается метода **equals()**, то при наследовании могут возникнуть определенные затруднения. В частности, класс **Manager** расширяет класс **Employee**, а следовательно, он реализует интерфейс **Comparable<Employee>**, но не интерфейс **Comparable<Manager>**, как показано ниже.

```
class Manager extends Employee
{
    public int compareTo(Employee other)
    {
        Manager otherManager = (Manager) other; // Так нельзя!
        . . .
    }
    . . .
}
```

Но в таком случае нарушается правило “антисимметрии”. Если объект **x** относится к классу **Employee**, а объект **y** — к классу **Manager**, то вызов **x.compareTo(y)** не приведет к исключению, поскольку **x** и **y** будут сравнены как объекты класса **Employee**. А при противоположном вызове **y.compareTo(x)** будет сгенерировано исключение типа **ClassCastException**. Аналогичная ситуация возникает при реализации метода **equals()**, которая обсуждалась в главе 5. Из этого затруднительного положения имеются два выхода.

Если у подклассов разные представления о сравнении, нужно запретить сравнение объектов, принадлежащих разным классам. Таким образом, каждый метод `compareTo()` должен начинаться со следующей проверки:

```
if (getClass() != other.getClass()) throw new ClassCastException();
```

Если же существует общий алгоритм сравнения объектов подклассов, то в суперклассе следует реализовать единый метод `compareTo()` и объявить его как `final`.

Допустим, что в организации руководящие работники считаются выше рядовых по рангу, независимо от зарплаты. Как в таком случае быть с другими подклассами, например `Executive` и `Secretary`? Если требуется учредить нечто вроде табеля о рангах, то в класс `Employee` следует ввести метод `rank()`. Тогда в каждом подклассе метод `rank()` должен переопределяться, а результаты его работы учитываться при выполнении метода `compareTo()`.

6.1.2. Свойства интерфейсов

Интерфейсы — это не классы. В частности, с помощью операции `new` нельзя создать экземпляр интерфейса следующим образом:

```
x = new Comparable(...); // Неверно!
```

Но, несмотря на то, что нельзя конструировать интерфейсные объекты объявлять интерфейсные переменные можно следующим образом:

```
Comparable x; // Верно!
```

При этом интерфейсная переменная должна ссылаться на объект класса, реализующего данный интерфейс, как в приведенном ниже фрагменте кода.

```
x = new Employee(...); // Верно, если класс Employee
                        // реализует интерфейс Comparable
```

Как известно, в ходе операции `instanceof` проверяется, принадлежит ли объект заданному классу. Но с помощью этой операции можно также проверить, реализует ли объект заданный интерфейс:

```
if (anObject instanceof Comparable) { ... }
```

Аналогично классам, интерфейсы также могут образовывать иерархию наследования. Это позволяет создавать цепочки интерфейсов в направлении от более абстрактных к более специализированным. Допустим, имеется следующий интерфейс `Moveable`:

```
public interface Moveable
{
    void move(double x, double y);
}
```

В таком случае можно создать интерфейс `Powered`, расширяющий интерфейс `Moveable` следующим образом:

```
public interface Powered extends Moveable
{
    double milesPerGallon();
}
```

И хотя у интерфейса не может быть ни полей экземпляров, ни статических методов, в нем можно объявлять константы:

```
public interface Powered extends Moveable
{
    double milesPerGallon();
    double SPEED_LIMIT = 95;
    // открытая статическая конечная константа
}
```

Как и методы, поля констант в интерфейсах автоматически становятся открытыми. А кроме того, они являются статическими и конечными (т.е. имеют по умолчанию модификаторы доступа `public static final`).



НА ЗАМЕТКУ! Если указать ключевое слово `public` при объявлении метода в интерфейсе, а поля обозначить как `public static final`, это не будет ошибкой. Некоторые программисты поступают так по привычке или для того, чтобы исходные коды их программ были более удобочитаемыми. Но в спецификации Java не рекомендуется употреблять лишние ключевые слова, и здесь мы следуем этим рекомендациям.

В некоторых интерфейсах объявляются только константы и ни одного метода. Например, в стандартной библиотеке имеется интерфейс `SwingConstants`, определяющий константы `NORTH`, `SOUTH`, `HORIZONTAL` и т.д. Любой класс, реализующий интерфейс `SwingConstants`, автоматически наследует эти константы. Его методы могут, например, непосредственно ссылаться на константу `NORTH`, не прибегая к громоздкому обозначению `SwingConstants.NORTH`. Тем не менее такое применение интерфейсов считается изжившим себя и поэтому не рекомендуется.

Любой класс в Java может иметь только один суперкласс, но в то же время любой класс может реализовывать несколько интерфейсов, что позволяет максимально гибко определять поведение класса. Например, в Java имеется очень важный интерфейс `Cloneable`, подробнее рассматриваемый в разделе 6.1.9. Так, если какой-нибудь класс реализует интерфейс `Cloneable`, то для создания точных копий его объектов можно применять метод `clone()` из класса `Object`. А если требуется не только создавать клоны объектов данного класса, но и сравнивать их, тогда в этом классе нужно реализовать оба интерфейса, `Cloneable` и `Comparable`, как показано ниже. Для разделения имен интерфейсов, задающих свойства классов, служит запятая.

```
class Employee implements Cloneable, Comparable
```

6.1.3. Интерфейсы и абстрактные классы

Если вы помните содержание раздела главы 5, посвященного абстрактным классам, то у вас могут возникнуть следующие резонные вопросы: зачем разработчики языка Java создали механизм интерфейсов и почему бы не сделать интерфейс `Comparable` абстрактным классом, например, так, как показано ниже?

```
abstract class Comparable // Почему бы и нет?
{
    public abstract int compareTo(Object other);
}
```

В этом случае рассматриваемый здесь класс `Employee` мог бы просто расширять абстрактный и реализовывать метод `compareTo()` следующим образом:

```
class Employee extends Comparable // Почему бы и нет?
{
    public int compareTo(Object other) { ... }
}
```


К сожалению, это породило бы массу проблем, связанных с использованием абстрактного базового класса для выражения обобщенного свойства. Ведь каждый класс может расширять только один класс. Допустим, класс `Employee` уже является подклассом какого-нибудь другого класса, скажем, `Person`. Это означает, что он уже не может расширять еще один класс следующим образом:

```
class Employee extends Person, Comparable // ОШИБКА!
```

Но в то же время каждый класс может реализовывать сколько угодно интерфейсов, как показано ниже.

```
class Employee extends Person
    implements Comparable // Верно!
```

В других языках программирования и, в частности, в C++ у классов может быть несколько суперклассов. Это языковое средство называется *множественным наследованием*. Создатели Java решили не поддерживать множественное наследование, поскольку оно делает язык слишком сложным (как C++) или менее эффективным (как Eiffel). В то же время интерфейсы обеспечивают большинство преимуществ множественного наследования, не усложняя язык и не снижая его эффективность.



НА ЗАМЕТКУ C++! В языке C++ допускается множественное наследование. Это вызывает много осложнений, связанных с виртуальными базовыми классами, правилами доминирования и приведением типов указателей. Множественным наследованием пользуются лишь немногие программирующие на C++. Некоторые вообще им не пользуются, а остальные рекомендуют применять множественное наследование только в "примешанном" виде. Это означает, что первичный базовый класс описывает родительский объект, а дополнительные базовые классы (так называемые *примеси*) описывают вспомогательные свойства. Такой подход чем-то напоминает те классы в Java, у которых имеется только один базовый класс и дополнительные интерфейсы. Но в C++ примеси позволяют добавлять некоторые виды поведения по умолчанию, тогда как интерфейсы в Java на это не способны.

6.1.4. Статические и закрытые методы

Начиная с версии Java 8, в интерфейсы разрешается вводить статические методы. Формальных причин, по которым в интерфейсе не могли бы присутствовать статические методы, никогда не существовало. Но такие методы не согласовывались с представлением об интерфейсах как об абстрактных спецификациях.

В прошлом статические методы зачастую определялись в дополнительном классе, сопутствующем интерфейсу. В стандартной библиотеке Java можно обнаружить пары интерфейсов и служебных классов, например `Collection/Collections` или `Path/Paths`. Такое разделение больше не требуется.

Рассмотрим в качестве примера класс `Paths`. У него имеется пара фабричных методов для составления пути к файлу или каталогу из последовательности символьных строк, например, таким образом: `Paths.get("jdk1-11", "conf", "security")`. В версии Java 11 равнозначные методы предоставляются в интерфейсе `Path` следующим образом:

```
public interface Path
{
    public static Path of(URI uri)
    { . . . }
    public static Path of(String first, String... more)
    { . . . }
    . . .
}
```

В таком случае потребность в классе `Paths` просто отпадает. Аналогично, реализуя собственные интерфейсы, больше не имеет смысла предоставлять отдельный сопутствующий класс для служебных методов.

Начиная с версии Java 9, методы в интерфейсе могут быть объявлены закрытыми (`private`). Закрытый метод может быть статическим или же методом экземпляра. А поскольку закрытые методы можно использовать в других методах самого интерфейса, то их применение ограничивается ролью вспомогательных методов для других методов данного интерфейса.

6.1.5. Методы с реализацией по умолчанию

Для любого интерфейсного метода можно предоставить реализацию *по умолчанию*. Такой метод следует пометить модификатором доступа `default`, как показано ниже.

```
public interface Comparable<T>
{
    default int compareTo(T other) { return 0; }
    // по умолчанию все элементы одинаковы
}
```

Безусловно, пользы от такого метода не очень много, поскольку в каждой настоящей реализации интерфейса `Comparable` он будет переопределен. Но иногда методы с реализацией по умолчанию оказываются все же полезными. Например, в главе 9 будет представлен интерфейс `Iterator` для обращения к элементам структуры данных. В этом интерфейсе объявляется метод `remove()`:

```
public interface Iterator<E>
{
    boolean hasNext();
    E next();
    default void remove() {
        throw new UnsupportedOperationException("remove");
    }
    . . .
}
```

Чтобы реализовать итератор, придется предоставить методы `hasNext()` и `next()`. Для этих методов отсутствуют реализуемые по умолчанию методы, поскольку они зависят от конкретной структуры данных, которую требуется обойти. Но если реализуемый итератор доступен только для чтения, то о методе `remove()` можно вообще не беспокоиться.

Из методов с реализацией по умолчанию можно вызывать другие методы. Например, в интерфейсе `Collection` можно определить служебный метод `isEmpty()`, как показано ниже. И тогда программисту, реализующему интерфейс `Collection`, не придется беспокоиться о реализации этого метода.

```
public interface Collection
{
    int size(); // абстрактный метод
    default boolean isEmpty()
    {
        return size() == 0;
    }
    . . .
}
```



НА ЗАМЕТКУ! На самом деле интерфейс `Collection` из прикладного интерфейса Java API ничего подобного не делает. Вместо этого в нем имеется абстрактный класс `AbstractCollection`, реализующий интерфейс `Collection` и определяющий метод `isEmpty()` с точки зрения метода `size()`. Тем, кто реализует коллекции, обычно рекомендуется расширять класс `AbstractCollection`. Но такой способ устарел, поскольку достаточно реализовать соответствующие методы в интерфейсе.

Методы с реализацией по умолчанию играют важную роль в дальнейшем развитии интерфейсов. Рассмотрим в качестве примера интерфейс `Collection`, многие годы входящий в состав стандартной библиотеки Java. Допустим, некогда был предоставлен следующий класс, реализующий интерфейс `Collection`:

```
public class Bag implements Collection
```

а впоследствии, начиная с версии Java 8, в этот интерфейс был внедрен метод `stream()`.

Допустим также, что метод `stream()` не является методом с реализацией по умолчанию. В таком случае класс `Bag` больше не компилируется, поскольку он не реализует новый метод из интерфейса `Collection`. Таким образом, внедрение в интерфейс метода с реализацией не по умолчанию нарушает *совместимость на уровне исходного кода*.

Но допустим, что этот класс не перекомпилируется и просто используется содержащий его старый архивный JAR-файл. Этот класс по-прежнему загружается, несмотря на отсутствующий в нем метод. В программах могут по-прежнему строиться экземпляры класса `Bag`, и ничего плохого не произойдет. (Внедрение метода в интерфейс *совместимо на уровне двоичного кода*.) Но если в программе делается вызов метода `stream()` для экземпляра класса `Bag`, то возникает ошибка типа `AbstractMethodError`.

Подобные затруднения можно устранить, если объявить метод `stream()` как `default`. И тогда класс `Bag` будет компилироваться снова. А если этот класс загружается без перекомпиляции и метод `stream()` вызывается для экземпляра класса `Bag`, то такой вызов происходит по ссылке `Collection.stream`.

6.1.6. Разрешение конфликтов с методами по умолчанию

Что, если один и тот же метод сначала определен по умолчанию в одном интерфейсе, а затем таким же образом в другом интерфейсе или как метод в суперклассе? В таких языках, как Scala и C++, действуют сложные правила разрешения подобных неоднозначностей. А в Java подобные правила оказываются намного более простыми и заключаются в следующем.

1. В конфликте верх одерживает суперкласс. Если суперкласс предоставляет конкретный метод, то методы по умолчанию с одинаковыми именами и типами параметров просто игнорируются.
2. Интерфейсы вступают в конфликт. Если суперинтерфейс предоставляет метод, а другой интерфейс — метод (по умолчанию или иначе) с таким же самым именем и типами параметров, то для разрешения конфликта необходимо переопределить этот метод.

Рассмотрим второе правило. Допустим, в другом интерфейсе определен метод `getName()` следующим образом:

```
interface Person
{
    default String getName() { return ""; };
}

interface Named
{
    default String getName()
    { return getClass().getName() + "_" + hashCode(); }
}
```

Что произойдет, если сформировать приведенный ниже класс, реализующий оба интерфейса?

```
class Student implements Person, Named { . . . }
```

Этот класс наследует оба конфликтующих метода `getName()`, предоставляемых интерфейсами `Person` и `Named`. Вместо того чтобы выбрать один из этих методов, компилятор Java выдаст сообщение об ошибке, предоставив программисту самому разрешать возникшую неоднозначность. Для этого достаточно предоставить метод `getName()` в классе `Student` и выбрать в нем один из конфликтующих методов следующим образом:

```
class Student implements Person, Named
{
    public String getName()
    { return Person.super.getName(); }
    . . .
}
```

А теперь допустим, что в интерфейсе `Named` не предоставляется реализация метода `getName()` по умолчанию:

```
interface Named
{
    String getName();
}
```

Может ли класс `Student` унаследовать метод по умолчанию из интерфейса `Person`? На первый взгляд это может показаться вполне обоснованным, но разработчики Java решили сделать выбор в пользу единообразия. Независимо от характера конфликта между двумя интерфейсами, хотя бы в одном из них предоставляется реализация искомого метода, а компилятор выдаст сообщение об ошибке, предоставив программисту возможность самому разрешать возникшую неоднозначность.



НА ЗАМЕТКУ! Если ни один из интерфейсов не предоставляет реализацию по умолчанию общего для них метода, то никакого конфликта не возникает, как, собственно, и было до версии Java 8. В таком случае для класса, реализующего эти интерфейсы, имеются два варианта выбора: реализовать метод или оставить его нереализованным и объявить класс как **abstract**.

Итак, мы рассмотрели конфликты между интерфейсами. Теперь рассмотрим класс, расширяющий суперкласс и реализующий интерфейс, наследуя от обоих один и тот же метод. Допустим, класс `Student`, наследующий от класса `Person` и реализующий интерфейс `Named`, определяется следующим образом:

```
class Student extends Person implements Named { . . . }
```

В таком случае значение имеет только метод из суперкласса, а любой метод с реализацией по умолчанию из интерфейса игнорируется. В данном примере класс `Student` наследует метод `getName()` из класса `Person`, и для него совершенно не важно, предоставляет ли интерфейс `Named` такой же самый метод по умолчанию. Ведь в этом случае соблюдается первое упомянутое выше правило, согласно которому в конфликте верх одерживает суперкласс.

Это правило гарантирует совместимость с версией Java 7. Если ввести методы с реализацией по умолчанию в интерфейс, это никак не скажется на работоспособности прикладного кода, написанного до появления подобных методов в интерфейсах.



ВНИМАНИЕ! Метод с реализацией по умолчанию, переопределяющий один из методов в классе `Object`, создать нельзя. Например, в интерфейсе нельзя определить метод с реализацией по умолчанию для метода `toString()` или `equals()`, даже если это и покажется привлекательным для таких интерфейсов, как `List`. Как следствие из правила, когда в конфликте верх одерживает суперкласс, такой метод вообще не смог бы одолеть метод `Object.toString()` или `Object.equals()`.

6.1.7. Интерфейсы и обратные вызовы

Характерным для программирования шаблоном является *обратный вызов*. В этом шаблоне указывается действие, которое должно произойти там, где наступает конкретное событие. Такое событие может, например, произойти в результате щелчка на экранной кнопке или выбора пункта меню в пользовательском интерфейсе. Но поскольку мы еще не касались вопросов реализации пользовательских интерфейсов, то рассмотрим похожую, но более простую ситуацию.

В состав пакета `javax.swing` входит класс `Timer`, с помощью которого можно уведомлять об истечении заданного промежутка времени. Так, если в части программы производится отсчет времени на циферблате часов, то можно организовать уведомление каждую секунду, чтобы обновлять циферблат.

При построении таймера задается промежуток времени и указывается, что он должен сделать по истечении заданного промежутка времени. Но как указать таймеру, что он должен сделать? Во многих языках программирования с этой целью предоставляется имя функции, которую таймер должен периодически вызывать. Но в классах из стандартной библиотеки Java принят объектно-ориентированный подход. В частности, таймеру передается объект некоторого класса, а таймер вызывает один из методов для объекта этого класса. Передача объекта считается более гибким подходом, чем передача функции, поскольку объект может нести дополнительную информацию.

Безусловно, таймеру должно быть известно, какой именно метод вызывать. Поэтому он требует указать объект класса, реализующего интерфейс `ActionListener` из пакета `java.awt.event`, объявление которого приведено ниже. Таким образом, таймер вызывает метод `actionPerformed()` из этого интерфейса по истечении заданного промежутка времени.

```
public interface ActionListener
{
    void actionPerformed(ActionEvent event);
}
```

Допустим, что каждые 10 секунд требуется выводить сообщение "At the tone, the time is . . ." (Время по звуковому сигналу). С этой целью можно определить класс, реализующий интерфейс `ActionListener`, разместив операторы, которые требуется выполнить, в теле метода `actionPerformed()`, как показано ниже.

```
class TimePrinter implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        System.out.println("At the tone, the time is "
            + Instant.ofEpochMilli(event.getWhen()));
        Toolkit.getDefaultToolkit().beep();
    }
}
```

Обратите внимание на параметр `event` типа `ActionEvent` в вызове метода `actionPerformed()`. Он предоставляет сведения о наступившем событии, в том числе время наступления события. Так, в результате вызова `event.getWhen()` возвращается время наступления события, которое отсчитывается в миллисекундах от начала так называемой “эпохи”, т.е. с 1 января 1970 г. Передав это время статическому методу `Instant.ofEpochMilli()`, можно в итоге получить более удобочитаемое его описание.

Далее конструируется объект класса `TimePrinter`, который затем передается конструктору класса `Timer`:

```
var = new TimePrinter();
Timer t = new Timer(10000, listener);
```

В качестве первого параметра конструктора класса `Timer` служит промежуток времени, который должен истекать между последовательными уведомлениями и который измеряется в миллисекундах, а в качестве второго параметра — объект приемника событий. И, наконец, таймер запускается следующим образом:

```
t.start();
```

В итоге каждую секунду на экран выводится сообщение, аналогичное приведенному ниже, и сопровождается соответствующим звуковым сигналом.

```
At the tone, the time is 2017-12-16T05:01:49.550Z
```

В листинге 6.3 демонстрируется применение такого таймера на практике. После запуска таймера программа открывает диалоговое окно для вывода сообщений и ожидает до тех пор, пока пользователь не щелкнет на экранной кнопке ОК, чтобы остановить ее выполнение. По ходу этого ожидания текущее время отображается через каждую секунду. (Если опустить открытие диалогового окна, данная программа завершится, как только произойдет возврат из метода `main()`.)

Листинг 6.3. Исходный код из файла `timer/TimerTest.java`

```
1 package timer;
2
3 /**
4     @version 1.02 2017-12-14
5     @author Cay Horstmann
6 */
7
8 import java.awt.*;
9 import java.awt.event.*;
10 import java.time.*;
11 import javax.swing.*;
12
```

```
13 public class TimerTest
14 {
15     public static void main(String[] args)
16     {
17         var listener = new TimePrinter();
18
19         // построить таймер, вызывающий приемник событий
20         // каждую секунду
21         var timer = new Timer(1000, listener);
22         timer.start();
23
24         // продолжить выполнение программы до тех пор, пока
25         // пользователь не выберет экранную кнопку "ОК"
26         JOptionPane.showMessageDialog(null,
27                                     "Quit program?");
28         System.exit(0);
29     }
30 }
31
32 class TimePrinter implements ActionListener
33 {
34     public void actionPerformed(ActionEvent event)
35     {
36         System.out.println("At the tone, the time is "
37                             + Instant.ofEpochMilli(event.getWhen()));
38         Toolkit.getDefaultToolkit().beep();
39     }
40 }
```

javax.swing.JOptionPane 1.2

- **static void showMessageDialog(Component parent, Object message)**

Отображает диалоговое окно с подсказкой сообщений и экранной кнопкой ОК. Это диалоговое окно располагается по центру родительского компонента, обозначаемого параметром **parent**. Если же параметр **parent** принимает пустое значение **null**, диалоговое окно располагается по центру экрана.

javax.swing.Timer 1.2

- **Timer(int interval, ActionListener listener)**

Строит таймер, уведомляющий указанный приемник событий **listener** всякий раз, когда истекает промежуток времени, заданный в миллисекундах.

- **void start()**

Запускает таймер. Как только таймер будет запущен, он вызывает метод **actionPerformed()** для приемников своих событий.

- **void stop()**

Останавливает таймер. Как только таймер будет остановлен, он больше не вызывает метод **actionPerformed()** для приемников своих событий.

```
java.awt.Toolkit 1.0
```

- **static Toolkit getDefaultToolkit()**
Получает набор инструментов, выбираемый по умолчанию. Этот набор содержит сведения о среде GUI.
- **void beep()**
Издает звуковой сигнал.

6.1.8. Интерфейс Comparator

В разделе 6.1.1. было показано, каким образом сортируется массив объектов, при условии, что они являются экземплярами классов, реализующих интерфейс Comparable. Например, можно отсортировать массив символьных строк, поскольку класс String реализует интерфейс Comparable<String>, а метод String.compareTo() сравнивает символьные строки в лексикографическом порядке.

А теперь допустим, что требуется отсортировать символьные строки по порядку увеличения их длины, а не в лексикографическом порядке. В классе String нельзя реализовать метод compareTo() двумя разными способами. К тому же этот класс относится к стандартной библиотеке Java и не подлежит изменению.

В качестве выхода из этого положения можно воспользоваться вторым вариантом метода Arrays.sort(), параметрами которого являются массив и *компаратор* — экземпляр класса, реализующего приведенный ниже интерфейс Comparator.

```
public interface Comparator<T> {  
    int compare(T first, T second);  
}
```

Чтобы сравнить символьные строки по длине, достаточно определить класс, реализующий интерфейс Comparator<String>:

```
class LengthComparator implements Comparator<String> {  
    public int compare(String first, String second) {  
        return first.length() - second.length();  
    }  
}
```

Чтобы произвести сравнение, фактически требуется получить экземпляр данного класса следующим образом:

```
var comp = new LengthComparator();  
if (comp.compare(words[i], words[j]) > 0) ...
```

Сравните этот фрагмент кода с вызовом words[i].compareTo(words[j]). Метод compare() вызывается для объекта компаратора, а не для самой символьной строки.



НА ЗАМЕТКУ! Несмотря на то что у объекта типа LengthComparator отсутствует состояние, его экземпляр все же требуется получить, чтобы вызвать метод compare(), который не является статическим.

Чтобы отсортировать массив, достаточно передать объект типа LengthComparator методу Arrays.sort() следующим образом:

```
String[] friends = { "Peter", "Paul", "Mary" };  
Arrays.sort(friends, new LengthComparator());
```


Теперь массив упорядочен как ["Paul", "Mary", "Peter"] или ["Mary", "Paul", "Peter"]. Далее, в разделе 6.2, будет показано, как лямбда-выражения упрощают пользование интерфейсом `Comparator`.

6.1.9. Клонирование объектов

В этом разделе рассматривается интерфейс `Cloneable`, который обозначает, что в классе предоставляется надежный метод `clone()`. Клонирование объектов производится нечасто, а подробности данного процесса носят совершенно технический характер, поэтому вы можете не обращаться к материалу этого раздела до тех пор, пока он вам не понадобится.

Чтобы понять назначение клонирования объектов, напомним, что же происходит, когда создается копия переменной, содержащей ссылку на объект. В этом случае оригинал и копия переменной содержат ссылки на один и тот же объект (рис. 6.1). Это означает, что изменение одной переменной повлечет за собой изменение другой:

```
var = new Employee("John Public", 50000);  
Employee copy = original;  
copy.raiseSalary(10); // Оригинал тоже изменился!
```

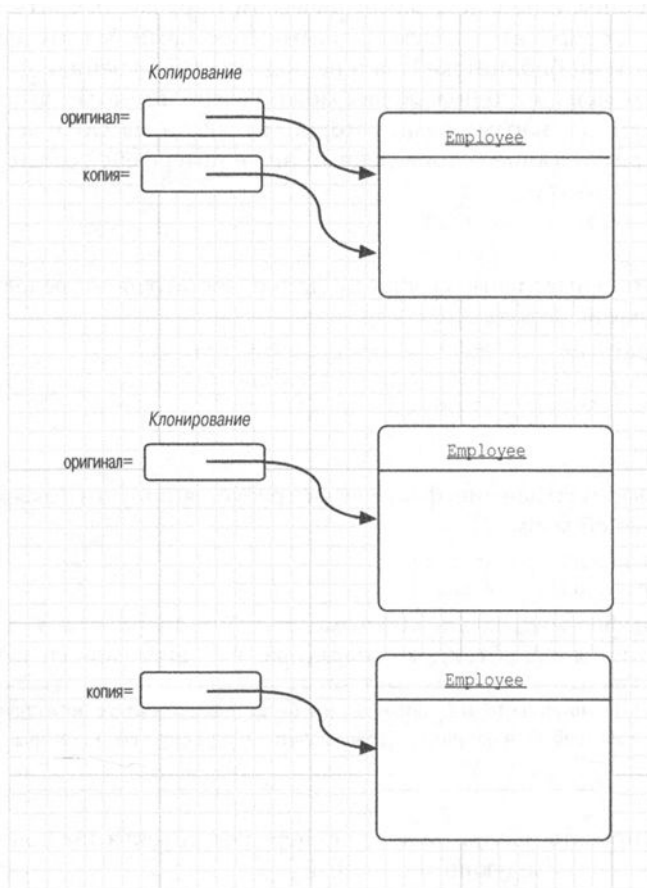


Рис. 6.1. Копирование и клонирование объектов

Если же требуется, чтобы переменная `copy` представляла новый объект, который в первый момент своего существования идентичен объекту `original`, но совершенно независим от него, в таком случае нужно воспользоваться методом `clone()` следующим образом:

```
Employee copy = original.clone();  
copy.raiseSalary(10); // Теперь оригинал не изменился!
```

Но не все так просто. Метод `clone()` является защищенным (`protected`) в классе `Object`, т.е. его нельзя вызвать непосредственно. И только класс `Employee` может клонировать объекты своего класса. Для этого ограничения имеется своя веская причина. Проанализируем, каким образом класс `Object` может реализовать метод `clone()`. Ему вообще ничего не известно об объекте, поэтому он может копировать лишь поля. Если все поля класса являются числовыми или имеют другой основной тип, их копирование выполняется нормально. Но если объект содержит ссылку на подобъект, то оригинал и клонированные объекты будут совместно использовать одни и те же данные.

Чтобы проиллюстрировать это явление, рассмотрим класс `Employee`, которым, начиная с главы 4, мы пользуемся для демонстрации различных особенностей работы с объектами. На рис. 6.2 показано, что происходит, когда метод `clone()` из класса `Object` применяется для клонирования объекта типа `Employee`. Как видите, операция клонирования по умолчанию является неполной — она не копирует объекты, на которые имеются ссылки в других объектах. (На рис. 6.2 показан общий объект `Date`. По причинам, которые станут ясны в дальнейшем, в данном примере используется вариант класса `Employee`, в котором день приема на работу представлен объектом типа `Date`.)

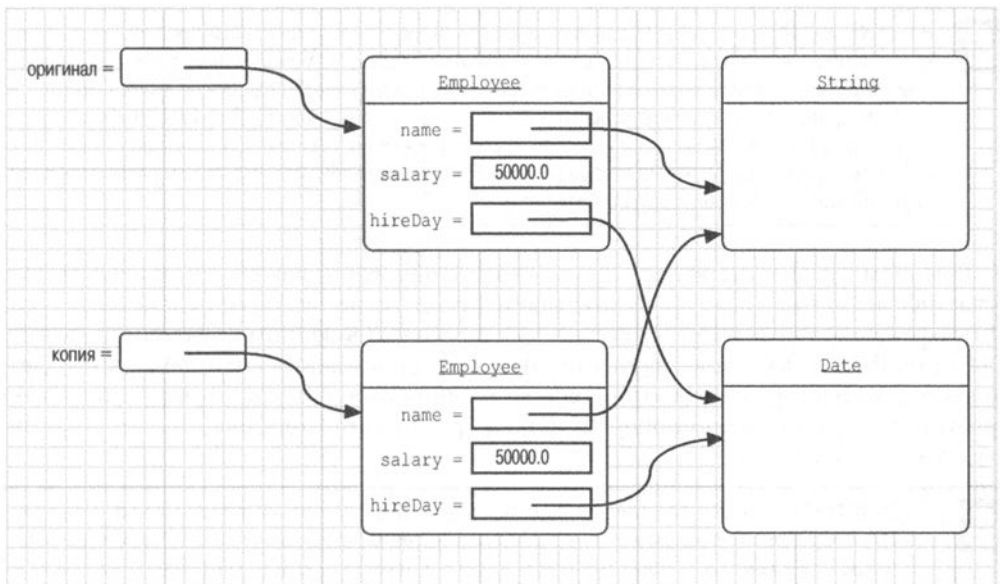


Рис. 6.2. Неполное копирование

Так ли уж плохо неполное копирование? Все зависит от конкретной ситуации. Если подобъект, используемый совместно как оригиналом, так и с неполным клоном, является неизменяемым, это вполне безопасно. Такое случается, если подобъект

является экземпляром неизменяемого класса, например `String`. С другой стороны, подобъект может оставаться постоянным на протяжении всего срока действия того объекта, который его содержит, не подвергаясь воздействию модифицирующих методов или методов, вычисляющих ссылку на него.

Но подобъекты зачастую подвергаются изменениям, поэтому приходится перепределять метод `clone()`, чтобы выполнить *полное* копирование, которое позволяет клонировать подобъекты наряду с содержащими их объектами. В данном примере поле `hireDay` ссылается на экземпляр изменяемого класса `Date`, и поэтому он должен быть также клонирован. (Здесь используется поле типа `Date`, а не типа `LocalDate` именно для того, чтобы продемонстрировать особенности процесса клонирования. Если бы поле `hireDay` ссылалось на экземпляр неизменяемого класса `LocalDate`, то никаких дополнительных действий не потребовалось бы.)

Для каждого класса нужно принять следующие решения.

1. Достаточно ли метода `clone()`, предоставляемого по умолчанию?
2. Можно ли доработать предоставляемый по умолчанию метод `clone()` таким образом, чтобы вызывать его для изменяемых объектов?
3. Следует ли вообще отказаться от применения метода `clone()`?

По умолчанию принимается последнее решение. А для принятия первого и второго решения класс должен удовлетворять следующим требованиям.

1. Реализация интерфейса `Cloneable`.
2. Переопределение метода `clone()` с модификатором доступа `public`.



НА ЗАМЕТКУ! Метод `clone()` объявлен в классе `Object` как защищенный (`protected`), поэтому его нельзя просто вызвать по ссылке `anObject.clone()`. Но разве недоступны защищенные методы для любого подкласса, и не является ли каждый класс подклассом класса `Object`? К счастью, правила защищенного доступа не такие строгие (см. главу 5). Подкласс может вызвать защищенный метод `clone()` только для клонирования своих собственных объектов. Чтобы клонировать другие объекты, метод `clone()` следует переопределить как открытый и разрешить клонирование объектов любым другим методом.

В данном случае интерфейс `Cloneable` используется не совсем обычным образом. В частности, метод `clone()` не объявляется в нем, а наследуется от класса `Object`. Интерфейс служит меткой, указывающей на то, что в данном случае разработчик класса понимает, как выполняется процесс клонирования. В языке Java наблюдается настолько настороженное отношение к клонированию объектов, что если объект требует выполнения данной операции, но не реализует интерфейс `Cloneable`, то генерируется исключение.



НА ЗАМЕТКУ! Интерфейс `Cloneable` — один из немногих помеченных интерфейсов в Java, иногда еще называемых маркерными интерфейсами. Напомним, что назначение обычных интерфейсов вроде `Comparable` — обеспечить реализацию в некотором классе конкретного метода или ряда методов, объявленных в данном интерфейсе. У маркерных интерфейсов отсутствуют методы, а их единственное назначение — разрешить выполнение операции `instanceof` для проверки типа следующим образом:

```
if (obj instanceof Cloneable) ...
```

Но пользоваться маркерными интерфейсами в прикладных программах все же не рекомендуется.

Даже если реализация метода `clone()` по умолчанию (неполное копирование) вполне подходит, все равно нужно реализовать также интерфейс `Cloneable`, переопределить метод `clone()` как открытый и сделать вызов `super.clone()`, как показано в следующем примере кода:

```
class Employee implements Cloneable
{
    // сделать метод открытым, изменить возвращаемый тип
    public Employee clone() throws CloneNotSupportedException
    {
        return (Employee) super.clone();
    }
    . . .
}
```



НА ЗАМЕТКУ! Вплоть до версии Java 1.4 метод `clone()` всегда возвращал объект типа `Object`. Поддерживаемые теперь ковариантные возвращаемые типы (см. главу 5) позволяют указывать в методе `clone()` правильный тип возвращаемого значения.

Рассмотренный выше метод `clone()` не добавляет никаких новых функциональных возможностей к методу `Object.clone()`, реализующему неполное копирование. Чтобы реализовать полное копирование, придется приложить дополнительные усилия и организовать клонирование изменяемых полей экземпляра. Ниже приведен пример реализации метода `clone()`, выполняющего полное копирование.

```
class Employee implements Cloneable
{
    . . .
    public Employee clone() throws CloneNotSupportedException
    {
        // вызвать метод Object.clone()
        Employee cloned = (Employee) super.clone();
        // клонировать изменяемые поля
        cloned.hireDay = (Date) hireDay.clone();

        return cloned;
    }
}
```

Метод `clone()` из класса `Object` может генерировать исключение типа `CloneNotSupportedException`. Это происходит в том случае, если метод `clone()` вызывается для объекта, не реализующего интерфейс `Cloneable`. Но поскольку классы `Employee` и `Date` реализуют этот интерфейс, то исключение не генерируется. Впрочем, компилятору об этом ничего неизвестно, и поэтому о возможном исключении приходится объявлять следующим образом:

```
public Employee clone() throws CloneNotSupportedException
```



НА ЗАМЕТКУ! Не лучше ли было бы вместо этого предусмотреть обработку исключения, как показано ниже? (Подробнее о перехвате исключений речь пойдет в главе 7.)

```
public Employee clone()
{
    try
    {
```

```

        return super.clone();
    }
    catch (CloneNotSupportedException e) { return null; }
    // Этого не произойдет, так как данный класс
    // реализует интерфейс Cloneable
}

```

Такое решение вполне подходит для конечных классов, объявляемых как **final**. А в остальном уместнее прибегнуть к помощи ключевого слова **throws**. В этом случае у подкласса останется возможность сгенерировать исключение типа **CloneNotSupportedException**, если он не в состоянии поддерживать клонирование.

Клонируя объекты подклассов, следует соблюдать особую осторожность. Так, если вы определите метод `clone()` в классе `Employee`, другие смогут воспользоваться им для клонирования объектов типа `Manager`. Сможет ли метод `clone()` из класса `Employee` справиться с подобной задачей? Это зависит от набора полей, объявленных в классе `Manager`. В данном случае никаких затруднений не возникнет, поскольку поле `bonus` относится к примитивному типу. Но ведь в класс `Manager` может быть введено поле, требующее полного копирования или вообще не допускающее клонирование. Нет никакой гарантии, что в подклассе реализован метод `clone()`, корректно решающий поставленную задачу. Именно поэтому метод `clone()` объявлен в классе `Object` как `protected`. Но если вы хотите, чтобы пользователи ваших классов могли вызывать метод `clone()`, то подобная роскошь остается для вас недоступной.

Следует ли реализовывать метод `clone()` в своих классах? Если пользователям требуется полное копирование, то ответ, конечно, должен быть положительным. Некоторые специалисты считают, что от метода `clone()` нужно вообще отказаться и реализовать вместо него другой метод, решающий аналогичную задачу. Можно, конечно, согласиться с тем, что метод `clone()` — не совсем удачное решение, но если вы передадите его функции другому методу, то столкнетесь с теми же трудностями. Как бы то ни было, клонирование применяется нечасто. Достаточно сказать, что метод `clone()` реализован менее чем в 5% классов из стандартной библиотеки.

В программе, исходный код которой приведен в листинге 6.4, сначала клонируются объекты класса `Employee` (из листинга 6.5), затем вызываются два модифицирующих метода. Метод `raiseSalary()` изменяет значение в поле `salary`, а метод `setHireDay()` — состояние в поле `hireDay`. Ни один из модифицирующих методов не воздействует на исходный объект, поскольку метод `clone()` переопределен и осуществляет полное копирование.



НА ЗАМЕТКУ! У всех видов массивов имеется открытый, а не защищенный метод `clone()`. Им можно воспользоваться для создания нового массива, содержащего копии всех элементов, как в следующем примере кода:

```

int[] luckyNumbers = { 2, 3, 5, 7, 11, 13 };
int[] cloned = (int[]) luckyNumbers.clone();
cloned[5] = 12; // не изменяет элемент
                // массива luckyNumbers[5]

```



НА ЗАМЕТКУ! В главе 2 второго тома настоящего издания представлен альтернативный механизм клонирования объектов с помощью средства сериализации объектов в Java. Этот механизм прост в реализации и безопасен, хотя и не очень эффективен.

Листинг 6.4. Исходный код из файла `clone/CloneTest.java`

```
1 package clone;
2
3 /**
4  * этой программе демонстрируется клонирование объектов
5  * @version 1.11 2018-03-16
6  * @author Cay Horstmann
7  */
8 public class CloneTest
9 {
10     public static void main(String[] args)
11         throws CloneNotSupportedException
12     {
13         var original = new Employee("John Q. Public",
14                                     50000);
15         original.setHireDay(2000, 1, 1);
16         Employee copy = original.clone();
17         copy.raiseSalary(10);
18         copy.setHireDay(2002, 12, 31);
19         System.out.println("original=" + original);
20         System.out.println("copy=" + copy);
21     }
22 }
```

Листинг 6.5. Исходный код из файла `clone/Employee.java`

```
1 package clone;
2
3 import java.util.Date;
4 import java.util.GregorianCalendar;
5
6 public class Employee implements Cloneable
7 {
8     private String name;
9     private double salary;
10    private Date hireDay;
11
12    public Employee(String name, double salary)
13    {
14        this.name = name;
15        this.salary = salary;
16        hireDay = new Date();
17    }
18
19    public Employee clone()
20        throws CloneNotSupportedException
21    {
22        // вызвать метод Object.clone()
23        Employee cloned = (Employee) super.clone();
24
25        // клонировать изменяемые поля
26        cloned.hireDay = (Date) hireDay.clone();
27    }
```

```
28     return cloned;
29 }
30
31 /**
32  * Устанавливает заданную дату приема на работу
33  * @param year Год приема на работу
34  * @param month Месяц приема на работу
35  * @param day День приема на работу
36 */
37 public void setHireDay(int year, int month, int day)
38 {
39     Date newHireDay = new GregorianCalendar(year,
40                                             month - 1, day).getTime();
41
42     // Пример изменения поля экземпляра
43     hireDay.setTime(newHireDay.getTime());
44 }
45
46 public void raiseSalary(double byPercent)
47 {
48     double raise = salary * byPercent / 100;
49     salary += raise;
50 }
51
52 public String toString()
53 {
54     return "Employee[name=" + name + ",salary="
55           + salary + ",hireDay=" + hireDay + "];"
56 }
57 }
```

6.2. Лямбда-выражения

А теперь можно приступить к рассмотрению лямбда-выражений — самого привлекательного изменения в языке Java за последние годы. В этом разделе будет показано, как пользоваться лямбда-выражениями для определения блоков кода с помощью лаконичного синтаксиса и как писать прикладной код, в котором употребляются лямбда-выражения.

6.2.1. Причины для употребления лямбда-выражений

Лямбда-выражение — это блок кода, который передается для последующего выполнения один или несколько раз. Прежде чем рассматривать синтаксис (или даже любопытное название) лямбда-выражений, вернемся немного назад, чтобы выяснить, где именно мы уже пользовались подобными блоками кода в Java.

В разделе 6.1.7 было показано, как обращаться с устанавливаемыми промежутками времени. Для этого достаточно ввести нужные операторы в тело метода `actionPerformed()` из интерфейса `ActionListener`, реализуемого в конкретном классе:

```
class Worker implements ActionListener
{
    public void actionPerformed(ActionEvent event)
```

```
{  
    // сделать что-нибудь  
}
```

Когда же этот код потребуется выполнить повторно, достаточно будет получить экземпляр класса `Worker`, а затем передать его объекту типа `Timer` при его построении. Но самое главное, что метод `actionPerformed()` содержит код, который требуется выполнить не сразу, а впоследствии.

Рассмотрим в качестве другого примера сортировку с помощью специального компаратора. Так, если требуется отсортировать символьные строки по длине, а не в выбираемом по умолчанию лексикографическом порядке, то методу `sort()` можно передать объект класса, реализующего интерфейс `Comparator`:

```
class LengthComparator implements Comparator<String>  
{  
    public int compare(String first, String second)  
    {  
        return first.length() - second.length();  
    }  
}  
...  
Arrays.sort(strings, new LengthComparator());
```

Метод `compare()` вызывается из метода `sort()` не сразу, а лишь тогда, когда требуется перестроить элементы в нужном порядке при сортировке массива. С этой целью методу `sort()` предоставляется блок кода, требующийся для сравнения элементов, и этот код настолько интегрирован в остальную часть логики, что о повторной его реализации можно даже не беспокоиться.

У обоих рассмотренных здесь примеров имеется нечто общее: блок кода, передаваемый таймеру или методу `sort()`. И этот блок кода вызывается не сразу, а впоследствии. До версии Java 8 передать блок кода на выполнение было не так-то просто. Ведь Java — объектно-ориентированный язык программирования, поэтому программирующим на нем приходится конструировать объект, относящийся к определенному классу, где имеется метод с требующимся кодом.

В других языках программирования допускается непосредственная передача блоков кода. Но разработчики долго противились внедрению такой возможности в Java. Ведь главная сила языка Java — в его простоте и согласованности. Этот язык может прийти в полный беспорядок, если внедрять в него каждое средство для получения чуть более лаконичного кода. И хотя в других языках программирования не так просто породить поток исполнения или зарегистрировать обработчик событий от щелчка кнопкой мыши, тем не менее, их прикладные интерфейсы API в основном оказываются более простыми, согласованными и эффективными. Аналогичный прикладной интерфейс API, принимающий объекты классов, реализующих конкретную функцию, можно было бы написать и на Java, но пользоваться таким прикладным интерфейсом API было бы неудобно.

И в какой-то момент пришлось решать не столько вопрос усовершенствования Java для функционального программирования, сколько вопрос о том, как это сделать. На выработку проектного решения, подходящего для Java, ушло несколько лет экспериментирования. В следующем разделе будет показано, как обращаться с блоками кода, начиная с версии Java 8.

6.2.2. Синтаксис лямбда-выражений

Обратимся снова к примеру сортировки символьных строк из предыдущего раздела. В этом примере определяется, является ли одна символьная строка короче другой. С этой целью вычисляется следующее выражение:

```
first.length() - second.length()
```

А что обозначают ссылки `first` и `second`? Они обозначают символьные строки. Язык Java является строго типизированным, и поэтому приведенное выше выражение можно написать следующим образом:

```
(String first, String second) ->
    first.length() - second.length()
```

Собственно говоря, это и есть *лямбда-выражение*. Такое выражение представляет собой блок кода вместе с указанием любых переменных, которые должны быть переданы коду.

А почему оно так называется? Много лет назад, задолго до появления компьютеров, математик и логик Алонсо Чёрч (Alonzo Church) формализовал, что же должно означать эффективное выполнение математической функции. (Любопытно, что имеются такие функции, о существовании которых известно, но никто не знает, как вычислить их значения.) Он употребил греческую букву лямбда (λ) для обозначения параметров функции аналогично следующему:

```
 $\lambda$ first. $\lambda$ second.first.length() - second.length()
```



НА ЗАМЕТКУ! Почему была выбрана именно буква λ ? Неужели Алонсо Чёрчу не хватило букв латинского алфавита? По давней традиции знаком ударения (^) в математике обозначаются параметры функции, что навело Алонсо Чёрча на мысль воспользоваться прописной буквой Λ . Но в конечном итоге он остановил свой выбор на строчной букве λ , и с тех пор выражение с переменными параметрами называют *лямбда-выражением*.

Выше была приведена одна из форм лямбда-выражений в Java. Она состоит из параметров, знака стрелки `->` и вычисляемого выражения. Если же в теле лямбда-выражения должно быть выполнено вычисление, которое не вписывается в одно выражение, его можно написать точно так же, как и тело метода, заключив в фигурные скобки `{ }` и явно указав операторы `return`, как показано в следующем примере кода:

```
(String first, String second) ->
{
    if (first.length() < second.length()) return -1;
    else if (first.length() > second.length()) return 1;
    else return 0;
}
```

В отсутствие параметров у лямбда-выражения следует указать пустые круглые скобки, как при объявлении метода без параметров:

```
() -> {
    for (int i = 100; i >= 0; i--)
        System.out.println(i);
}
```

Если же типы параметров лямбда-выражения могут быть выведены, их можно опустить, как в следующем примере кода:

```
Comparator<String> comp
    =(first, second) // То же, что
                      // и (String first, String second)
    -> first.length() - second.length();
```

В данном примере компилятор может сделать вывод, что ссылки `first` и `second` должны обозначать символьные строки, поскольку результат вычисления лямбда-выражения присваивается компаратору символьных строк. (Эту операцию присваивания мы рассмотрим более подробно в следующем разделе.)

Если у метода имеется единственный параметр выводимого типа, то можно даже опустить круглые скобки, как показано ниже.

```
ActionListener listener = event ->
    System.out.println("The time is "
        + Instant.ofEpochMilli(event.getWhen()));
    // вместо выражения (event) -> . . .
    // или (ActionEvent event) -> . . .
```

Результат вычисления лямбда-выражения вообще не указывается. Тем не менее компилятор выводит его из тела лямбда-выражения и проверяет, соответствует ли он ожидаемому результату. Например, следующее выражение:

```
String first, String second) ->
    first.length() - second.length()
```

может быть использовано в контексте, где ожидается результат типа `int`.



НА ЗАМЕТКУ! Недопустимо, чтобы значение возвращалось в одних ветвях лямбда-выражения, но не возвращалось в других его ветвях. Например, следующее лямбда-выражение недопустимо:

```
(int x) -> { if (x >= 0) return 1; }
```

В примере программы из листинга 6.6 демонстрируется применение лямбда-выражений для построения компаратора и приемника действий.

Листинг 6.6. Исходный код из файла `lambda/LambdaTest.java`

```
1 package lambda;
2
3 import java.util.*;
4
5 import javax.swing.*;
6 import javax.swing.Timer;
7
8 /**
9  * В этой программе демонстрируется
10  * применение лямбда-выражений
11  * @version 1.0 2015-05-12
12  * @author Cay Horstmann
13  */
14 public class LambdaTest
15 {
16     public static void main(String[] args)
17     {
18         String[] planets = new String[]
19             { "Mercury", "Venus", "Earth",
```

```
20         "Mars", "Jupiter", "Saturn",
21         "Uranus", "Neptune" );
22     System.out.println(Arrays.toString(planets));
23     System.out.println("Sorted in dictionary order:");
24     Arrays.sort(planets);
25     System.out.println(Arrays.toString(planets));
26     System.out.println("Sorted by length:");
27     Arrays.sort(planets, (first, second) ->
28         first.length() - second.length());
29     System.out.println(Arrays.toString(planets));
30
31     Timer t = new Timer(1000, event ->
32         System.out.println("The time is " + new Date()));
33     t.start();
34
35     // выполнять программу до тех пор, пока пользователь
36     // не выберет экранную кнопку "OK"
37     JOptionPane.showMessageDialog(null,
38         "Quit program?");
39     System.exit(0);
40 }
41 }
```

6.2.3. Функциональные интерфейсы

Как обсуждалось ранее, в Java имеется немало интерфейсов, инкапсулирующих блоки кода, в том числе интерфейсы `ActionListener` и `Comparator`. Лямбда-выражения совместимы с этими интерфейсами. Лямбда-выражение можно предоставить всякий раз, когда ожидается объект, реализующий интерфейс с единственным абстрактным методом. Такой интерфейс называется *функциональным*.



НА ЗАМЕТКУ! У вас может возникнуть резонный вопрос: почему функциональный интерфейс должен иметь *единственный* абстрактный метод? Разве все методы в интерфейсе не являются абстрактными? На самом деле в интерфейсах всегда имелась возможность повторно объявить такие методы из класса `Object`, как `toString()` или `clone()`, не делая их абстрактными. (В некоторых интерфейсах из прикладного программного интерфейса Java API методы из класса `Object` объявляются повторно с целью присоединить к ним документирующие комментарии, составляемые с помощью утилиты `javadoc`. Характерным тому примером служит интерфейс `Comparator`.) Но важнее другое: в интерфейсах допускается объявлять методы неабстрактными, как было показано, в разделе 6.1.5.

Чтобы продемонстрировать преобразование в функциональный интерфейс, рассмотрим снова метод `Arrays.sort()`. В качестве второго параметра ему требуется экземпляр типа `Comparator` — интерфейса с единственным методом. Вместо него достаточно предоставить лямбда-выражение следующим образом:

```
Arrays.sort(words,
    (first, second) -> first.length() - second.length());
```

Подспудно метод `Arrays.sort()` принимает объект некоторого класса, реализующего интерфейс `Comparator`. В результате вызова метода `compare()` для этого объекта выполняется тело лямбда-выражения. Управление такими объектами и классами полностью зависит от конкретной реализации и может быть намного более эффективным, чем применение традиционных внутренних классов. Поэтому

лямбда-выражение лучше всего рассматривать как функцию, а не объект, приняв к сведению, что оно может быть передано функциональному интерфейсу.

Именно такое преобразование в функциональные интерфейсы и делает столь привлекательными лямбда-выражения. Их синтаксис прост и лаконичен. Рассмотрим еще один пример употребления лямбда-выражения. Приведенный ниже код намного легче читать, чем его альтернативный вариант с классом, реализующим интерфейс `ActionListener`.

```
var timer = new Timer(1000, event ->
{
    System.out.println("At the tone, the time is "
        + Instant.ofEpochMilli(event.getWhen()));
    Toolkit.getDefaultToolkit().beep();
});
```

В действительности *единственное*, что можно сделать с лямбда-выражением в Java, — преобразовать его в функциональный интерфейс. В других языках программирования, поддерживающих функциональные литералы, можно объявлять типы функций вроде `(String, String) -> int`, объявлять переменные подобных типов, присваивать этим переменным функции и вызывать их. Но разработчики Java решили придерживаться знакомого им понятия интерфейсов вместо того, чтобы вводить в язык функциональные типы.



НА ЗАМЕТКУ! Лямбда-выражение нельзя присвоить переменной типа `Object`, т.е. общего супер-типа для всех классов в Java. Ведь `Object` — это класс, а не функциональный интерфейс.

В стандартной библиотеке Java предоставляется целый ряд весьма универсальных функциональных интерфейсов, входящих в пакет `java.util.function`. К их числу относится функциональный интерфейс `BiFunction<T, U, R>`, описывающий функции с помощью параметров типа `T` и `U` и возвращаемого типа `R`. В частности, лямбда-выражение для сравнения символьных строк можно сохранить в переменной такого типа следующим образом:

```
BiFunction<String, String, Integer> comp
    = (first, second) -> first.length() - second.length();
```

Тем не менее это никак не помогает при сортировке, поскольку отсутствует такой метод `Arrays.sort()`, которому требовался бы параметр типа `BiFunction`. Если у вас имеется прежний опыт функционального программирования, такая ситуация может показаться вам необычной. Но для программирующих на Java она вполне естественна. У такого интерфейса, как `Comparator`, имеется конкретное назначение, а не только метод с заданными типами параметров и возвращаемого значения. Так, если требуется сделать что-нибудь с лямбда-выражением, придется по-прежнему учитывать его назначение, имея для него конкретный функциональный интерфейс.

Особенно удобным оказывается функциональный интерфейс `Predicate` из пакета `java.util.function`, который определяется следующим образом:

```
public interface Predicate<T> {
    boolean test(T t);
    // Дополнительные методы по умолчанию и
    // статические методы
}
```

В классе `ArrayList` имеется метод `removeIf()` с параметром типа `Predicate`, специально предназначенный для передачи ему лямбда-выражения. Например, в следующем выражении из списочного массива удаляются все пустые значения `null`:

```
list.removeIf(e -> e == null);
```

Еще одним полезным функциональным интерфейсом является интерфейс `Supplier<T>`, определяемый следующим образом:

```
public interface Supplier<T>
{
    T get();
}
```

В этом функциональном интерфейсе вызывается поставщик, не имеющий аргументов, но возвращающий значение типа `T`. Поставщики служат для *отложенного вычисления*. Рассмотрим в качестве примера следующий вызов:

```
LocalDate hireDay = Objects.requireNonNullOrElse(day,
    new LocalDate(1970, 1, 1));
```

Такой вызов нельзя считать оптимальным. Ведь параметр `day` редко принимает пустое значение `null`, и поэтому стандартный объект типа `LocalDate` требуется построить лишь тогда, когда в этом возникнет необходимость. А, используя поставщик, можно отложить вычисление. Так, в приведенном ниже примере кода метод `requireNonNullOrElseGet()` обращается к поставщику лишь в том случае, когда ему требуется значение.

```
LocalDate hireDay = Objects.requireNonNullOrElseGet(day,
    () -> new LocalDate(1970, 1, 1));
```

6.2.4. Ссылки на методы

Иногда уже имеется метод, выполняющий именно то действие, которое требуется передать какому-нибудь другому коду. Допустим, требуется просто выводить объект события от таймера всякий раз, когда это событие наступает. Безусловно, для этого можно было бы сделать следующий вызов:

```
var = new Timer(1000, event -> System.out.println(event));
```

Но было бы лучше просто передать метод `println()` конструктору класса `Timer`. И сделать это можно следующим образом:

```
var = new Timer(1000, System.out::println);
```

Выражение `System.out::println` обозначает *ссылку на метод*. Она предписывает компилятору получить экземпляр функционального интерфейса, переопределив его единственный абстрактный метод, чтобы вызвать заданный метод. В данном примере получается экземпляр типа `ActionListener`, в методе `actionPerformed(ActionEvent e)` которого вызывается метод `System.out.println(e)`.



НА ЗАМЕТКУ! Как и лямбда-выражение, ссылка на метод не является объектом. Она лишь приводит к порождению объекта, когда присваивается переменной, типом которой является функциональный интерфейс.



НА ЗАМЕТКУ! В классе `PrintStream`, экземпляром которого служит объект `System.out`, имеется десять перегружаемых методов типа `println`. Поэтому компилятору приходится выяснять, какой из этих методов следует вызвать в зависимости от контекста. В рассматриваемом здесь примере ссылка на метод `System.out::println` должна быть преобразована в экземпляр типа `ActionListener` со следующим методом:

```
void actionPerformed(ActionEvent e)
```

Метод `println(Object x)` выбирается из десяти перегружаемых методов типа `println` потому, что тип `Object` лучше всего соответствует типу `ActionEvent`. И когда вызывается метод `actionPerformed()`, в стандартный поток вывода `System.out` направляется объект наступившего события.

А теперь допустим, что та же самая ссылка присваивается другому функциональному интерфейсу:

```
Runnable task = System.out::println;
```

У функционального интерфейса `Runnable` имеется единственный абстрактный метод `void run()` без параметров, поэтому в данном случае вызывается метод `println()`, не имеющий параметров. Таким образом, в результате вызова `task.run()` в стандартный поток вывода `System.out` направляется пустая строка.

В качестве еще одного примера допустим, что требуется отсортировать символьные строки независимо от регистра букв. В таком случае методу сортировки можно передать следующее выражение:

```
Arrays.sort(strings, String::compareToIgnoreCase)
```

Как следует из приведенных выше примеров, операция `::` отделяет имя метода от имени класса или объекта. Ниже приведены три разновидности этой операции.

- **Объект::МетодЭкземпляра**
- **Класс::МетодЭкземпляра**
- **Класс::СтатическийМетод**

В первой разновидности ссылка на метод равнозначна лямбда-выражению, параметры которого передаются методу. Так, ссылка на метод `System.out::println` состоит из объекта `System.out` и выражения-метода, равнозначного лямбда-выражению `x -> System.out.println(x)`.

Во второй разновидности ссылки на метод первый ее параметр становится неявным параметром метода. Например, ссылка на метод `String::compareToIgnoreCase` равнозначна лямбда-выражению `(x, y) -> x.compareToIgnoreCase(y)`.

И в третьей разновидности ссылки на метод все параметры передаются статическому методу. Например, ссылка на метод `Math::pow` равнозначна лямбда-выражению `(x, y) -> Math.pow(x, y)`. Другие примеры ссылок на методы приведены в табл. 6.1.

Таблица 6.1. Примеры ссылок на методы

Ссылка на метод	Равнозначное лямбда-выражение	Примечания
<code>separator::equals</code>	<code>x -> separator.equals(x)</code>	Это выражение-метод с заданным объектом и методом экземпляра. Параметр лямбда-выражения передается как явный параметр метода
<code>String::trim</code>	<code>x -> x.trim()</code>	Это выражение-метод с заданным классом и методом экземпляра. Параметр лямбда-выражения становится неявным параметром

Ссылка на метод	Равнозначное лямбда-выражение	Примечания
<code>String::concat</code>	<code>(x, y) -> x.concat(y)</code>	И здесь вызывается метод экземпляра, но на этот раз с явным параметром. Как и прежде, первый параметр лямбда-выражения становится неявным параметром, а остальные параметры передаются методу
<code>Integer::valueOf</code>	<code>x -> Integer.valueOf(x)</code>	Это выражение-метод с заданным статическим методом. Параметр лямбда-выражения передается этому статическому методу
<code>Integer::sum</code>	<code>(x, y) -> Integer.sum(x, y)</code>	Это еще один пример вызова статического метода, но на этот раз с двумя параметрами. Оба параметра лямбда-выражения передаются статическому методу. В частности, метод <code>Integer.sum()</code> создается, чтобы служить в качестве ссылки на метод. В качестве альтернативы можно было бы просто составить лямбда-выражение <code>(x, y) -> x + y</code>
<code>Integer::new</code>	<code>x -> new Integer(x)</code>	Это ссылка на конструктор (см. далее раздел 6.2.5). Параметры лямбда-выражения передаются конструктору
<code>Integer[]::new</code>	<code>n -> new Integer[n]</code>	Это ссылка на конструктор массива (см. далее раздел 6.2.5). Параметр лямбда-выражения определяет длину массива

Следует, однако, иметь в виду, что лямбда-выражение можно превратить в ссылку на метод лишь в том случае, если в его теле вызывается единственный метод и больше ничего делается. Рассмотрим в качестве примера следующее лямбда-выражение:

```
s -> s.length() == 0
```

В этом лямбда-выражении вызывается единственный метод. Но, кроме того, в нем производится сравнение, и поэтому здесь нельзя воспользоваться ссылкой на метод.



НА ЗАМЕТКУ! Если имеется несколько переопределяемых методов с одинаковым именем, то компилятор попытается выяснить из контекста назначение каждого из них. Например, имеются два варианта метода `Math.max()`: один — для целочисленных значений, другой — для числовых значений с плавающей точкой типа `double`. Выбор конкретного варианта этого метода зависит от параметров типа функционального интерфейса, к которому приводится ссылка на метод `Math::max`. Как и лямбда-выражения, ссылки на методы не действуют обособленно. Они всегда преобразуются в экземпляры функциональных интерфейсов.



НА ЗАМЕТКУ! Иногда в состав прикладного интерфейса API включаются методы, специально предназначенные для применения в качестве ссылок на другие методы. Например, в классе `Objects` имеется метод `isNull()`, позволяющий проверить, является ли ссылка пустой (`null`). На первый взгляд, пользоваться таким методом неудобно, поскольку проверочное выражение `obj == null` выглядит более удобочитаемым, чем вызов `Objects.isNull(obj)`. Но этому методу можно передать ссылку на любой другой метод с параметром типа `Predicate`. Например, чтобы удалить все пустые ссылки из списка, достаточно сделать следующий вызов:

```
list.removeIf(Objects::isNull);
// эту строку кода легче читать, чем
// строку кода list.removeIf(e -> e == null);
```



НА ЗАМЕТКУ! Ссылка на метод по заданному объекту несколько отличается от равнозначного ей лямбда-выражения. В качестве примера рассмотрим ссылку на метод `separator::equals`. Если ссылка `separator` оказывается пустой (`null`), то исключение типа `NullPointerException` будет сгенерировано, как только сформируется выражение-метод `separator::equals`. Это же исключение будет сгенерировано лишь при вызове равнозначного лямбда-выражения `x -> separator.equals(x)`.

В ссылке на метод допускается указывать ссылку `this`. Например, ссылка на метод `this::equals` равнозначна лямбда-выражению `x -> this.equals(x)`. Это же относится и к ссылке `super`. Так, в следующей ссылке на метод:

`super::МетодЭкземпляра`

ссылка `super` является целевой и вызывает вариант заданного метода экземпляра из суперкласса. Ниже приведен искусственный пример, который, впрочем, наглядно демонстрирует подобный механизм ссылок.

```
class Greeter
{
    public void greet(ActionEvent event)
    {
        System.out.println("Hello, the time is "
            + Instant.ofEpochMilli(event.getWhen()));
    }
}

class RepeatedGreeter extends Greeter
{
    public void greet(ActionEvent event)
    {
        var timer = new Timer(1000, super::greet);
        timer.start();
    }
}
```

При вызове метода `RepeatedGreeter.greet()` конструируется объект типа `Timer`, выполняющий метод `greet()` из суперкласса по ссылке на него `super::greet` на каждом такте работы таймера.

6.2.5. Ссылки на конструкторы

Ссылки на конструкторы действуют таким же образом, как и ссылки на методы, за исключением того, что вместо имени метода указывается операция `new`. Например, ссылка `Person::new` делается на конструктор класса `Person`. Если же у класса имеется несколько конструкторов, то конкретный конструктор выбирается по ссылке в зависимости от контекста.

Рассмотрим пример, демонстрирующий применение ссылки на конструктор. Допустим, имеется список символьных строк. Его можно преобразовать в массив объектов класса `Person`, вызывая конструктор этого класса для каждой символьной строки:

```
ArrayList<String> names = . . . ;
Stream<Person> stream = names.stream().map(Person::new);
List<Person> people = stream.collect(Collectors.toList());
```


Более подробно методы `stream()`, `map()` и `collect()` будут рассматриваться в главе 1 второго тома настоящего издания, а до тех пор достаточно сказать, что для каждого элемента списка в методе `map()` вызывается конструктор `Person(String)`. Если же у класса `Person` имеется несколько конструкторов, то компилятор выбирает среди них конструктор с параметром типа `String`, поскольку он заключает из контекста, что конструктор должен вызываться с символьной строкой.

Ссылки на конструкторы можно формировать вместе с типами массивов. Например, ссылка на конструктор `int[]::new` делается с одним параметром длины массива. Она равнозначна лямбда-выражению `x -> new int[x]`.

Ссылки на конструкторы массивов удобны для преодоления следующего ограничения: в Java нельзя построить массив обобщенного типа `T`. Так, выражение `new T[n]` ошибочно, поскольку оно будет приведено путем стирания типов к выражению `new Object[n]`. У создателей библиотек это вызывает определенные затруднения. Допустим, требуется создать массив из объектов типа `Person`. В интерфейсе `Stream` имеется метод `toArray()`, возвращающий массив типа `Object`, как показано ниже.

```
Object[] people = stream.toArray();
```

Но этого явно недостаточно. Пользователю требуется массив ссылок на объекты типа `Person`, а не на объекты типа `Object`. В библиотеке потоков данных это затруднение разрешается с помощью ссылок на конструкторы. Достаточно передать методу `toArray()` следующую ссылку на конструктор `Person[]::new`:

```
Person[] people = stream.toArray(Person[]::new);
```

чтобы вызвать этот конструктор в методе `toArray()` и получить в итоге массив нужного типа. Этот массив будет далее заполнен объектами типа `Person` и возвращен вызывающей части программы.

6.2.6. Область видимости переменных

Нередко в лямбда-выражении требуется доступ к переменным из объемлющего метода или класса. Рассмотрим в качестве примера следующий фрагмент кода:

```
public static void repeatMessage(String text, int delay)
{
    ActionListener listener = event ->
    {
        System.out.println(text);
        Toolkit.getDefaultToolkit().beep();
    };
    new Timer(delay, listener).start();
}
```

Рассмотрим далее следующий вызов:

```
repeatMessage("Hello", 1000); // выводит слово Hello
                             // 1000 раз в отдельном потоке исполнения
```

Обратите внимание на переменную `text` в лямбда-выражении. Она определяется не в самом лямбда-выражении, а в качестве параметра метода `repeatMessage()`.

Если хорошенько подумать, то можно прийти к выводу, что здесь происходит нечто не совсем обычное. Код лямбда-выражения может быть выполнен спустя немало времени после возврата из вызванного метода `repeatMessage()`, когда переменные параметров больше не существуют. Каким же образом переменная `text` сохраняется до момента выполнения лямбда-выражения?

Чтобы понять происходящее, следует уточнить представление о лямбда-выражении. Лямбда-выражение имеет следующие составляющие.

1. Блок кода.
2. Параметры.
3. Значения *свободных* переменных, т.е. таких переменных, которые не являются параметрами и не определены в коде.

В рассматриваемом здесь примере лямбда-выражение содержит одну свободную переменную: `text`. В структуре данных, представляющей лямбда-выражение, должны храниться значения свободных переменных (в данном примере — символьная строка "Hello"). В таком случае говорят, что эти значения *захвачены* лямбда-выражением. (Механизм захвата значений зависит от конкретной реализации. Например, лямбда-выражение можно преобразовать в объект единственным методом, чтобы скопировать значения свободных переменных в переменные экземпляра данного объекта.)



НА ЗАМЕТКУ! Формально блок кода вместе со значениями свободных переменных называется *замыканием*. В языке Java лямбда-выражения служат в качестве замыканий.

Как видите, лямбда-выражение может захватывать значение переменной из окружающей области видимости. Но для того чтобы захваченное значение было вполне определено, в Java накладывается следующее важное ограничение: в лямбда-выражении можно ссылаться только на те переменные, значения которых не изменяются. Так, следующий фрагмент кода недопустим:

```
public static void countDown(int start, int delay)
{
    ActionListener listener = event ->
    {
        start--; // ОШИБКА: изменить захваченную
                // переменную нельзя!
        System.out.println(start);
    };
    new Timer(delay, listener).start();
}
```

Для такого ограничения имеется веское основание: изменение переменных в лямбда-выражениях ненадежно, если несколько действий выполняются параллельно. Ничего подобного не произойдет при выполнении рассмотренных выше действий, но, в общем, это довольно серьезный вопрос, который более подробно рассматривается в главе 12.

Не допускается также ссылаться в лямбда-выражении на переменную, которая изменяется *извне*. Например, следующий фрагмент кода недопустим:

```
public static void repeat(String text, int count)
{
    for (int i = 1; i <= count; i++)
    {
        ActionListener listener = event ->
        {
            System.out.println(i + ": " + text);
            // ОШИБКА: нельзя ссылаться на
            // изменяемую переменную i
        };
    }
}
```

```

    new Timer(1000, listener).start();
}
}

```

Правило гласит: любая захваченная переменная в лямбда-выражении должна быть *действительно конечной*, т.е. такой переменной, которой вообще не присваивается новое значение после ее инициализации. В данном примере переменная `text` всегда ссылается на один и тот же объект типа `String`, и ее допускается захватывать. Но значение переменной `i` изменяется, и поэтому захватить ее нельзя.

Тело лямбда-выражения находится в *той же области действия, что и вложенный блок кода*. На него распространяются те же правила, что и при конфликте имен и сокрытии переменных. В частности, не допускается объявлять параметр или переменную в лямбда-выражении с таким же именем как и у локальной переменной:

```

Path first = Paths.get("/usr/bin");
Comparator<String> comp = (first, second) ->
    first.length() - second.length();
// ОШИБКА: переменная first уже определена!

```

В теле метода не допускаются две локальные переменные с одинаковым именем, поэтому такие переменные нельзя внедрить и в лямбда-выражение. Если в лямбда-выражении указывается ссылка `this`, она делается на параметр метода, создающего это лямбда-выражение. Рассмотрим в качестве примера следующий фрагмент кода:

```

public class Application()
{
    public void init()
    {
        ActionListener listener = event ->
        {
            System.out.println(this.toString());
            . . .
        }
        . . .
    }
}

```

В выражении `this.toString()` вызывается метод `toString()` из объекта типа `Application`, но не экземпляра типа `ActionListener`. В применении ссылки `this` в лямбда-выражении нет ничего особенного. Область действия лямбда-выражения оказывается вложенной в тело метода `init()`, а ссылка `this` имеет такое же назначение, как и в любом другом месте этого метода.

6.2.7. Обработка лямбда-выражений

До сих пор пояснялось, как составлять лямбда-выражения и передавать их методу, ожидающему функциональный интерфейс. А далее будет показано, как писать собственные методы, в которых можно употреблять лямбда-выражения.

Лямбда-выражения применяются для *отложенного выполнения*. Ведь если некоторый код требуется выполнить сразу, это можно сделать, не заключая его в лямбда-выражение. Для отложенного выполнения кода имеется немало причин, в том числе следующие.

- Выполнение кода в отдельном потоке.
- Неоднократное выполнение кода.
- Выполнение кода в нужный момент по ходу алгоритма (например, выполнение операции сравнения при сортировке).

- Выполнение кода при наступлении какого-нибудь события (щелчка на экранной кнопке, поступления данных и т.д.).
- Выполнение кода только по мере надобности.

Рассмотрим простой пример. Допустим, некоторое действие требуется повторить *n* раз. Это действие и количество его повторений передаются методу `repeat()` следующим образом:

```
repeat(10, () -> System.out.println("Hello, World!"));
```

Чтобы принять лямбда-выражение в качестве параметра, нужно выбрать (а в редких случаях — предоставить) функциональный интерфейс. В данном примере для этого достаточно воспользоваться интерфейсом `Runnable`, как показано ниже. Обратите внимание на то, что тело лямбда-выражения выполняется при вызове `action.run()`.

```
public static void repeat(int n, Runnable action) {
    for (int i = 0; i < n; i++) action.run();
}
```

А теперь немного усложним рассматриваемый здесь простой пример, чтобы уведомить действие, на каком именно шаге цикла оно должно произойти. Для этого нужно выбрать функциональный интерфейс с методом, принимающим параметр типа `int` и ничего не возвращающим (`void`). Вместо написания собственного функционального интерфейса лучше воспользоваться одним из стандартных интерфейсов, перечисленных в табл. 6.2. Ниже приведен стандартный функциональный интерфейс для обработки значений типа `int`.

```
public interface IntConsumer {
    void accept(int value);
}
```

Усовершенствованный вариант метода `repeat()` выглядит следующим образом:

```
public static void repeat(int n, IntConsumer action) {
    for (int i = 0; i < n; i++) action.accept(i);
}
```

А вызывается он таким образом:

```
repeat(10, i ->
    System.out.println("Countdown: " + (9 - i)));
```

Таблица 6.2. Наиболее употребительные функциональные интерфейсы

Функциональный интерфейс	Типы параметров	Возвращаемый тип	Имя абстрактного метода	Описание	Другие методы
<code>Runnable</code>	Отсутствуют	<code>void</code>	<code>run</code>	Выполняет действие без аргументов или возвращаемого значения	
<code>Supplier<T></code>	Отсутствуют	<code>T</code>	<code>get</code>	Предоставляет значение типа <code>T</code>	
<code>Consumer<T></code>	<code>T</code>	<code>void</code>	<code>accept</code>	Употребляет значение типа <code>T</code>	<code>andThen</code>

Окончание табл. 6.2

Функциональный интерфейс	Типы параметров	Возвращаемый тип	Имя абстрактного метода	Описание	Другие методы
<code>BiConsumer<T, U></code>	<code>T, U</code>	<code>void</code>	<code>accept</code>	Употребляет значения типа <code>T</code> и <code>U</code>	<code>andThen</code>
<code>Function<T, R></code>	<code>T</code>	<code>R</code>	<code>apply</code>	Функция с аргументом <code>T</code>	<code>compose</code> , <code>andThen</code> , <code>identity</code>
<code>BiFunction<T, U, R></code>	<code>T, U</code>	<code>R</code>	<code>apply</code>	Функция с аргументами <code>T</code> и <code>U</code>	<code>andThen</code>
<code>UnaryOperator<T></code>	<code>T</code>	<code>T</code>	<code>apply</code>	Унарная операция над типом <code>T</code>	<code>compose</code> , <code>andThen</code> , <code>identity</code>
<code>BinaryOperator<T></code>	<code>T, T</code>	<code>T</code>	<code>apply</code>	Двоичная операция над типом <code>T</code>	<code>andThen</code> , <code>maxBy</code> , <code>minBy</code>
<code>Predicate<T></code>	<code>T</code>	<code>boolean</code>	<code>test</code>	Булевозначная функция	<code>and</code> , <code>or</code> , <code>negate</code> , <code>isEqual</code>
<code>BiPredicate<T, U></code>	<code>T, U</code>	<code>boolean</code>	<code>test</code>	Булевозначная функция с аргументами	<code>and</code> , <code>or</code> , <code>negate</code>

В табл. 6.3 перечислены 34 доступные специализации функциональных интерфейсов для примитивных типов `int`, `long` и `double`. Как будет показано в главе 8, пользоваться этими специализациями намного эффективнее, чем обобщенными интерфейсами. Именно по этой причине в приведенном выше примере кода использовался интерфейс `IntConsumer` вместо интерфейса `Consumer<Integer>`.

Таблица 6.3. Функциональные интерфейсы для примитивных типов, где обозначения *p*, *q* относятся к типам `int`, `long`, `double`; а *P*, *Q* — к типам `Int`, `Long`, `Double`

Функциональный интерфейс	Типы параметров	Возвращаемый тип	Имя абстрактного метода
<code>BooleanSupplier</code>	Отсутствует	<code>boolean</code>	<code>getAsBoolean</code>
<code>PSupplier</code>	Отсутствует	<i>p</i>	<code>getAsP</code>
<code>PConsumer</code>	<i>p</i>	<code>void</code>	<code>accept</code>
<code>ObjPConsumer<T></code>	<code>T, p</code>	<code>void</code>	<code>accept</code>
<code>PFunction<T></code>	<i>p</i>	<code>T</code>	<code>apply</code>
<code>PToQFunction</code>	<code>T</code>	<i>q</i>	<code>applyAsQ</code>
<code>ToPFunction<T></code>	<code>T</code>	<code>T</code>	<code>applyAsP</code>
<code>ToPBFunction<T, U></code>	<code>T, U</code>	<i>p</i>	<code>applyAsP</code>
<code>PUnaryOperator</code>	<i>p</i>	<i>p</i>	<code>applyAsP</code>
<code>PBinaryOperator</code>	<i>p, p</i>	<i>p</i>	<code>applyAsP</code>
<code>PPredicate</code>	<i>p</i>	<code>boolean</code>	<code>test</code>



СОВЕТ. Функциональными интерфейсами из табл. 6.2 и 6.3 рекомендуется пользоваться при всякой удобной возможности. Допустим, требуется написать метод для обработки файлов по заданному критерию. Для этой цели имеется устаревший интерфейс `java.io.FileFilter`, но лучше воспользоваться стандартным интерфейсом `Predicate<File>`. Единственная причина не делать этого может возникнуть в том случае, если в прикладном коде уже имеется немало методов, где получают экземпляры типа `FileFilter`.



НА ЗАМЕТКУ! У большинства стандартных функциональных интерфейсов имеются неабстрактные методы получения или объединения функций. Например, вызов метода `Predicate.isEqual(a)` равнозначен ссылке на метод `a::equals`, но он действует и в том случае, если аргумент `a` имеет пустое значение `null`. Для объединения предикатов имеются методы по умолчанию `and()`, `or()`, `negate()`. Например, вызов `Predicate.isEqual(a).or(Predicate.isEqual(b))` равнозначен лямбда-выражению `x -> a.equals(x) || b.equals(x)`.



НА ЗАМЕТКУ! Если вы разрабатываете собственный интерфейс с единственным абстрактным методом, можете пометить его аннотацией `@FunctionalInterface`. Это дает следующие преимущества. Во-первых, компилятор проверяет, что аннотируемый элемент является интерфейсом с единственным абстрактным методом, и выдает сообщение об ошибке, если вы случайно введете неабстрактный метод в свой интерфейс. И во-вторых, страница документирующих комментариев включает в себя пояснение, что объявляемый интерфейс является функциональным.

Пользоваться аннотацией совсем не обязательно. Ведь любой интерфейс с единственным абстрактным методом по определению является функциональным. Тем не менее такой интерфейс полезно пометить аннотацией `@FunctionalInterface`.

6.2.8. Еще о компараторах

В интерфейсе `Comparator` имеется целый ряд удобных статических методов для создания компараторов. Эти методы предназначены для применения вместе с лямбда-выражениями и ссылками на методы.

Статический метод `comparing()` принимает функцию извлечения ключей, приводящую обобщенный тип `T` к сравниваемому типу (например, `String`). Эта функция применяется к сравниваемым объектам, а сравнение производится по возвращаемым ею ключам. Допустим, имеется массив объектов типа `Person`. Ниже показано, как отсортировать их по имени.

```
Arrays.sort(people, Comparator.comparing(Person::getName));
```

Сделать это, безусловно, намного проще, чем реализовывать интерфейс `Comparator` вручную. Кроме того, исходный код становится более понятным, поскольку совершенно ясно, что людей требуется сравнивать по имени. Компараторы можно связывать в цепочку с помощью метода `thenComparing()` для разрыва связей, как показано в приведенном ниже примере кода. Если у двух людей оказываются одинаковые фамилии, то применяется второй компаратор.

```
Arrays.sort(people, Comparator
    .comparing(Person::getLastName)
    .thenComparing(Person::getFirstName));
```

Имеется немало разновидностей этих методов. В частности, можно указать компаратор для сравнения по ключам, извлекаемым методами `comparing()` и `thenComparing()`. Например, в следующем фрагменте кода люди сортируются по длине их имен:

```
Arrays.sort(people, Comparator.comparing(Person::getName,
    (s, t) -> Integer.compare(s.length(), t.length())));
```

Кроме того, у методов `comparing()` и `thenComparing()` имеются варианты, включающие упаковку значений типа `int`, `long` или `double`. Так, приведенную выше операцию можно упростить следующим образом:

```
Arrays.sort(people, Comparator.comparingInt(
    p -> p.getName().length()));
```

Если функция извлечения ключа может вернуть пустое значение `null`, то удобно воспользоваться статическими методами `nullsFirst()` и `nullsLast()`, принимающими имеющийся компаратор и модифицирующими его таким образом, чтобы он не генерировал исключение при появлении пустых значений `null`, но интерпретировал их как более мелкие, чем обычные значения. Допустим, метод `getMiddleName()` возвращает пустое значение `null` без отчества. В таком случае можно сделать следующий вызов:

```
Comparator.comparing(Person::getMiddleName(),
    Comparator.nullsFirst(...))
```

Методу `nullsFirst()` требуется компаратор, сравнивающий в данном случае две символьные строки. А метод `naturalOrder()` создает компаратор для любого класса, реализующего интерфейс `Comparable`. В данном примере это компаратор `Comparator.<String>naturalOrder()`. Ниже приведен полностью сформированный вызов для сортировки людей по потенциально пустым отчествам. Для того чтобы сделать этот вызов более удобочитаемым, предварительно производится статический импорт по директиве `java.util.Comparator.*`. Следует, однако, иметь в виду, что тип метода `naturalOrder()` выводится автоматически.

```
Arrays.sort(people, comparing(Person::getMiddleName(),
    nullsFirst(naturalOrder())));
```

И, наконец, статический метод `reverseOrder()` производит сортировку в порядке, обратном естественному. Чтобы обратиться к любому компаратору, достаточно вызвать метод экземпляра `reversed()`. Например, вызов `naturalOrder().reversed()` равнозначен вызову метода `reverseOrder()`.

6.3. Внутренние классы

Внутренним называется один класс, определенный в другом классе. А зачем он вообще нужен? На то имеются следующие причины.

- Внутренний класс можно скрыть от других классов того же пакета.
- Объект внутреннего класса имеет доступ к данным объекта, в котором он определен, включая закрытые данные.

Раньше внутренние классы играли важную роль в лаконичной реализации обратных вызовов, а ныне с этой ролью лучше справляются лямбда-выражения. Тем не менее внутренние классы могут оказаться весьма полезными для структуризации прикладного кода. Именно об этом и пойдет речь в последующих разделах.



НА ЗАМЕТКУ C++! В языке C++ имеются вложенные классы. Вложенный класс находится в области действия объемлющего класса. Ниже приведен типичный пример, где в классе для связанного списка определен класс, содержащий связи, а также класс, в котором определяется позиция итератора.

```
class LinkedList
{
public:
    class Iterator // вложенный класс
    {
```

```

public:
    void insert(int x);
    int erase();
    . . .
};
. . .
private:
    class Link // вложенный класс
    {
    public:
        Link* next;
        int data;
    };
    . . .
};

```

Вложенные классы подобны внутренним классам в Java. Но внутренние классы в Java обладают дополнительными возможностями, которые делают их более полезными и богатыми функционально, чем вложенные классы в C++. Объект, происходящий из внутреннего класса, содержит неявную ссылку на объект внешнего класса, где получается его экземпляр. По этой ссылке он получает доступ к состоянию всего внешнего объекта. Например, объекту класса `Iterator` в Java не потребуется явная ссылка на тот объект класса `LinkedList`, на который он ссылается. Такая дополнительная ссылка отсутствует только у статических внутренних классов в Java. Именно они и являются полным аналогом вложенных классов в C++.

6.3.1. Доступ к состоянию объекта с помощью внутреннего класса

Синтаксис, применяемый для внутренних классов, довольно сложен. Поэтому, для того чтобы продемонстрировать применение внутренних классов, рассмотрим простой, хотя и надуманный в какой-то степени пример. Итак, реорганизуем класс `TimerTest` из рассмотренного ранее примера, чтобы сформировать из него класс `TalkingClock`. Для организации работы “говорящих часов” применяются два параметра: интервал между последовательными сообщениями и признак, позволяющий включать или отключать звуковой сигнал. Соответствующий код приведен ниже.

```

public class TalkingClock
{
    private int interval;
    private boolean beep;

    public TalkingClock(int interval, boolean beep) { . . . }
    public void start() { . . . }

    public class TimePrinter implements ActionListener
    // внутренний класс
    {
        . . .
    }
}

```

Обратите внимание на то, что класс `TimePrinter` теперь расположен в классе `TalkingClock`. Это не означает, что каждый экземпляр класса `TalkingClock` содержит поле типа `TimePrinter`. Как станет ясно в дальнейшем, объекты типа `TimePrinter` создаются методами из класса `TalkingClock`. Рассмотрим класс `TimePrinter` более подробно. В приведенном ниже коде обратите внимание на то,

что в методе `actionPerformed()` признак `beep` проверяется перед тем, как воспроизвести звуковой сигнал.

```
public class TimePrinter implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        System.out.println("At the tone, the time is "
            + Instant.ofEpochMilli(event.getWhen()));
        if (beep) Toolkit.getDefaultToolkit().beep();
    }
}
```

А теперь начинается самое интересное. Нетрудно заметить, что в классе `TimePrinter` отсутствует поле `beep`. Вместо этого метод `actionPerformed()` обращается к соответствующему полю объекта типа `TalkingClock`. А это уже нечто новое. Обычно метод обращается к полям объекта. Но оказывается, что внутренний класс имеет доступ не только к своим полям, но и к полям создавшего его объекта, т.е. экземпляра внешнего класса. Для того чтобы это стало возможным, внутренний класс должен содержать ссылку на объект внешнего класса (рис. 6.3).

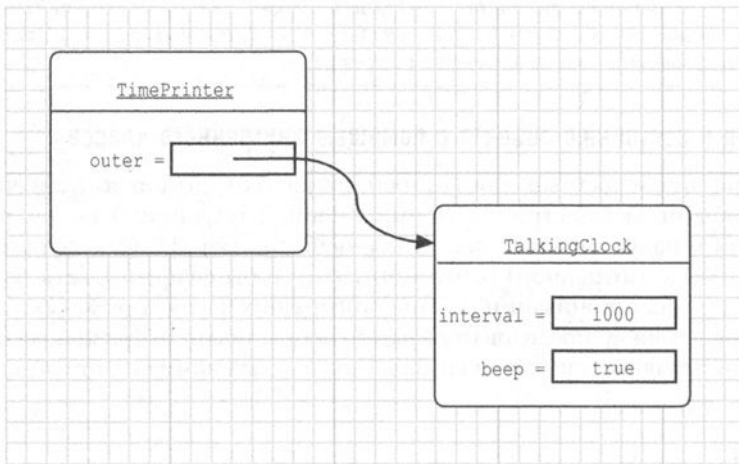


Рис. 6.3. Объект внутреннего класса содержит ссылку на объект внешнего класса

В определении внутреннего класса эта ссылка не присутствует явно. Чтобы продемонстрировать, каким образом она действует, введем в код ссылку `outer`. Тогда метод `actionPerformed()` будет выглядеть следующим образом:

```
public void actionPerformed(ActionEvent event)
{
    System.out.println("At the tone, the time is "
        + Instant.ofEpochMilli(event.getWhen()));
    if (outer.beep) Toolkit.getDefaultToolkit().beep();
}
```

Ссылка на объект внешнего класса задается в конструкторе. Компилятор видоизменяет все конструкторы внутреннего класса, добавляя параметр для ссылки на внешний класс. А поскольку конструкторы в классе `TalkingClock` не определены, то

компилятор автоматически формирует конструктор без аргументов, генерируя код, подобный следующему:

```
public TimePrinter(TalkingClock clock)
    // автоматически генерируемый код
{
    outer = clock;
}
```

Еще раз обращаем ваше внимание на то, что слово `outer` не является ключевым в Java. Оно используется только для иллюстрации механизма, задействованного во внутренних классах.

После того как метод `start()` создаст объект класса `TimePrinter`, компилятор передаст конструктору текущего объекта ссылку `this` на объект типа `TalkingClock` следующим образом:

```
var = new TimePrinter(this);
// параметр добавляется автоматически
```

В листинге 6.7 приведен исходный код завершенного варианта программы, проверяющей внутренний класс. Если бы `TimePrinter` был обычным классом, он должен был бы получить доступ к признаку `beep` через открытый метод из класса `TalkingClock`. Применение внутреннего класса усовершенствует код, поскольку отпадает необходимость предоставлять специальный метод доступа, представляющий интерес только для какого-нибудь другого класса.



НА ЗАМЕТКУ! Класс `TimePrinter` можно было бы объявить как закрытый (`private`). И тогда конструировать объекты типа `TimePrinter` могли бы только методы из класса `TalkingClock`. Закрытыми могут быть только внутренние классы. А обычные классы всегда доступны в пределах пакета или же полностью открыты.

Листинг 6.7. Исходный код из файла `innerClass/InnerClassTest.java`

```
1 package innerClass;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.time.*;
6
7 import javax.swing.*;
8
9 /**
10  * В этой программе демонстрируется применение
11  * внутренних классов
12  * @version 1.11 2017-12-14
13  * @author Cay Horstmann
14  */
15 public class InnerClassTest
16 {
17     public static void main(String[] args)
18     {
19         var clock = new TalkingClock(1000, true);
20         clock.start();
21
22         // выполнять программу до тех пор, пока пользователь
```

```
23 // не щелкнет на экранной кнопке "OK"
24 JOptionPane.showMessageDialog(null, "Quit program?");
25 System.exit(0);
26 }
27 }
28
29 /**
30  * Часы, выводящие время через регулярные промежутки
31  */
32 class TalkingClock
33 {
34     private int interval;
35     private boolean beep;
36
37     /**
38      * Конструирует "говорящие часы"
39      * @param interval Интервал между сообщениями
40      *                (в миллисекундах)
41      * @param beep Истинно, если часы должны издавать
42      *            звуковой сигнал
43      */
44     public TalkingClock(int interval, boolean beep)
45     {
46         this.interval = interval;
47         this.beep = beep;
48     }
49
50     /**
51      * Запускает часы
52      */
53     public void start()
54     {
55         var listener = new TimePrinter();
56         var timer = new Timer(interval, listener);
57         timer.start();
58     }
59
60     public class TimePrinter implements ActionListener
61     {
62         public void actionPerformed(ActionEvent event)
63         {
64             System.out.println("At the tone, the time is "
65                 + Instant.ofEpochMilli(event.getWhen()));
66             if (beep) Toolkit.getDefaultToolkit().beep();
67         }
68     }
69 }
```

6.3.2. Специальные синтаксические правила для внутренних классов

В предыдущем разделе ссылка на внешний класс была названа *outer* для того, чтобы стало понятнее, что это ссылка из внутреннего класса на внешний. На самом деле синтаксис для внешних ссылок немного сложнее. Так, приведенное ниже выражение обозначает внешнюю ссылку.

ВнешнийКласс.this

Например, во внутреннем классе `TimePrinter` можно создать метод `actionPerformed()` следующим образом:

```
public void actionPerformed(ActionEvent event)
{
    . . .
    if (TalkingClock.this.beep)
        Toolkit.getDefaultToolkit().beep();
}
```

С другой стороны, конструктор внутреннего класса можно записать более явным образом, используя следующий синтаксис:

```
ОбъектВнешнегоКласса.new
    ВнутреннийКласс(параметры конструктора)
```

Например, в приведенной ниже строке кода ссылка на внешний класс из вновь созданного объекта типа `TimePrinter` получает ссылку `this` на метод, создающий объект внутреннего класса.

```
ActionListener listener = this.new TimePrinter();
```

Такой способ применяется чаще всего, хотя явное указание ссылки `this` здесь, как всегда, излишне. Тем не менее это позволяет явно указать другой объект в ссылке на объект внешнего класса. Так, если класс `TimePrinter` является открытым внутренним классом, его объекты можно создать для любых “говорящих часов”:

```
TalkingClock jabberer = new TalkingClock(1000, true);
TalkingClock.TimePrinter listener =
    jabberer.new TimePrinter();
```

Но если ссылка на внутренний класс делается за пределами области действия внешнего класса, то она указывается следующим образом:

```
ВнешнийКласс.ВнутреннийКласс
```



НА ЗАМЕТКУ! Любые статические поля должны быть объявлены во внутреннем классе как конечные (`final`), а также инициализированы константой на стадии компиляции. Ведь если поле не содержит константу, его значение не может считаться однозначным.

У внутреннего класса не может быть статических методов. В спецификации на язык Java не поясняются причины для такого ограничения. Во внутреннем классе можно было бы разрешить лишь те статические методы, которые имели бы доступ только к статическим полям и методам из объемлющего класса. Очевидно, что разработчики Java решили, что сложности внедрения такой возможности перевешивают те преимущества, которые она может дать.

6.3.3. О пользе, необходимости и безопасности внутренних классов

Внутренние классы были впервые внедрены в версии Java 1.1. Многие программисты встретили появление внутренних классов настороженно, считая их отступлением от принципов, положенных в основу Java. По их мнению, главным преимуществом языка Java над C++ является его простота. Внутренние классы действительно сложны. (Такой вывод вы, скорее всего, сделаете, ознакомившись с анонимными внутренними классами, которые будут рассматриваться далее в этой главе.) Взаимодействие внутренних классов с другими языковыми средствами не совсем очевидно, и особенно это касается вопросов управления доступом и безопасности.

Зачем же создатели языка Java пожертвовали преимуществами, которыми он выгодно отличался от других языков программирования, в пользу изящного и, безусловно, интересного механизма, выгода от которого, впрочем, сомнительна? Не пытаюсь дать исчерпывающий ответ на этот вопрос, заметим только, что обращение с внутренними классами происходит на уровне *компилятора*, а не виртуальной машины. Для их обозначения используется знак \$, разделяющий имена внешних и внутренних классов. Таким образом, для виртуальной машины внутренние классы неотличимы от внешних.

Например, класс `TimePrinter`, входящий в состав класса `TalkingClock`, преобразуется в файл `TalkingClock$TimePrinter.class`. Для того чтобы посмотреть, каким образом действует этот механизм, попробуйте провести следующий эксперимент: запустите на выполнение программу `ReflectionTest` (см. главу 5) и выполните рефлексии класса `TalkingClock$TimePrinter`. С другой стороны, можно воспользоваться утилитой `javap` следующим образом:

```
javap -private ИмяКласса
```



НА ЗАМЕТКУ! Если вы пользуетесь UNIX, не забудьте экранировать знак \$, указывая имя класса в командной строке. Следовательно, запускайте программу `ReflectionTest` или утилиту `javap` таким способом:

```
java reflection.ReflectionTest ВнутреннийКласс.TalkingClock\$TimePrinter
```

или же таким:

```
javap -private ВнутреннийКласс.TalkingClock\$TimePrinter
```

В итоге будет получен следующий результат:

```
public class innerClass.TalkingClock$TimePrinter
    implements java.awt.event.ActionListener
{
    final innerClass.TalkingClock this$0;
    public innerClass.TalkingClock$TimePrinter(
        innerClass.TalkingClock);
    public void actionPerformed(java.awt.event.ActionEvent);
}
```

Нетрудно заметить, что компилятор генерирует дополнительное поле `this$0` для ссылки на внешний класс. (Имя `this$0` синтезируется компилятором, поэтому сослаться на него нельзя.) Кроме того, у конструктора можно обнаружить параметр `TalkingClock`. Если компилятор автоматически выполняет преобразования, нельзя ли реализовать подобный механизм вручную? Попробуем сделать это, превратив `TimePrinter` в обычный класс, определяемый за пределами класса `TalkingClock`, а затем передав ему ссылку `this` на создавший его объект:

```
class TalkingClock
{
    . . .
    public void start()
    {
        var listener = new TimePrinter(this);
        var timer = new Timer(interval, listener);
        timer.start();
    }
}
```

```
class TimePrinter implements ActionListener
{
    private TalkingClock outer;
    . . .
    public TimePrinter(TalkingClock clock)
    {
        outer = clock;
    }
}
```

Рассмотрим теперь метод `actionPerformed()`. Ему требуется доступ к полю `outer.beep` следующим образом:

```
if (outer.beep) . . . // ОШИБКА!
```

И здесь возникает ошибка. Внутренний класс может иметь доступ к закрытым данным лишь того внешнего класса, в который он входит. Но ведь класс `TimePrinter` уже не является внутренним, а следовательно, не имеет такого доступа.

Таким образом, внутренние классы, по существу, намного эффективнее, чем обычные, поскольку они обладают более высокими правами доступа. В связи с этим возникает следующий вопрос: каким образом внутренние классы получают дополнительные права доступа, если они преобразуются в обычные, а виртуальной машине вообще о них ничего не известно? Чтобы раскрыть эту тайну, воспользуемся еще раз программой `ReflectionTest`, отслеживающей поведение класса `TalkingClock`, получив следующий результат:

```
class TalkingClock
{
    private int interval;

    private boolean beep;

    public TalkingClock(int, boolean);

    static boolean access$0(TalkingClock);
    public void start();
}
```

Обратите внимание на статический метод `access$0`, добавленный компилятором во внешний класс. Этот метод возвращает значение из поля `beep` объекта, переданного ему в качестве параметра. (Имя этого метода может отличаться в зависимости от компилятора, например `access$000`.)

Этот метод вызывается из внутреннего класса. В методе `actionPerformed()` из класса `TimePrinter` имеется следующий оператор:

```
if (beep)
```

Он преобразуется компилятором в приведенный ниже вызов.

```
if (TalkingClock.access$0(outer))
```

Не опасно ли это? В принципе опасно! Посторонний может легко вызвать метод `access$0()` и прочесть данные из закрытого поля `beep`. Конечно, `access$0` не является допустимым именем для метода в Java. Но злоумышленники, знакомые со структурой файлов классов, легко могут создать свой аналогичный файл и вызвать данный метод с помощью соответствующих команд виртуальной машины. Разумеется, такой

файл должен формироваться вручную (например, с помощью редактора шестнадцатеричного кода). Но поскольку область действия секретных методов доступа ограничена пакетом, атакующий код должен размещаться в том же самом пакете, что и атакуемый класс.

Итак, если внутренний класс имеет доступ к закрытым полям, можно создать другой класс, добавить его в тот же самый пакет и получить доступ к закрытым данным. Правда, для этого требуется опыт и решительность. Программист не может получить такой доступ случайно, не создавая для этих целей специальный файл, содержащий видоизмененные классы.



НА ЗАМЕТКУ! Синтезированные методы и конструкторы могут быть довольно запутанными (материал этой врезки не для слабонервных, так что можете его пропустить). Допустим, класс **TimePrinter** превращен в закрытый внутренний класс. Но для виртуальной машины не существует внутренних классов, поэтому компилятор делает все возможное, чтобы произвести доступный в пределах пакета класс с закрытым конструктором следующим образом:

```
private TalkingClock$TimePrinter(TalkingClock);
```

Безусловно, никто не сможет вызвать такой конструктор. Поэтому требуется второй конструктор, доступный в пределах пакета и вызывающий первый:

```
TalkingClock$TimePrinter(TalkingClock, TalkingClock$1);
```

Компилятор преобразует вызов конструктора в методе **start()** из класса **TalkingClock** следующим образом:

```
new TalkingClock$TimePrinter(this, null)
```

6.3.4. Локальные внутренние классы

Если внимательно проанализировать исходный код класса **TalkingClock**, то можно обнаружить, что имя класса **TimePrinter** используется лишь однажды: при создании объекта данного типа в методе **start()**. В подобных случаях класс можно определить локально в отдельном методе, как выделено ниже полужирным.

```
public void start()
{
    class TimePrinter implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            System.out.println("At the tone, the time is "
                + Instant.ofEpochMilli(event.getWhen()));
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
    }

    var listener = new TimePrinter();
    var timer = new Timer(interval, listener);
    timer.start();
}
```

Локальные внутренние классы никогда не объявляются с помощью модификаторов доступа (например, **public** и **protected**). Их область действия всегда ограничивается блоком, в котором они объявлены. У локальных внутренних классов имеется следующее большое преимущество: они полностью скрыты от внешнего кода и даже

от остальной части класса `TalkingClock`. Ни одному из методов, за исключением `start()`, ничего неизвестно о классе `TimePrinter`.

6.3.5. Доступ к конечным переменным из внешних методов

Локальные внутренние классы выгодно отличаются от обычных внутренних классов еще и тем, что имеют доступ не только к полям своего внешнего класса, но и к локальным переменным! Но такие локальные переменные должны быть объявлены как *действительно конечные*. Это означает, что такие переменные нельзя изменить после их инициализации путем присваивания им первоначальных значений. Обратимся к характерному примеру, переместив параметры `interval` и `beep` из конструктора `TalkingClock` в метод `start()`, как выделено ниже полужирным.

```
public void start(int interval, boolean beep)
{
    class TimePrinter implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            System.out.println("At the tone, the time is "
                               + Instant.ofEpochMilli(event.getWhen()));
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
    }

    var listener = new TimePrinter();
    var timer = new Timer(interval, listener);
    timer.start();
}
```

Обратите внимание на то, что в классе `TimePrinter` больше не нужно хранить переменную экземпляра `beep`. Он просто ссылается на параметр метода, содержащего определение данного класса. Возможно, это не так уж и неожиданно. В конце концов, приведенная ниже строка кода находится в теле метода `start()`, так почему бы не иметь в ней доступ к переменной `beep`?

```
if (beep) . . .
```

Локальные внутренние классы обладают рядом особенностей. Чтобы понять их, рассмотрим подробнее логику их управления на приведенном выше примере кода.

1. Вызывается метод `start()`.
2. При вызове конструктора внутреннего класса `TimePrinter` инициализируется объектная переменная `listener`.
3. Передается ссылка `listener` конструктору класса `Timer`, запускается таймер, и метод `start()` прекращает свою работу. В этот момент параметр `beep` метода `start()` уже не существует.
4. Некоторое время спустя выполняется условный оператор `if (beep)...` в методе `actionPerformed()`.

Для того чтобы метод `actionPerformed()` выполнялся успешно, в классе `TimePrinter` должна быть создана копия поля `beep` до того, как оно перестанет существовать в качестве локальной переменной метода `start()`. Именно это и происходит. В данном примере компилятор синтезирует для локального внутреннего

класса имя `TalkingClock$1TimePrinter`. Если применить снова программу `ReflectionTest` для анализа класса `TalkingClock$1TimePrinter`, то получится следующий результат:

```
class TalkingClock$1TimePrinter
{
    TalkingClock$1TimePrinter(TalkingClock, boolean);

    public void actionPerformed(java.awt.event.ActionEvent);

    final boolean val$beep;
    final TalkingClock this$0;
}
```

Обратите внимание на параметр конструктора, имеющий тип `boolean`, а также переменную экземпляра `val$beep`. При создании объекта переменная `beep` передается конструктору и размещается в поле `val$beep`. Чтобы это стало возможным, разработчикам компилятора пришлось немало потрудиться. Компилятор должен обнаруживать доступ к локальным переменным, создавать для каждой из них соответствующие поля, а затем копировать локальные переменные в конструкторе таким образом, чтобы поля данных инициализировались копиями локальных переменных.

6.3.6. Анонимные внутренние классы

Работая с локальными внутренними классами, можно воспользоваться еще одной интересной возможностью. Так, если требуется создать единственный объект некоторого класса, этому классу можно вообще не присваивать имени. Такой класс называется *анонимным внутренним классом*, как показано ниже.

```
public void start(int interval, boolean beep)
{
    var listener = new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            System.out.println("At the tone, the time is "
                + Instant.ofEpochMilli(event.getWhen()));
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
    };

    var timer = new Timer(interval, listener);
    timer.start();
}
```

Следует, однако, признать, что синтаксис анонимных внутренних классов довольно сложен. На самом деле приведенный выше фрагмент кода означает следующее: создается новый объект класса, реализующего интерфейс `ActionListener`, где в фигурных скобках определен требующийся метод `actionPerformed()`. Ниже приведена общая форма определения анонимных внутренних классов.

```
new СуперТип(параметры конструирования объектов)
{
    методы и данные внутреннего класса
}
```

Здесь *СуперТип* может быть интерфейсом, например `ActionListener`, и тогда внутренний класс реализует данный интерфейс. *СуперТип* может быть также классом, и в этом случае внутренний класс расширяет данный суперкласс.

Анонимный внутренний класс не может иметь конструкторов, поскольку имя конструктора должно совпадать с именем класса, а в данном случае у класса отсутствует имя. Вместо этого параметры, необходимые для создания объекта, передаются конструктору *суперкласса*. Так, если вложенный класс реализует какой-нибудь интерфейс, параметры конструктора можно не указывать. Тем не менее они должны быть указаны в круглых скобках следующим образом:

```
new ТипИнтерфейса()
{
    методы и данные
}
```

Следует внимательно и аккуратно проводить различие между созданием нового объекта некоторого класса и конструированием объекта анонимного внутреннего класса, расширяющего данный класс. Если за скобками со списком параметров, необходимых для создания объекта, следует открытая фигурная скобка, то определяется анонимный вложенный класс:

```
Person queen = new Person("Mary");
// объект типа Person
Person count = new Person("Dracula") { . . . };
// объект внутреннего класса, расширяющего класс Person
```



НА ЗАМЕТКУ! Несмотря на то что у анонимного класса могут отсутствовать конструкторы, это никоим образом не мешает предоставить блок инициализации объекта, как показано ниже.

```
var count = new Person("Dracula")
{
    { инициализация }
    . . .
};
```

В листинге 6.8 приведен исходный код завершенной версии программы, реализующей “говорящие часы”, где применяется анонимный внутренний класс. Сравнив эту версию программы с ее версией из листинга 6.7, можете сами убедиться, что применение анонимного внутреннего класса сделало программу немного короче, но не проще для понимания, хотя для этого требуются опыт и практика.

Многие годы программирующие на Java регулярно пользовались анонимными внутренними классами для организации приемников событий и прочих обратных вызовов. А ныне вместо них лучше употреблять лямбда-выражения. Например, метод `start()`, упоминавшийся в начале этого раздела, можно написать в более лаконичной форме, воспользовавшись лямбда-выражением, как выделено ниже полужирным.

```
public void start(int interval, boolean beep)
{
    var timer = new Timer(interval, event -> {
        System.out.println("At the tone, the time is "
            + Instant.ofEpochMilli(event.getWhen()));
        if (beep) Toolkit.getDefaultToolkit().beep();
    });
    timer.start();
}
```



НА ЗАМЕТКУ! Существует специальный прием, называемый инициализацией в двойных фигурных скобках и выгодно использующий преимущества синтаксиса внутренних классов. Допустим, требуется составить списочный массив и передать ему метод следующим образом:

```
var friends = new ArrayList<>();
favorites.add("Harry");
favorites.add("Tony");
invite(friends);
```

Если списочный массив больше не понадобится, то было бы неплохо сделать его анонимным. Но как тогда вводить в него дополнительные элементы? А вот как:

```
invite(new ArrayList<String>() {{ add("Harry"); add("Tony"); }})
```

Обратите внимание на двойные фигурные скобки в приведенной выше строке кода. Внешние фигурные скобки образуют анонимный подкласс `ArrayList`, а внутренние фигурные скобки — блок конструирования объектов (см. главу 4).

На практике такой прием редко приносит пользу. Вероятнее всего, метод `invite()` должен был бы принимать любой список типа `List<String>` в качестве параметра, что дало бы возможность передать вызов `List.of("Harry", "Tony")`.



ВНИМАНИЕ! Зачастую анонимный подкласс удобно сделать почти, но не совсем таким же, как и его суперкласс. Но в этом случае следует соблюдать особую осторожность в отношении метода `equals()`. Как рекомендовалось в главе 5, в методе `equals()` необходимо организовать следующую проверку:

```
if (getClass() != other.getClass()) return false;
```

Но анонимный подкласс ее не пройдет.



СОВЕТ. При выдаче регистрирующих или отладочных сообщений в них нередко требуется включить имя текущего класса, как в приведенной ниже строке кода.

```
System.err.println("Something awful happened in " + getClass());
```

Но такой прием не годится для статического метода. Ведь вызов метода `getClass()`, по существу, означает вызов `this.getClass()`. Но ссылка `this` на текущий объект для статического метода не годится. В таком случае можно воспользоваться следующим выражением:

```
new Object(){}.getClass().getEnclosingClass()
// получить класс статического метода
```

Здесь в операции `new Object(){}` создается объект анонимного подкласса, производного от класса `Object`, а метод `getEnclosingClass()` получает объемлющий его класс, т.е. класс, содержащий статический метод.

Листинг 6.8. Исходный код из файла `anonymousInnerClass/AnonymousInnerClassTest.java`

```
1 package anonymousInnerClass;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.time.*;
6
7 import javax.swing.*;
8
```

```
9  /**
10 * В этой программе демонстрируется применение
11 * анонимных внутренних классов
12 * @version 1.12 2017-12-14
13 * @author Cay Horstmann
14 */
15 public class AnonymousInnerClassTest
16 {
17     public static void main(String[] args)
18     {
19         var clock = new TalkingClock();
20         clock.start(1000, true);
21
22         // выполнять программу до тех пор, пока пользователь
23         // не щелкнет на экранной кнопке "OK"
24         JOptionPane.showMessageDialog(null,
25                                     "Quit program?");
26         System.exit(0);
27     }
28 }
29
30 /**
31 * Часы, выводящие время через регулярные промежутки
32 */
33 class TalkingClock
34 {
35     /**
36     * Запускает часы
37     * @param interval Интервал между сообщениями
38     *                  (в миллисекундах)
39     * @param beep Истинно, если часы должны издавать
40     *             звуковой сигнал
41     */
42     public void start(int interval, boolean beep)
43     {
44         var listener = new ActionListener()
45         {
46             public void actionPerformed(ActionEvent event)
47             {
48                 System.out.println("At the tone, the time is "
49                                     + Instant.ofEpochMilli(event.getWhen()));
50                 if (beep) Toolkit.getDefaultToolkit().beep();
51             }
52         };
53         var timer = new Timer(interval, listener);
54         timer.start();
55     }
56 }
```

6.3.7. Статические внутренние классы

Иногда внутренний класс требуется лишь для того, чтобы скрыть его в другом классе, тогда как ссылка на объект внешнего класса не нужна. Подавить формирование такой ссылки можно, объявив внутренний класс статическим (i.e. `static`).

Допустим, в массиве требуется найти максимальное и минимальное числа. Для этого можно было бы, конечно, написать два метода: один — для нахождения максимального числа, а другой — для нахождения минимального числа. Но при вызове обоих методов массив просматривается дважды. Было бы намного эффективнее просматривать массив только один раз, одновременно определяя в нем как максимальное, так и минимальное число:

```
double min = Double.MAX_VALUE;
double max = Double.MIN_VALUE;
for (double v : values)
{
    if (min > v) min = v;
    if (max < v) max = v;
}
```

Но в таком случае метод должен возвращать два значения. Сделать это можно, определив класс `Pair` с двумя полями для хранения числовых значений, как показано ниже.

```
class Pair
{
    private double first;
    private double second;

    public Pair(double f, double s)
    {
        first = f;
        second = s;
    }

    public double getFirst() { return first; }
    public double getSecond() { return second; }
}
```

Тогда метод `minmax()` сможет вернуть объект типа `Pair` следующим образом:

```
class ArrayAlg
{
    public static Pair minmax(double[] values)
    {
        . . .
        return new Pair(min, max);
    }
}
```

Таким образом, для получения максимального и минимального числовых значений достаточно вызвать методы `getFirst()` и `getSecond()`:

```
Pair p = ArrayAlg.minmax(d);
System.out.println("min = " + p.getFirst());
System.out.println("max = " + p.getSecond());
```

Разумеется, имя `Pair` слишком широко распространено, и при выполнении крупного проекта у другого программиста может возникнуть такая же блестящая идея, вот только класс `Pair` у него будет содержать не числовые, а строковые поля. Это вполне вероятное затруднение можно разрешить, сделав класс `Pair` внутренним и определенным в классе `ArrayAlg`. Тогда подлинным именем этого класса будет не `Pair`, а `ArrayAlg.Pair`:

```
ArrayAlg.Pair p = ArrayAlg.minmax(d);
```

Но, в отличие от внутренних классов, применявшихся в предыдущих примерах, ссылка на другой объект в классе `Pair` не требуется. Ее можно подавить, объявив внутренний класс статическим:

```
class ArrayAlg
{
    public static class Pair
    {
        . . .
    }
    . . .
}
```

Разумеется, только внутренние классы можно объявлять статическими. Статический внутренний класс ничем не отличается от любого другого внутреннего класса, за исключением того, что его объект не содержит ссылку на создавший его объект внешнего класса. В данном примере следует применять статический внутренний класс, поскольку объект внутреннего класса создается в теле статического метода, как показано ниже.

```
public static Pair minmax(double[] d)
{
    . . .
    return new Pair(min, max);
}
```

Если бы класс `Pair` не был объявлен статическим, компилятор сообщил бы, что при инициализации объекта внутреннего класса объект типа `ArrayAlg` недоступен.



НА ЗАМЕТКУ! Статический вложенный класс применяется в тех случаях, когда доступ к объекту внутреннего класса не требуется. Некоторые программисты для обозначения статических внутренних классов пользуются термином *вложенные классы*.



НА ЗАМЕТКУ! В отличие от обычных внутренних классов, у статических внутренних классов могут быть статические поля и методы.



НА ЗАМЕТКУ! Внутренние классы, определенные в интерфейсах, автоматически считаются статическими и открытыми (т.е. `static` и `public`).

В листинге 6.9 приведен весь исходный код класса `ArrayAlg` и вложенного в него класса `Pair`.

Листинг 6.9. Исходный код из файла `staticInnerClass/StaticInnerClassTest.java`

```
1 package staticInnerClass;
2
3 /**
4  * В этой программе демонстрируется применение
5  * статического внутреннего класса
6  * @version 1.02 2015-05-12
7  * @author Cay Horstmann
```

```
8  */
9  public class StaticInnerClassTest
10 {
11     public static void main(String[] args)
12     {
13         double[] d = new double[20];
14         for (int i = 0; i < d.length; i++)
15             d[i] = 100 * Math.random();
16         ArrayAlg.Pair p = ArrayAlg.minmax(d);
17         System.out.println("min = " + p.getFirst());
18         System.out.println("max = " + p.getSecond());
19     }
20 }
21
22 class ArrayAlg
23 {
24     /**
25      * Пара чисел с плавающей точкой
26      */
27     public static class Pair
28     {
29         private double first;
30         private double second;
31
32         /**
33          * Составляет пару из двух чисел с плавающей точкой
34          * @param f Первое число
35          * @param s Второе число
36          */
37         public Pair(double f, double s)
38         {
39             first = f;
40             second = s;
41         }
42
43         /**
44          * Возвращает первое число из пары
45          * @return Возврат первого числа
46          */
47         public double getFirst()
48         {
49             return first;
50         }
51
52         /**
53          * Возвращает второе число из пары
54          * @return Возврат второго числа
55          */
56         public double getSecond()
57         {
58             return second;
59         }
60     }
61
62     /**
63      * Определяет минимальное и максимальное
```

```

64  * числа в массиве
65  * @param values Массив чисел с плавающей точкой
66  * @return Пара, первым элементом которой является
67  *         минимальное число, а вторым элементом –
68  *         максимальное число
69  */
70  public static Pair minmax(double[] values)
71  {
72      double min = Double.POSITIVE_INFINITY;
73      double max = Double.NEGATIVE_INFINITY;
74      for (double v : values)
75      {
76          if (min > v) min = v;
77          if (max < v) max = v;
78      }
79      return new Pair(min, max);
80  }
81  }

```

6.4. Загрузчики служб

Иногда приходится разрабатывать приложение с архитектурой подключаемых модулей. Такой подход поощряется на некоторых платформах, применяемых в средах разработки (например, OSG; <http://osgi.org>), серверах приложений и прочих сложных приложениях. И хотя рассмотрение подобных платформ выходит за рамки данной книги, в комплекте JDK предоставляется простой механизм для загрузки подключаемых модулей, как поясняется ниже. Этот механизм поддерживается в модульной системе на платформе Java, как поясняется в главе 9 второго тома настоящего издания.

Если предоставляется служба, то разработчику прикладной программы необходимо дать определенную свободу в реализации функциональных средств данной службы. Было бы также желательно предоставить на выбор несколько реализаций службы. Загрузку служб, соответствующих общему интерфейсу, позволяет упростить класс `ServiceLoader`.

С этой целью определяется интерфейс (или суперкласс) с методами, которые должен предоставлять каждый экземпляр загружаемой службы. Допустим, служба обеспечивает шифрование данных, как показано ниже.

```

package serviceLoader;

public interface Cipher
{
    byte[] encrypt(byte[] source, byte[] key);
    byte[] decrypt(byte[] source, byte[] key);
    int strength();
}

```

Поставщик услуг предоставляет один или несколько классов, реализующих эту службу, например:

```

package serviceLoader.impl;

public class CaesarCipher implements Cipher
{

```



```

public byte[] encrypt(byte[] source, byte[] key)
{
    var = new byte[source.length];
    for (int i = 0; i < source.length; i++)
        result[i] = (byte) (source[i] + key[0]);
    return result;
}

public byte[] decrypt(byte[] source, byte[] key)
{
    return encrypt(source, new byte[] { (byte) -key[0] });
}

public int strength() { return 1; }
}

```

Классы, реализующие данную службу, могут находиться в любом пакете — совсем не обязательно в том же пакете, где находится интерфейс этой службы. Но в каждом из них должен быть непременно конструктор без аргументов.

Теперь в текстовый файл, имя которого совпадает с полностью уточненным именем класса, можно ввести имена классов в кодировке UTF-8 и сохранить его в каталоге META-INF/services. Так, файл META-INF/services/serviceLoader.Cipher будет содержать следующую строку кода:

```
serviceLoader.impl.CaesarCipher
```

В данном примере предоставляется единственный класс, реализующий нужную службу. Но можно предоставить несколько таких классов для последующего выбора наиболее подходящего из них.

После таких подготовительных действий загрузчик службы инициализируется в прикладной программе приведенным ниже образом, причем лишь один раз.

```

public static ServiceLoader<Cipher> cipherLoader =
    ServiceLoader.load(Cipher.class);

```

Метод `iterator()` загрузчика службы возвращает итератор для перебора всех предоставляемых реализаций службы. (Подробнее об итераторах речь пойдет в главе 9.) Но для их перебора проще всего организовать цикл `for`, выбирая в нем подходящий объект для реализации службы:

```

public static Cipher getCipher(int minStrength)
{
    for (Cipher cipher : cipherLoader)
        // неявно вызывает метод cipherLoader.iterator()
    {
        if (cipher.strength() >= minStrength) return cipher;
    }
    return null;
}

```

А с другой стороны, можно воспользоваться потоками данных (см. главу 1 второго тома настоящего издания), чтобы обнаружить требующуюся службу. В частности, метод `stream()` выдает поток экземпляров типа `ServiceLoader.Provider`. Этот тип относится к интерфейсу с двумя методами `type()` и `get()` для получения класса и экземпляра поставщика услуг. Так, если требуется выбрать поставщика услуг по типу, то для этого достаточно вызвать метод `type()`, не прибегая к необходимости получать экземпляры служб, как показано ниже.

```
public static Optional<Cipher> getCipher2(int minStrength)
{
    return cipherLoader.stream()
        .filter(descr -> descr.type() ==
            serviceLoader.impl.CaesarCipher.class)
        .findFirst()
        .map(ServiceLoader.Provider::get);
}
```

И, наконец, если требуется получить экземпляр любой службы, то для этого достаточно вызвать метод `findFirst()`, как показано ниже. Более подробно класс `Optional` описывается в главе 1 второго тома настоящего издания.

```
Optional<Cipher> cipher = cipherLoader.findFirst();
```

java.util.ServiceLoader<S> 1.6

- **static <S> ServiceLoader<S> load(Class<S> service)**
Создает загрузчик службы для загрузки классов, реализующих заданную службу.
- **Iterator<S> iterator()**
Предоставляет итератор, загружающий по требованию классы, реализующие данную службу. Это означает, что класс загружается всякий раз, когда итератор продвигается дальше по ходу итерации.
- **Stream<ServiceLoader.Provider<S>> stream() 9**
Возвращает поток дескрипторов поставщика услуг, чтобы загрузить по требованию поставщика требующегося класса.
- **Optional<S> findFirst() 9**
Обнаруживает первого же поставщика услуг, если таковой имеется.

java.util.ServiceLoader.Provider<S> 9

- **Class<? extends S> type()**
Получает тип данного поставщика услуг.
- **S get()**
Получает экземпляр данного поставщика услуг.

6.5. Прокси-классы

В последнем разделе этой главы мы обсудим понятие *прокси-классов*, которые зачастую называют *классами-посредниками*. Они предназначены для того, чтобы создавать во время выполнения программы новые классы, реализующие заданные интерфейсы. Прокси-классы требуются в том случае, если на стадии компиляции еще неизвестно, какие именно интерфейсы следует реализовать. В прикладном программировании такая ситуация возникает крайне редко. Но в некоторых приложениях системного программирования гибкость, обеспечиваемая прокси-классами, может оказаться весьма уместной.

6.5.1. О применении прокси-классов

Допустим, требуется сконструировать объект класса, реализующего один или несколько интерфейсов, конкретные характеристики которых во время компиляции неизвестны. Допустим также, что требуется создать объект класса, реализующего эти интерфейсы. Сделать это не так-то просто. Для построения конкретного класса достаточно воспользоваться методом `newInstance()` из класса `Class` или механизмом рефлексии, чтобы найти конструктор этого класса. Но получить экземпляр интерфейса нельзя. Следовательно, определить новый класс во время выполнения не удастся.

В качестве выхода из этого затруднительного положения в некоторых программах генерируется размещаемый в файле код, вызывается компилятор, а затем полученный файл класса. Естественно, что это очень медленный процесс, который к тому же требует разворачивания компилятора вместе с программой. Механизм прокси-классов предлагает более изящное решение. Прокси-класс может создавать во время выполнения совершенно новые классы и реализует те интерфейсы, которые указывает программист. В частности, в прокси-классе содержатся следующие методы.

- Все методы, которые требуют указанные интерфейсы.
- Все методы, определенные в классе `Object` (в том числе `toString()`, `equals()` и т.д.).

Но определить новый код для этих методов в ходе выполнения программы нельзя. Вместо этого программист должен предоставить *обработчик вызовов*, т.е. объект любого класса, реализующего интерфейс `InvocationHandler`. В этом интерфейсе единственный метод объявляется следующим образом:

```
Object invoke(Object proxy, Method method, Object[] args)
```

При вызове какого-нибудь метода для прокси-объекта автоматически вызывается метод `invoke()` из обработчика вызовов, получающий объект класса `Method` и параметры исходного вызова. Обработчик вызовов должен быть в состоянии обработать вызов.

6.5.2. Создание прокси-объектов

Для создания прокси-объекта служит метод `newProxyInstance()` из класса `Proxy`. Этот метод получает следующие три параметра.

- *Загрузчик классов.* Модель безопасности в Java позволяет использовать загрузчики разных классов, в том числе системных классов, загружаемых из Интернета, и т.д. Загрузчики классов обсуждаются в главе 9 второго тома настоящего издания. А до тех пор в приведенных далее примерах кода будет указываться пустое значение `null`, чтобы использовать загрузчик классов, предусмотренный по умолчанию.
- Массив объектов типа `Class` — по одному на каждый реализуемый интерфейс.
- Обработчик вызовов.

Остается решить еще два вопроса: как определить обработчик и что можно сделать с полученным в итоге прокси-объектом? Разумеется, ответы на эти вопросы зависят от конкретной задачи, которую требуется решить с помощью механизма прокси-объектов. В частности, их можно применять для достижения следующих целей.

- Переадресация вызовов методов на удаленный сервер.
- Связывание событий, происходящих в пользовательском интерфейсе, с определенными действиями, выполняемыми в программе.
- Отслеживание вызовов методов при отладке.

В рассматриваемом здесь примере программы прокси-объекты и обработчики вызовов применяются для отслеживания обращений к методам. С этой целью определяется приведенный ниже класс `TraceHandler`, заключающий некоторый объект в оболочку. Его метод `invoke()` лишь выводит на экран имя и параметры того метода, к которому выполнялось обращение, а затем вызывает сам метод, задавая в качестве неявного параметра объект, заключенный в оболочку.

```
class TraceHandler implements InvocationHandler
{
    private Object target;
    public TraceHandler(Object t)
    {
        target = t;
    }

    public Object invoke(Object proxy, Method m,
                        Object[] args)
        throws Throwable
    {
        // вывести метод и его параметры
        . . .
        // вызвать конкретный метод
        return m.invoke(target, args);
    }
}
```

Ниже показано, каким образом создается прокси-объект, позволяющий отслеживать вызов одного из его методов. Теперь всякий раз, когда метод вызывается для объекта прокси, выводятся его имя и параметры, а затем происходит обращение к соответствующему методу объекта `value`.

```
Object value = . . .;
// сконструировать оболочку
var handler = new TraceHandler(value);
// сконструировать прокси-объект для одного
// или нескольких интерфейсов
var interfaces = new Class[] { Comparable.class};
Object proxy = Proxy.newProxyInstance(
ClassLoader.getSystemClassLoader(),
new Class[] { Comparable.class } , handler);
```

В программе, приведенной в листинге 6.10, прокси-объекты служат для отслеживания результатов двоичного поиска. Сначала заполняется массив, состоящий из прокси-объектов для целых чисел от 1 до 1000. Затем для поиска случайного целого числа в массиве вызывается метод `binarySearch()` из класса `Arrays`. И, наконец, на экран выводится элемент, совпадающий с критерием поиска, как показано ниже.

```
var elements = new Object[1000];
// заполнить элементы прокси-объектами
// для целых чисел от 1 до 1000
```

```
for (int i = 0; i < elements.length; i++)
{
    Integer value = i + 1;
    elements[i] = Proxy.newProxyInstance(. . .);
    // прокси-объект для конкретного значения
}

// сформировать случайное целое число
Integer key = new Random().nextInt(elements.length) + 1;

// выполнить поиск по заданному критерию key
int result = Arrays.binarySearch(elements, key);

// вывести совпавший элемент, если таковой найден
if (result >= 0) System.out.println(elements[result]);
```

Класс `Integer` реализует интерфейс `Comparable`. Прокси-объекты принадлежат классу, определяемому во время выполнения. (Его имя выглядит как `$Proxy0`.) Этот класс также реализует интерфейс `Comparable`. Но в его методе `compareTo()` вызывается метод `invoke()` из обработчика вызовов прокси-объекта.



НА ЗАМЕТКУ! Как отмечалось в начале этой главы, в классе `Integer` фактически реализован интерфейс `Comparable<Integer>`. Но во время выполнения сведения об обобщенных типах удаляются, а при создании прокси-объекта используется объект базового типа `Comparable`.

Метод `binarySearch()` осуществляет вызов, аналогичный следующему:

```
if (elements[i].compareTo(key) < 0) ...
```

Массив заполнен прокси-объектами, и поэтому в методе `compareTo()` вызывается метод `invoke()` из класса `TraceHandler`. В этом методе выводится имя вызванного метода и его параметры, а затем вызывается метод `compareTo()` для заключенного в оболочку объекта типа `Integer`.

В конце программы из рассматриваемого здесь примера выводятся результаты ее работы, для чего служит следующая строка кода:

```
System.out.println(elements[result]);
```

Из метода `println()` вызывается метод `toString()` для прокси-объекта, а затем этот вызов также переадресуется обработчику вызовов. Ниже приведены результаты полной трассировки при выполнении программы.

```
500.compareTo(288)
250.compareTo(288)
375.compareTo(288)
312.compareTo(288)
281.compareTo(288)
296.compareTo(288)
288.compareTo(288)
288.toString()
```

Как видите, при двоичном поиске на каждом шаге интервал уменьшается вдвое. Обратите внимание на то, что и метод `toString()` представлен прокси-объектом, несмотря на то, что он не относится к интерфейсу `Comparable`. Как станет ясно в дальнейшем, некоторые методы из класса `Object` всегда представлены прокси-объектами.

Листинг 6.10. Исходный код из файла `proxy/ProxyTest.java`

```
1 package proxy;
2
3 import java.lang.reflect.*;
4 import java.util.*;
5
6 /**
7  * В этой программе демонстрируется
8  * применение прокси-объектов
9  * @version 1.01 2018-04-10
10 * @author Cay Horstmann
11 */
12 public class ProxyTest
13 {
14     public static void main(String[] args)
15     {
16         var elements = new Object[1000];
17
18         // заполнить массив elements прокси-объектами
19         // целых чисел в пределах от 1 до 1000
20         for (int i = 0; i < elements.length; i++)
21         {
22             Integer value = i + 1;
23             var handler = new TraceHandler(value);
24             Object proxy = Proxy.newProxyInstance(
25                 ClassLoader.getSystemClassLoader(),
26                 new Class[] { Comparable.class }, handler);
27             elements[i] = proxy;
28         }
29
30         // сформировать случайное целое число
31         Integer key = new Random().nextInt(
32             elements.length) + 1;
33
34         // выполнить поиск по критерию key
35         int result = Arrays.binarySearch(elements, key);
36
37         // вывести совпавший элемент, если таковой найден
38         if (result >= 0)
39             System.out.println(elements[result]);
40     }
41 }
42
43 /**
44 * Обработчик вызовов, выводящий сначала имя метода
45 * и его параметры, а затем вызываемый исходный метод
46 */
47 class TraceHandler implements InvocationHandler
48 {
49     private Object target;
50
51     /**
52      * Конструирует объекты типа TraceHandler
53      * @param t неявный параметр вызова метода
54      */
```

```
55 public TraceHandler(Object t)
56 {
57     target = t;
58 }
59
60 public Object invoke(Object proxy, Method m,
61                     Object[] args)
62     throws Throwable
63 {
64     // вывести неявный аргумент
65     System.out.print(target);
66     // вывести имя метода
67     System.out.print(".") + m.getName() + "(");
68     // вывести явные аргументы
69     if (args != null)
70     {
71         for (int i = 0; i < args.length; i++)
72         {
73             System.out.print(args[i]);
74             if (i < args.length - 1)
75                 System.out.print(", ");
76         }
77     }
78     System.out.println(")");
79
80     // вызвать конкретный метод
81     return m.invoke(target, args);
82 }
83 }
```

6.5.3. Свойства прокси-классов

Итак, показав прокси-классы в действии, вернемся к анализу некоторых их свойств. Напомним, что прокси-классы создаются во время выполнения, но затем они становятся обычными классами, как и все остальные классы, обрабатываемые виртуальной машиной.

Все прокси-классы расширяют класс `Proxy`. Такой класс содержит только одну переменную экземпляра, которая ссылается на обработчик вызовов, определенный в суперклассе `Proxy`. Любые дополнительные данные, необходимые для задач, решаемых прокси-объектами, должны храниться в обработчике вызовов. Например, в программе из листинга 6.10 при создании прокси-объектов, представляющих интерфейс `Comparable`, класс `TraceHandler` служит оболочкой для конкретных объектов.

Во всех прокси-классах переопределяются методы `toString()`, `equals()` и `hashCode()` из класса `Object`. Эти методы лишь вызывают метод `invoke()` для обработчика вызовов. Другие методы из класса `Object` (например, `clone()` и `getClass()`) не переопределяются. Имена прокси-классов не определены. В виртуальной машине формируются имена классов, начинающиеся со строки `$Proxy`.

Для конкретного загрузчика классов и заданного набора интерфейсов может существовать только один прокси-класс. Это означает, что, если дважды вызвать метод `newProxyInstance()` для одного и того же загрузчика классов и массива интерфейсов, будут получены два объекта одного и того же класса. Имя этого класса можно определить с помощью метода `getProxyClass()` следующим образом:

```
Class proxyClass = Proxy.getProxyClass(null, interfaces);
```

Прокси-класс всегда является открытым (`public`) и конечным (`final`). Если все интерфейсы, которые реализуются прокси-классом, объявлены как `public`, этот класс не принадлежит ни одному конкретному пакету. В противном случае все интерфейсы, в объявлении которых не указан модификатор доступа `public`, а следовательно, и сам прокси-класс, должны принадлежать одному пакету. Вызвав метод `isProxyClass()` из класса `Proxy`, можно проверить, представляет ли объект типа `Class` определенный прокси-класс.

`java.lang.reflect.InvocationHandler 1.3`

- `Object invoke(Object proxy, Method method, Object[] args)`

Этот метод определяется с целью задать действие, которое должно быть выполнено при вызове какого-нибудь метода для прокси-объекта.

`java.lang.reflect.Proxy 1.3`

- `static Class getProxyClass(ClassLoader loader, Class[] interfaces)`
Возвращает прокси-класс, реализующий заданные интерфейсы.
- `static Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler handler)`
Создает новый экземпляр прокси-класса, реализующего заданные интерфейсы. Во всех методах вызывается метод `invoke()` для объекта, указанного в качестве обработчика вызовов.
- `static boolean isProxyClass(Class c1)`
Возвращает логическое значение `true`, если параметр `c1` оказывается прокси-классом.

Этой главой завершается изложение основ языка Java. С понятиями интерфейсов, лямбда-выражений и внутренних классов вам придется встречаться еще не раз. В отличие от них, загрузчики классов, прокси-классы, клонирование и усовершенствованные методики представляют интерес в основном для разработчиков инструментальных средств, а не прикладных программ. Итак, овладев основами языка Java, вы можете перейти к изучению механизма обработки исключительных ситуаций, которые поясняются в главе 7.

Исключения, утверждения и протоколирование

В этой главе...

- ▶ Обработка ошибок
- ▶ Перехват исключений
- ▶ Рекомендации по обработке исключений
- ▶ Применение утверждений
- ▶ Протоколирование
- ▶ Рекомендации по отладке программ

В идеальном случае пользователи никогда не делают ошибок при вводе данных; файлы, которые они пытаются открыть, всегда существуют, а программы всегда работают правильно. Исходный код в приведенных до сих пор примерах был рассчитан именно на такой идеальный случай. Но теперь пришла пора опуститься с небес на землю, где людям свойственно делать ошибки при программировании и вводе данных, и рассмотреть механизмы Java, позволяющие выявлять и обрабатывать неизбежные ошибки в программах и данных.

Иметь дело с ошибками всегда неприятно. Если пользователь однажды потеряет все результаты своей работы из-за скрытой ошибки в программе или непредвиденного стечения обстоятельств, он может навсегда отвернуться от такой программы. В подобной ситуации необходимо сделать, по крайней мере, следующее.

- Сообщить пользователю об обнаруженной программной ошибке.
- Сохранить все результаты его работы.
- Дать ему возможность благополучно завершить программу.

В исключительных ситуациях, например, при вводе данных с ошибками, которые могут привести к нарушению нормальной работы программы, в Java применяется механизм перехвата ошибок, который называется *обработкой исключений*. Этот механизм аналогичен обработке исключений в C++ или Delphi и описывается в первой части этой главы.

Во время тестирования программы реализуется целый ряд проверок, чтобы убедиться в правильности ее работы. Но эти проверки зачастую оказываются слишком дорогостоящими по времени и ресурсам, а по окончании тестирования — излишними. Проверки можно удалить, а когда потребуется дополнительное тестирование, вернуть их обратно, хотя это и потребует определенных затрат труда. Вторая часть главы посвящена применению средств диагностических утверждений для выборочной активизации проверок.

Когда прикладная программа делает что-то не так, не всегда имеется возможность оказать пользователю своевременную помощь или прервать работу программы. Вместо этого можно зарегистрировать возникшую неполадку для последующего анализа. Поэтому третья часть главы посвящена средствам протоколирования в Java.

7.1. Обработка ошибок

Допустим, что в ходе выполнения программы, написанной на Java, обнаружена ошибка. Она может быть вызвана неверной информацией в файле, ненадежным сетевым соединением или выходом за допустимые границы массива, как бы неприятно ни было об этом упоминать, а может быть и попыткой использовать ссылку, которая не указывает на какой-то конкретный объект. Вполне естественно, пользователи надеются, что программа самостоятельно справится с возникшими затруднениями. Если из-за ошибки какая-нибудь операция не может быть завершена благополучно, программа должна сделать одно из двух.

- Вернуться в безопасное состояние и разрешить пользователю выполнить другие команды.
- Дать пользователю возможность сохранить результаты своей работы и аккуратно завершить работу.

Добиться этого не так-то просто: фрагмент кода, в котором обнаруживается ошибка (или даже тот фрагмент кода, выполнение которого приводит к ошибке), обычно находится очень далеко от кода, который может восстановить данные и сохранить результаты, полученные пользователем. Поэтому основное назначение механизма обработки исключений — передать данные обработчику исключений из того места, где возник сбой. Предусматривая обработку исключений в программе, следует предвидеть возможные ошибки и связанные с ними осложнения. Какие же ошибки следует рассмотреть в первую очередь?

- *Ошибки ввода.* В дополнение к неизбежным опечаткам пользователи часто предпочитают действовать так, как они считают нужным, а не следовать инструкциям разработчиков прикладных программ. Допустим, пользователю требуется перейти на веб-сайт, но он допустил синтаксическую ошибку при

вводе веб-адреса (URL) этого сайта. Программа должна была бы проверить синтаксис URL, но, предположим, что ее автор забыл это сделать. Тогда сетевое программное обеспечение сообщит об ошибке.

- *Сбои оборудования.* Оборудование не всегда работает должным образом. Принтер может оказаться выключенным, а веб-страница — временно недоступной. Зачастую оборудование перестает нормально работать в ходе выполнения задания, например, бумага в принтере может закончиться на середине печатаемой страницы.
- *Физические ограничения.* Диск может оказаться переполненным, а оперативная память — исчерпанной.
- *Ошибки программирования.* Какой-нибудь метод может работать неправильно. Например, он может возвращать неверный результат или некорректно вызывать другие методы. Выход за допустимые границы массива, попытка найти несуществующий элемент в хеш-таблице, извлечение элемента из пустого стека — все это характерные примеры ошибок программирования.

Обычно метод сообщает об ошибке, возвращая специальный код, который анализируется вызывающим методом. Например, методы, вводящие данные из файлов, обычно возвращают значение `-1` по достижении конца файла. Такой способ обработки ошибок часто оказывается эффективным. В других случаях в качестве признака ошибки возвращается пустое значение `null`.

К сожалению, возможность возвращать код ошибки имеется далеко не всегда. Иногда правильные данные не удается отличить от признаков ошибок. Так, метод, возвращающий целое значение, вряд ли возвратит значение `-1`, обнаружив ошибку, поскольку оно может быть получено в результате вычислений.

Как упоминалось в главе 5, в Java имеется возможность предусмотреть в каждом методе альтернативный выход, которым следует воспользоваться, если нормальное выполнение задания нельзя довести до конца. В этом случае метод не станет возвращать значение, а *сгенерирует* объект, инкапсулирующий сведения о возникшей ошибке. Следует, однако, иметь в виду, что выход из метода происходит незамедлительно, и он не возвращает своего нормального значения. Более того, возобновить выполнение кода, вызвавшего данный метод, невозможно. Вместо этого начинается поиск *обработчика исключений*, который может справиться с возникшей ошибочной ситуацией.

Исключения имеют свой собственный синтаксис и являются частью особой иерархии наследования. Сначала мы рассмотрим особенности синтаксиса исключений, а затем дадим ряд рекомендаций относительно эффективного применения обработчиков исключений.

7.1.1. Классификация исключений

В языке Java объект исключения всегда является экземпляром класса, производного от класса `Throwable`. Как станет ясно в дальнейшем, если стандартных классов недостаточно, можно создавать и свои собственные классы исключений. На рис. 7.1 показана в упрощенном виде иерархия наследования исключений в Java.

Обратите внимание на то, что иерархия наследования исключений сразу же разделяется на две ветви: `Error` и `Exception`, хотя общим предшественником для всех исключений является класс `Throwable`. Иерархия класса `Error` описывает внутренние ошибки и ситуации, возникающие в связи с нехваткой ресурсов в исполняющей

системе Java. Ни один объект этого класса нельзя сгенерировать самостоятельно. При возникновении внутренней ошибки в такой системе возможности разработчика прикладной программы крайне ограничены. Можно лишь уведомить пользователя и попытаться аккуратно прервать выполнение программы, хотя такие ситуации достаточно редки.

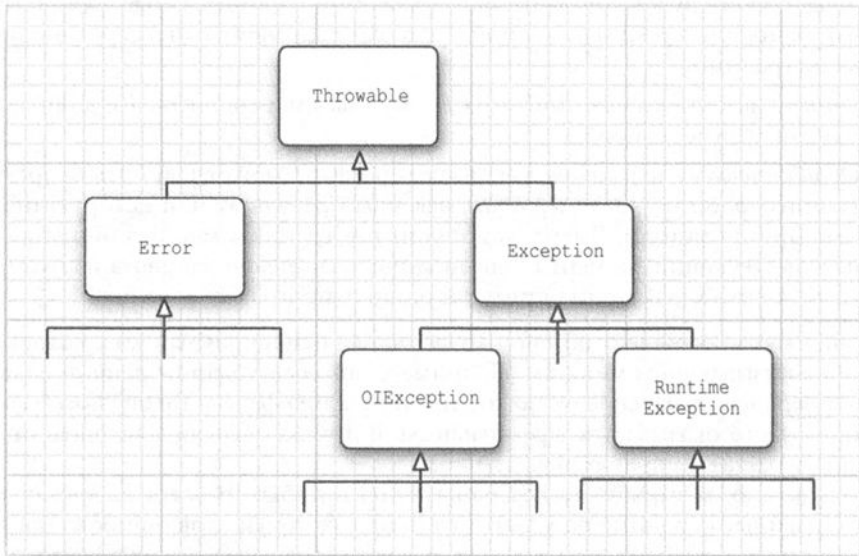


Рис. 7.1. Иерархия наследования исключений в Java

При программировании на Java основное внимание следует уделять иерархии класса `Exception`. Эта иерархия также разделяется на две ветви: исключения, производные от класса `RuntimeException`, и остальные. Исключения типа `RuntimeException` возникают вследствие ошибок программирования. Все другие виды исключений являются следствием непредвиденного стечения обстоятельств, например, ошибок ввода-вывода, возникающих при выполнении вполне корректных программ.

Исключения, производные от класса `RuntimeException`, связаны со следующими программными ошибками.

- Неверное приведение типов.
- Выход за пределы массива.
- Обращение к объекту по пустой ссылке `null`.

Остальные исключения возникают в следующих случаях.

- Попытка чтения по достижении конца файла.
- Попытка открыть несуществующий файл.
- Попытка получить объект типа `Class`, если в символьной строке указан несуществующий класс.

Исключения типа `RuntimeException` практически всегда возникают по вине программиста. Так, исключения типа `ArrayIndexOutOfBoundsException` можно избежать, если всегда проверять индексы массива. А исключение типа `NullPointerException`

никогда не возникнет, если перед тем, как воспользоваться переменной, проверить, не содержит ли она пустое значение `null`.

А как быть, если файл не существует? Нельзя ли сначала проверить, существует ли он вообще? Дело в том, что файл может быть удален сразу же после проверки его существования. Следовательно, понятие существования файла зависит от среды исполнения, а не от кода программы.

В спецификации языка Java любое исключение, производное от класса `Error` или `RuntimeException`, называется *непроверяемым*. Все остальные исключения называются *проверяемыми*. Для всех проверяемых исключений компилятор проверяет наличие соответствующих обработчиков.



НА ЗАМЕТКУ! Имя класса `RuntimeException` иногда вводит в заблуждение. Разумеется, все рассматриваемые здесь ошибки возникают во время выполнения программы.



НА ЗАМЕТКУ C++! Если вы знакомы с (намного более ограниченной) иерархией исключений из стандартной библиотеки C++, то, возможно, заметили некоторое противоречие в именах. В языке C++ имеются два основных класса, описывающих исключения: `runtime_error` и `logic_error`. Класс `logic_error` равнозначен классу `RuntimeException` в Java. А класс `runtime_error` является базовым для исключений, возникающих при непредвиденном стечении обстоятельств. Он равнозначен всем исключениям, предусмотренным в Java, но не относящимся к типу `RuntimeException`.

7.1.2. Объявление проверяемых исключений

Метод может генерировать исключения, если возникает ситуация, с которой он не в состоянии справиться. Идея проста: метод не только сообщает компилятору, какие значения он может возвращать, но и предсказывает, какие ошибки *могут* возникнуть. Например, в коде, вводящем данные из файла, можно предположить, что такого файла не существует или он пуст. Следовательно, код, предназначенный для ввода из файла, должен сообщить компилятору, что он может сгенерировать исключение типа `IOException`.

Объявление о том, что данный метод может генерировать исключение, делается в его заголовке. Ниже в качестве примера приведено объявление одного из конструкторов класса `FileInputStream` из стандартной библиотеки. (Подробнее о вводе-выводе речь пойдет в главе 1 второго тома настоящего издания.)

```
public FileInputStream(String name)
    throws FileNotFoundException
```

Это объявление означает, что данный конструктор создает объект типа `FileInputStream`, исходя из параметра `name` типа `String`, но в определенных случаях, когда что-нибудь пойдет не так, он может также сгенерировать исключение типа `FileNotFoundException`. Если это произойдет, исполняющая система начнет поиск обработчика событий, предназначенного для обработки объектов типа `FileNotFoundException`.

Создавая свой собственный метод, не нужно объявлять все возможные исключения, которые этот метод фактически может сгенерировать. Чтобы лучше понять, когда и что следует описывать с помощью оператора `throws`, имейте в виду, что исключение генерируется в следующих четырех случаях.

- Вызывается метод, генерирующий проверяемое исключение, например конструктор класса `FileInputStream`.
- Обнаружена ошибка, и с помощью оператора `throw` явным образом генерируется проверяемое исключение (оператор `throw` обсуждается в следующем разделе).
- Обнаружена ошибка программирования, например, в программе присутствует выражение `a[-1] = 0`, вследствие чего возникает непроверяемое исключение, например, типа `ArrayIndexOutOfBoundsException`.
- Возникает внутренняя ошибка в виртуальной машине или библиотеке исполняющей системы.

В любом из двух первых случаев нужно сообщить тем, кто будет пользоваться данным методом, что возможно исключение. Зачем? А затем, что любой метод, генерирующий исключение, представляет собой потенциально опасное место в прикладной программе. Если своевременно не предусмотреть в ней обработку данного типа исключения, то исполнение текущего потока прервется.

Методы являются составными частями предоставляемых классов, и поэтому объявить, что метод способен сгенерировать исключение, можно, включив в его заголовок *описание исключения*, как показано ниже.

```
class MyAnimation
{
    . . .
    public Image loadImage(String s) throws IOException
    {
        . . .
    }
}
```

Если же метод способен сгенерировать несколько проверяемых исключений, все они должны быть перечислены в его заголовке через запятую, как демонстрируется в следующем примере:

```
class MyAnimation
{
    . . .
    public Image loadImage(String s)
        throws FileNotFoundException, EOFException
    {
        . . .
    }
}
```

Но внутренние ошибки, т.е. исключения, производные от класса `Error`, объявлять не нужно. Такие исключения могут генерироваться любыми методами, а самое главное, что они не поддаются никакому контролю.

Точно так же совсем не обязательно объявлять непроверяемые исключения, производные от класса `RuntimeException`:

```
class MyAnimation
{
    . . .
    void drawImage(int i)
        throws ArrayIndexOutOfBoundsException
```

```
// не рекомендуется!  
{  
    . . .  
}  
}
```

Ответственность за подобные ошибки полностью возлагается на разработчика прикладной программы. Так, если вас беспокоит возможность выхода индекса за допустимые границы массива, лучше уделите больше внимания предотвращению этой исключительной ситуации, вместо того чтобы объявлять о потенциальной опасности ее возникновения.

Таким образом, в методе должны быть объявлены все *проверяемые* исключения, которые он может сгенерировать. А *непроверяемые* исключения находятся вне контроля разработчика данного метода (класс `Error`) или же являются следствием логических ошибок, которые не следовало допускать (класс `RuntimeException`). Если же в объявлении метода не сообщается обо всех проверяемых исключениях, компилятор выдаст сообщение об ошибке.

Разумеется, вместо объявления исключений, как было показано выше, их можно перехватывать. В этом случае исключение не генерируется, и объявлять его в операторе `throws` не нужно. Далее в этой главе мы обсудим, что лучше: перехватывать исключение самостоятельно или поручить это кому-нибудь другому.



ВНИМАНИЕ! Метод из подкласса не может генерировать более общие исключения, чем переопределяемый им метод из суперкласса. (В методе подкласса можно сгенерировать только конкретные исключения или не генерировать вообще никаких исключений.) Если, например, метод суперкласса вообще не генерирует проверяемые исключения, то и подкласс этого сделать не может. Так, если вы переопределяете метод `JComponent.paintComponent()`, то ваш метод `paintComponent()` не должен генерировать ни одного проверяемого исключения, поскольку соответствующий метод суперкласса этого не делает.

Если в объявлении метода указывается, что он способен сгенерировать исключение определенного класса, то он может сгенерировать исключения и его подклассов. Например, в конструкторе класса `FileInputStream` можно объявить о возможности генерирования исключения типа `IOException`, причем неизвестно, какого именно. Это может быть, в частности, простое исключение класса `IOException` или же объект одного из производных от него классов, в том числе класса `FileNotFoundException`.



НА ЗАМЕТКУ C++! Спецификатор `throws` в Java совпадает со спецификатором `throws` в C++. Но у них имеется одно существенное отличие. Спецификатор `throws` в C++ действует только во время выполнения, а не на стадии компиляции. Это означает, что компилятор C++ игнорирует объявление исключений. Но если исключение генерируется в функции, не включенной в список спецификатора `throws`, то вызывается функция `unexpected()` и по умолчанию выполнение программы прекращается. Если же спецификатор `throws` в коде C++ не указан, то функция может сгенерировать любое исключение. А в Java метод без спецификатора `throws` может вообще не генерировать проверяемые исключения.

7.1.3. Порядок генерирования исключений

Допустим, в прикладной программе произошло нечто ужасное. Так, в ней имеется метод `readData()`, вводящий данные из файла, в заголовке которого указано следующее:

```
Content-length: 1024
```


Но после ввода 733 символов достигнут конец файла. Такую ситуацию можно посчитать настолько ненормальной, что придется сгенерировать исключение. Далее необходимо решить, какого типа должно быть генерируемое исключение. В данном случае подходит одно из исключений типа `IOException`. В документации можно найти следующее описание исключения типа `EOFException`: “Сигнализирует о том, что во время ввода данных неожиданно обнаружен признак конца файла EOF”. Как раз то, что нужно! Такое исключение можно сгенерировать следующим способом:

```
throw new EOFException();
```

```
throw new EOFException();
```

или, если угодно, таким:

```
EOFException e = new EOFException();
throw e;
```

Ниже показано, как генерирование такого исключения реализуется непосредственно в коде.

```
String readData(Scanner in) throws EOFException
{
    . . .
    while ( . . . )
    {
        if (!in.hasNext()) // достигнут конец файла
                           // (признак EOF)
        {
            if (n < len)
                throw new EOFException();
        }
        . . .
    }
    return s;
}
```

В классе `EOFException` имеется второй конструктор, получающий в качестве параметра символьную строку. Его можно использовать для более подробного описания исключения, как показано ниже.

```
String gripe = "Content-length: " + len + ", Received: " + n;
throw new EOFException(gripe);
```

Как видите, если один из существующих классов исключений подходит для конкретных целей прикладной программы, то сгенерировать в ней исключение не составит большого труда. В этом случае необходимо сделать следующее.

1. Найти подходящий класс.
2. Создать экземпляр этого класса.
3. Сгенерировать исключение.

После того как исключение будет сгенерировано в методе, управление уже не возвратится в вызывающую часть программы. Это означает, что беспокоиться о возвращении значения, предусмотренного по умолчанию, или кода ошибки не нужно.



НА ЗАМЕТКУ C++! Генерирование исключений в C++ и Java происходит почти одинаково, за одним незначительным отличием. В языке Java можно генерировать только объекты классов, производных от класса `Throwable`, а в C++ — объекты любого класса.

7.1.4. Создание классов исключений

В прикладной программе может возникнуть ситуация, не предусмотренная ни в одном из стандартных классов исключений. В этом случае нетрудно создать свой собственный класс исключения. Очевидно, что он должен быть подклассом, производным от класса `Exception` или одного из его подклассов, например `IOException`, как показано ниже. Этот класс можно снабдить конструктором по умолчанию и конструктором, содержащим подробное сообщение. (Это сообщение, возвращаемое методом `toString()` из суперкласса `Throwable`, может оказаться полезным при отладке программы.)

```
class FileFormatException extends IOException
{
    public FileFormatException() {}
    public FileFormatException(String gripe)
    {
        super(gripe);
    }
}
```

А теперь можно сгенерировать исключение собственного типа следующим образом:

```
String readData(BufferedReader in) throws FileFormatException
{
    . . .
    while ( . . . )
    {
        if (ch == -1) // достигнут конец файла (признак EOF)

        {
            if (n < len)
                throw new FileFormatException();
        }
        . . .
    }
    return s;
}
```

`java.lang.Throwable` 1.0

- **`Throwable()`**
Создает новый объект типа **`Throwable`**, не сопровождая его подробным сообщением.
- **`Throwable(String message)`**
Создает новый объект типа **`Throwable`**, сопровождая его подробным сообщением. По соглашению все классы исключений должны содержать два конструктора: конструктор по умолчанию и конструктор с подробным сообщением.
- **`String getMessage()`**
Получает подробное сообщение, предусмотренное в объекте типа **`Throwable`**.

7.2. Перехват исключений

Теперь вы знаете, как генерировать исключения. Как видите, ничего сложного в этом нет: сгенерировав исключение, о нем можно попросту забыть. Разумеется, должен существовать код, позволяющий перехватить и обработать исключение. Именно об этом и пойдет речь в последующих разделах.

7.2.1. Перехват одного исключения

Если исключение возникает и нигде не перехватывается, то выполнение программы сразу же прерывается, а на консоль выводится сообщение о типе исключения и содержимое стека. Программы с графическим интерфейсом (настольные и веб-приложения) перехватывают исключения и выводят аналогичное сообщение, возвращаясь затем в цикл обработки событий в пользовательском интерфейсе. (Отлаживая программы с графическим интерфейсом, лучше не сворачивать консольное окно на экране.)

Перехват исключения осуществляется в блоке операторов `try/catch`. В простейшем случае этот блок имеет следующий вид:

```
try
{
    код
    дополнительный код
    дополнительный код
}
catch (ТипИсключения e)
{
    обработчик исключений данного типа
}
```

Если фрагмент кода в блоке оператора `try` генерирует исключение типа, указанного в заголовке блока оператора `catch`, то происходит следующее.

1. Программа пропускает оставшуюся часть кода в блоке оператора `try`.
2. Программа выполняет код обработчика в блоке оператора `catch`.

Если код в блоке оператора `try` не генерирует исключение, то программа пропускает блок оператора `catch`. А если какой-нибудь из операторов из блока `try` сгенерирует исключение, отличающееся от типа, указанного в блоке `catch`, то выполнение данной части программы (в частности, вызываемого метода) немедленно прекращается. (Остается только надеяться, что в вызывающей части программы все же предусмотрен перехват исключения данного типа.)

Чтобы продемонстрировать, как все описанное выше действует на практике, рассмотрим следующий типичный код для ввода данных:

```
public void read(String filename)
{
    try
    {
        InputStream in = new FileInputStream(filename);
        int b;
        while ((b = in.read()) != -1)
        {
            обработать введенные данные
        }
    }
}
```

```
    }  
  }  
  catch (IOException exception)  
  {  
    exception.printStackTrace();  
  }  
}
```

Обратите внимание на то, что большая часть кода в блоке оператора `try` выполняет простейшие действия: вводит и обрабатывает данные отдельными байтами до тех пор, пока не будет достигнут конец файла. Как указано в документации на прикладной программный интерфейс API, существует вероятность того, что метод `read()` сгенерирует исключение `IOException`. В таком случае пропускается весь остальной цикл `while`, происходит непосредственный переход к блоку оператора `catch` и генерируется трассировка стека. Для небольших программ это вполне благоразумный способ обработки исключений. А какие еще существуют возможности?

Зачастую лучше вообще ничего не делать, а просто передать исключение вызывающей части программы. Если в методе `read()` возникнет ошибка ввода, ответственность за обработку этой исключительной ситуации следует возложить на вызывающую часть программы! Придерживаясь такого подхода, достаточно указать, что метод `read()` способен сгенерировать исключение типа `IOException` следующим образом:

```
public void read(String filename) throws IOException  
{  
    InputStream in = new FileInputStream(filename);  
    int b;  
    while ((b = in.read()) != -1)  
    {  
        обработать введенные данные  
    }  
}
```

Напомним, что компилятор строго следит за спецификаторами `throws`. Вызывая метод, генерирующий проверяемое исключение, следует обработать его самостоятельно в этом методе или делегировать его обработку вызывающей части программы. Что же лучше? Как правило, следует перехватывать лишь те исключения, которые можно обработать самостоятельно, а остальные передавать дальше. Передавая исключение, следует добавить спецификатор `throws`, чтобы предупредить об этом вызывающую часть программы.

Просматривая документацию на прикладной интерфейс API, обратите внимание на методы, которые способны генерировать исключения. А затем решите, следует ли обрабатывать исключение самостоятельно или включить его в список спецификатора `throws`. В последнем варианте нет ничего постыдного: лучше предоставить обработку исключения более компетентному обработчику, чем пытаться организовать ее самостоятельно.

Как упоминалось ранее, из этого правила имеется одно исключение. Если создается метод, переопределяющий метод суперкласса, но не генерирующий ни одного исключения (например, метод `paintComponent()` из класса `JComponent`), каждое проверяемое исключение следует *непрерывно* перехватывать в теле этого метода самостоятельно. В этот метод из подкласса нельзя добавить спецификатор `throws`, отсутствующий в переопределяемом методе из суперкласса.



НА ЗАМЕТКУ C++! Перехват исключений в Java и C++ осуществляется почти одинаково. Следующий оператор в Java:

```
catch (Exception e) // Java
```

является аналогом приведенного ниже оператора в C++:

```
catch (Exception& e) // C++
```

Но в Java не существует аналога оператора `catch (. . .)`, доступного в C++. В языке Java он не требуется, потому что все исключения являются производными от общего суперкласса.

7.2.2. Перехват нескольких исключений

В блоке оператора `try` можно перехватить несколько исключений, обработав их по отдельности. Для каждого типа исключения следует предусмотреть свой блок оператора `catch` следующим образом:

```
try
{
    код, способный генерировать исключения
}
catch (FileNotFoundException e)
{
    чрезвычайные действия, если отсутствуют нужные файлы
}
catch (UnknownHostException e)
{
    чрезвычайные действия, если хосты неизвестны
}
catch (IOException e)
{
    чрезвычайные действия во всех остальных
    случаях появления ошибок ввода-вывода
}
```

Объект исключения содержит сведения о нем. Чтобы получить дополнительные сведения об этом объекте из подробного сообщения об ошибке, достаточно сделать вызов `e.getMessage()`, а для того чтобы получить конкретный тип объекта исключения — вызов `e.getClass().getName()`.

Начиная с версии Java 7, разнотипные исключения можно перехватывать в одном и том же блоке оператора `catch`. Допустим, что чрезвычайные действия, если нужные файлы отсутствуют или хосты неизвестны, одинаковы. В таком случае блоки оператора `catch` для перехвата обоих исключений можно объединить, как показано ниже. К такому способу перехвата исключений следует прибегать лишь в тех случаях, когда перехватываемые исключения не являются подклассами друг для друга.

```
try
{
    код, способный генерировать исключения
}
catch (FileNotFoundException | UnknownHostException e)
{
    чрезвычайные действия, если нужные файлы отсутствуют
    или неизвестны хосты
}
catch (IOException e)
```

```
{
    чрезвычайные действия во всех остальных
    случаях появления ошибок ввода-вывода
}
```



НА ЗАМЕТКУ! При перехвате нескольких исключений переменная исключения неявно считается конечной (`final`). Например, переменной `e` нельзя присвоить другое значение в следующем блоке:

```
catch (FileNotFoundException | UnknownHostException e) { ... }
```



НА ЗАМЕТКУ! Перехват нескольких исключений не делает код ни проще, ни эффективнее. Формируемые в итоге байт-коды содержат единственный блок общего оператора `catch`.

7.2.3. Повторное генерирование и связывание исключений в цепочку

Исключение можно генерировать и в блоке `catch`, образуя тем самым цепочку исключений. Обычно это делается для того, чтобы изменить тип исключения. Так, если разрабатывается подсистема для применения другими разработчиками, то имеет смысл генерировать такие исключения, которые давали бы возможность сразу определить, что ошибка возникла именно в этой подсистеме. В качестве характерного примера можно привести исключение типа `ServletException`. Вполне возможно, что в коде, где выполняется сервлет, совсем не обязательно иметь подробные сведения о том, какая именно возникла ошибка, а важно лишь знать, что сервлет работает неверно. В приведенном ниже фрагменте кода показано, каким образом перехватывается и повторно генерируется исключение.

```
try
{
    получить доступ к базе данных
}
catch (SQLException e)
{
    throw new ServletException("database error: "
                               + e.getMessage());
}
```

В данном случае текст сообщения об исключении формируется в конструкторе класса `ServletException`. Но предыдущее исключение лучше сделать причиной, т.е. источником последующего исключения:

```
try
{
    получить доступ к базе данных
}
catch (SQLException original)
{
    var e = new ServletException("database error");
    e.initCause(original);
    throw e;
}
```

Теперь при перехвате последующего исключения предыдущее исключение можно извлечь следующим образом:

```
Throwable original = caughtException.getCause();
```

Настоятельно рекомендуется именно такой способ заключения исключений в оболочку. Ведь он позволяет генерировать исключения более высокого уровня, не теряя подробных сведений об исходном исключении.



СОВЕТ. Рассмотренный выше способ заключения в оболочку оказывается удобным в том случае, если перехват исключения осуществляется в методе, которому не разрешается генерировать проверяемые исключения. Проверяемое исключение можно перехватить и заключить его в оболочку исключения времени выполнения.

Иногда требуется зарегистрировать исключение и сгенерировать его повторно без всяких изменений:

```
try
{
    получить доступ к базе данных
}
catch (Exception e)
{
    logger.log(level, message, e);
    throw e;
}
```

До версии Java 7 такому подходу был присущ существенный недостаток. Допустим, в теле метода имеется следующая строка кода:

```
public void updateRecord() throws SQLException
```

Раньше компилятор анализировал оператор `throw` в блоке `catch`, а затем тип переменной `e` и выдавал предупреждение, что данный метод может сгенерировать любое исключение типа `Exception`, а не только типа `SQLException`. Этот недостаток был устранен в версии Java 7. Теперь компилятор проверяет только тот факт, что переменная `e` происходит из блока `try`. Если в данном блоке экземплярами класса `SQLException` являются только проверяемые исключения и содержимое переменной `e` не изменяется в блоке `catch`, то в объявлении объемлющего метода можно с полным основанием указать генерирование исключения следующим образом: `throws SQLException`.

7.2.4. Блок оператора **finally**

Когда в методе генерируется исключение, оставшиеся в нем операторы не выполняются. Если же в методе задействованы какие-нибудь локальные ресурсы, о которых известно лишь ему, то освободить их уже нельзя. Можно, конечно, перехватить и повторно сгенерировать все исключения, но это не совсем удачное решение, поскольку ресурсы придется освобождать в двух местах: в обычном коде и в коде исключения. Разрешить подобное затруднение можно с помощью оператора `finally`.



НА ЗАМЕТКУ! В версии Java 7 появилась возможность разрешить подобное затруднение более изящным способом, используя оператор `try` с ресурсами, рассматриваемый в следующем разделе. А механизм освобождения ресурсов с помощью оператора `finally` подробно описывается здесь потому, что он закладывает концептуальное основание под обработку исключений. Хотя на практике вам, вероятнее всего, придется пользоваться оператором `try` с ресурсами чаще, чем оператором `finally`.

Код в блоке оператора `finally` выполняется независимо от того, возникло исключение или нет. Так, в приведенном ниже примере кода графический контекст освобождается при любых условиях.

```
InputStream in = new FileInputStream(...);
try
{
    // 1
    код, способный генерировать исключения
    // 2
}
catch (IOException e)
{
    // 3
    вывести сообщение об ошибке
    // 4
}
finally
{
    // 5
    in.close();
}
// 6
```

Рассмотрим три возможных ситуации, в которых программа выполняет блок оператора `finally`.

1. Код не генерирует никаких исключений. В этом случае программа сначала полностью выполняет блок оператора `try`, а затем блок оператора `finally`. Иными словами, выполнение программы последовательно проходит через точки 1, 2, 5 и 6.
2. Код генерирует исключение, которое перехватывается в блоке оператора `catch` (в данном примере это исключение типа `IOException`). В этом случае программа сначала выполняет блок оператора `try` до той точки, в которой возникает исключение, а остальная часть блока оператора `try` пропускается. Затем программа выполняет код из соответствующего блока оператора `catch` и, наконец, код из блока оператора `finally`.

Если в блоке оператора `catch` исключения не генерируются, то выполнение программы продолжается с первой строки, следующей после блока оператора `try`. Таким образом, выполнение программы последовательно проходит через точки 1, 3, 4, 5 и 6. Если же исключение генерируется в блоке оператора `catch`, то управление передается вызывающей части программы и выполнение программы проходит только через точки 1, 3 и 5.

3. Код генерирует исключение, которое не обрабатывается в блоке оператора `catch`. В этом случае программа выполняет блок `try` вплоть до той точки, в которой генерируется исключение, а оставшаяся часть блока оператора `try` пропускается. Затем программа выполняет код из блока оператора `finally` и передает исключение обратно вызывающей части программы. Таким образом, выполнение программы проходит только через точки 1 и 5.

Блок оператора `finally` можно использовать и без блока оператора `catch`. Рассмотрим следующий пример кода:


```
InputStream in = ...;
try
{
    код, способный генерировать исключения
}
finally
{
    in.close();
}
```

Оператор `in.close()` из блока `finally` выполняется независимо от того, возникает ли исключение в блоке `try`. Разумеется, если исключение возникает, оно будет перехвачено в очередном блоке `catch`, как показано ниже.

```
InputStream in = ...;
try
{
    try
    {
        код, способный генерировать исключения
    }
    finally
    {
        in.close();
    }
}
catch (IOException e)
{
    вывести сообщение об ошибке
}
```

Здесь внутренний блок оператора `try` отвечает только за закрытие потока ввода, а внешний блок оператора `try` сообщает об ошибках. Такой код не только более понятен, но и более функционален, поскольку ошибки выявляются и в блоке оператора `finally`.



ВНИМАНИЕ! Если в блоке `finally` имеется оператор `return`, результаты его выполнения могут быть неожиданными. Допустим, в середине блока оператора `try` происходит возврат из метода с помощью оператора `return`. Перед тем как передать управление вызывающей части программы, следует выполнить блок оператора `finally`. Если и в нем имеется оператор `return`, то первоначально возвращаемое значение будет замаскировано. Рассмотрим следующий пример кода:

```
public static int parseInt(String s)
{
    try
    {
        return Integer.parseInt(s);
    }
    finally
    {
        return 0; // ОШИБКА!
    }
}
```

На первый взгляд, если сделать вызов `parseInt("42")`, то из блока оператора `try` должно вернуться целочисленное значение `42`. Но блок оператора `finally` выполняется прежде возврата из метода `Integer.parseInt()`, и поэтому данный метод возвратит нулевое значение, пренебрегая первоначальным возвращаемым значением.

Хуже того, если сделать вызов `parseInt("zero")`, то в методе `Integer.parseInt()` будет сгенерировано исключение типа `NumberFormatException`. А затем будет выполнен блок оператора `finally`, где оператор `return` поглотит исключение!

Блок оператора `finally` предназначен для освобождения ресурсов. Поэтому в нем не следует размещать операторы (`return`, `throw`, `break`, `continue`), изменяющие порядок выполнения прикладного кода.

7.2.5. Оператор `try` с ресурсами

В версии Java 7 внедрена следующая удобная конструкция, упрощающая код обработки исключений, где требуется освобождать используемые ресурсы:

```
открыть ресурс
try
{
    использовать ресурс
}
finally
{
    закрыть ресурс
}
```

Следует, однако, иметь в виду, что эта конструкция эффективна при одном условии: используемый ресурс принадлежит классу, реализующему интерфейс `AutoCloseable`. В этом интерфейсе имеется единственный метод, объявляемый следующим образом:

```
void close() throws Exception
```



НА ЗАМЕТКУ! Имеется также интерфейс `Closeable`, производный от интерфейса `AutoCloseable`. В нем также присутствует единственный метод `close()`, но он объявляется для генерирования исключения типа `IOException`.

В своей простейшей форме оператор `try` с ресурсами выглядит следующим образом:

```
try (Resource res = ...)
{
    использовать ресурс res
}
```

Если в коде имеется блок оператора `try`, то метод `res.close()` вызывается автоматически. Ниже приведен типичный пример ввода всего текста из файла и последующего его вывода.

```
try (var in = new Scanner(new FileInputStream(
    "/usr/share/dict/words"), StandardCharsets.UTF_8))
{
    while (in.hasNext())
        System.out.println(in.next());
}
```

Независимо от того, происходит ли выход из блока оператора `try` нормально, или же в нем возникает исключение, метод `in.close()` вызывается в любом случае,

как и при использовании блока оператора `finally`. В блоке оператора `try` можно также указать несколько ресурсов, как в приведенном ниже примере кода.

```
try (var in = new Scanner(new FileInputStream(
    "/usr/share/dict/words"), StandardCharsets.UTF_8);
    var out = new PrintWriter("out.txt",
        StandardCharsets.UTF_8))
{
    while (in.hasNext())
        out.println(in.next().toUpperCase());
}
```

Таким образом, независимо от способа завершения блока оператора `try` оба потока ввода `in` и вывода `out` благополучно закрываются. Если бы такое освобождение ресурсов пришлось программировать вручную, для этого пришлось бы составлять вложенные блоки операторов `try/finally`.

В версии Java 9 появилась возможность предоставлять предварительно объявляемые действительно конечные переменные в заголовке оператора `try`, как показано ниже.

```
public static void printAll(String[] lines,
                           PrintWriter out)
{
    try (out) { // действительно конечная переменная
        for (String line : lines)
            out.println(line);
    } // здесь делается вызов out.close()
}
```

Трудности возникают в том случае, если исключение генерируется не только в блоке оператора `try`, но и в методе `close()`. Оператор `try` с ресурсами предоставляет довольно изящный выход из столь затруднительного положения. Исходное исключение генерируется повторно, а любые исключения, генерируемые в методе `close()`, считаются “подавленными”. Они автоматически перехватываются и добавляются к исходному исключению с помощью метода `addSuppressed()`. И если они представляют какой-то интерес с точки зрения обработки, то следует вызвать метод `getSuppressed()`, получающий массив подавленных исключений из метода `close()`.

Но самое главное, что все это не нужно программировать вручную. Всякий раз, когда требуется освободить используемый ресурс, достаточно применить оператор `try` с ресурсами.



НА ЗАМЕТКУ! Оператор `try` с ресурсами может также иметь сопутствующие операторы `catch` и `finally`. Блоки этих операторов выполняются после освобождения используемых ресурсов.

7.2.6. Анализ элементов трассировки стека

Трассировка стека — это список вызовов методов в данной точке выполнения программы. Вам, скорее всего, не раз приходилось видеть подобные списки. Ведь они выводятся всякий раз, когда при выполнении программы на Java возникает непроверяемое или необрабатываемое исключение.

Для получения текстового описания трассировки стека достаточно вызвать метод `printStackTrace()` из класса `Throwable`, как показано ниже.

```
var t = new Throwable();
var out = new StringWriter();
t.printStackTrace(new PrintWriter(out));
String description = out.toString();
```

Более удобный способ состоит в употреблении класса `StackWalker`, предоставляющего поток экземпляров класса `StackWalker.StackFrame`, каждый из которых описывает один фрейм стека. Обойти фреймы стека можно, сделав следующий вызов:

```
StackWalker walker = StackWalker.getInstance();
walker.forEach(frame -> проанализировать frame)
```

Если нужно обработать поток типа `Stream<StackWalker.StackFrame>` по требованию, то следует сделать приведенный ниже вызов. Более подробно обработка потоков данных рассматривается в главе 1 второго тома настоящего издания.

```
walker.walk(stream -> обработать stream)
```

В классе `StackWalker.StackFrame` имеются методы для получения имени файла и номера строки кода, а также объекта анализируемого класса и имени метода из исполняемой строки кода. В частности, метод `toString()` выдает отформатированную символьную строку, содержащую все эти сведения.



НА ЗАМЕТКУ! До версии Java 9 метод `Throwable.getStackTrace()` выдавал массив `StackTraceElement[]` с такой же информацией, как и в потоке экземпляров класса `StackWalker.StackFrame`. Но его вызов оказывается менее эффективным, поскольку он охватывает весь стек, несмотря на то, что в вызывающем коде может потребоваться лишь несколько фреймов стека. А кроме того, он предоставляет доступ лишь к именам классов, но не к их объектам из ожидающих методов.

В листинге 7.1 представлен исходный код программы, в которой выводится трассировка стека для рекурсивной функции вычисления факториала. Например, при вычислении факториала числа 3 получаются следующие результаты трассировки стека вызова `factorial(3)`:

```
factorial(3):
StackTraceTest.factorial(StackTraceTest.java:18)
StackTraceTest.main(StackTraceTest.java:34)
factorial(2):
StackTraceTest.factorial(StackTraceTest.java:18)
StackTraceTest.factorial(StackTraceTest.java:24)
StackTraceTest.main(StackTraceTest.java:34)
factorial(1):
StackTraceTest.factorial(StackTraceTest.java:18)
StackTraceTest.factorial(StackTraceTest.java:24)
StackTraceTest.factorial(StackTraceTest.java:24)
StackTraceTest.main(StackTraceTest.java:34)
return 1
return 2
return 6
```

Листинг 7.1. Исходный код из файла `stackTrace/StackTraceTest.java`

```
1 package stackTrace;
2
3 import java.util.*;
```

```
4
5 /**
6  * В этой программе выводится трассировка
7  * стека вызовов рекурсивного метода
8  * @version 1.10 2017-12-14
9  * @author Cay Horstmann
10 */
11 public class StackTraceTest
12 {
13     * Вычисляет факториал заданного числа
14     * @param n Положительное целое число
15     * @return n! = 1 * 2 * . . . * n
16     */
17     public static int factorial(int n)
18     {
19         System.out.println("factorial(" + n + "):");
20         var walker = StackWalker.getInstance();
21         walker.forEach(System.out::println);
22         int r;
23         if (n <= 1) r = 1;
24         else r = n * factorial(n - 1);
25         System.out.println("return " + r);
26         return r;
27     }
28
29     public static void main(String[] args)
30     {
31         try (var in = new Scanner(System.in))
32         {
33             System.out.print("Enter n: ");
34             int n = in.nextInt();
35             factorial(n);
36         }
37     }
38 }
```

java.lang.Throwable 1.0

- **Throwable(Throwable cause) 1.4**
- **Throwable(String message, Throwable cause) 1.4**
Создают объект типа **Throwable** по заданному объекту **cause**, который описывает причины возникновения исключения.
- **Throwable initCause(Throwable cause) 1.4**
Указывает причину создания данного объекта. Если причина создания объекта уже указана, генерирует исключение. Возвращает ссылку **this**.
- **Throwable getCause() 1.4**
Возвращает объект исключения, который был указан в качестве причины для создания данного объекта. Если причина создания данного объекта не указана, возвращается пустое значение **null**.
- **StackTraceElement[] getStackTrace() 1.4**
Возвращает трассировку стека в момент создания объекта.

java.lang.Throwable 1.0 (окончание)

- **void addSuppressed(Throwable t) 7**

Добавляет к данному исключению “подавленное” исключение. Это происходит в операторе **try** с ресурсами, где **t** — исключение, генерируемое методом **close()**.

- **Throwable[] getSuppressed() 7**

Получает все исключения, “подавленные” данным исключением. Как правило, это исключения, генерируемые методом **close()** в операторе **try** с ресурсами.

java.lang.Exception 1.0

- **Exception(Throwable cause) 1.4**

- **Exception(String message, Throwable cause)**

Создают объект типа **Exception** по заданному объекту **cause**.

java.lang.StackWalker.StackFrame 9

- **static StackWalker getInstance()**

- **static StackWalker getInstance(StackWalker.Option option)**

- **static StackWalker getInstance(Set<StackWalker.Option> options)**

Получают экземпляр класса **StackWalker**. В качестве параметров можно указать константы **RETAIN_CLASS_REFERENCE**, **SHOW_HIDDEN_FRAMES** и **SHOW_REFLECT_FRAMES** из перечисления **StackWalker.Option**.

- **forEach(Consumer<? super StackWalker.StackFrame> action)**

Выполняет заданное действие над каждым фреймом стека, начиная с самого последнего из вызванных методов.

- **walk(Function<? super Stream<StackWalker.StackFrame>, ? extends T> function)**

Применяет заданную функцию к потоку фреймов стека и возвращает результат выполнения данной функции.

java.lang.StackWalker.StackFrame 9

- **String getFileName()**

Получает имя исходного файла, содержащего точку исполнения данного элемента, а если сведения о нем недоступны — пустое значение **null**.

- **int getLineNumber()**

Получает номер строки кода из исходного файла, содержащего точку исполнения данного элемента, а если сведения о нем недоступны — значение **-1**.

java.lang.StackWalker.StackFrame 9 *(окончание)*

- **String getClassNames()**

Получает полностью уточненное имя класса, метод которого содержит точку исполнения данного элемента.

- **String getDeclaringClass()**

Получает объект типа **Class**, метод которого содержит точку исполнения данного элемента. Если обходчик стека построен с параметром **RETAIN_CLASS_REFERENCE**, то генерируется исключение.

- **String getMethodName()**

Получает имя метода, содержащего точку исполнения данного элемента. Имя конструктора обозначается как **<init>**, а имя статического инициализатора — как **<clinit>**. Хотя отличить перегружаемые методы с одинаковыми именами невозможно.

- **boolean isNativeMethod()**

Возвращает логическое значение **true**, если точка исполнения данного элемента находится в платформенно-ориентированном методе.

- **String toString()**

Возвращает отформатированную символьную строку, содержащую имя класса, метода и файла, а также номер строки кода по мере их доступности.

java.lang.StackTraceElement 1.4

- **String getFileName()**

Получает имя исходного файла, содержащего точку, в которой выполняется данный элемент. Если эти сведения недоступны, возвращается пустое значение **null**.

- **int getLineNumber()**

Получает номер строки кода, содержащей точку, в которой выполняется данный элемент. Если эти сведения недоступны, возвращается значение **-1**.

- **String getClassNames()**

Возвращает полное имя класса, содержащего точку, в которой выполняется данный элемент.

- **String getMethodName()**

Возвращает имя метода, содержащего точку, в которой выполняется данный элемент. Имя конструктора — **<init>**, а имя статического инициализатора — **<clinit>**. Перегружаемые методы с одинаковыми именами не различаются.

- **boolean isNativeMethod()**

Возвращает логическое значение **true**, если точка выполнения данного элемента находится в собственном методе.

- **String toString()**

Возвращает отформатированную символьную строку, содержащую имя класса и метода, а также имя файла и номер строки кода (если эти сведения доступны).

7.3. Рекомендации по обработке исключений

Специалисты придерживаются разных мнений относительно обработки исключений. Одни считают, что проверяемые исключения — это не более чем досадная помеха, а другие готовы затратить дополнительное время и труд на их обработку. На наш взгляд, исключения (особенно проверяемые) — полезный механизм, но, применяя его, не стоит слишком увлекаться. В этом разделе дается ряд рекомендаций относительно применения и обработки исключений в прикладных программах.

1. Обработка исключений не может заменить собой простую проверку.

Для иллюстрации этого положения ниже приведен фрагмент кода, в котором используется встроенный класс `Stack`. В этом коде делается 10 миллионов попыток извлечь элемент из пустого стека. Сначала в нем выясняется, пуст ли стек:

```
if (!s.empty()) s.pop();
```

Затем элемент принудительно извлекается из стека, независимо от того, пуст он или заполнен, как показано ниже. После этого перехватывается исключение типа `EmptyStackException`, которое предупреждает, что так делать нельзя.

```
try()
{
    s.pop();
}
catch (EmptyStackException e)
{
}
```

Время, затраченное на тестовом компьютере для вызова метода `isEmpty()`, составило 646 миллисекунд, а на перехват исключения типа `EmptyStackException` — 21739 миллисекунд.

Как видите, перехват исключения занял намного больше времени, чем простая проверка. Из этого следует вывод: пользуйтесь исключениями только в тех случаях, когда это оправданно, что зачастую бывает лишь в исключительных ситуациях.

2. Не доводите обработку исключений до абсурдных мелочей.

Многие программисты заключают едва ли не каждый оператор в отдельный блок оператора `try`. Ниже показано, к чему приводит подобная мелочность при программировании на Java.

```
PrintStream out;
Stack s;
for (i = 0; i < 100; i++)
{
    try
    {
        n = s.pop();
    }
    catch (EmptyStackException e)
    {
        // стек оказался пустым
    }
    try
```



```
{
    out.writeInt(n);
}
catch (IOException e)
{
    // сбой при записи данных в файл
}
}
```

При таком подходе объем кода значительно увеличивается. Поэтому хорошенько подумайте о той задаче, которую должна решать ваша программа. В данном случае требуется извлечь из стека 100 чисел и записать их в файл. (Неважно, зачем это нужно; ведь это всего лишь пример.) Одно только генерирование исключения не является выходом из положения. Ведь если стек пуст, то он не заполнится как по мановению волшебной палочки. А если при записи числовых данных в файл возникает ошибка, то она не исчезнет сама собой. Следовательно, имеет смысл разместить в блоке `try` *весь* фрагмент кода для решения поставленной задачи, как показано ниже. Если хотя бы одна из операций в этом блоке даст сбой, можно отказаться от решения всей задачи, а не отдельных ее частей.

```
try
{
    for (i = 0; i < 100; i++)
    {
        n = s.pop();
        out.writeInt(n);
    }
}
catch (IOException e)
{
    // сбой при записи данных в файл
}
catch (EmptyStackException e)
{
    // стек оказался пустым
}
```

Этот фрагмент кода уже выглядит намного яснее. Он выполняет одну из своих основных обязанностей — *отделять* нормальную обработку данных от обработки исключений.

3. *Правильно пользуйтесь возможностями, которые предоставляет иерархия наследования исключений.*

Не ограничивайтесь генерированием только исключения типа `RuntimeException`. Найдите подходящий подкласс или создайте собственный. Не перехватывайте исключение типа `Throwable`. При таком подходе ваш код становится трудным для понимания и сопровождения.

Правильно различайте проверяемые и непроверяемые исключения. Для обработки проверяемых исключений требуются дополнительные усилия, поэтому не применяйте их для уведомления о логических ошибках. (В библиотеке, поддерживающей механизм рефлексии, это правило не соблюдается. Поэтому ее пользователям часто приходится писать код для перехвата тех исключений, которые могут вообще не возникнуть.)

Смело преобразуйте, если требуется, один тип исключения в другой, более подходящий в данной ситуации. Если вы, скажем, выполняете синтаксический анализ целого значения, вводимого из файла, перехватывайте исключение класса `NumberFormatException` и преобразуйте его в исключение подкласса, производного от класса `IOException` или `MySubsystemException`, для последующего генерирования.

4. Не подавляйте исключения.

При программировании на Java существует большой соблазн подавить исключения. Допустим, требуется написать метод, вызывающий другой метод, который может сгенерировать исключение один раз в сто лет. Компилятор предупредит об этом, поскольку исключение не указано в списке оператора `throws` при объявлении данного метода. Но если нет никакого желания указывать его в списке оператора `throws`, чтобы компилятор не выдавал сообщения об ошибках во всех методах, вызывающих данный метод, то такое исключение можно просто подавить следующим образом:

```
public Image loadImage(String s)
{
    try
    {
        код, способный генерировать проверяемые исключения
    }
    catch (Exception e)
    {} // ничего не делать!
}
```

Теперь код будет скомпилирован. Он будет прекрасно работать, но не в исключительной ситуации. А если она возникнет, то будет просто проигнорирована. Но если вы все-таки считаете, что подобные исключения важны, приложите усилия для их обработки.

5. Обнаруживая ошибки, проявляйте необходимую твердость вместо излишней терпимости.

Некоторые программисты колеблются, стоит ли генерировать исключение, когда обнаруживается ошибка. Может быть, лучше вернуть фиктивное значение, чем генерировать исключение, когда методу передаются неверные параметры? А когда стек оказывается пустым, то следует ли возвращать из метода `Stack.pop()` пустое значение `null` вместо того, чтобы генерировать исключение? На наш взгляд, лучше сгенерировать исключение типа `EmptyStackException` в той точке, где возникла ошибка, чем исключение типа `NullPointerException` впоследствии.

6. Не бойтесь передавать исключения для обработки в коде, разрабатываемом другими.

Многие программисты считают себя обязанными перехватывать все исключения. Вызывая метод, генерирующий некоторое исключение, например, конструктор класса `FileInputStream` или метод `readLine()`, они инстинктивно перехватывают исключения, которые могут быть при этом сгенерированы. Но зачастую предпочтительнее *передать* исключение другому обработчику, а не обрабатывать его самостоятельно, как показано ниже.

```
public void readStuff(String filename)
    throws IOException // Ничтоже сумняшеся!
{
    var in = new FileInputStream(filename,
                                StandardCharsets.UTF_8);
    . . .
}
```

Методы более высокого уровня лучше оснащены средствами уведомления пользователей об ошибках или отмены выполнения неверных операций.



НА ЗАМЕТКУ! Рекомендации в пп.5 и 6 можно свести к следующему общему правилу: генерировать исключения раньше, а перехватывать их позже.

7.4. Применение утверждений

Утверждения — широко распространенное средство так называемого *безопасного* программирования. В последующих разделах будет показано, как пользоваться ими эффективно.

7.4.1. Понятие утверждения

Допустим, вы убеждены, что конкретное свойство уже задано, и обращаетесь к нему в своей программе. Так, при вычислении следующего выражения вы уверены, что значение параметра *x* не является отрицательным:

```
double y = Math.sqrt(x);
```

Возможно, оно получено в результате других вычислений, которые не могут порождать отрицательные числовые значения, или же является параметром метода, которому можно передавать только положительные числовые значения. Но вы все же хотите еще раз проверить свои предположения, чтобы в ходе выполнения программы не возникло ошибки. Разумеется, можно было бы просто сгенерировать исключение следующим образом:

```
if (x < 0) throw new IllegalArgumentException("x < 0");
```

Но у такого подхода имеется следующий существенный недостаток: приведенный выше код остается в программе даже после ее тестирования. Если в исходном коде программы расставлено много таких проверок, ее выполнение может замедлиться. Механизм утверждений позволяет вводить в исходный код программы проверки для ее тестирования, а затем удалять их из окончательного варианта. Для этой цели имеется ключевое слово `assert`. Ниже приведены две его основные формы.

```
assert условие;
```

и

```
assert условие : выражение;
```

Оба приведенных выше оператора проверяют заданное *условие* и генерируют исключение типа `AssertionError`, если оно не выполняется. Во втором операторе *выражение* передается конструктору объекта типа `AssertionError` и преобразуется в символьную строку сообщения.



НА ЗАМЕТКУ! Единственная цель, которую преследует **выражение** в операторе **assert**, — получить символьную строку сообщения. В объекте типа **AssertionError** конкретное значение выражения не хранится, поэтому его нельзя запросить в дальнейшем. В документации на JDK снисходительно сообщается, что это сделано для того, чтобы “стимулировать программистов устранять ошибки, выявляемые в результате непрохождения тестов на утверждения, которые противоречат основному назначению программных средств”.

Чтобы проверить, является ли числовое значение переменной *x* неотрицательным, достаточно написать следующий оператор:

```
assert x >= 0;
```

Кроме того, можно передать конкретное значение переменной *x* объекту типа **AssertionError**, чтобы впоследствии вывести его, как показано ниже.

```
assert x >= 0 : x;
```



НА ЗАМЕТКУ C++! Макрокоманда **assert** в C преобразует проверяемое условие в символьную строку, которая выводится, если условие не выполняется. Так, если выражение **assert(x>=0)** имеет логическое значение **false**, выводится символьная строка, сообщающая, что условие **x >= 0** не выполняется. А в Java условие не включается автоматически в сообщение об ошибке. Если же условие требуется отобразить, его можно передать в виде символьной строки объекту типа **AssertionError** следующим образом: **assert x >= 0 : "x >= 0"**.

7.4.2. Разрешение и запрет утверждений

По умолчанию утверждения запрещены. Разрешить их в ходе выполнения программы можно с помощью параметра **-enableassertions** или **-ea**, указываемого в командной строке следующим образом:

```
java -enableassertions MyApp
```

Следует, однако, иметь в виду, что для разрешения и запрета утверждений компилировать программу заново не нужно. Это задача *загрузчика классов*. Если диагностические утверждения запрещены, загрузчик классов проигнорирует их код, чтобы не замедлять выполнение программы.

Разрешать утверждения можно в отдельных классах или в целых пакетах следующим образом:

```
java -ea:MyClass -ea:com.mycompany.mylib ... MyApp
```

Эта команда разрешает утверждения в классе **MyClass**, а также во всех классах из пакета **com.mycompany.mylib** и его *подчиненных* пакетов. Параметр **-ea...** разрешает утверждения во всех классах из пакета, выбираемого по умолчанию.

Кроме того, утверждения можно запретить в определенных классах и пакетах. Для этого достаточно указать параметр **-disableassertions** или **-da** в командной строке следующим образом:

```
java -ea:... -da:MyClass MyApp
```

Некоторые классы загружаются не загрузчиком классов, а непосредственно виртуальной машиной. Для выборочного разрешения или запрета утверждений, содержащихся в этих классах, можно также указать параметр **-ea** и **-da** в командной строке. Но эти параметры разрешают и запрещают все утверждения одновременно, и поэтому их нельзя применять в системных классах без загрузчика классов. Для разрешения утверждений, находящихся в системных классах, следует указать параметр **-enablesystemassertions/-esa**. Имеется также возможность программно

управлять состоянием утверждений. Подробнее об этом можно узнать в документации на прикладной программный интерфейс API.

7.4.3. Проверка параметров с помощью утверждений

В языке Java предусмотрены следующие три механизма обработки системных сбоев.

- Генерирование исключений.
- Протоколирование.
- Применение утверждений.

Когда же следует применять утверждения? Чтобы правильно ответить на этот вопрос, необходимо учесть следующее.

- Утверждения оказываются ложными, если произошла неустраняемая ошибка.
- Утверждения разрешаются только на время отладки и тестирования программ. (Иногда это положение шутливо формулируют так: надевайте спасательный жилет, плавая у берега, а в открытом море он вам все равно не поможет.)

Итак, утверждения не следует применять в качестве индикаторов несущественных и легко исправимых ошибок. Они предназначены для выявления серьезных неполадок во время тестирования.

Рассмотрим типичную ситуацию, возникающую при проверке параметров метода. Следует ли применять утверждения, чтобы проверить, не выходит ли индекс за допустимые пределы, или выявить пустую ссылку? Чтобы ответить на этот вопрос, необходимо обратиться к документации, описывающей данный метод. В качестве примера рассмотрим метод `Array.sort()` из стандартной библиотеки.

```
/**
 * Упорядочивает указанную часть заданного массива
 * по нарастающей. Упорядочиваемая часть массива
 * начинается с индекса fromIndex и заканчивается
 * индексом, предшествующим toIndex, т.е. элемент с
 * индексом toIndex упорядочению не подлежит.
 * @param a Упорядочиваемый массив
 * @param fromIndex Индекс первого элемента упорядочиваемой
 *                  части массива (включительно)
 * @param toIndex Индекс первого элемента, находящегося за
 *                 пределами упорядочиваемой части массива
 * @throws IllegalArgumentException Если fromIndex > toIndex,
 *                                генерируется исключение
 * @throws ArrayIndexOutOfBoundsException Если fromIndex < 0
 *                                или toIndex > a.length, генерируется исключение
 */
static void sort[] a, int fromIndex, int toIndex
```

В документации утверждается, что данный метод генерирует исключение, если значение индекса задано неверно. Такое поведение метода является частью контракта, который он заключает с вызывающим кодом. Если вы реализуете этот метод, то должны придерживаться данного контракта и генерировать соответствующее исключение. В этом случае утверждения не подходят.

Следует ли выявлять с помощью утверждений пустые ссылки? Как и прежде, ответ отрицательный. В документации на рассматриваемый здесь метод ничего не говорится о его реакции в том случае, если параметр `a` имеет пустое значение `null`. В вызывающей части программы имеются все основания предполагать, что в этом

случае метод будет выполнен и не сгенерирует исключение типа `AssertionError`. Допустим, однако, что в контракт на метод внесено следующее изменение:

@param a Упорядочиваемый массив. (Не должен быть пустым)

Теперь в вызывающей части программы известно, что передавать пустой массив рассматриваемому здесь методу запрещено. Поэтому данный метод может начинаться со следующего утверждения:

```
assert(a != null);
```

Специалисты по вычислительной технике называют подобный контракт *предусловием*. В исходном методе на параметры не накладывалось никаких предусловий, так как предполагалось, что он будет правильно действовать во всех случаях. А в видоизмененном методе накладывается следующее предусловие: параметр `a` не должен принимать пустое значение `null`. Если в вызывающей части программы данное предусловие нарушается, то контракт не соблюдается и метод не обязан его выполнять. На самом деле, имея утверждение, метод может повести себя непредсказуемо, если он вызывается неверно. В одних случаях он может сгенерировать исключение типа `assertionError`, а в других — исключение типа `NullPointerException`, что зависит от конкретной конфигурации загрузчика классов.

7.4.4. Документирование предположений с помощью утверждений

Многие программисты оформляют свои предположения о работе программы в виде комментариев. Рассмотрим следующий пример из документации, доступной по адресу <https://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html>:

```
if (i % 3 == 0)
    . . .
else if (i % 3 == 1)
    . . .
else // (i % 3 == 2)
```

В данном случае намного больше оснований использовать утверждения следующим образом:

```
if (i % 3 == 0)
    . . .
else if (i % 3 == 1)
    . . .
else
{
    assert i % 3 == 2;
    . . .
}
```

Безусловно, стоило бы тщательнее сформулировать проверки. В частности, какие значения может принимать выражение `i % 3`? Если переменная `i` принимает положительное числовое значение, то остаток может быть равным 0, 1 или 2. А если она принимает отрицательное значение, то остаток может быть равным -1 или -2. Таким образом, намного логичнее предположить, что переменная `i` не содержит отрицательное числовое значение. Для этого можно составить следующее утверждение, введя его перед условным оператором `if`:

```
assert(i >= 0);
```

Во всяком случае, рассмотренный выше пример показывает, что утверждения удобны для самопроверки в процессе программирования. Они являются тактическим средством для тестирования и отладки, тогда как протоколирование — стратегический механизм, сопровождающий программу на протяжении всего срока ее службы. Подробнее о протоколировании речь пойдет в следующем разделе.

`java.lang.ClassLoader 1.0`

- `void setDefaultAssertionStatus(boolean b) 1.4`
Разрешает или запрещает утверждения во всех классах, загруженных указанным загрузчиком классов, в которых состояние утверждений не задано явным образом.
- `void setClassAssertionStatus(String className, boolean b) 1.4`
Разрешает или запрещает утверждения в заданном классе и его подклассах.
- `void setPackageAssertionStatus(String packageName, boolean b) 1.4`
Разрешает или запрещает утверждения во всех классах заданного пакета и его подчиненных пакетов.
- `void clearAssertionStatus() 1.4`
Удаляет все утверждения, состояние которых задано явным образом, а также запрещает все утверждения в классах, загруженных данным загрузчиком.

7.5. Протоколирование

Программирующие на Java часто прибегают к вызовам метода `System.out.println()`, чтобы отладить код и уяснить поведение программы. Разумеется, после устранения ошибки следует убрать промежуточный вывод и ввести его в другом месте программы для выявления очередной неполадки. С целью упростить отладку программ в подобном режиме предусмотрен прикладной программный интерфейс API для протоколирования. Ниже перечислены основные преимущества его применения.

- Все протокольные записи нетрудно запретить или разрешить.
- Запрещенные протокольные записи отнимают немного ресурсов и не влияют на эффективность работы приложения.
- Протокольные записи можно направить разным обработчикам, вывести на консоль, записать в файл и т.п.
- Регистраторы и обработчики способны фильтровать записи. Фильтры отбрасывают ненужные записи по критериям, предоставляемым теми, кто реализует фильтры.
- Протокольные записи допускают форматирование. Их можно, например, представить в виде простого текста или в формате XML.
- В приложениях можно использовать несколько протоколов, имеющих иерархические имена, подобные именам пакетов, например `com.myscompany.myapp`.
- Конфигурирование протоколирования определяется в файле конфигурации.



НА ЗАМЕТКУ! Во многих приложениях применяются другие библиотеки протоколирования, в том числе Log4J 2 (<https://logging.apache.org/log4j/2.x>) и Logback (<https://logback.qos.ch>), обеспечивающие более высокую производительность, чем стандартная для Java библиотека протоколирования. Такие библиотеки несколько отличаются своими прикладными интерфейсами API. Такие фасады протоколирования, как SLF4J (<https://www.slf4j.org>) и Commons Logging

(<https://commons.apache.org/proper/commons-logging/>), предоставляют единообразный прикладной интерфейс API, чтобы можно было сменить библиотеку протоколирования, не переписывая прикладной код. Дело усложняется еще и тем, что библиотека Log4J 2 может служить фасадом для таких компонентов, как SLF4J. В настоящем издании рассматривается стандартная библиотека протоколирования в Java. Поэтому, преследуя самые разные цели, целесообразно изучить прикладной интерфейс API этой библиотеки, чтобы лучше понять имеющиеся ее альтернативы.



НА ЗАМЕТКУ! В версии Java 9 на платформе Java появилась отдельная легковесная система протоколирования, не зависящая от модуля `java.logging`, где содержится стандартная библиотека протоколирования в Java. Эта система предназначена для применения только в прикладном интерфейсе Java API. Если модуль `java.logging` присутствует, протокольные сообщения автоматически направляются на обработку именно ему. В сторонних библиотеках протоколирования могут предоставляться адаптеры для приема протокольных сообщений из платформы Java. Протоколирование на уровне платформы Java здесь не рассматривается потому, что оно не предназначено для прикладного программирования на Java.

7.5.1. Элементарное протоколирование

Для элементарного протоколирования служит глобальный регистратор, который вызывается следующим образом:

```
Logger.getGlobal().info("File->Open menu item selected");
```

По умолчанию выводится следующая протокольная запись:

```
May 10, 2013 10:12:15 PM LoggingImageViewer fileOpen  
INFO: File->Open menu item selected
```

Но если в соответствующем месте программы, например, в начале метода `main()`, ввести приведенную ниже строку кода, то протокольная запись будет запрещена.

```
Logger.getGlobal().setLevel(Level.OFF)
```

7.5.2. Усовершенствованное протоколирование

Итак, рассмотрев элементарное протоколирование, перейдем к средствам протоколирования, применяемым при создании программ по промышленным стандартам. В профессионально разработанном приложении, как правило, все записи не накапливаются в одном глобальном протоколе, поэтому можно определить свои собственные средства протоколирования. Так, для создания или извлечения регистратора вызывается метод `getLogger()`:

```
private static final Logger myLogger =  
    Logger.getLogger("com.mycompany.myapp");
```



СОВЕТ. Регистратор, на который больше не делается ссылка ни в одной из переменных, собирается в "мусор". Во избежание этого следует сохранить ссылку на регистратор в статической переменной, как показано в приведенном выше примере кода.

Как и имена пакетов, имена регистраторов образуют иерархию. На самом деле они являются еще более иерархическими, чем имена пакетов. Если между пакетом и его предшественником нет никакой семантической связи, то регистратор и производные от него регистраторы обладают общими свойствами. Так, если в регистраторе `"com.mycompany"` задать определенный уровень протоколирования, то производный от него регистратор унаследует этот уровень.

Существует семь уровней протоколирования:

- SEVERE
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST

По умолчанию используются первые три уровня, остальные нужно задавать, вызывая метод `setLevel()` следующим образом:

```
logger.setLevel(Level.FINE);
```

В итоге будут регистрироваться все сообщения, начиная с уровня `FINE` и выше. Кроме того, для разрешения протоколирования на всех уровнях можно воспользоваться константой `Level.ALL`, а константой `Level.OFF` — для запрета протоколирования. Для всех уровней определены методы протоколирования, как показано в приведенном ниже примере.

```
logger.warning(message);  
logger.fine(message);
```

С другой стороны, можно воспользоваться методом `log()`, чтобы явно указать нужный уровень:

```
logger.log(Level.FINE, message);
```



СОВЕТ. По умолчанию регистрируются все записи, имеющие уровень **INFO** и выше. Следовательно, для отладочных сообщений, требующихся для диагностики программ, но совершенно не нужных пользователям, следует указывать уровни **CONFIG**, **FINE**, **FINER** и **FINEST**.



ВНИМАНИЕ! Если уровень протоколирования превышает величину **INFO**, следует изменить конфигурацию обработчика протоколов. По умолчанию обработчик протоколов блокирует сообщения, имеющие уровень ниже **INFO**.

Протокольная запись, создаваемая по умолчанию, состоит из имени класса и метода, содержащего вызов регистратора. Но если виртуальная машина оптимизирует процесс исполнения кода, то точные сведения о вызываемых методах могут стать недоступными. Для уточнения вызывающего класса и метода следует применить метод `logp()` следующим образом:

```
void logp(Level l, String className, String methodName,  
          String message)
```

Для отслеживания порядка выполнения диагностируемой программы имеются следующие удобные методы:

```
void entering(String className, String methodName)  
void entering(String className, String methodName,  
              Object param)  
void entering(String className, String methodName,  
              Object[] params)  
void exiting(String className, String methodName)
```

```
void exiting(String className, String methodName,
             Object result)
```

Ниже приведен пример применения этих методов непосредственно в коде. При их вызове формируются протокольные записи, имеющие уровень FINER и начинающиеся с символьных строк ENTRY и RETURN.

```
int read(String file, String pattern)
{
    logger.entering("com.mycompany.mylib.Reader", "read",
                   new Object[] { file, pattern });
    . . .
    logger.exiting("com.mycompany.mylib.Reader",
                  "read", count);
    return count;
}
```



НА ЗАМЕТКУ! В будущем методы протоколирования будут поддерживать переменное число параметров. Тогда появится возможность делать вызовы вроде следующего: `logger.entering("com.mycompany.mylib.Reader", "read", file, pattern)`.

Обычно протоколирование служит для записи неожиданных исключений. Для этого имеются два удобных метода, позволяющих ввести имя исключения в протокольную запись следующим образом:

```
void throwing(String className, String methodName,
              Throwable t)
void log(Level l, String message, Throwable t)
```

Ниже приведены типичные примеры применения этих методов в коде. При вызове метода `throwing()` регистрируется протокольная запись, имеющая уровень FINER, а также сообщение, начинающееся со строки THROW.

```
if (. . .)
{
    IOException exception = new IOException(". . .");
    logger.throwing("com.mycompany.mylib.Reader",
                  "read", exception);
    throw exception;
}
и
try
{
    . . .
}
catch (IOException e)
{
    Logger.getLogger("com.mycompany.myapp").
        log(Level.WARNING, "Reading image", e);
}
```

7.5.3. Смена диспетчера протоколирования

Свойства системы протоколирования можно изменить, редактируя конфигурационный файл, который по умолчанию находится по следующему пути: `conf/logging.properties` (или по пути `jre/lib/logging.properties` до версии Java 9).

Если требуется сменить конфигурационный файл, при запуске приложения необходимо установить свойство `java.util.logging.config.file` следующим образом:

```
java -Djava.util.logging.config.file=  
конфигурационный_файл ГлавныйКласс
```

Чтобы изменить уровень протоколирования, принятый по умолчанию, необходимо отредактировать конфигурационный файл, изменив в нем следующую строку:

```
.level=INFO
```

А в собственных регистраторах уровни протоколирования задаются с помощью строк вроде приведенной ниже. Иными словами, необходимо добавить к имени регистратора суффикс `.level`.

```
com.mycompany.myapplication.level=FINE
```

Как станет ясно в дальнейшем, регистраторы на самом деле не направляют сообщения на консоль, поскольку это задача обработчиков протоколов. Для обработчиков протоколов также определены уровни. Чтобы вывести на консоль сообщения, имеющие уровень `FINE`, необходимо сделать следующую установку:

```
java.util.logging.ConsoleHandler.level=FINE
```



ВНИМАНИЕ! Параметры настройки диспетчера протоколирования не являются системными свойствами. Запуск прикладной программы с параметром `-Dcom.mycompany.myapplication.level=FINE` никак не отражается на действиях регистратора.

Диспетчер протоколирования инициализируется во время запуска виртуальной машины, но еще до выполнения метода `main()`. Если же требуется специальная настройка свойств протоколирования, но без запуска прикладной программы с параметром `-Djava.util.logging.config.file` из командной строки, в таком случае следует сделать вызов `System.setProperty("java.util.logging.config.file", file)` в данной программе. Но тогда придется также сделать вызов `LogManager.getLogManager().readConfiguration()`, чтобы повторно инициализировать диспетчер протоколирования.

Начиная с версии Java 9, вместо этого можно обновить конфигурацию протоколирования, сделав следующий вызов:

```
LogManager.getLogManager().updateConfiguration(mapper);
```

Новая конфигурация читается из того места, которое указано в системном свойстве `java.util.logging.config.file`. И тогда для определения значений по всем ключам в прежней или новой конфигурации применяется специальный сопоставитель типа `Function<String, BiFunction<String, String, String>>`. Он сопоставляет ключи из существующей конфигурации с заменяющими функциями. Каждая заменяющая функция принимает прежнее и новое значения, связанные с отдельным ключом (или пустое значение `null`, если соответствующее значение отсутствует), а также заменяет прежнее значение на новое или пустое значение `null`, если данный ключ должен быть опущен в обновленной конфигурации.

Чтобы стал более понятным описанный выше сложный механизм обновления конфигурации, рассмотрим пару примеров. В качестве удобной схемы сопоставления можно было бы объединить прежние и новые конфигурации, отдав предпочтение новому значению, когда ключ присутствует как в прежних, так и в новых конфигурациях. В таком случае сопоставитель можно выразить следующим образом:

```
key -> ((oldValue, newValue) -> newValue == null
      ? oldValue : newValue)
```

А если требуется лишь обновить ключи, начиная с `com.mycompany`, оставив остальные ключи без изменения, то сопоставитель можно выразить таким образом:

```
key -> key.startsWith("com.mycompany")
      ? ((oldValue, newValue) -> newValue)
      : ((oldValue, newValue) -> oldValue)
```

Имеется также возможность изменить уровни протоколирования в выполняемой программе с помощью утилиты **jconsole**. Подробнее об этом см. по адресу www.oracle.com/technetwork/articles/java/jconsole-1564139.html#LoggingControl.



НА ЗАМЕТКУ! Файл, содержащий свойства системы протоколирования, обрабатывается классом `java.util.logging.LogManager`. Задав имя подкласса в качестве системного свойства `java.util.logging.manager`, можно указать другой диспетчер протоколирования. Кроме того, можно оставить стандартный диспетчер протоколирования, пропустив инициализационные установки в файле свойств. В системном свойстве `java.util.logging.config.class` следует установить имя класса, в котором как-то иначе задаются свойства диспетчера протоколирования. Подробное описание класса `LogManager` см. в документации на прикладной интерфейс Java API.

7.5.4. Локализация

Нередко возникает потребность вывести протокольные сообщения на разных языках, чтобы они стали понятны пользователям из других стран. Интернационализация прикладных программ обсуждается в главе 7 второго тома настоящего издания. Ниже вкратце поясняется, что следует иметь в виду при локализации протокольных сообщений.

Данные, требующиеся для интернационализации приложения, содержатся в комплектах ресурсов. Комплект ресурсов представляет собой набор сопоставлений для различных региональных настроек (например, в США или Германии). Например, в комплекте ресурсов можно сопоставить символьную строку `"readingFile"` с текстовой строкой `"Reading file"` на английском языке или со строкой `"Achtung! Datei wird eingelesen"` на немецком языке.

В состав прикладной программы может входить несколько комплектов ресурсов, например, один — для меню, другой — для протокольных сообщений. У каждого комплекта ресурсов имеется свое имя (например, `"com.mycompany.logmessages"`). Для того чтобы ввести сопоставление в комплект ресурсов, следует предоставить файл для региональных настроек на соответствующем языке. Сопоставления сообщений на английском языке находятся в файле `com/mycompany/logmessages_en.properties`, а сопоставления сообщений на немецком языке — в файле `com/mycompany/logmessages_de.properties`. (Здесь пары символов `en` и `de` обозначают стандартные коды языков.) Объединение файлов осуществляется с помощью классов прикладной программы. В частности, класс `ResourceBundle` обнаруживает их автоматически. Содержимое этих файлов состоит из записей простым текстом, подобных приведенным ниже.

```
readingFile=Achtung! Datei wird eingelesen
renamingFile=Datei wird umbenannt
...
```

При вызове регистратора сначала указывается конкретный комплект ресурсов:

```
Logger logger =
    Logger.getLogger(loggerName,
        "com.mycompany.logmessages");
```

Затем для составления протокольного сообщения указывается ключ из комплекта ресурсов, но не строка самого сообщения:

```
logger.info("readingFile");
```

Нередко возникает потребность включать какие-нибудь аргументы в локализованные сообщения. В таком случае сообщение должно иметь заполнители {0}, {1} и т.д. Допустим, в протокольное сообщение требуется ввести имя файла. С этой целью заполнитель используется следующим образом:

```
Reading file {0}.  
Achtung! Datei {0} wird eingelesen.
```

Заполнители заменяются соответствующими значениями при вызове одного из следующих методов:

```
logger.log(Level.INFO, "readingFile", fileName);  
logger.log(Level.INFO, "renamingFile", new Object[]  
    { oldName, newName });
```

А начиная с версии Java 9, в методе `logrb()` можно указать объект комплекта ресурсов, а не его имя:

```
logger.logrb(Level.INFO, bundle, "renamingFile",  
    oldName, newName);
```



НА ЗАМЕТКУ! Это единственный метод протоколирования, где допускается употреблять переменные аргументы, задаваемые для параметров сообщения.

7.5.5. Обработчики протоколов

По умолчанию регистраторы посылают протокольные записи объекту класса `ConsoleHandler`, который выводит их в поток сообщений об ошибках `System.err`. Точнее говоря, регистраторы посылают протокольные записи родительскому обработчику протоколов, а у его окончательного предшественника (под именем `"`) имеется объект типа `ConsoleHandler`.

Подобно регистраторам, у обработчиков протоколов имеются свои уровни. Уровень протоколирования записи должен превышать порог как для регистратора, так и для обработчика. Уровень консольного обработчика по умолчанию задается в файле, содержащем параметры настройки диспетчера протоколирования, как показано ниже.

```
java.util.logging.ConsoleHandler.level=INFO
```

Для регистрации записи, имеющей уровень `FINE`, в конфигурационном файле следует изменить исходный уровень протоколирования как регистратора, так и обработчика. С другой стороны, можно вообще пренебречь файлом конфигурации и установить свой собственный обработчик протоколов следующим образом:

```
Logger logger = Logger.getLogger("com.mycompany.myapp");  
logger.setLevel(Level.FINE);  
logger.setUseParentHandlers(false);  
var handler = new ConsoleHandler();  
handler.setLevel(Level.FINE);  
logger.addHandler(handler);
```

По умолчанию регистратор посылает протокольные записи как своим, так и родительским обработчикам. Рассматриваемый здесь регистратор наследует от изначального регистратора (под именем `"`), посылающего на консоль все протокольные записи уровня `INFO` и выше. Но вряд ли нужно, чтобы все протокольные записи

выводились дважды, поэтому в свойстве `useParentHandlers` следует установить логическое значение `false`.

Чтобы послать протокольную запись еще куда-нибудь, следует ввести другой обработчик протоколов. Для этого в прикладном интерфейсе API для протоколирования предоставляются два класса: `FileHandler` и `SocketHandler`. Обработчик типа `SocketHandler` посылает протокольные записи на указанный хост и в порт. Наибольший интерес представляет обработчик типа `FileHandler`, выводящий протокольные записи в файл. Эти записи можно просто передать обработчику типа `FileHandler`, устанавливаемому по умолчанию:

```
var handler = new FileHandler();
logger.addHandler(handler);
```

Протокольные записи выводятся в файл `java.n.log`, который находится в начальном каталоге пользователя, где `n` — однозначный номер, отличающий этот файл от других аналогичных файлов. Если на платформе начальные каталоги пользователей не поддерживаются (например, в Windows 95/98/ME), файл записывается в каталог, предусмотренный по умолчанию, например `C:\Windows`. По умолчанию протокольные записи хранятся в формате XML. Обычная протокольная запись имеет следующий вид:

```
<record>
  <date>2002-02-04T07:45:15</date>
  <millis>1012837515710</millis>
  <sequence>1</sequence>
  <logger>com.mycompany.myapp</logger>
  <level>INFO</level>
  <class>com.mycompany.mylib.Reader</class>
  <method>read</method>
  <thread>10</thread>
  <message>Reading file corejava.gif</message>
</record>
```

Поведение исходного обработчика протоколов типа `FileHandler` можно изменить, задав другие параметры настройки в диспетчере протоколирования (табл. 7.1) или воспользовавшись другим конструктором (см. краткое описание прикладного интерфейса API для протоколирования в конце этого раздела). Как правило, имя файла протокола, предлагаемое по умолчанию, не используется. Следовательно, для него нужно задать другой шаблон, например `%h/myapp.log` (переменные шаблона описаны в табл. 7.2).

Таблица 7.1. Параметры настройки обработчика протоколов типа `FileHandler`

Настраиваемое свойство	Описание	Значение по умолчанию
<code>java.util.logging.FileHandler.level</code>	Уровень обработчика	Level.ALL
<code>java.util.logging.FileHandler.append</code>	Определяет, должен ли обработчик добавлять записи в существующий файл или же открывать новый файл при очередном запуске программы	false
<code>java.util.logging.FileHandler.limit</code>	Приблизительная оценка максимального размера файла. При превышении этого размера открывается новый файл (0 — размер файла не ограничен)	0 (т.е. без ограничений) в классе FileHandler ; 50000 в исходной конфигурации диспетчера протоколирования

Окончание табл. 7.1

Настраиваемое свойство	Описание	Значение по умолчанию
<code>java.util.logging. FileHandler.pattern</code>	Шаблон имени файла протокола (см. табл. 7.2)	<code>%h/java%u.log</code>
<code>java.util.logging. FileHandler.count</code>	Количество файлов протокола, участвующих в ротации	1 (ротация не производится)
<code>java.util.logging. FileHandler.filter</code>	Класс, используемый для фильтрации	Фильтрация отсутствует
<code>java.util.logging. FileHandler.encoding</code>	Применяемая кодировка	Кодировка, принятая на текущей платформе
<code>java.util.logging. FileHandler.formatter</code>	Средство форматирования протокольных записей	<code>java.util.logging. XMLFormatter</code>

Таблица 7.2. Переменные шаблона для имени файла протокола

Переменная	Описание
<code>%h</code>	Значение системного свойства <code>user.home</code>
<code>%t</code>	Временный системный каталог
<code>%u</code>	Однозначный номер, позволяющий избежать конфликта имен
<code>%g</code>	Определяет режим формирования номеров для подвергаемых ротации файлов протоколов. (Если ротация производится, а в шаблоне отсутствует переменная <code>%g</code> , то используется суффикс <code>.%g</code> .)
<code>%%</code>	Знак процента

Если в нескольких приложениях (или нескольких копиях одного и того же приложения) используется тот же самый файл протокола, следует установить признак `append`, определяющий режим ввода данных в конце файла. Кроме того, можно воспользоваться шаблоном имени файла `%u` таким образом, чтобы каждое приложение создавало свою особую копию файла протокола.

Целесообразно также установить режим ротации файлов протоколов. В этом случае имена файлов протоколов формируются следующим образом: `myapp.log.0`, `myapp.log.1`, `myapp.log.2` и т.д. Как только размер файла превысит допустимый предел, самый старый файл протокола удаляется, остальные переименовываются, а новый файл получает имя с номером 0.



СОВЕТ. Многие программисты пользуются средствами протоколирования для упрощения технической поддержки своих прикладных программ. Если в какой-то момент программа начинает работать некорректно, пользователь может обратиться к файлу протокола. Чтобы это стало возможным, следует установить признак `append`, организовать ротацию протоколов или сделать и то и другое.

Имеется также возможность определить свои собственные обработчики протоколов, расширив класс `Handler` или `StreamHandler`. Такой обработчик будет определен в программе, рассматриваемой далее в качестве примера. Этот обработчик отображает протокольные записи в окне (рис. 7.2).

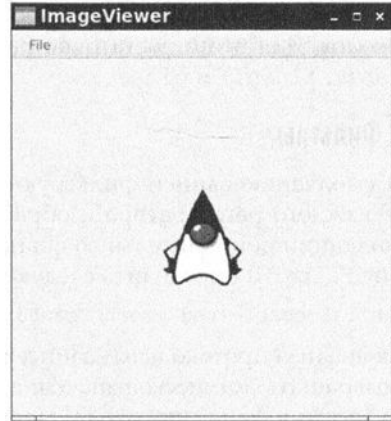
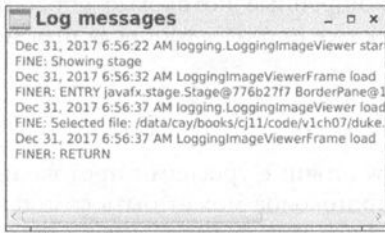


Рис. 7.2. Обработчик, отображающий протокольные записи в окне

Данный обработчик протоколов расширяет класс `StreamHandler` и устанавливает поток вывода с методами `write()` для отображения в текстовой области данных, выводимых в этот поток:

```
class WindowHandler extends StreamHandler
{
    public WindowHandler()
    {
        . . .
        var output = new JTextArea();
        setOutputStream(new
            OutputStream()
            {
                public void write(int b) {} // не вызывается!
                public void write(byte[] b, int off, int len)
                {
                    output.append(new String(b, off, len));
                }
            });
        . . .
    }
}
```

При таком подходе возникает лишь одно затруднение: обработчик размещает протокольные записи в буфере и направляет их в поток вывода только тогда, когда буфер заполнен. Следовательно, необходимо переопределить метод `publish()`, чтобы выводить содержимое буфера после каждой протокольной записи. Это делается следующим образом:

```
class WindowHandler extends StreamHandler
{
    . . .
    public void publish(LogRecord record)
    {
        super.publish(record);
        flush();
    }
}
```


По желанию можно написать и более изощренные потоковые обработчики протоколов. Для этого достаточно расширить класс `Handler` и определить методы `publish()`, `flush()` и `close()`.

7.5.6. Фильтры

По умолчанию записи фильтруются в соответствии с уровнями протоколирования. У каждого регистратора и обработчика протоколов может быть свой фильтр, выполняющий дополнительную фильтрацию. Для этого достаточно реализовать интерфейс `Filter` и определить следующий метод:

```
boolean isLoggable(LogRecord record)
```

Для анализа протокольных записей можно выбрать любой критерий, а метод должен возвращать логическое значение `true` для тех протокольных записей, которые нужно ввести в файл протокола. Например, фильтр может пропускать только протокольные записи, созданные в начале выполнения метода и при возврате из него. Фильтр должен вызвать метод `record.getMessage()` и проверить, начинается ли запись со строки `ENTRY` или `RETURN`.

Чтобы задать фильтр в регистраторе или обработчике протоколов, следует вызвать метод `setFilter()`. Но фильтры можно применять только по очереди.

7.5.7. Средства форматирования

Классы `ConsoleHandler` и `FileHandler` порождают протокольные записи в текстовом виде и в формате XML. Но для них можно определить свой собственный формат. Для этого необходимо расширить класс `Formatter` и переопределить следующий метод:

```
String format(LogRecord record)
```

Далее протокольная запись форматируется избранным способом и возвращается получаемая в итоге символьная строка. В методе `format()` может потребоваться вызов приведенного ниже метода. Этот метод форматирует сообщение, составляющее часть протокольной записи, подставляя параметры и выполняя интернационализацию.

```
String formatMessage(LogRecord record)
```

Во многих форматах файлов (например, XML) предполагается существование начальной и конечной частей файла, заключающих в себе отформатированные протокольные записи. В таком случае нужно переопределить следующие методы:

```
String getHead(Handler h)
String getTail(Handler h)
```

И, наконец, вызывается метод `setFormatter()`, устанавливающий средства форматирования в обработчике протоколов. Имея столько возможностей для протоколирования, в них легко запутаться. Поэтому ниже приведены краткие рекомендации по выполнению основных операций протоколирования.

7.5.8. “Рецепт” протоколирования

Ниже приведен “рецепт” организации протоколирования, в котором сведены наиболее употребительные операции.

1. Для простых приложений выбирайте один регистратор. Желательно, чтобы имя регистратора совпадало с именем основного пакета приложения, например `com.mycompany.myprog`. Создать регистратор можно, сделав следующий вызов:
`Logger logger = Logger.getLogger("com.mycompany.myprog");`

2. Для удобства в те классы, где интенсивно используется протоколирование, можно добавить статические поля:

```
private static final Logger logger =  
    Logger.getLogger("com.mycompany.myprog");
```

3. По умолчанию все сообщения, имеющие уровень `INFO` и выше, выводятся на консоль. Пользователи могут изменить конфигурацию, предусмотренную по умолчанию, но, как пояснялось ранее, это довольно сложный процесс. Следовательно, лучше задать более оправданные настройки прикладной программы по умолчанию. Приведенный ниже код гарантирует, что все сообщения будут зарегистрированы в файле протокола, связанном с конкретным приложением. Введите этот код в тело метода `main()` своего приложения.

```
if (System.getProperty(  
    "java.util.logging.config.class") == null  
    && System.getProperty(  
        "java.util.logging.config.file") == null)  
{  
    try  
    {  
        Logger.getLogger("").setLevel(Level.ALL);  
        final int LOG_ROTATION_COUNT = 10;  
        var handler = new FileHandler("%h/myapp.log",  
                                       0, LOG_ROTATION_COUNT);  
        Logger.getLogger("").addHandler(handler);  
    }  
    catch (IOException e)  
    {  
        logger.log(Level.SEVERE, "Can't create log file handler", e);  
    }  
}
```

4. Теперь все готово для протоколирования. Помните, что все сообщения, имеющие уровень протоколирования `INFO`, `WARNING` и `SEVERE`, выводятся на консоль. Следовательно, эти уровни протоколирования нужно зарезервировать для сообщений, представляющих ценность для пользователей вашей программы. Уровень `FINE` лучше выделить для сообщений, предназначенных для программистов. В тех местах кода, где вы обычно вызывали метод `System.out.println()`, регистрируйте сообщения следующим образом:

```
logger.fine("File open dialog canceled");
```

5. Рекомендуется также регистрировать неожиданные исключения, например, так, как показано ниже.

```
try  
{  
    . . .  
}
```

```
catch (SomeException e)
{
    logger.log(Level.FINE, "explanation", e);
}
```

В листинге 7.2 приведен исходный код программы, составленной по описанному выше “рецепту” протоколирования, но с одним существенным дополнением: сообщения не только регистрируются, но и отображаются в протокольном окне.

Листинг 7.2. Исходный код из файла `logging/LoggingImageViewer.java`

```
1 package logging;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.io.*;
6 import java.util.logging.*;
7 import javax.swing.*;
8
9 /**
10  * Это видоизмененный вариант программы просмотра,
11  * в которой регистрируются различные события
12  * @version 1.03 2015-08-20
13  * @author Cay Horstmann
14  */
15 public class LoggingImageViewer
16 {
17     public static void main(String[] args)
18     {
19         if (System.getProperty(
20             "java.util.logging.config.class") == null
21             && System.getProperty(
22                 "java.util.logging.config.file") == null)
23         {
24             try
25             {
26                 Logger.getLogger("com.horstmann.corejava")
27                     .setLevel(Level.ALL);
28                 final int LOG_ROTATION_COUNT = 10;
29                 var handler = new FileHandler(
30                     "%h/LoggingImageViewer.log",
31                     0, LOG_ROTATION_COUNT);
32                 Logger.getLogger("com.horstmann.corejava")
33                     .addHandler(handler);
34             }
35             catch (IOException e)
36             {
37                 Logger.getLogger("com.horstmann.corejava")
38                     .log(Level.SEVERE,
39                         "Can't create log file handler", e);
40             }
41         }
42
43         EventQueue.invokeLater(() ->
44         {
45             Handler windowHandler = new WindowHandler();
46             windowHandler.setLevel(Level.ALL);
```

```
47     Logger.getLogger("com.horstmann.corejava")
48         .addHandler(windowHandler);
49
50     JFrame frame = new ImageViewerFrame();
51     frame.setTitle("LoggingImageViewer");
52     frame.setDefaultCloseOperation(
53         JFrame.EXIT_ON_CLOSE);
54
55     Logger.getLogger("com.horstmann.corejava")
56         .fine("Showing frame");
57     frame.setVisible(true);
58 });
59 }
60 }
61
62 /**
63  * Фрейм, в котором показывается изображение
64  */
65 class ImageViewerFrame extends JFrame
66 {
67     private static final int DEFAULT_WIDTH = 300;
68     private static final int DEFAULT_HEIGHT = 400;
69
70     private JLabel label;
71     private static Logger logger =
72         Logger.getLogger("com.horstmann.corejava");
73
74     public ImageViewerFrame()
75     {
76         logger.entering("ImageViewerFrame", "<init>");
77         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
78
79         // установить строку меню
80         JMenuBar menuBar = new JMenuBar();
81         setJMenuBar(menuBar);
82
83         JMenu menu = new JMenu("File");
84         menuBar.add(menu);
85
86         JMenuItem openItem = new JMenuItem("Open");
87         menu.add(openItem);
88         openItem.addActionListener(new FileOpenListener());
89
90         JMenuItem exitItem = new JMenuItem("Exit");
91         menu.add(exitItem);
92         exitItem.addActionListener(new ActionListener()
93         {
94             public void actionPerformed(ActionEvent event)
95             {
96                 logger.fine("Exiting.");
97                 System.exit(0);
98             }
99         });
100
101         // использовать метку для обозначения изображений
102         label = new JLabel();
103         add(label);
104         logger.exiting("ImageViewerFrame", "<init>");
```

```
105 }
106
107 private class FileOpenListener
108     implements ActionListener
109 {
110     public void actionPerformed(ActionEvent event)
111     {
112         logger.entering(
113             "ImageViewerFrame.FileOpenListener",
114             "actionPerformed", event);
115
116         // установить селектор файлов
117         JFileChooser chooser = new JFileChooser();
118         chooser.setCurrentDirectory(new File("."));
119
120         // принять все файлы с расширением .gif
121         chooser.setFileFilter(
122             new javax.swing.filechooser.FileFilter()
123             {
124                 public boolean accept(File f)
125                 {
126                     return f.getName().toLowerCase()
127                         .endsWith(".gif") || f.isDirectory();
128                 }
129
130                 public String getDescription()
131                 {
132                     return "GIF Images";
133                 }
134             }
135         );
136
137         // показать диалоговое окно селектора файлов
138         int r = chooser.showOpenDialog(
139             ImageViewerFrame.this);
140
141         // если файл изображения подходит, выбрать
142         // его в качестве пиктограммы для метки
143         if (r == JFileChooser.APPROVE_OPTION)
144         {
145             String name = chooser.getSelectedFile()
146                 .getPath();
147             logger.log(Level.FINE, "Reading file {0}",
148                 name);
149             label.setIcon(new ImageIcon(name));
150         }
151         else logger.fine("File open dialog canceled.");
152         logger.exiting("ImageViewerFrame
153             .FileOpenListener", "actionPerformed");
154     }
155 }
156 }
157
158 /**
159  * Обработчик для отображения протокольных
160  * записей в окне
161  */
162 class WindowHandler extends StreamHandler
```

```
163 {
164     private JFrame frame;
165
166     public WindowHandler()
167     {
168         frame = new JFrame();
169         final JTextArea output = new JTextArea();
170         output.setEditable(false);
171         frame.setSize(200, 200);
172         frame.add(new JScrollPane(output));
173         frame.setFocusableWindowState(false);
174         frame.setVisible(true);
175         setOutputStream(new OutputStream()
176         {
177             public void write(int b)
178             {
179                 } // не вызывается!
180
181             public void write(byte[] b, int off, int len)
182             {
183                 output.append(new String(b, off, len));
184             }
185         });
186     }
187
188     public void publish(LogRecord record)
189     {
190         if (!frame.isVisible()) return;
191         super.publish(record);
192         flush();
193     }
194 }
```

java.util.logging.Logger 1.4

- **Logger getLogger(String loggerName)**
- **Logger getLogger(String loggerName, String bundleName)**

Возвращают регистратор протоколов с указанным именем. Если регистратор не существует, он создается.

- **void severe(String message)**
- **void warning(String message)**
- **void info(String message)**
- **void config(String message)**
- **void fine(String message)**
- **void finer(String message)**
- **void finest(String message)**

Протоколируют запись на уровне, соответствующем имени метода, вместе с заданным сообщением *message*.

java.util.logging.Logger 1.4 (продолжение)

- **void entering(String className, String methodName)**
- **void entering(String className, String methodName, Object param)**
- **void entering(String className, String methodName, Object[] param)**
- **void exiting(String className, String methodName)**
- **void exiting(String className, String methodName, Object result)**

Протоколируют запись, описывающую вход и выход из метода с заданными параметрами или возвращаемым значением.

- **void throwing(String className, String methodName, Throwable t)**
- Протоколирует запись, описывающую процесс генерирования объекта исключения.
- **void log(Level level, String message)**
- **void log(Level level, String message, Object obj)**
- **void log(Level level, String message, Object[] objs)**
- **void log(Level level, String message, Throwable t)**

Протоколируют запись на заданном уровне вместе с указанным сообщением. Запись может содержать обычные объекты или объект исключения. Для этой цели в сообщении предусмотрены заполнители {0}, {1} и т.д.

- **void logp(Level level, String className, String methodName, String message)**
- **void logp(Level level, String className, String methodName, String message, Object obj)**
- **void logp(Level level, String className, String methodName, String message, Object[] objs)**
- **void logp(Level level, String className, String methodName, String message, Throwable t)**

- Протоколируют запись на заданном уровне с данными о вызове и указанным сообщением. Запись может включать обычные объекты или объекты исключений.

- **void logrb(Level level, String className, String methodName, ResourceBundle bundle, String message, Object... params) 9**
- **void logrb(Level level, String className, String methodName, ResourceBundle bundle, String message, Throwable thrown) 9**

Протоколируют запись на заданном уровне с данными о вызове, указанным сообщением и именем комплекта ресурсов. Запись может включать обычные объекты или объекты исключений.

- **Level getLevel()**
- **void setLevel(Level l)**

Получают и устанавливают уровень данного регистратора.

- **Logger getParent()**
- **void setParent(Logger l)**

Получают и устанавливают родительский регистратор.

- **Handler[] getHandlers()**
- Получает все обработчики протоколов для данного регистратора.

- **void addHandler(Handler h)**
- **void removeHandler(Handler h)**

Добавляют и удаляют обработчик протоколов для данного регистратора.

java.util.logging.Logger 1.4 (окончание)

- **boolean getUseParentHandlers()**
- **void setUseParentHandlers(boolean b)**
Получают и устанавливают свойство, определяющее режим использования родительского обработчика протоколов. Если это свойство принимает логическое значение **true**, регистратор направляет все протоколируемые записи обработчикам своего родительского регистратора.
- **Filter getFilter()**
- **void setFilter(Filter f)**
Получают и устанавливают фильтр для данного регистратора.

java.util.logging.Handler 1.4

- **abstract void publish(LogRecord record)**
Посылает протокольную запись по указанному адресу.
- **abstract void flush()**
Выводит данные из буфера.
- **abstract void close()**
Выводит данные из буфера и освобождает все ресурсы.
- **Filter getFilter()**
- **void setFilter(Filter f)**
Получает и устанавливает фильтр для данного обработчика протоколов.
- **Formatter getFormatter()**
- **void setFormatter(Formatter f)**
Получают и устанавливают средство форматирования для данного обработчика протоколов.
- **Level getLevel()**
- **void setLevel(Level l)**
Получают и устанавливают уровень данного обработчика протоколов.

java.util.logging.ConsoleHandler 1.4

- **ConsoleHandler()**
Создает новый консольный обработчик протоколов.

java.util.logging.FileHandler 1.4

- **FileHandler(String pattern)**
- **FileHandler(String pattern, boolean append)**
- **FileHandler(String pattern, int limit, int count)**

java.util.logging.FileHandler 1.4 (окончание)

- **FileHandler(String pattern, int limit, int count, boolean append)**
 - **FileHandler(String pattern, long limit, int count, boolean append)** 9
- Создают файловый обработчик протоколов. Формат шаблонов приведен в табл. 7.2. Параметр **limit** определяет максимальное количество байтов перед открытием файла протоколирования, а параметр **count** — количество файлов в ротационной последовательности. Если параметр **append** принимает логическое значение **true**, протокольные записи должны присоединяться к существующему файлу протоколирования.

java.util.logging.LogRecord 1.4

- **Level getLevel()**
Получает уровень протоколирования для данной записи.
- **String getLoggerName()**
Получает имя регистратора, создавшего данную запись.
- **ResourceBundle getResourceBundle()**
- **String getResourceBundleName()**
Получает комплект ресурсов, предназначенный для интернационализации сообщения, или его имя. Если комплект ресурсов не предусмотрен, возвращается пустое значение **null**.
- **String getMessage()**
Получает исходное сообщение, не прошедшее интернационализацию и форматирование.
- **Object[] getParameters()**
Получает параметры или пустое значение **null**, если они отсутствуют.
- **Throwable getThrown()**
Получает объект сгенерированного исключения или пустое значение **null**, если объект не существует.
- **String getSourceClassName()**
- **String getSourceMethodName()**
Определяют местоположение кода, зарегистрировавшего данную запись. Эти сведения могут быть предоставлены кодом регистрации или автоматически выведены, исходя из анализа стека выполнения программы. Если код регистрации содержит неверное значение или программа была оптимизирована таким образом, что определить местоположение кода невозможно, эти сведения могут оказаться неточными.
- **long getMillis()**
Определяет время создания данной протокольной записи в миллисекундах, отсчитывая его от даты 1 января 1970 г.
- **Instant getInstant()** 9
Получает время создания в виде объекта типа **java.time.Instant** (см. главу 6 второго тома настоящего издания).
- **long getSequenceNumber()**
Получает однозначный порядковый номер записи.
- **int getThreadID()**
Получает однозначный идентификатор потока, в котором была создана данная протокольная запись. Такие идентификаторы присваиваются классом **LogRecord** и не имеют отношения к другим идентификаторам потоков.

java.util.logging.LogManager 1.4

- **static LogManager getLogger()**
Получает глобальный экземпляр типа **LogManager**.
- **void readConfiguration()**
- **void readConfiguration(InputStream in)**
Читают конфигурацию протоколирования из файла, указанного в системном свойстве **java.util.logging.config.file**, или из заданного потока ввода.
- **void updateConfiguration(InputStream in, Function<String, BiFunction<String, String, String>> mapper)** 9
- **void updateConfiguration(Function<String, BiFunction<String, String, String>> mapper)** 9
Объединяют конфигурацию протоколирования с содержимым файла, указанного в системном свойстве **java.util.logging.config.file**, или с заданным потоком ввода. Описание параметра **mapper** см. в разделе 7.5.3.

java.util.logging.Filter 1.4

- **boolean isLoggable(LogRecord record)**
Возвращает логическое значение **true**, если указанная запись должна быть зарегистрирована.

java.util.logging.Formatter 1.4

- **abstract String format(LogRecord record)**
- Возвращает символьную строку, полученную в результате форматирования заданной протокольной записи.
- **String getHead(Handler h)**
- **String getTail(Handler h)**
Возвращают символьные строки, которые должны появиться в начале и в конце документа, содержащего данную протокольную запись. Эти методы определены в суперклассе **Formatter** и по умолчанию возвращают пустую символьную строку. При необходимости их следует переопределить.
- **String formatMessage(LogRecord record)**
Возвращает интернационализированное и форматированное сообщение, являющееся частью протокольной записи.

7.6. Рекомендации по отладке программ

Допустим, вы написали программу и оснастили ее средствами для перехвата и соответствующей обработки исключений. Затем вы запускаете ее на выполнение, и она работает неправильно. Что же делать дальше? (Если у вас никогда не возникал такой вопрос, можете не читать оставшуюся часть этой главы.)

Разумеется, лучше всего было бы иметь под рукой удобный и эффективный отладчик. Такие отладчики являются частью профессиональных IDE, например Eclipse

или NetBeans. В завершение этой главы дадим несколько рекомендаций, которые стоит взять на вооружение, прежде чем запускать отладчик.

1. Значение любой переменной можно вывести с помощью выражения

```
System.out.println("x=" + x);
```

или

```
Logger.getGlobal().info("x=" + x);
```

Если в переменной *x* содержится числовое значение, оно преобразуется в символьную строку. Если же переменная *x* ссылается на объект, то вызывается метод `toString()`. Состояние объекта, используемого в качестве неявного параметра, поможет определить следующий вызов:

```
Logger.getGlobal().info("this=" + this);
```

В большинстве классов Java метод `toString()` переопределяется таким образом, чтобы предоставить пользователю как можно больше полезной информации. Это очень удобно для отладки. Об этом следует позаботиться и в своих классах.

2. Один малоизвестный, но очень полезный прием состоит в том, чтобы включить в каждый класс отдельный метод `main()` и разместить в нем код, позволяющий протестировать этот класс отдельно от других, как показано ниже.

```
public class MyClass
{
    методы и поля данного класса
    . . .
    public static void main(String[] args)
    {
        тестовый код
    }
}
```

Создайте несколько объектов, вызовите все методы и проверьте, правильно ли они действуют. Все это делается в теле методов `main()`, а интерпретатор вызывается для каждого файла класса по отдельности. Если выполняется апплет, то ни один из этих методов `main()` вообще не будет вызван. А если выполняется приложение, то интерпретатор вызовет только метод `main()` из запускающего класса.

3. Если предыдущая рекомендация пришлась вам по душе, попробуйте поработать в среде JUnit, которая доступна по адресу <http://junit.org>. JUnit представляет собой широко распространенную среду модульного тестирования, которая существенно упрощает работу по созданию наборов тестов и контрольных примеров. Запускайте тесты после каждого видоизменения класса. При обнаружении программных ошибок добавляйте новый контрольный пример для последующего тестирования.
4. *Протоколирующий прокси-объект* представляет собой экземпляр подкласса, который перехватывает вызовы методов, протоколирует их и обращается к суперклассу. Так, если возникнут трудности при вызове метода `nextDouble()` из класса `Random`, можно создать прокси-объект в виде экземпляра анонимного подкласса следующим образом:

```
var generator = new Random()
{
```

```
public double nextDouble()
{
    double result = super.nextDouble();
    Logger.getGlobal().info("nextDouble: " + result);
    return result;
}
};
```

При каждом вызове метода `nextDouble()` будет формироваться протокольное сообщение. Ниже поясняется, как выявить ту часть кода, из которой вызывается данный метод, и как произвести трассировку стека.

5. Вызвав метод `printStackTrace()` из класса `Throwable`, можно получить трассировку стека из любого объекта исключения. В приведенном ниже фрагменте кода перехватывается любое исключение, выводится объект исключения и трассировка стека, а затем повторно генерируется исключение, чтобы найти предназначенный для него обработчик.

```
try
{
    . . .
}
catch (Throwable t)
{
    t.printStackTrace();
    throw t;
}
```

Для трассировки стека не нужно даже перехватывать исключение. Просто введите следующую строку в любом месте кода:

```
Thread.dumpStack();
```

6. Как правило, результаты трассировки стека выводятся в поток сообщений об ошибках `System.err`. Если же требуется протоколировать или отобразить результаты трассировки стека, то ниже показано, как заключить их в символьную строку.

```
var out = new ByteArrayOutputStream();
new Throwable().printStackTrace(out);
String description = out.toString();
```

7. Ошибки, возникающие при выполнении программы, удобно записывать в файл. Но обычно ошибки посылаются в поток сообщений об ошибках `System.err`, а не в поток вывода `System.out`. Поэтому ошибки нельзя вывести в файл по обычной команде следующим образом:

```
java MyProgram > errors.txt
```

Ошибки лучше перехватывать по такой команде:

```
java MyProgram 2> errors.txt
```

А для того чтобы направить потоки `Stream.err` и `Stream.out` в один и тот же файл, воспользуйтесь командой

```
java MyProgram 1> errors.txt 2>&1
```

Эта команда действует как в командной оболочке `bash`, так и в `Windows`.

8. Результаты трассировки стека необрабатываемых исключений свидетельствуют о том, что поток сообщений об ошибках `System.err` далек от идеала. Выводимые в него сообщения смущают конечных пользователей, если им случается их увидеть, и они недоступны для диагностических целей, когда в этом возникает потребность. Поэтому более правильный подход состоит в том, чтобы протоколировать их в файл. Обработчик для необрабатываемых исключений можно заменить статическим методом `Thread.setDefaultUncaughtExceptionHandler()` следующим образом:

```
Thread.setDefaultUncaughtExceptionHandler(  
    new Thread.UncaughtExceptionHandler()  
    {  
        public void uncaughtException(Thread t, Throwable e)  
        {  
            сохранить данные в файле протокола  
        };  
    });
```

9. Чтобы отследить загрузку класса, запустите виртуальную машину Java с параметром **-verbose**. На экране появятся строки, подобные следующим:

```
[0.012s][info][class,load] opened: /opt/jdk-9.0.1  
/lib/modules  
[0.034s][info][class,load] java.lang.Object  
source: jrt: /java.base  
[0.035s][info][class,load] java.io.Serializable  
source: jrt:/java.base  
[0.035s][info][class,load] java.lang.Comparable  
source: jrt:/java.base  
[0.035s][info][class,load] java.lang.CharSequence  
source: jrt:/java.base  
[0.035s][info][class,load] java.lang.String  
source: jrt:/java.base  
[0.036s][info][class,load] java.lang.reflect.AnnotatedElement  
source: jrt:/java.base  
[0.036s][info][class,load]  
java.lang.reflect.GenericDeclaration  
source: jrt:/java.base  
[0.036s][info][class,load] java.lang.reflect.Type  
source: jrt:/java.base  
[0.036s][info][class,load] java.lang.Class  
source: jrt:/java.base  
[0.036s][info][class,load] java.lang.Cloneable  
source: jrt:/java.base  
[0.037s][info][class,load] java.lang.ClassLoader  
source: jrt:/java.base  
[0.037s][info][class,load] java.lang.System  
source: jrt:/java.base  
[0.037s][info][class,load] java.lang.Throwable  
source: jrt:/java.base  
[0.037s][info][class,load] java.lang.Error  
source: jrt:/java.base  
[0.037s][info][class,load] java.lang.ThreadDeath  
source: jrt:/java.base  
[0.037s][info][class,load] java.lang.Exception source:  
jrt:/java.base
```

```
[0.037s][info][class,load] java.lang.RuntimeException
source: jrt:/java.base
[0.038s][info][class,load] java.lang.SecurityManager
source: jrt:/java.base
. . .
```

Такой способ может оказаться полезным для диагностики ошибок, связанных с путями к классам.

10. Параметр **-Xlint** указывает компилятору выявлять типичные ошибки в исходном коде программы. Если, например, вызвать компилятор так, как показано ниже, он уведомит о пропущенных операторах `break` в ветвях оператора `switch`.

```
javac -Xlint:fallthrough
```

Термин *lint* применялся ранее для описания инструментального средства, предназначенного для обнаружения потенциальных ошибок в программах, написанных на С. Теперь он обозначает все инструментальные средства, отмечающие языковые конструкции, которые могут оказаться спорными, хотя и допустимыми. Ниже описаны допустимые значения параметра **-Xlint**.

В итоге получаются сообщения, аналогичные следующему:

```
warning: [fallthrough] possible fall-through into case1
```

Символьная строка в квадратных скобках обозначает категорию предупреждения. Вывод каждой категории можно разрешить или запретить. Но поскольку большинство этих категорий весьма полезны для отладки программ, то лучше оставить их на месте, запретив лишь те из них, которые не представляют интерес, как показано ниже.

```
javac -Xlint:all,-fallthrough,-serial sourceFiles
```

Список предупреждений можно получить по следующей команде:

```
javac --help -X
```

11. В виртуальной машине Java реализована поддержка *контроля и управления* приложениями, написанными на Java. Это позволяет установить в ней агенты, которые будут отслеживать расходование памяти, использование потоков исполнения, загрузку классов и т.п. Такая поддержка важна для работы с крупномасштабными прикладными программами, работающими в течение длительного времени, например, с серверами приложений. Для демонстрации новых возможностей в комплект поставки JDK включена графическая утилита **jconsole**, которая отображает статистические данные о производительности виртуальной машины (рис. 7.3). Запустите на выполнение сначала свою программу, а затем утилиту **jconsole**. Далее выберите свою программу из списка выполняющихся программ на Java. На консоль будет выведено немало полезных сведений о вашей программе. Подробнее об этом см. по адресу www.oracle.com/technetwork/articles/java/jconsole-1564139.html.

¹ предупреждение["провал"] возможный "провал" в ветви выбора `case`

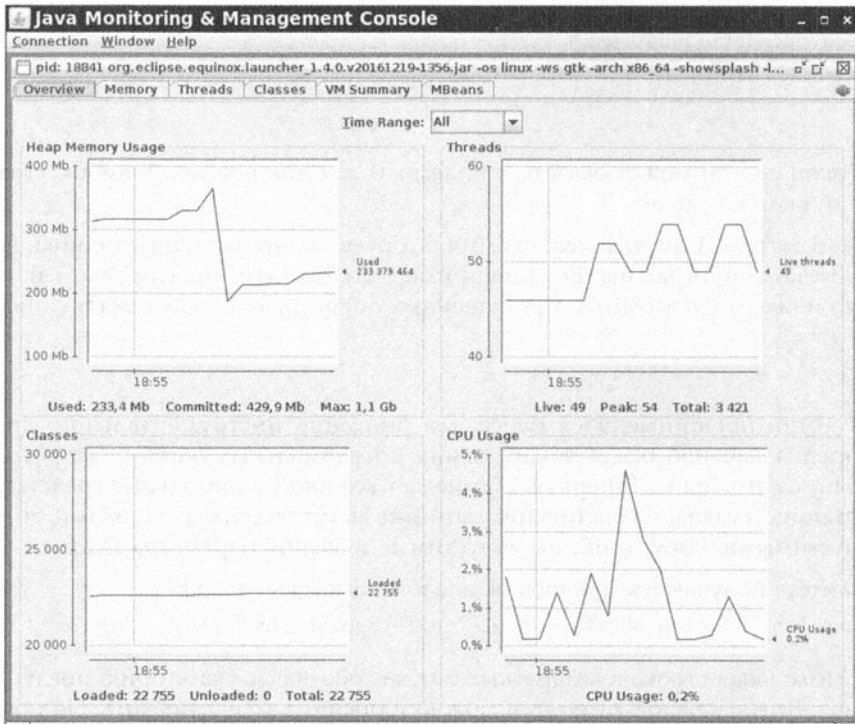


Рис. 7.3. Окно утилиты jconsole

12. В комплект Oracle JDK входит инструментальное средство Java Mission Control, предназначенное для профилирования и диагностики прикладных программ на профессиональном уровне. Им можно свободно пользоваться на стадии разработки приложений, тогда как на стадии их эксплуатации потребуется коммерческая лицензия на данное средство. Со временем его версия с открытым исходным кодом будет доступна как составная часть комплекта OpenJDK. Подобно утилите **jconsole**, инструментальное средство Java Mission Control можно присоединить к действующей виртуальной машине. Оно способно также анализировать результаты, выводимые инструментальным средством Java Flight Recorder, собирающим данные, касающиеся диагностики и профилирования выполняющегося приложения на Java. Подробнее об этих инструментальных средствах см. по адресу <https://docs.oracle.com/javacomponents/index.html>.

В этой главе был представлен механизм обработки исключений, а также даны полезные рекомендации по поводу тестирования и отладки прикладных программ. В двух последующих главах речь пойдет об обобщенном программировании и его наиболее важном применении: каркасе коллекций Java.

Обобщенное программирование

В этой главе...

- ▶ Назначение обобщенного программирования
- ▶ Определение простого обобщенного класса
- ▶ Обобщенные методы
- ▶ Ограничения на переменные типа
- ▶ Обобщенный код и виртуальная машина
- ▶ Ограничения и пределы обобщений
- ▶ Правила наследования обобщенных типов
- ▶ Подстановочные типы
- ▶ Рефлексия и обобщения

У обобщенных классов и методов имеются параметры типа. Это дает возможность точно описать, что именно должно произойти, когда получаются их экземпляры конкретного типа. До появления обобщенных классов программистам приходилось пользоваться классом `Object` для написания кода, способного обрабатывать разнотипные данные. Но это было неудобно и ненадежно.

С внедрением обобщений в Java появилась выразительная система типов, позволяющая разработчикам подробно описывать порядок смены типов переменных и методов. В простых случаях реализовать обобщенный код нетрудно, а в более сложных — намного труднее. Ведь цель состоит в том, чтобы предоставить такие классы и методы, которыми другие программисты могли бы пользоваться без всяких неожиданностей.

Внедрение обобщений в версии Java 5 — самое важное изменение в языке программирования Java с момента его первоначального выпуска. При этом главной

проектной целью было обеспечение обратной совместимости с прежними выпусками. В итоге на обобщения в Java был наложен ряд неудобных ограничений. О выгодах и трудностях обобщенного программирования речь пойдет в этой главе.

8.1. Назначение обобщенного программирования

Обобщенное программирование означает написание кода, который может быть неоднократно применен к разнотипным объектам. Так, если нет желания программировать отдельные классы для составления коллекций из объектов типа `String` и `File`, достаточно собрать эти объекты в коллекцию, воспользовавшись единственным обобщенным классом `ArrayList`. И это всего лишь один простой пример обобщенного программирования.

Фактически класс `ArrayList` имелся в Java еще до появления обобщенных классов. Итак, исследуем механизм обобщенного программирования и его назначение как для тех, кто его реализует, так и для тех, кто им пользуется.

8.1.1. Преимущества параметров типа

До внедрения обобщенных классов обобщенное программирование на Java всегда реализовывалось посредством наследования. Так, в классе `ArrayList` просто поддерживался массив ссылок на класс `Object` следующим образом:

```
public class ArrayList // до появления обобщенных классов
{
    private Object[] elementData;
    . . .
    public Object get(int i) { . . . }
    public void add(Object o) { . . . }
}
```

Такой подход порождает две серьезные проблемы. Всякий раз, когда извлекается значение, приходится выполнять приведение типов, как показано ниже.

```
ArrayList files = new ArrayList();
. . .
String filename = (String) files.get(0);
```

Более того, в таком коде отсутствует проверка ошибок. Ничто не мешает добавить значения любого класса следующим образом:

```
files.add(new File(". . ."));
```

Этот вызов компилируется и выполняется без ошибок. Но затем попытка привести результат выполнения метода `get()` к типу `String` приведет к ошибке.

Обобщения предлагают лучшее решение: *параметры типа*. Класс `ArrayList` теперь принимает параметр типа, обозначающий тип элементов коллекции, как показано ниже. Благодаря этому код получается более удобочитаемым. Теперь становится сразу понятно, что данный конкретный списочный массив содержит объекты типа `String`.

```
var files = new ArrayList<String>();
```



НА ЗАМЕТКУ! Если объявить переменную, явно указав ее тип вместо ключевого слова `var`, то в конструкторе можно опустить параметр типа, используя ромбовидный синтаксис, как показано ниже. Опущенный тип объекта автоматически выводится из типа переменной.

```
ArrayList<String> files = new ArrayList<>();
```

В версии Java 9 применение ромбовидного синтаксиса распространено на те случаи, когда это было раньше не приемлемо. Например, ромбовидный синтаксис можно теперь применять к анонимным подклассам, как показано ниже.

```
ArrayList<String> passwords = new ArrayList<>()
    // допустимое применение ромбовидного синтаксиса
    // в версии Java 9
{
    public String get(int n)
    { return super.get(n).replaceAll(".", "*"); }
};
```

Компилятору также известно, что у метода `add()` из класса `ArrayList<String>` имеется параметр типа `String`. Это намного безопаснее, чем иметь дело с параметром типа `Object`. Теперь компилятор может проконтролировать, не подставлен ли объект неверного типа в исходном коде. Например, следующая строка кода не скомпилируется:

```
files.add(new File(". . ."));
// в коллекцию типа ArrayList<String>
// можно вводить только объекты типа String
```

Ошибка компиляции — это намного лучше, чем исключение в связи с неправильным приведением типов во время выполнения. Привлекательность параметров типа в том и состоит, что они делают исходный код программы более удобочитаемым и безопасным.

8.1.2. На кого рассчитано обобщенное программирование

Пользоваться такими обобщенными классами, как `ArrayList`, очень просто. И большинство программирующих на Java просто применяют типы вроде `ArrayList<String>`, как будто они являются такой же неотъемлемой частью языка Java, как и массивы типа `String[]`. (Разумеется, списочные массивы лучше простых массивов, поскольку они допускают автоматическое расширение.)

Но реализовать обобщенный класс не так-то просто. Те, кто будет пользоваться обобщенным кодом, попытаются подставлять всевозможные классы вместо предусмотренных параметров типа. Они надеются, что все будет работать без досадных ограничений и запутанных сообщений об ошибках. Поэтому главная задача обобщенно программирующего — предвидеть все возможные в дальнейшем применения разработанного им обобщенного класса.

Насколько трудно этого добиться? Приведем пример типичного затруднения, которое пришлось преодолевать разработчикам стандартной библиотеки классов Java. Класс `ArrayList` содержит метод `addAll()`, предназначенный для добавления элементов из другой коллекции. Так, у программиста может возникнуть потребность добавить все элементы из коллекции типа `ArrayList<Manager>` в коллекцию типа `ArrayList<Employee>`. Но обратное, разумеется, недопустимо. Как же разрешить один вызов и запретить другой? Создатели Java нашли искусный выход из этого затруднительного положения, внедрив понятие *подстановочного типа*. Подстановочные типы довольно абстрактны, но они позволяют разработчику библиотеки сделать методы как можно более гибкими.

Обобщенное программирование разделяется на три уровня квалификации. На элементарном уровне можно просто пользоваться обобщенными классами (как

правило, коллекциями типа `ArrayList`, даже не задумываясь, как они работают). Большинство прикладных программистов предпочитают оставаться на этом уровне до тех пор, пока дело не заладится. Сочетая разные обобщенные классы или же имея дело с унаследованным кодом, в котором ничего неизвестно о параметрах типа, можно столкнуться с непонятным сообщением об ошибках. В подобной ситуации необходимо иметь достаточно ясное представление об обобщениях в Java, чтобы устранить неполадку системно, а не “методом тыка”. И, наконец, можно решиться на реализацию своих собственных обобщенных классов и методов.

Прикладным программистам, вероятнее всего, не придется писать много обобщенного кода. Разработчики JDK уже выполнили самую трудную работу и предусмотрели параметры типа для всех классов коллекций. В связи с этим можно сформулировать следующее эмпирическое правило: от применения параметров типа выигрывает только тот код, в котором традиционно присутствует много операций приведения от самых общих типов, как, например, класс `Object` или интерфейс `Comparable`.

В этой главе поясняется все, что следует знать для реализации собственного обобщенного кода. Надеемся, однако, что читатели воспользуются почерпнутыми из этой главы знаниями, прежде всего, для отыскания причин неполадок в своих программах, а также для удовлетворения любопытства относительно внутреннего устройства параметризованных классов коллекций.

8.2. Определение простого обобщенного класса

Обобщенным называется класс с одной или несколькими переменными типа. В этой главе в качестве примера рассматривается простой обобщенный класс `Pair`. Он выбран для примера потому, что позволяет сосредоточить основное внимание на механизме обобщений, не вдаваясь в подробности сохранения данных. Ниже приведен исходный код обобщенного класса `Pair`.

```
public class Pair<T>
{
    private T first;
    private T second;

    public Pair() { first = null; second = null; }
    public Pair(T first, T second)
        { this.first = first; this.second = second; }

    public T getFirst() { return first; }
    public T getSecond() { return second; }

    public void setFirst(T newValue) { first = newValue; }
    public void setSecond(T newValue) { second = newValue; }
}
```

В классе `Pair` вводится переменная типа `T`, заключенная в угловые скобки (`<>`) после имени самого класса. У обобщенного класса может быть больше одной переменной типа. Например, класс `Pair` можно было бы определить с разными типами для первого и второго поля следующим образом:

```
public class Pair<T, U> { . . . }
```

Переменные типа используются повсюду в определении класса для обозначения типов, возвращаемых методами, а также типов полей и локальных переменных. Ниже приведен пример объявления переменной типа.

```
private T first; // использовать переменную типа
```



НА ЗАМЕТКУ! В именах переменных типа принято употреблять прописные буквы и стремиться к их краткости. Так, в стандартной библиотеке Java буквой **E** обозначается тип элемента коллекции, буквами **K** и **V** — типы ключей и значений в таблице, а буквой **T** (и соседними с ней буквами **S** и **U** при необходимости) — “любой тип вообще”, как поясняется в документации.

Экземпляр обобщенного типа получается путем подстановки имени типа вместо переменной типа:

```
Pair<String>
```

Результат такой подстановки следует рассматривать как обычный класс с конструкторами:

```
Pair<String>()
Pair<String>(String, String)
```

и методами:

```
String getFirst()
String getSecond()
void setFirst(String)
void setSecond(String)
```

Иными словами, обобщенный класс действует как фабрика обычных классов.

В примере программы из листинга 8.1 применяется класс `Pair`. В статическом методе `minmax()` осуществляется перебор элементов массива с одновременным вычислением минимального и максимального значений. Для возврата обоих значений используется объект типа `Pair`. Напомним, что в методе `compareTo()` сравниваются две строки и возвращается нулевое значение, если символьные строки одинаковы; отрицательное числовое значение — если первая символьная строка следует прежде второй в лексикографическом порядке; а иначе — положительное числовое значение.



НА ЗАМЕТКУ C++! На первый взгляд обобщенные классы в Java похожи на шаблонные классы в C++. Единственное очевидное отличие состоит в том, что в Java не предусмотрено специальное ключевое слово `template`. Но, как будет показано далее в этой главе, у этих двух механизмов обобщений имеются существенные отличия.

Листинг 8.1. Исходный код из файла `pair1/PairTest1.java`

```
1 package pair1;
2
3 /**
4  * @version 1.01 2012-01-26
5  * @author Cay Horstmann
6  */
7 public class PairTest1
8 {
9     public static void main(String[] args)
10    {
```

```

11     String[] words =
12         { "Mary", "had", "a", "little", "lamb" };
13     Pair<String> mm = ArrayAlg.minmax(words);
14     System.out.println("min = " + mm.getFirst());
15     System.out.println("max = " + mm.getSecond());
16 }
17 }
18
19 class ArrayAlg
20 {
21     /**
22      * Получает символьные строки с минимальным и
23      * максимальным значениями среди элементов массива
24      * @param a Массив символьных строк
25      * @return Пара минимального и максимального значений
26      *         или пустое значение, если параметр a имеет
27      *         пустое значение
28      */
29     public static Pair<String> minmax(String[] a)
30     {
31         if (a == null || a.length == 0) return null;
32         String min = a[0];
33         String max = a[0];
34         for (int i = 1; i < a.length; i++)
35         {
36             if (min.compareTo(a[i]) > 0) min = a[i];
37             if (max.compareTo(a[i]) < 0) max = a[i];
38         }
39         return new Pair<>(min, max);
40     }
41 }

```

8.3. Обобщенные методы

В предыдущем разделе было показано, как определяется обобщенный класс. Имеется также возможность определить отдельный метод с параметрами (или переменными) типа следующим образом:

```

class ArrayAlg
{
    public static <T> T getMiddle(T... a)
    {
        return a[a.length / 2];
    }
}

```

Этот метод определен в обычном, а не в обобщенном классе. Тем не менее это обобщенный метод, на что указывают угловые скобки и переменная типа. Обратите внимание на то, что переменная типа вводится после модификаторов доступа (в данном случае `public static`) и перед возвращаемым типом.

Обобщенные методы можно определять как в обычных, так и в обобщенных классах. Когда вызывается обобщенный метод, ему можно передать конкретные типы данных, заключая их в угловые скобки перед именем метода, как показано ниже.

```

String middle = ArrayAlg.<String>getMiddle("John", "Q.",
                                         "Public");

```

В данном случае (как и вообще) при вызове метода можно пропустить параметр типа `String`. У компилятора имеется достаточно информации, чтобы вывести из такого обобщения именно тот метод, который требуется вызвать. Он сопоставляет тип аргументов с обобщенным типом `T...` и приходит к выводу, что вместо обобщенного типа `T` следует подставить конкретный тип `String`, что равнозначно следующему вызову:

```
String middle = ArrayAlg.getMiddle("John", "Q.", "Public");
```

Выводимость типов в обобщенных методах практически всегда действует исправно. Но иногда компилятор ошибается, и тогда приходится разбираться в тех ошибках, о которых он сообщает. Рассмотрим в качестве примера следующую строку кода:

```
double middle = ArrayAlg.getMiddle(3.14, 1729, 0);
```

При выполнении этой строки кода компилятор выведет сообщение об ошибке заголовочного содержания, но суть его в том, что данный код можно интерпретировать двояко и в обоих случаях — правильно. По существу, компилятор выполняет автоупаковку параметра в один объект типа `Double` и два объекта типа `Integer`, а затем пытается найти для них общий суперттип. И таких супретипов два: класс `Number` и интерфейс `Comparable`, который сам является обобщенным. В этом случае для устранения ошибки методу следует передать все параметры со значениями типа `double`.



СОВЕТ. Петер Ван Дер Ахе (Peter von der Ahé) рекомендует следующий прием, если требуется выяснить, какой тип компилятор выводит при вызове обобщенного метода: намеренно допустить ошибку и изучите полученное в итоге сообщение об ошибке. В качестве примера рассмотрим следующий вызов: `ArrayAlg.getMiddle("Hello", 0, null)`. Если присвоить полученный результат переменной ссылки на объект типа `JBUTTON`, что заведомо неверно, то в конечном итоге будет получено следующее сообщение об ошибке:

```
found:
java.lang.Object&java.io.Serializable&java.lang.Comparable<? extends
java.lang.Object&java.io.Serializable&java.lang.Comparable<?>>
```

Это означает, что результат вызова данного метода можно присвоить переменным ссылки на объекты типа `Object`, `Serializable` или `Comparable`.



НА ЗАМЕТКУ C++! В языке C++ параметр типа указывается после имени метода, что может привести к неприятной неоднозначности при синтаксическом анализе кода. Например, выражение `g(f<a,b>(c))` может означать следующее: "вызвать метод `g()` с результатом вызова `f<a,b>(c)`" или же "вызвать метод `g()` с двумя логическими значениями `f<a` и `b>(c)`".

8.4. Ограничения на переменные типа

Иногда класс или метод нуждается в наложении ограничений на переменные типа. Приведем характерный пример, в котором требуется вычислить наименьший элемент массива следующим образом:

```
class ArrayAlg
{
    public static <T> T min(T[] a) // почти верно
    {
        if (a == null || a.length == 0) return null;
        T smallest = a[0];
```

```

    for (int i = 1; i < a.length; i++)
        if (smallest.compareTo(a[i]) > 0) smallest = a[i];
    return smallest;
}
}

```

Но здесь возникает затруднение. Обратите внимание на тело метода `min()`. Переменная `smallest` относится к типу `T`, а это означает, что она может быть объектом произвольного класса. Но откуда известно, имеется ли метод `compareTo()` в том классе, к которому относится тип `T`?

Выход из этого затруднительного положения состоит в том, чтобы наложить ограничение на тип `T` и вместо него подставлять только класс, реализующий `Comparable` — стандартный интерфейс с единственным методом `compareTo()`. Для этого достаточно наложить *ограничение* на переменную типа `T` следующим образом:

```
public static <T extends Comparable> T min(T[] a) . . .
```

В действительности интерфейс `Comparable` сам является обобщенным. Но мы не будем пока что принимать во внимание это обстоятельство и соответствующие предупреждения компилятора. В разделе 8.8 мы обсудим, как правильно пользоваться параметрами типа вместе с интерфейсом `Comparable`.

Теперь обобщенный метод `min()` может вызываться только с массивами классов, реализующих интерфейс `Comparable`, в том числе `String`, `Date` и т.п. А вызов `min()` с массивом типа `Rectangle` приведет к ошибке во время компиляции, поскольку класс `Rectangle` не реализует интерфейс `Comparable`.



НА ЗАМЕТКУ C++! В языке C++ нельзя ограничивать типы в параметрах шаблонов. Если создать экземпляр шаблона с неправильным типом, появится (зачастую невнятное) сообщение об ошибке в коде шаблона.

Вас может удивить, почему здесь используется ключевое слово `extends` вместо `implements`, ведь `Comparable` — это интерфейс? Так, следующее обозначение:

```
<T extends ограничивающий_тип>
```

означает, что тип `T` должен быть *подтипом* ограничивающего типа, причем к типу `T` и ограничивающему типу может относиться как интерфейс, так и класс. Ключевое слово `extends` выбрано потому, что это вполне благоразумное приближение понятия подтипа, и создатели Java не сочли нужным вместо этого вводить в язык новое ключевое слово.

Переменная типа или подстановка может иметь несколько ограничений, как показано в приведенном ниже примере кода. Ограничивающие типы разделяются знаком `&`, потому что запятые служат для разделения переменных типа.

```
T extends Comparable & Serializable
```

Как и в механизме наследования в Java, у интерфейсов может быть сколько угодно супертипов, но только один из ограничивающих типов может быть классом. Если для ограничения служит класс, он должен быть первым в списке накладываемых ограничений.

В следующем примере программы из листинга 8.2 метод `minmax()` переделан на обобщенный. В этом методе вычисляется минимальное и максимальное значения в обобщенном массиве и возвращается объект обобщенного типа `Pair<T>`.

Листинг 8.2. Исходный код из файла `pair2/PairTest2.java`

```
1 package pair2;
2
3 import java.time.*;
4
5 /**
6  * @version 1.02 2015-06-21
7  * @author Cay Horstmann
8  */
9 public class PairTest2
10 {
11     public static void main(String[] args)
12     {
13         LocalDate[] birthdays =
14         {
15             LocalDate.of(1906, 12, 9), // G. Hopper
16             LocalDate.of(1815, 12, 10), // A. Lovelace
17             LocalDate.of(1903, 12, 3), // J. von Neumann
18             LocalDate.of(1910, 6, 22), // K. Zuse
19         };
20         Pair<LocalDate> mm = ArrayAlg.minmax(birthdays);
21         System.out.println("min = " + mm.getFirst());
22         System.out.println("max = " + mm.getSecond());
23     }
24 }
25
26 class ArrayAlg
27 {
28     /**
29     * Получает минимальное и максимальное значения
30     * из массива объектов типа T
31     * @param a Массив объектов типа T
32     * @return a Пара минимального и максимального значений
33     *         или пустое значение, если параметр a имеет
34     *         пустое значение
35     */
36     public static <T extends Comparable>
37         Pair<T> minmax(T[] a)
38     {
39         if (a == null || a.length == 0) return null;
40         T min = a[0];
41         T max = a[0];
42         for (int i = 1; i < a.length; i++)
43         {
44             if (min.compareTo(a[i]) > 0) min = a[i];
45             if (max.compareTo(a[i]) < 0) max = a[i];
46         }
47         return new Pair<>(min, max);
48     }
49 }
```


8.5. Обобщенный код и виртуальная машина

Виртуальная машина не оперирует объектами обобщенных типов — все объекты принадлежат обычным классам. В первоначальных вариантах реализации обобщений можно было даже компилировать программу с обобщениями в файлы классов, которые способна исполнять виртуальная машина версии 1.0! В последующих разделах будет показано, каким образом компилятор “стирает” параметры типа, а также пояснены последствия этого процесса для программирующих на Java.

8.5.1. Стирание типов

Всякий раз, когда определяется обобщенный тип, автоматически создается соответствующий ему базовый (так называемый “сырой”) тип. Имя этого типа совпадает с именем обобщенного типа с удаленными параметрами типа. Переменные типа *стираются* и заменяются ограничивающими типами (или типом `Object`, если переменная не имеет ограничений). Например, базовый тип для обобщенного типа `Pair<T>` выглядит следующим образом:

```
public class Pair
{
    private Object first;
    private Object second;
    public Pair(Object first, Object second)
    {
        this.first = first;
        this.second = second;
    }

    public Object getFirst() { return first; }
    public Object getSecond() { return second; }
    public void setFirst(Object newValue)
    { first = newValue; }
    public void setSecond(Object newValue)
    { second = newValue; }
}
```

Если `T` — неограниченная переменная типа, то ее тип просто заменяется на `Object`. В итоге получается обычный класс вроде тех, что реализовывались до появления обобщений в Java. Прикладные программы могут содержать разные варианты обобщенного класса `Pair`, в том числе `Pair<String>` или `Pair<GregorianCalendar>`, но в результате стирания все они приводятся к базовым типам `Pair`.



НА ЗАМЕТКУ C++! В этом отношении обобщения в Java заметно отличаются от шаблонов в C++. Для каждого экземпляра шаблона в C++ получается свой тип. Это неприятное явление называется “раздуванием кода шаблона”. Этим недостатком Java не страдает.

Базовый тип заменяет тип переменных первым накладываемым на них ограничением или же типом `Object`, если никаких ограничений не предусмотрено. Например, у переменной типа в обобщенном классе `Pair<T>` отсутствуют явные ограничения, поэтому базовый тип заменяет обобщенный тип `T` на `Object`. Допустим, однако, что объявлен несколько иной обобщенный тип:

```
public class Interval<T extends Comparable &
    Serializable> implements Serializable
{
    private T lower;
    private T upper;
    . . .
    public Interval(T first, T second)
    {
        if (first.compareTo(second) <= 0)
            { lower = first; upper = second; }
        else { lower = second; upper = first; }
    }
}
```

Соответствующий ему базовый тип выглядит следующим образом:

```
public class Interval implements Serializable
{
    private Comparable lower;
    private Comparable upper;
    . . .
    public Interval(Comparable first, Comparable second)
    { . . . }
}
```



НА ЗАМЕТКУ! А что, если поменять местами ограничения: `class Interval<Serializable & Comparable>`? В этом случае базовый тип заменит обобщенный тип `T` на `Serializable`, а компилятор произведет там, где требуется, приведение к типу `Comparable`. Поэтому ради эффективности в конце списка ограничений следует размещать отмечающие интерфейсы (т.е. интерфейсы без методов).

8.5.2. Преобразование обобщенных выражений

Когда в программе вызывается обобщенный метод, компилятор вводит операции приведения типов при стирании возвращаемого типа. Рассмотрим в качестве примера следующую последовательность операторов:

```
Pair<Employee> buddies = . . . ;
Employee buddy = buddies.getFirst();
```

В результате стирания из метода `getFirst()` возвращается тип `Object`. Поэтому компилятор автоматически вводит приведение к типу `Employee`. Это означает, что компилятор преобразует вызов данного метода в следующие две команды для виртуальной машины:

- вызвать метод базового типа `Pair.getFirst()`;
- привести тип `Object` возвращаемого объекта к типу `Employee`.

Операции приведения типов вводятся также при обращении к обобщенному полю. Допустим, что поля `first` и `second` открытые, т.е. объявлены как `public`. (Возможно, это и не самый лучший, но вполне допустимый в Java стиль программирования.) Тогда при преобразовании приведенного ниже выражения в получаемый в конечном итоге байт-код также будут введены операции приведения типов.

```
Employee buddy = buddies.first;
```

8.5.3. Преобразование обобщенных методов

Стирание типов происходит и в обобщенных методах. Программисты обычно воспринимают обобщенные методы как целое семейство методов вроде следующего:

```
public static <T extends Comparable> T min(T[] a)
```

Но после стирания типов остается только один приведенный ниже метод. Обратите внимание на то, что параметр обобщенного типа `T` стирается, а остается только ограничивающий тип `Comparable`.

```
public static Comparable min(Comparable[] a)
```

Стирание типов в обобщенных методах приводит к некоторым осложнениям. Рассмотрим следующий пример кода:

```
class DateInterval extends Pair<LocalDate>
{
    public void setSecond(LocalDate second)
    {
        if (second.compareTo(getFirst()) >= 0)
            super.setSecond(second);
    }
    . . .
}
```

В этом фрагменте кода интервал дат задается парой объектов типа `LocalDate`, поэтому соответствующие методы требуется переопределить, чтобы второе сравниваемое значение не было меньше первого. В результате стирания данный класс преобразуется в следующий:

```
class DateInterval extends Pair // после стирания
{
    public void setSecond(LocalDate second) { . . . }
    . . .
}
```

Как ни странно, но имеется и другой метод `setSecond()`, унаследованный от класса `Pair`, а именно:

```
public void setSecond(Object second)
```

И это совершенно иной метод, поскольку он имеет параметр другого типа: `Object` вместо `LocalDate`. Но он *не* должен быть другим. Рассмотрим следующую последовательность операторов:

```
DateInterval interval = new DateInterval(. . .);
Pair<LocalDate> pair = interval; // допускается
                                // присваивание суперклассу
pair.setSecond(aDate);
```

Предполагается, что вызов метода `setSecond()` является полиморфным, и поэтому вызывается соответствующий метод. А поскольку переменная `pair` ссылается на объект типа `DateInterval`, это должен быть вызов `DateInterval.setSecond()`. Но дело в том, что стирание типов мешает соблюдению принципа полиморфизма. В качестве выхода из этого затруднительного положения компилятор формирует следующий *мостовой метод* в классе `DateInterval`:

```
public void setSecond(Object second)
{ setSecond((LocalDate) second); }
```

Чтобы стал понятнее этот механизм, проанализируем выполнение приведенного ниже оператора.

```
pair.setSecond(aDate)
```

В объявлении переменной `pair` указан тип `Pair<LocalDate>`, к которому относится только один метод под именем `setSecond`, а именно `setSecond(Object)`. Виртуальная машина вызывает этот метод для того объекта, на который ссылается переменная `pair`. Этот объект относится к типу `DateInterval`, и поэтому вызывается метод `DateInterval.setSecond(Object)`. Именно он и является синтезированным мостовым методом. Ведь он, в свою очередь, вызывает метод `DateInterval.setSecond(LocalDate)`, что, собственно говоря, и требуется.

Мостовые методы могут быть еще более необычными. Допустим, в классе `DateInterval` переопределяется также метод `getSecond()`:

```
class DateInterval extends Pair<LocalDate>
{
    public LocalDate getSecond()
    { return (LocalDate) super.getSecond().clone(); }
    . . .
}
```

В классе `DateInterval` имеются следующие два метода под именем `getSecond`:

```
Date getSecond()    // определен в классе DateInterval
Object getSecond()  // переопределяет метод из класса Pair
                    // для вызова первого метода
```

Написать такой код на Java без параметров нельзя. Ведь было бы неверно иметь два метода с одинаковыми типами параметров. Но в виртуальной машине типы параметров и *возвращаемый тип* определяют метод. Поэтому компилятор может сформировать байт-код для двух методов, отличающихся только возвращаемым типом, и виртуальная машина правильно ведет себя в подобной ситуации.



НА ЗАМЕТКУ! Применение мостовых методов не ограничивается только обобщенными типами. Как упоминалось в главе 5, вполне допустимо определять в методе более ограниченный возвращаемый тип, когда он переопределяет другой метод. Так, в приведенном ниже фрагменте кода методы `Object.clone()` и `Employee.clone()` имеют так называемые *ковариантные возвращаемые типы*.

```
public class Employee implements Cloneable
{
    public Employee clone()
        throws CloneNotSupportedException { ... }
}
```

На самом деле в классе `Object` имеются два таких метода:

```
Employee clone() // определен выше
Object clone()   // синтезированный мостовой метод, переопределяющий
                  // метод Object clone()
```

При этом синтезированный мостовой метод вызывает вновь определенный метод.

Таким образом, о преобразовании обобщений в Java нужно запомнить следующее.

- Для виртуальной машины обобщений не существует, но имеются только обычные классы и методы.

- Все параметры типа заменяются ограничивающими типами.
- Мостовые методы синтезируются для соблюдения принципа полиморфизма.
- Операции приведения типов вводятся по мере надобности для обеспечения типовой безопасности.

8.5.4. Вызов унаследованного кода

Внедрением обобщений в Java преследовалась главная цель: обеспечить совместимость обобщенного и унаследованного кода. Обратимся к конкретному примеру. В библиотеке Swing для разработки графических интерфейсов предоставляется класс `JSlider`, реализующий компонент ползунок, где отметки могут быть специально определены с метками, содержащими текст или изображения. Метки устанавливаются с помощью следующего вызова:

```
void setLabelTable(Dictionary table)
```

В классе `Dictionary` целые числа сопоставляются с метками. До версии Java 5 данный класс был реализован в виде отображения экземпляров типа `Object`. А в версии Java 5 класс `Dictionary` был сделан обобщенным, хотя класс `JSlider` вообще не был обновлен. Таким образом, класс `Dictionary` без параметров типа относится к базовому типу. Именно здесь и вступает в действие совместимость.

При заполнении словаря можно воспользоваться базовым типом, как показано ниже.

```
Dictionary<Integer, Component> labelTable =  
    new Hashtable<>();  
labelTable.put(0, new JLabel(new ImageIcon("nine.gif")));  
labelTable.put(20, new JLabel(new ImageIcon("ten.gif")));  
...
```

А при попытке передать объект обобщенного типа `Dictionary<Integer, Component>` методу `setLabelTable()`, как демонстрируется в приведенной ниже строке кода, компилятор выдаст соответствующее предупреждение.

```
slider.setLabelTable(labelTable); // ПРЕДУПРЕЖДЕНИЕ!
```

Ведь компилятору неизвестно намерен, что именно метод `setLabelTable()` может сделать с объектом типа `Dictionary`. Так, этот метод может заменить все ключи символьными строками, нарушив гарантию того, что ключи должны иметь тип `Integer`. Поэтому при выполнении последующих операций приведения типов могут возникнуть неприятные исключения.

Над этим стоит поразмыслить, чтобы выяснить, что именно компонент типа `JSlider` собирается делать с данным объектом типа `Dictionary`. В данном случае совершенно ясно, что компонент типа `JSlider` только вводит информацию, так что предупреждением компилятора можно пренебречь.

А теперь рассмотрим противоположный случай, когда объект базового типа получается от унаследованного класса. Его можно присвоить переменной обобщенного типа, но тогда будет выдано предупреждение, как показано в приведенной ниже строке кода.

```
Dictionary<Integer, Components> labelTable =  
    slider.getLabelTable(); // ПРЕДУПРЕЖДЕНИЕ!
```

И в этом случае следует проанализировать предупреждение и убедиться в том, что таблица меток действительно содержит объекты типа `Integer` и `Component`,

хотя нет никакой гарантии, что они там присутствуют. В частности, злоумышленник может установить другой объект типа `Dictionary` в компоненте типа `JSlider`. Но опять же эта ситуация не хуже той, что была до внедрения обобщений. В худшем случае программа сгенерирует исключение.

Итак, обратив внимание на предупреждение компилятора, можно воспользоваться аннотацией для того, чтобы оно исчезло. Такую аннотацию следует разместить перед локальной переменной следующим образом:

```
@SuppressWarnings("unchecked")
Dictionary<Integer, Components> labelTable =
    slider.getLabelTable(); // без предупреждения
```

Аналогичным образом можно снабдить аннотацией весь метод, как показано ниже. Такая аннотация отменяет проверку всего кода в теле метода.

```
@SuppressWarnings("unchecked")
public void configureSlider() { . . . }
```

8.6. Ограничения и пределы обобщений

В последующих разделах будет рассмотрен целый ряд ограничений, которые следует учитывать, работая с обобщениями в Java. Большинство этих ограничений являются следствием стирания типов.

8.6.1. Параметрам типа нельзя приписывать простые типы

Примитивный тип нельзя подставить вместо типа параметра. Это означает, что не бывает объекта типа `Pair<double>`, а только объект типа `Pair<Double>`. Причину, конечно, следует искать в стирании типов. После такого стирания в классе `Pair` отсутствуют поля типа `Object`, и поэтому их нельзя использовать для хранения значений типа `double`.

И хотя такое ограничение досадно, тем не менее, он согласуется с особым положением примитивных типов в Java. Этот недостаток не носит фатального характера. Ведь существует всего восемь простых типов данных, а обработать их всегда можно с помощью отдельных классов и методов, когда нельзя подставить вместо них типы-оболочки.

8.6.2. Во время выполнения можно запрашивать только базовые типы

В виртуальной машине объекты всегда имеют определенный необобщенный тип. Поэтому все запросы типов во время выполнения дают только базовый тип. Например, следующий оператор на самом деле только проверяет, является ли а экземпляром `Pair` любого типа:

```
if (a instanceof Pair<String>) // ОШИБКА!
```

Это же справедливо и в отношении следующего оператора:

```
if (a instanceof Pair<T>) // ОШИБКА!
```

или такого оператора приведения типов:

```
Pair<String> p = (Pair<String>) a; // ПРЕДУПРЕЖДЕНИЕ!
// Проверить можно только принадлежность
// переменной a к типу Pair
```

Чтобы напомнить о возможной опасности, компилятор выдает ошибку (при операции `instanceof`) или предупреждение (при приведении типов) всякий раз, когда делается запрос, принадлежит ли объект к обобщенному типу. Аналогично метод `getClass()` всегда возвращает базовый тип, как показано в приведенном ниже примере кода. Результатом сравнения оказывается логическое значение `true`, потому что при обоих вызовах метода `getClass()` возвращается объект типа `Pair.class`.

```
Pair<String> stringPair = . . .;
Pair<Employee> employeePair = . . .;
if (stringPair.getClass() ==
    employeePair.getClass()) // равны!
```

8.6.3. Массивы параметризованных типов недопустимы

Нельзя объявить массив параметризованных типов вроде следующего:

```
var table = new Pair<String>[10]; // ОШИБКА!
```

Что же здесь не так? После стирания типом таблицы становится `Pair[]`. Но его можно преобразовать в тип `Object[]` следующим образом:

```
Object[] objarray = table;
```

В массиве запоминается тип его элементов и генерируется исключение типа `ArrayStoreException`, если попытаться сохранить в нем элемент неверного типа, как показано ниже.

```
objarray[0] = "Hello"; // ОШИБКА! Типом компонента
                       // является Pair
```

Но стирание делает этот механизм неэффективным для обобщенных типов. Приведенное ниже присваивание пройдет проверку на сохранение в массиве, но выдаст ошибку соблюдения типов. Именно поэтому массивы параметризованных типов не допускаются.

```
objarray[0] = new Pair<Employee>();
```

Следует, однако, иметь в виду, что не допускается только создание подобных массивов. И хотя разрешается, например, объявить переменную типа `Pair<String>[]`, ее нельзя инициализировать с помощью операции `new Pair<String>[10]`.



НА ЗАМЕТКУ! Допускается объявлять массивы элементов подстановочных типов, а затем приводить их к соответствующим типам следующим образом:

```
var table = (Pair<String>[]) new Pair<?>[10];
```

Хотя результат такой операции не гарантирует полную безопасность. Так, если сначала сохранить объект типа `Pair<Employee>` в элементе массива `table[0]`, а затем вызвать для него метод `table[0].getFirst()` из класса `String`, то будет сгенерировано исключение типа `ClassCastException`.



СОВЕТ. Если объекты параметризованных типов требуется хранить в коллекциях, пользуйтесь обобщенным классом `ArrayList`, например, `ArrayList<Pair<String>>`. Это безопасно и эффективно.

8.6.4. Предупреждения о переменном числе аргументов

Как было показано в предыдущем разделе, в Java не поддерживаются массивы обобщенных типов. А в этом разделе будет рассмотрен следующий связанный с этим вопрос: передача экземпляров обобщенных типов методу с переменным числом аргументов.

Рассмотрим в качестве примера следующий простой метод с переменным числом аргументов:

```
public static <T> void addAll(Collection<T> coll, T... ts)
{
    for (T t : ts) coll.add(t);
}
```

Напомним, что параметр `ts` на самом деле является массивом, содержащим все предоставляемые аргументы. А теперь рассмотрим вызов этого метода в приведенном ниже фрагменте кода.

```
Collection<Pair<String>> table = ...;
Pair<String> pair1 = ...;
Pair<String> pair2 = ...;
addAll(table, pair1, pair2);
```

Для вызова этого метода в виртуальной машине Java придется сформировать массив объектов типа `Pair<String>`, что не по правилам. Но эти правила были ослаблены, чтобы уведомлять в подобных случаях только о предупреждении.

Подавить выдачу такого предупреждения можно двумя способами. Во-первых, ввести аннотацию `@SuppressWarnings("unchecked")` в тело метода, содержащего вызов метода `addAll()`. И, во-вторых, начиная с версии Java 7, ввести аннотацию `@SafeVarargs` в тело самого метода `addAll()`, как показано ниже.

```
@SafeVarargs
public static <T> void addAll(Collection<T> coll, T... ts)
```

Теперь этот метод можно вызывать с аргументами обобщенных типов. Приведенную выше аннотацию можно ввести в любые методы, выбирающие элементы из массива параметров — наиболее характерного примера употребления переменного числа аргументов.

Аннотацию `@SafeVarargs` можно употреблять в конструкторах и методах типа `static`, `final` или `private` (начиная с версии Java 9). Любой другой метод можно было бы переопределить, но тогда употребление данной аннотации потеряло бы всякий смысл.



НА ЗАМЕТКУ! С помощью аннотации `@SafeVarargs` можно преодолеть ограничение на создание обобщенного массива. Для этой цели служит метод

```
@SafeVarargs static <E> E[] array(E... array) { return array; }
```

который можно далее вызвать следующим образом:

```
Pair<String>[] table = array(pair1, pair2);
```

На первый взгляд такой прием удобен, но он чреват следующей опасностью:

```
Object[] objarray = table;
objarray[0] = new Pair<Employee>();
```

Этот фрагмент кода будет выполняться без исключения типа `ArrayStoreException`, поскольку в массиве сохраняются данные только стертого типа. Но если обратиться к элементу массива `table[0]` в другом месте кода, то такое исключение будет сгенерировано.

8.6.5. Нельзя создавать экземпляры переменных типа

Переменные типа нельзя использовать в выражениях вроде `new T(...)`. Например, следующий конструктор `Pair<T>` недопустим:

```
public Pair() { first = new T(); second = new T(); } // ОШИБКА!
```

Стирание типов может изменить обобщенный тип `T` на `Object`, а вызывать конструктор `new Object()`, конечно, не стоит. Начиная с версии Java 8, можно прибегнуть к наилучшему обходному приему, предоставив в вызывающем коде ссылку на конструктор, как показано в следующем примере:

```
Pair<String> p = Pair.makePair(String::new);
```

Метод `makePair()` получает ссылку на функциональный интерфейс для вызова функции без аргументов и возврата результата типа `T` следующим образом:

```
public static <T> Pair<T> makePair(Supplier<T> constr)
{
    return new Pair<>(constr.get(), constr.get());
}
```

В качестве более традиционного обходного приема обобщенные объекты можно конструировать через рефлексию, вызывая метод `Constructor.newInstance()`. К сожалению, это совсем не простой прием. Так, сделать приведенный ниже вызов нельзя. Выражение `T.class` недопустимо, поскольку оно будет приведено путем стирания к выражению `Object.class`.

```
first = T.class.getConstructor().newInstance(); // ОШИБКА!
```

Вместо этого придется разработать прикладной интерфейс API, чтобы передать методу `makePair()` объект типа `Class`, как показано ниже.

```
public static <T> Pair<T> makePair(Class<T> cl)
{
    try {
        return new Pair<>(cl.getConstructor().newInstance(),
                        cl.getConstructor().newInstance());
    }
    catch (Exception ex) { return null; }
}
```

Этот метод может быть вызван следующим образом:

```
Pair<String> p = Pair.makePair(String.class);
```

Следует, однако, иметь в виду, что класс `Class` сам является обобщенным. Например, `String.class` — это экземпляр (на самом деле единственный) типа `Class<String>`. Поэтому в методе `makePair()` может быть автоматически выведен тип создаваемой пары.

8.6.6. Нельзя строить обобщенные массивы

Как нельзя получить единственный обобщенный экземпляр, так нельзя построить обобщенный массив. Но причина здесь другая: массив заполняется пустыми значениями `null`, что может показаться вполне безопасным для его построения. Но ведь массив относится к определенному типу, который служит в виртуальной машине для контроля значений, сохраняемых в массиве. А этот тип стирается. Рассмотрим в качестве примера следующую строку кода:

```
public static <T extends Comparable> T[] minmax(T[] a)
{ T[] mm = new T[2]; . . . } // ОШИБКА!
```

Если массив используется только как закрытое поле экземпляра класса, его можно объявить как `Object[]` и прибегнуть к приведению типов при извлечении из него элементов. Например, класс `ArrayList` может быть реализован следующим образом:

```
public class ArrayList<E>
{
    private Object[] elements;
    . . .
    @SuppressWarnings("unchecked") public E get(int n)
    { return (E) elements[n]; }
    public void set(int n, E e) { elements[n] = e; }
    // приведение типов не требуется!
}
```

Но конкретная его реализация не настолько ясна. Так, в следующем фрагменте кода приведение к типу `E[]` совершенно ложно, но стирание типов делает это незаметным:

```
public class ArrayList<E>
{
    private E[] elements;
    . . .
    public ArrayList() { elements = (E[]) new Object[10]; }
}
```

Такой прием не подходит для метода `minmax()`, поскольку он возвращает массив обобщенного типа `T[]`. И если сделать ложное заключение о типе, то во время выполнения кода возникнет ошибка. Допустим, обобщенный метод `minmax()` реализуется следующим образом:

```
public static <T extends Comparable> T[] minmax(T... a)
{
    Object[] mm = new Object[2];
    . . .
    return (T[]) mm; // компилируется без предупреждения
}
```

Тогда приведенный ниже вызов данного метода компилируется без всяких предупреждений.

```
String[] ss = minmax("Tom", "Dick", "Harry");
```

Исключение типа `ClassCastException` возникает, когда ссылка на объект типа `Object[]` приводится к типу `Comparable[]` при возврате из метода. В таком случае лучше всего предложить пользователю предоставить ссылку на конструктор массива следующим образом:

```
String[] ss = ArrayAlg.minmax(String[]::new, "Tom",
                               "Dick", "Harry");
```

Ссылка на конструктор `String::new` обозначает функцию для построения массива типа `String` заданной длины. Она служит в качестве параметра метода для получения массива нужного типа, как показано ниже.

```
public static <T extends Comparable> T[] minmax(
    IntFunction<T[]> constr, T... a)
{
```

```
T[] mm = constr.apply(2);
...
}
```

Более традиционный способ состоит в том, чтобы воспользоваться рефлексией и сделать вызов `Array.newInstance()` следующим образом:

```
public static <T extends Comparable> T[] minmax(T... a)
{
    T[] mm = (T[]) Array.newInstance(a.getClass()
                                     .getComponentType(), 2);
    ...
}
```

Методу `toArray()` из класса `ArrayList` повезло в меньшей степени. Ему нужно построить массив типа `T[]`, но у него нет типа элементов. Поэтому возможны два варианта:

```
Object[] toArray()
T[] toArray(T[] result)
```

Второй метод принимает параметр в виде массива. Если этот массив достаточно велик, то он используется. В противном случае создается новый массив подходящего размера с элементами типа `result`.

8.6.7. Переменные типа в статическом контексте обобщенных классов недействительны

На переменные типа нельзя ссылаться в статических полях или методах. Например, следующая замечательная идея не работает:

```
public class Singleton<T>
{
    private static T singleInstance; // ОШИБКА!

    public static T getSingleInstance() // ОШИБКА!
    {
        if (singleInstance == null)
            сконструировать новый экземпляр типа T
            return singleInstance;
    }
}
```

Если бы такое было возможно, то программа должна была бы сконструировать экземпляр типа `Singleton<Random>` для общего генератора случайных чисел и экземпляр типа `Singleton <JFileChooser>` для общего диалогового окна выбора файлов. Но такая уловка не работает. Ведь после стирания типов остается только один класс `Singleton` и только одно поле `singleInstance`. Именно по этой причине статические поля и методы с переменными типа просто недопустимы.

8.6.8. Нельзя генерировать или перехватывать экземпляры обобщенного класса в виде исключений

Генерировать или перехватывать объекты обобщенного класса в виде исключений не допускается. На самом деле обобщенный класс не может расширять класс `Throwable`. Например, приведенное ниже определение обобщенного класса не будет скомпилировано.

```
public class Problem<T> extends Throwable
{ /* . . . */ } // ОШИБКА!
    // Класс Throwable расширить нельзя!
```

Кроме того, переменную типа нельзя использовать в блоке `catch`. Например, следующий метод не будет скомпилирован:

```
public static <T extends Throwable> void doWork(Class<T> t)
{
    try
    {
        выполнить нужные действия
    }
    catch (T e) // ОШИБКА! Перехватывать
        // переменную типа нельзя
    {
        Logger.global.info(...)
    }
}
```

Но в то же время переменные типа можно использовать в определениях исключений. Так, приведенный ниже метод вполне допустим.

```
public static <T extends Throwable> void doWork(T t)
    throws T // Допустимо!
{
    try
    {
        выполнить нужные действия
    }
    catch (Throwable realCause)
    {
        t.initCause(realCause);
        throw t;
    }
}
```

8.6.9. Преодоление ограничения на обработку проверяемых исключений

Как гласит основополагающий принцип обработки исключений в Java, для всех проверяемых исключений должны быть предоставлены соответствующие обработчики. Но это ограничение можно преодолеть, используя обобщения. И главным средством для достижения этой цели служит следующий метод:

```
@SuppressWarnings("unchecked")
public static <T extends Throwable>
    void throwAs(Throwable e) throws T
{
    throw (T) e;
}
```

Допустим, этот метод содержится в интерфейсе `Task`. Если сделать следующий вызов при наличии проверяемого исключения `e`:

```
Task.<RuntimeException>throwAs(e);
```

то компилятор посчитает `e` непроверяемым исключением. В следующем фрагменте кода все исключения будут расценены компилятором как непроверяемые:

```
try
{
    выполнить нужные действия
}
catch (Throwable t)
{
    Task.<RuntimeException>throwAs(t);
}
```

Воспользуемся этим обстоятельством, чтобы разрешить досадное недоразумение. В частности, чтобы выполнить прикладной код в потоке исполнения, его придется расположить в теле метода `run()` из класса, реализующего интерфейс `Runnable`. Но ведь этому методу не разрешается генерировать проверяемые исключения. Поэтому предоставим для интерфейса `Runnable` адаптер из интерфейса `Task`, методу `run()` которого разрешается генерировать произвольные исключения:

```
interface Task
{
    void run() throws Exception;
    @SuppressWarnings("unchecked")
    static <T extends Throwable> void throwAs(Throwable t)
        throws T
    {
        throw (T) t;
    }
    static Runnable asRunnable(Task task)
    {
        return () ->
        {
            try
            {
                task.run();
            }
            catch (Exception e)
            {
                Task.<RuntimeException>throwAs(e);
            }
        };
    }
}
```

В следующем примере программы исполняется поток, в котором будет сгенерировано проверяемое исключение:

```
public class Test
{
    public static void main(String[] args)
    {
        var thread = new Thread(Task.asRunnable(() ->
        {
            Thread.sleep(1000);
            System.out.println("Hello, World!");
            throw new Exception("Check this out!");
        }));
        thread.start();
    }
}
```

В данном примере метод `Thread.sleep()` объявляется для генерирования исключения типа `InterruptedException`, которое больше не требуется перехватывать. Это исключение не будет сгенерировано, поскольку исполнение потока не прерывается. Тем не менее в данной программе генерируется проверяемое исключение. В результате выполнения данной программы будет получена трассировка стека.

Что же в этом такого особенного? Как правило, все исключения, возникающие в методе `run()` исполняемого потока, приходится перехватывать и заключать в оболочку непроверяемых исключений. Ведь метод `run()` объявляется без генерирования проверяемых исключений. Но в данном примере эти исключения не заключаются в оболочку. Вместо этого просто генерируется исключение, которое компилятор вынужден не воспринимать как проверяемое. С помощью обобщенных классов, стирания типов и аннотации `@SuppressWarnings` в данном примере удалось преодолеть весьма существенное ограничение, накладываемое системой типов в Java.

8.6.10. Остерегайтесь конфликтов после стирания типов

Не допускается создавать условия, приводящие к конфликтам после стирания обобщенных типов. Допустим, в обобщенный класс `Pair` вводится метод `equals()`:

```
public class Pair<T>
{
    public boolean equals(T value)
    { return first.equals(value) && second.equals(value); }
    . . .
}
```

Рассмотрим далее класс `Pair<String>`. По существу, в нем имеются два метода `equals()`:

```
boolean equals(String) // определен в обобщенном
                        // классе Pair<T>
boolean equals(Object) // унаследован от класса Object
```

Но интуиция вводит в заблуждение. В результате стирания типов метод `boolean equals(T)` становится методом `boolean equals(Object)`, который вступает в конфликт с методом `Object.equals()`. Разумеется, для разрешения подобного конфликта следует переименовать метод, из-за которого возникает конфликт.

В описании обобщений приводится другое правило: “Для поддержки преобразования путем стирания типов накладывается следующее ограничение: класс или переменная типа не могут одновременно быть подтипами двух типов, представляющих разные виды параметризации одного и того же интерфейса”. Например, следующий код недопустим:

```
class Employee implements Comparable<Employee> { . . . }
class Manager extends Employee
    implements Comparable<Manager>
{ . . . } // ОШИБКА!
```

В этом коде класс `Manager` должен одновременно реализовать оба типа, `Comparable<Employee>` и `Comparable<Manager>`, представляющие разные виды параметризации одного и того же интерфейса. Неясно, каким образом это ограничение согласуется со стиранием типов. Ведь вполне допустим следующий необобщенный вариант реализации интерфейса `Comparable`:

```
class Employee implements Comparable { . . . }
class Manager extends Employee implements Comparable
{ . . . }
```

Причина недоразумения намного менее явная, поскольку может произойти конфликт с синтезированными мостовыми методами. Класс, реализующий обобщенный интерфейс `Comparable<X>`, получает приведенный ниже мостовой метод. Но дело в том, что наличие двух таких методов с разными обобщенными типами `X` не допускается.

```
public int compareTo(Object other)
{ return compareTo((X) other); }
```

8.7. Правила наследования обобщенных типов

Для правильного обращения с обобщенными классами необходимо усвоить ряд правил, касающихся наследования и подтипов. Начнем с ситуации, которую многие программисты считают интуитивно понятной. Рассмотрим в качестве примера класс `Employee` и подкласс `Manager`. Является ли обобщенный класс `Pair<Manager>` подклассом, производным от обобщенного класса `Pair<Employee>`? Как ни странно, не является. Например, следующий код не подлежит компиляции:

```
Manager[] topHonchos = . . . ;
Pair<Employee> result =
    ArrayAlg.minmax(topHonchos); // ОШИБКА!
```

Метод `minmax()` возвращает объект типа `Pair<Manager>`, но не тип `Pair<Employee>`, а присваивать их друг другу недопустимо. В общем случае между классами `Pair<S>` и `Pair<T>` нет никаких отношений наследования, как бы ни соотносились друг с другом обобщенные типы `S` и `T` (рис. 8.1).

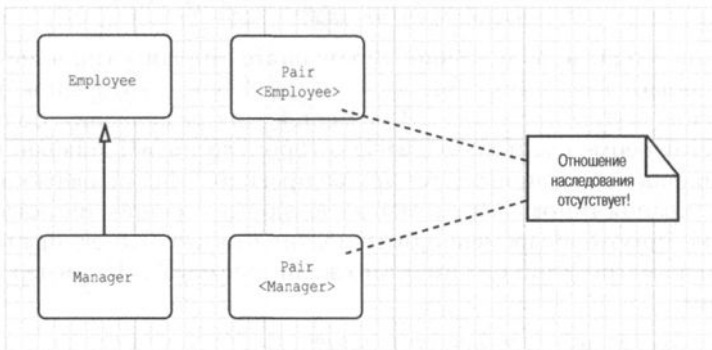


Рис. 8.1. Отношение наследования между обобщенными классами `Pair` отсутствует

Такое ограничение может показаться слишком строгим, но оно необходимо для соблюдения типовой безопасности. Допустим, объект класса `Pair<Manager>` все же разрешается преобразовать в объект класса `Pair<Employee>`, как показано в приведенном ниже фрагменте кода.

```
Pair<Manager> managerBuddies = new Pair<>(ceo, cfo);
Pair<Employee> employeeBuddies = managerBuddies;
```

```
// недопустимо, но предположим, что разрешено
employeeBuddies.setFirst(lowlyEmployee);
```

Даже если последний оператор в данном фрагменте кода и допустим, переменные `employeeBuddies` и `managerBuddies` все равно ссылаются на один и тот же объект. В итоге получается, что высшее руководство организации приравнивается к рядовым сотрудникам, что недопустимо для класса `Pair<Manager>`.



НА ЗАМЕТКУ! Выше было проведено очень важное отличие между обобщенными типами и массивами в Java. Так, массив `Manager[]` можно присвоить переменной типа `Employee[]` следующим образом:

```
Manager[] managerBuddies = { ceo, cfo };
Employee[] employeeBuddies = managerBuddies; // Допустимо!
```

Но массивы снабжены дополнительной защитой. Если попытаться сохранить объект рядового сотрудника в элементе массива `employeeBuddies[0]`, то виртуальная машина сгенерирует исключение типа `ArrayStoreException`.

Параметризованный тип можно всегда преобразовать в базовый. Например, `Pair<Employee>` — это подтип базового типа `Pair`. Такое преобразование необходимо для взаимодействия с унаследованным кодом. А можно ли преобразовать базовый тип и тем самым вызвать ошибку соблюдения типов? К сожалению, да. Рассмотрим следующий пример кода:

```
Pair<Manager> managerBuddies = new Pair<>(ceo, cfo);
Pair rawBuddies = managerBuddies; // Допустимо!
rawBuddies.setFirst(new File(". . .")); // допускается, но
// только с предупреждением во время компиляции
```

Такой код, откровенно говоря, настораживает. Но в этом отношении ситуация оказывается не хуже, чем в прежних версиях Java. Защита виртуальной машины не безгранична. Когда внешний объект извлекается методом `getFirst()` и присваивается переменной типа `Manager`, генерируется исключение типа `ClassCastException`, как и в старые добрые времена. Но в этом случае код лишается дополнительной защиты, которую обычно предоставляет обобщенное программирование.

И, наконец, одни обобщенные классы могут расширять или реализовывать другие обобщенные классы. В этом отношении они ничем не отличаются от обычных классов. Например, обобщенный класс `ArrayList<T>` реализует обобщенный интерфейс `List<T>`. Это означает, что объект типа `ArrayList<Manager>` может быть преобразован в объект типа `List<Manager>`. Но, как было показано выше, объект типа `ArrayList<Manager>` — это не объект типа `ArrayList<Employee>` или `List<Employee>`. На рис. 8.2 схематически показаны все отношения наследования между этими обобщенными классами и интерфейсами.

8.8. Подстановочные типы

Исследователям систем типов уже давно известно, что пользоваться жесткими системами обобщенных типов весьма неприятно. Поэтому создатели Java придумали изящный (и тем не менее безопасный) выход из положения: *подстановочные типы*. В последующих разделах поясняется, как обращаться с подстановочными типами.

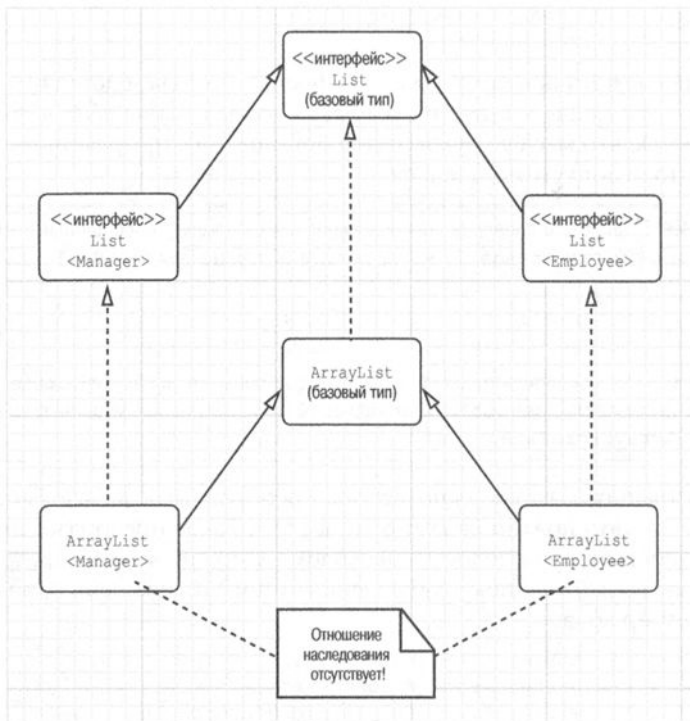


Рис. 8.2. Отношения наследования подтипов между обобщенными типами списков

8.8.1. Понятие подстановочного типа

В подстановочном типе параметр типа может быть переменным. Например, следующий подстановочный тип обозначает любой обобщенный тип `Pair`, параметр типа которого представляет подкласс, производный от класса `Employee`, в частности, класс `Pair<Manager>`, но не класс `Pair<String>`.

```
Pair<? extends Employee>
```

Допустим, требуется написать следующий метод, выводящий Ф.И.О. работников парами на экран:

```
public static void printBuddies(Pair<Employee> p)
{
    Employee first = p.getFirst();
    Employee second = p.getSecond();
    System.out.println(first.getName() + " and "
        + second.getName() + " are buddies.");
}
```

Как было показано в предыдущем разделе, передать объект типа `Pair<Manager>` этому методу нельзя, что не совсем удобно. Но из этого положения имеется простой выход — воспользоваться подстановочным типом следующим образом:

```
public static void printBuddies(Pair<? extends Employee> p)
```

Тип `Pair<Manager>` является подтипом `Pair<? extends Employee>` (рис. 8.3).

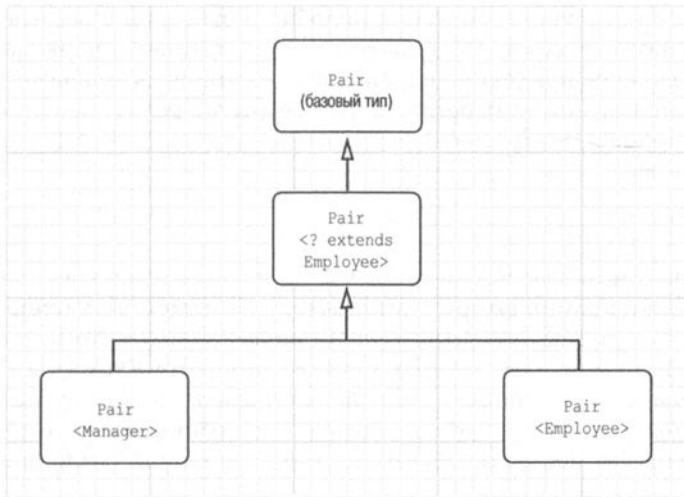


Рис. 8.3. Отношения наследования подтипов с подстановками

А могут ли подстановки нарушить тип `Pair<Manager>` по ссылке `Pair<? extends Employee>`? Не могут, как показано в приведенном ниже фрагменте кода.

```

Pair<Manager> managerBuddies = new Pair<>(ceo, cfo);
Pair<? extends Employee> wildcardBuddies =
    managerBuddies; // Допустимо!
wildcardBuddies.setFirst(lowlyEmployee);
// Ошибка при компиляции!

```

Но в то же время вызов метода `setFirst()` приведет к ошибке соблюдения типов. Чтобы стала понятнее причина этой ошибки, проанализируем тщательнее обобщенный класс типа `Pair<? extends Employee>`. У него имеются следующие методы:

```

? extends Employee getFirst()
void setFirst(? extends Employee)

```

Это делает невозможным вызов метода `setFirst()`. Компилятору требуется какой-нибудь подтип `Employee`, но неизвестно, какой именно. Он отказывается передать любой конкретный тип, поскольку знак подстановки `?` может и не совпасть с этим типом. При вызове метода `getFirst()` такое затруднение не возникает. Значение, возвращаемое методом `getFirst()`, вполне допустимо присвоить переменной ссылки на объект типа `Employee`. В этом, собственно, и состоит главный смысл ограниченных подстановок. С их помощью можно теперь отличать безопасные методы доступа от небезопасных модифицирующих методов.

8.8.2. Ограничения супертипа на подстановки

Ограничения на подстановки подобны ограничениям на переменные типа. Но у них имеется дополнительная возможность — определить *ограничение супертипа* следующим образом:

```
? super Manager
```

Такая подстановка ограничивается всеми супертипами `Manager`. (По удачному стечению обстоятельств существующего в Java ключевого слова `super` оказалось

достаточно для столь точного описания подобного отношения наследования.) А зачем это может понадобиться? Подстановка с ограничением супертипа дает поведение, противоположное поведению подстановочных типов, описанному в разделе 8.8. В частности, методом можно передавать параметры, но нельзя использовать возвращаемые ими значения. Например, в классе типа `Pair<? super Manager>` имеются следующие методы:

```
void setFirst(? super Manager)
? super Manager getFirst()
```

И хотя это на самом деле не синтаксис Java, тем не менее, он наглядно показывает, что именно известно компилятору. Компилятору не может быть точно известен тип метода `setFirst()`, и поэтому он не может допустить вызов этого метода с любым объектом типа `Employee` или `Object` в качестве аргумента. Он способен лишь передать объект типа `Manager` или его подтипа, например `Executive`. Более того, если вызывается метод `getFirst()`, то нет никакой гарантии относительно типа возвращаемого объекта. Его можно присвоить только переменной ссылки на объект типа `Object`.

Рассмотрим типичный пример. Допустим, имеется массив руководящих работников организации и сведения о минимальном и максимальном размере премии одного из них требуется ввести в объект класса `Pair`. К какому же типу относится класс `Pair`? Будет справедливо, если это окажется тип `Pair<Employee>` или `Pair<Object>` (рис. 8.4). Следующий метод примет любой подходящий объект типа `Pair` в качестве своего параметра:

```
public static void minmaxBonus(Manager[] a,
                               Pair<? super Manager> result)
{
    if (a == null || a.length == 0) return;
    Manager min = a[0];
    Manager max = a[0];
    for (int i = 1; i < a.length; i++)
    {
        if (min.getBonus() > a[i].getBonus()) min = a[i];
        if (max.getBonus() < a[i].getBonus()) max = a[i];
    }
    result.setFirst(min);
    result.setSecond(max);
}
```

Если рассуждать интуитивно, то подстановки с ограничениями супертипа позволяют записывать данные в обобщенный объект, а подстановки с ограничениями подтипа — читать данные из обобщенного объекта. Рассмотрим другой пример наложения ограничений супертипа. Интерфейс `Comparable` сам является обобщенным и объявляется следующим образом:

```
public interface Comparable<T>
{
    public int compareTo(T other);
}
```

где переменная типа обозначает тип параметра `other`. Например, класс `String` реализует интерфейс `Comparable<String>`, а его метод `compareTo()` объявляется следующим образом:

```
public int compareTo(String other)
```

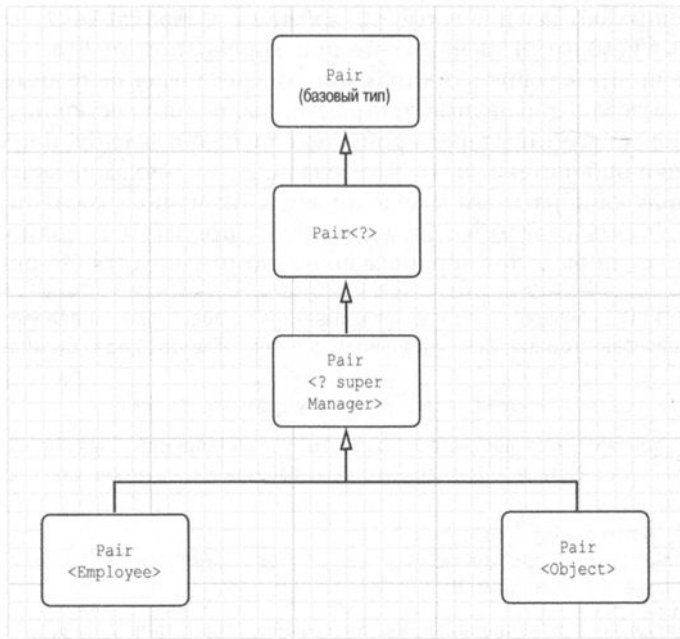


Рис. 8.4. Подстановка с ограничением супертипа

И это хорошо. Ведь явно задаваемый параметр имеет правильный тип. До появления обобщений параметр `other` относился к типу `Object`, и поэтому в реализации данного метода требовалось приведение типов. А теперь интерфейс `Comparable` относится к обобщенному типу, что позволяет внести следующее усовершенствование в объявление метода `min()` в классе `ArrayList`:

```
public static <T extends Comparable<T>> T min(T[] a)
```

Такое объявление метода оказывается более основательным, чем простое расширение типа `T extends Comparable`, и должно вполне подходить для многих классов. Так, если определяется минимальное значение в массиве типа `String`, то обобщенный тип `T` становится типом `String`, а тот — подтипом `Comparable<String>`. Но при обработке массива объектов типа `LocalDate` возникает затруднение. Оказывается, что класс `LocalDate` реализует интерфейс `ChronoLocalDate`, а тот расширяет интерфейс `Comparable<ChronoLocalDate>`. Поэтому и класс `LocalDate` реализует интерфейс `Comparable<ChronoLocalDate>`, а не интерфейс `Comparable<LocalDate>`. В подобных случаях на помощь приходят супертипы, как показано ниже.

```
public static <T extends Comparable<? super T>>
    T min(T[] a) . . .
```

Теперь метод `compareTo()` имеет следующую форму:

```
int compareTo(? super T)
```

Возможно, он объявляется для принятия в качестве параметра объекта обобщенного типа `T` или даже супертипа `T`, если, например, `T` обозначает тип `GregorianCalendar`. Как бы то ни было, методу `compareTo()` можно безопасно передать объект обобщенного типа `T`.

Непосвященных объявление вроде `<T extends Comparable<? super T>>` может повергнуть в невольный трепет. И это прискорбно, потому что такое объявление призвано помочь прикладным программистам, исключая ненужные ограничения на параметры вызова. Прикладные программисты, не интересующиеся обобщениями, вероятно, предпочтут поскорее пропустить такие объявления и принять на веру, что разработчики библиотеки знали, что делали. А тем, кто занимается разработкой библиотек, придется обратиться к подстановкам, иначе пользователи их библиотек помянут их недобрым словом и вынуждены будут прибегать в своем коде к разного рода приведению типов до тех пор, пока он не скомпилируется без ошибок.



НА ЗАМЕТКУ! Ограничения супертипа накладываются на подстановки и при указании аргумента типа функционального интерфейса. Например, в интерфейсе `Collection` имеется следующий метод:

```
default boolean removeIf(Predicate<? super E> filter)
```

Этот метод удаляет из коллекции все элементы, удовлетворяющие заданному предикату. Так, если не нравятся нечетные хеш-коды некоторых работников, их можно удалить следующим образом:

```
ArrayList<Employee> staff = . . .;
Predicate<Object> oddHashCode = obj -> obj.hashCode() % 2 != 0;
staff.removeIf(oddHashCode);
```

В данном случае функциональному интерфейсу требуется передать аргумент типа `Predicate<Object>`, а не просто `Predicate<Employee>`. И это позволяет сделать ограничение `super` на подстановку.

8.8.3. Неограниченные подстановки

Подстановками можно пользоваться вообще без ограничений, например `Pair<?>`. На первый взгляд этот тип похож на базовый тип `Pair`. На самом же деле типы отличаются. В классе типа `Pair<?>` имеются методы, аналогичные приведенным ниже.

```
? getFirst()
void setFirst(?)
```

Значение, возвращаемое методом `getFirst()`, может быть присвоено только переменной ссылки на объект типа `Object`. А метод `setFirst()` вообще нельзя вызывать — даже с параметром типа `Object`. Существенное отличие типов `Pair<?>` и `Pair` в том и состоит, что метод `setObject()` из класса базового типа `Pair` нельзя вызывать с *любым* объектом типа `Object`.



НА ЗАМЕТКУ! Тем не менее можно сделать вызов `setFirst(null)`.

А зачем вообще может понадобиться такой непонятный тип? Оказывается, он удобен для выполнения очень простых операций. Например, в приведенном ниже методе проверяется, содержит ли пара пустую ссылку на объект. Такому методу вообще не требуется конкретный тип.

```
public static boolean hasNulls(Pair<?> p)
{
    return p.getFirst() == null || p.getSecond() == null;
}
```

Применения подстановочного типа можно было бы избежать, превратив метод `contains()` в обобщенный, как показано ниже. Но его вариант с подстановочным типом выглядит более удобочитаемым.

```
public static <T> boolean hasNulls(Pair<T> p)
```

8.8.4. Захват подстановок

Рассмотрим следующий метод, меняющий местами составные элементы пары:

```
public static void swap(Pair<?> p)
```

Подстановка не является переменной типа, поэтому знак `?` нельзя указывать вместо типа. Иными словами, следующий фрагмент кода написан неверно:

```
? t = p.getFirst(); // ОШИБКА!
p.setFirst(p.getSecond());
p.setSecond(t);
```

В связи с этим возникает затруднение, поскольку при перестановке приходится временно запоминать первый составной элемент пары. Правда, это затруднение можно разрешить весьма любопытным способом, написав вспомогательный метод `swapHelper()` так, как показано ниже.

```
public static <T> void swapHelper(Pair<T> p)
{
    T t = p.getFirst();
    p.setFirst(p.getSecond());
    p.setSecond(t);
}
```

Следует, однако, иметь в виду, что метод `swapHelper()` — обобщенный, тогда как метод `swap()` — необобщенный. У него имеется фиксированный параметр типа `Pair<?>`. Теперь метод `swapHelper()` можно вызвать из метода `swap()` следующим образом:

```
public static void swap(Pair<?> p) { swapHelper(p); }
```

В данном случае параметр обобщенного типа `T` *захватывает* подстановку во вспомогательном методе `swapHelper()`. Неизвестно, какой именно тип обозначает подстановка, но это совершенно определенный тип. Поэтому определение метода `<T> swapHelper()` имеет конкретный смысл, когда обобщение `T` обозначает этот тип.

Безусловно, пользоваться в данном случае подстановкой совсем не обязательно. Вместо этого можно было бы напрямую реализовать метод `<T> void swap(Pair<T> p)` как обобщенный и без подстановок. Но рассмотрим следующий пример, в котором подстановочный тип появляется естественным образом посреди вычислений:

```
public static void maxminBonus(
    Manager[] a, Pair<? super Manager> result)
{
    minmaxBonus(a, result);
    PairAlg.swap(result); // Допустимо, поскольку в
    // методе swapHelper() захватывается подстановочный тип
}
```

В данном случае обойтись без механизма захвата подстановки нельзя. Захват подстановки допускается в очень ограниченных случаях. Компилятор должен иметь возможность гарантировать, что подстановка представляет один, вполне определенный

тип. Например, обобщение `T` в объявлении класса `ArrayList<Pair<T>>` вообще не сможет захватить подстановку в объявлении класса `ArrayList<Pair<?>>`. Списочный массив может содержать два элемента типа `Pair<?>`, в каждом из которых вместо знака `?` будет подставлен свой тип.

Пример тестовой программы из листинга 8.3 подытоживает разнообразные способы обращения с обобщениями, рассмотренные в предыдущих разделах, чтобы продемонстрировать их непосредственно в коде.

Листинг 8.3. Исходный код из файла `pair3/PairTest3.java`

```
1 package pair3;
2
3 /**
4  * @version 1.01 2012-01-26
5  * @author Cay Horstmann
6  */
7 public class PairTest3
8 {
9     public static void main(String[] args)
10    {
11        Manager ceo = new Manager("Gus Greedy", 800000,
12                                   2003, 12, 15);
13        Manager cfo = new Manager("Sid Sneaky", 600000,
14                                   2003, 12, 15);
15        Pair<Manager> buddies = new Pair<>(ceo, cfo);
16        printBuddies(buddies);
17
18        ceo.setBonus(1000000);
19        cfo.setBonus(500000);
20        Manager[] managers = { ceo, cfo };
21
22        Pair<Employee> result = new Pair<>();
23        minmaxBonus(managers, result);
24        System.out.println("first: "
25                             + result.getFirst().getName() + ", second: "
26                             + result.getSecond().getName());
27        maxminBonus(managers, result);
28        System.out.println("first: "
29                             + result.getFirst().getName() + ", second: "
30                             + result.getSecond().getName());
31    }
32
33    public static void printBuddies(
34        Pair<? extends Employee> p)
35    {
36        Employee first = p.getFirst();
37        Employee second = p.getSecond();
38        System.out.println(first.getName() + " and "
39                             + second.getName() + " are buddies.");
40    }
41
42    public static void minmaxBonus(Manager[] a,
43                                    Pair<? super Manager> result)
44    {
45        if (a == null || a.length == 0) return;
```

```
46     Manager min = a[0];
47     Manager max = a[0];
48     for (int i = 1; i < a.length; i++)
49     {
50         if (min.getBonus() > a[i].getBonus()) min = a[i];
51         if (max.getBonus() < a[i].getBonus()) max = a[i];
52     }
53     result.setFirst(min);
54     result.setSecond(max);
55 }
56 public static void maxminBonus(Manager[] a,
57                               Pair<? super Manager> result)
58 {
59     minmaxBonus(a, result);
60     PairAlg.swapHelper(result); // в методе swapHelper()
61                               // захватывается подстановочный тип
62 }
63 }
64
65 class PairAlg
66 {
67     public static boolean hasNulls(Pair<?> p)
68     {
69         return p.getFirst() == null
70             || p.getSecond() == null;
71     }
72
73     public static void swap(Pair<?> p)
74     { swapHelper(p); }
75
76     public static <T> void swapHelper(Pair<T> p)
77     {
78         T t = p.getFirst();
79         p.setFirst(p.getSecond());
80         p.setSecond(t);
81     }
82 }
```

8.9. Рефлексия и обобщения

С помощью рефлексии произвольные объекты можно анализировать во время выполнения. Если же объекты являются экземплярами обобщенных классов, то с помощью рефлексии удастся получить немного сведений о параметрах обобщенного типа, поскольку они стираются. Тем не менее в последующих разделах поясняется, какие сведения об обобщенных классах все-таки позволяет выявить рефлексия.

8.9.1. Обобщенный класс Class

Класс Class теперь является обобщенным. Например, `String.class` — на самом деле объект (по существу, единственный) класса `Class<String>`. Параметр типа удобен, потому что он позволяет методам обобщенного класса `Class<T>` быть более точными в отношении возвращаемых типов. В следующих методах из класса `Class<T>` используются преимущества параметра типа:


```

newInstance()
T cast(Object obj)
T[] getEnumConstants()
Class<? super T> getSuperclass()
Constructor<T> getConstructor(Class... parameterTypes)
Constructor<T> getDeclaredConstructor(
    Class... parameterTypes)

```

Метод `newInstance()` возвращает экземпляр класса, полученный от конструктора по умолчанию. Его возвращаемый тип теперь может быть объявлен как `T`, т.е. тот же тип, что и у класса, описанного в обобщении `Class<T>`. Такой прием позволяет сэкономить на приведении типов.

Метод `cast()` возвращает данный объект, объявленный теперь как имеющий обобщенный тип `T`, если его тип действительно является подтипом `T`. В противном случае он генерирует исключение типа `BadCastException`. А метод `getEnumConstants()` возвращает пустое значение `null`, если данный класс не относится к типу перечислений или массивов перечислимых значений, о которых известно, что они принадлежат к обобщенному типу `T`.

И, наконец, методы `getConstructor()` и `getDeclaredConstructor()` возвращают объект обобщенного класса `Constructor<T>`. Класс `Constructor` также стал обобщенным, поэтому у метода `newInstance()` теперь имеется правильный возвращаемый тип.

java.lang.Class<T> 1.0

- **T newInstance()**
Возвращает новый экземпляр, созданный конструктором по умолчанию.
- **T cast(Object obj)**
Возвращает заданный объект *obj*, если он пустой [`null`] или может быть преобразован в обобщенный тип `T`, а иначе генерирует исключение типа `BadCastException`.
- **T[] getEnumConstants()** 5
Возвращает массив всех значений, если `T` — перечислимый тип, а иначе — пустое значение `null`.
- **Class<? super T> getSuperclass()**
Возвращает суперкласс данного класса или пустое значение `null`, если обобщенный тип `T` не обозначает класс или же обозначает класс `Object`.
- **Constructor<T> getConstructor(Class... parameterTypes)** 1.1
- **Constructor<T> getDeclaredConstructor(Class... parameterTypes)** 1.1
Получают открытый конструктор или же конструктор с заданными типами параметров.

java.lang.reflect.Constructor<T> 1.1

- **T newInstance(Object... parameters)**
Возвращает новый экземпляр, созданный конструктором с заданными параметрами.

8.9.2. Сопоставление типов с помощью параметров `Class<T>`

Иногда оказывается полезно сопоставить переменную типа параметра `Class<T>` в обобщенном методе. Ниже приведен характерный тому пример.

```
public static <T> Pair<T> makePair(Class<T> c)
    throws InstantiationException, IllegalAccessException
{
    return new Pair<>(c.newInstance(), c.newInstance());
}
```

Если сделать приведенный ниже вызов, то параметр `Employee.class` будет означать объект типа `Class<Employee>`. Параметр обобщенного типа `T` в методе `makePair()` сопоставляется с классом `Employee`, откуда компилятор может заключить, что данный метод возвращает объект типа `Pair<Employee>`.

```
makePair(Employee.class)
```

8.9.3. Сведения об обобщенных типах в виртуальной машине

Одной из примечательных особенностей обобщений в Java является стирание обобщенных типов в виртуальной машине. Как ни странно, но классы, подвергшиеся стиранию типов, все еще сохраняют некоторую память о своем обобщенном происхождении. Например, базовому классу `Pair` известно, что он происходит от обобщенного класса `Pair<T>`, несмотря на то, что объект типа `Pair` не может отличить, был ли он сконструирован как объект типа `Pair<String>` или как объект типа `Pair<Employee>`.

Аналогично, следующий метод:

```
public static Comparable min(Comparable[] a)
```

является результатом стирания типов в приведенном ниже обобщенном методе.

```
public static <T extends Comparable<? super T>>
    T min(T[] a)
```

Прикладной интерфейс API для рефлексии можно использовать, чтобы определить следующее.

- Имеет ли обобщенный метод параметр типа `T`.
- Имеет ли параметр типа ограниченный подтип, который сам является обобщенным.
- Имеет ли ограничивающий тип подставляемый параметр.
- Имеет ли подставляемый параметр ограниченный супертип.
- Имеет ли обобщенный метод обобщенный массив в качестве своего параметра.

Иными словами, с помощью рефлексии можно реконструировать все, что было объявлено в реализации обобщенных классов и методов. Но вряд ли можно узнать, каким образом разрешались параметры типа для конкретных объектов и вызовов методов.

Чтобы выразить объявления обобщенных типов, следует воспользоваться интерфейсом `Type`. У этого интерфейса имеются следующие подтипы.

- Класс `Class`, описывающий конкретные типы.
- Интерфейс `TypeVariable`, описывающий переменные типа (например, `Textends Comparable<? super T>`).

- Интерфейс `WildcardType`, описывающий подстановки (например, `? super T`).
- Интерфейс `ParameterizedType`, описывающий обобщенный класс или типы интерфейсов (например, `Comparable<? super T>`).
- Интерфейс `GenericArrayType`, описывающий обобщенные массивы (например, `T[]`).

На рис. 8.5 схематически показана иерархия наследования интерфейса `Type`. Обратите внимание на то, что последние четыре подтипа являются интерфейсами. Виртуальная машина создает экземпляры подходящих классов, реализующих эти интерфейсы.

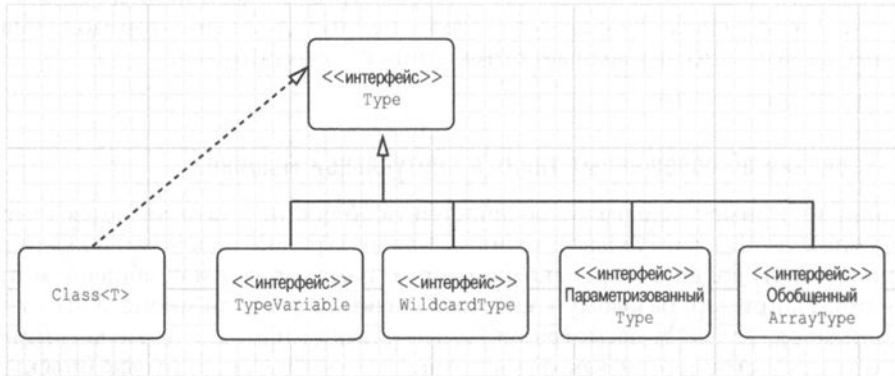


Рис. 8.5. Иерархия наследования интерфейса `Type`

В исходном коде примера программы из листинга 8.4 вывод сведений о данном классе на экран организуется с помощью прикладного интерфейса API для рефлексии обобщений. Если выполнить эту программу вместе с классом `Pair`, то в результате будет получен следующий отчет:

```

class Pair<T> extends java.lang.Object
public T getFirst()
public T getSecond()
public void setFirst(T)
public void setSecond(T)
  
```

А если выполнить эту программу вместе с классом `ArrayAlg` в каталоге `PairTest2`, то в отчете появится следующий метод:

```

public static <T extends java.lang.Comparable>
    Pair<T> minmax(T[])
  
```

Листинг 8.4. Исходный код из файла `genericReflection/GenericReflectionTest.java`

```

1  package genericReflection;
2
3  import java.lang.reflect.*;
4  import java.util.*;
5
6  /**
7   * @version 1.11 2018-04-10
  
```

```
8  * @author Cay Horstmann
9  */
10 public class GenericReflectionTest
11 {
12     public static void main(String[] args)
13     {
14         // прочитать имя класса из аргументов командной
15         // строки или введенных пользователем данных
16         String name;
17         if (args.length > 0) name = args[0];
18         else
19         {
20             try (var in = new Scanner(System.in))
21             {
22                 System.out.println("Enter class name (e.g.,
23                                     java.util.Collections): ");
24                 name = in.next();
25             }
26         }
27
28         try
29         {
30             // вывести обобщенные сведения о классе
31             // и его открытых методах
32             Class<?> cl = Class.forName(name);
33             printClass(cl);
34             for (Method m : cl.getDeclaredMethods())
35                 printMethod(m);
36         }
37         catch (ClassNotFoundException e)
38         {
39             e.printStackTrace();
40         }
41     }
42
43     public static void printClass(Class<?> cl)
44     {
45         System.out.print(cl);
46         printTypes(cl.getTypeParameters(), "<", " ", ">",
47                     true);
48         Type sc = cl.getGenericSuperclass();
49         if (sc != null)
50         {
51             System.out.print(" extends ");
52             printType(sc, false);
53         }
54         printTypes(cl.getGenericInterfaces(),
55                     " implements ", " ", " ", false);
56         System.out.println();
57     }
58
59     public static void printMethod(Method m)
60     {
61         String name = m.getName();
62         System.out.print(Modifier.toString(
63                             m.getModifiers()));
64         System.out.print(" ");
```

```
65     printTypes(m.getTypeParameters(),
66                "<", " ", "> ", true);
67
68     printType(m.getGenericReturnType(), false);
69     System.out.print(" ");
70     System.out.print(name);
71     System.out.print("(");
72     printTypes(m.getGenericParameterTypes(),
73                "", " ", " ", false);
74     System.out.println(")");
75 }
76
77 public static void printTypes(Type[] types,
78                               String pre, String sep, String suf,
79                               boolean isDefinition)
80 {
81     if (pre.equals(" extends ") && Arrays.equals(types,
82            new Type[] { Object.class }))
83         return;
84     if (types.length > 0) System.out.print(pre);
85     for (int i = 0; i < types.length; i++)
86     {
87         if (i > 0) System.out.print(sep);
88         printType(types[i], isDefinition);
89     }
90     if (types.length > 0) System.out.print(suf);
91 }
92
93 public static void printType(Type type,
94                               boolean isDefinition)
95 {
96     if (type instanceof Class)
97     {
98         var t = (Class<?>) type;
99         System.out.print(t.getName());
100     }
101     else if (type instanceof TypeVariable)
102     {
103         var t = (TypeVariable<?>) type;
104         System.out.print(t.getName());
105         if (isDefinition)
106             printTypes(t.getBounds(),
107                        " extends ", " & ", "", false);
108     }
109     else if (type instanceof WildcardType)
110     {
111         var t = (WildcardType) type;
112         System.out.print("?");
113         printTypes(t.getUpperBounds(),
114                  " extends ", " & ", "", false);
115         printTypes(t.getLowerBounds(),
116                  " super ", " & ", "", false);
117     }
118     else if (type instanceof ParameterizedType)
119     {
120         var t = (ParameterizedType) type;
121         Type owner = t.getOwnerType();
```

```

122     if (owner != null)
123     {
124         printType(owner, false);
125         System.out.print(".");
126     }
127     printType(t.getRawType(), false);
128     printTypes(t.getActualTypeArguments(),
129               "<", " ", ">", false);
130 }
131 else if (type instanceof GenericArrayType)
132 {
133     var t = (GenericArrayType) type;
134     System.out.print("[");
135     printType(t.getGenericComponentType(),
136               isDefinition);
137     System.out.print("]");
138 }
139 }
140 }

```

8.9.4. Литералы типов

Иногда требуется управлять поведением программы по типу значения. Например, пользователю механизма сохранения требуется предоставить возможность указывать порядок сохранения объекта конкретного класса. Как правило, такая возможность реализуется связыванием объекта типа `Class` с конкретным действием.

Но в связи со стиранием типов в обобщенных классах возникает следующий вопрос: как, например, обеспечить разные действия для классов `ArrayList<Integer>` и `ArrayList<String>`, в которых стирается один и тот же базовый тип `ArrayList`? Ответить на этот вопрос позволяет специальный прием, способный разрешить подобное затруднение в некоторых случаях. Этот прием состоит в том, чтобы зафиксировать экземпляр упоминавшегося ранее интерфейса `Type`. Для этого следует создать анонимный подкласс, как показано ниже.

```

var type = new TypeLiteral<ArrayList<Integer>>(){}
// обратите внимание на фигурные скобки {}

```

Конструктор класса `TypeLiteral` фиксирует обобщенный подтип, как демонстрируется в следующем примере кода:

```

class TypeLiteral
{
    public TypeLiteral()
    {
        Type parentType = getClass().getGenericSuperclass();
        if (parentType instanceof ParameterizedType)
        {
            type = ((ParameterizedType) parentType)
                    .getActualTypeArguments()[0];
        }
        else
            throw new UnsupportedOperationException(
                "Construct as new TypeLiteral< . . >(){}");
    }
    . . .
}

```

Если во время выполнения имеется обобщенный тип, его можно сопоставить с типом `TypeLiteral`. Обобщенный тип нельзя получить из конкретного объекта, поскольку он стирается. Но, как пояснялось в предыдущем разделе, обобщенные типы полей и параметров методов сохраняются в виртуальной машине.

Литералы типов употребляются в таких каркасах внедрения зависимостей, как CDI и Guice, для управления внедрением зависимостей обобщенных типов. В программе из листинга 8.5 демонстрируется более простой пример употребления литералов типов. Так, если имеется конкретный объект, то можно перечислить те его поля, обобщенные типы которых оказываются доступными, а также обнаружить соответствующие форматирующие действия. В частности, объект типа `ArrayList<Integer>` форматируется разделением значений пробелами, а объект типа `ArrayList<Character>` — присоединением символов к строке. Любые другие списочные массивы форматируются методом `ArrayList.toString()`.

Листинг 8.5. Исходный код из файла `genericReflection/TypeLiterals.java`

```
1  package genericReflection;
2
3  /**
4   * @version 1.01 2018-04-10
5   * @author Cay Horstmann
6   */
7
8  import java.lang.reflect.*;
9  import java.util.*;
10 import java.util.function.*;
11
12 /**
13  * Литерал типа описывает тип, который может быть
14  * обобщенным, например, ArrayList<String>.
15  */
16 class TypeLiteral<T>
17 {
18     private Type type;
19
20     /**
21      * Этот конструктор должен вызываться из анонимного
22      * подкласса как в операции new TypeLiteral< . . .>() {}
23      */
24     public TypeLiteral()
25     {
26         Type parentType = getClass().getGenericSuperclass();
27         if (parentType instanceof ParameterizedType)
28         {
29             type = ((ParameterizedType) parentType)
30                 .getActualTypeArguments()[0];
31         }
32         else
33             throw new UnsupportedOperationException(
34                 "Construct as new TypeLiteral< . . .>() {}");
35     }
36
37     private TypeLiteral(Type type)
38     {
```

```
39     this.type = type;
40 }
41
42 /**
43  * Выдает литерал типа, описывающий заданный тип
44  */
45 public static TypeLiteral<?> of(Type type)
46 {
47     return new TypeLiteral<Object>(type);
48 }
49
50 public String toString()
51 {
52     if (type instanceof Class)
53         return ((Class<?>) type).getName();
54     else
55         return type.toString();
56 }
57
58 public boolean equals(Object otherObject)
59 {
60     return otherObject instanceof TypeLiteral
61         && type.equals(((TypeLiteral<?>)
62             otherObject).type);
63 }
64
65 public int hashCode()
66 {
67     return type.hashCode();
68 }
69 }
70
71 /**
72  * Форматирует объекты по правилам, связывающим типы
73  * с функциями форматирования
74  */
75 class Formatter
76 {
77     private Map<TypeLiteral<?>, Function<?, String>>
78         rules = new HashMap<>();
79
80     /**
81      * Присоединяет правило форматирования к данному
82      * средству форматирования
83      * @param type Тип, к которому применяется
84      *           данное правило
85      * @param formatterForType Функция, форматирующая
86      *           объекты данного типа
87      */
88     public <T> void forType(TypeLiteral<T> type,
89         Function<T, String> formatterForType)
90     {
91         rules.put(type, formatterForType);
92     }
93
94     /**
95      * Форматирует все поля объекта по правилам
```



```

96     * данного средства форматирования
97     * @param obj an object
98     * @return Возвращает символьную строку с именами
99     *         всех полей и отформатированными значениями
100    */
101    public String formatFields(Object obj) throws
102        IllegalArgumentException, IllegalAccessException
103    {
104        var result = new StringBuilder();
105        for (Field f : obj.getClass().getDeclaredFields())
106        {
107            result.append(f.getName());
108            result.append("=");
109            f.setAccessible(true);
110            Function<?, String> formatterForType = rules.get(
111                TypeLiteral.of(f.getGenericType()));
112            if (formatterForType != null)
113            {
114                {
115                    // параметр formatterForType имеет тип ?,
116                    // поэтому его методу apply() ничего не
117                    // передается. Тип параметра приводится к
118                    // типу Object, чтобы можно было вызвать
119                    // данный метод
120                    @SuppressWarnings("unchecked")
121                    Function<Object, String> objectFormatter =
122                        (Function<Object, String>) formatterForType;
123                    result.append(objectFormatter
124                        .apply(f.get(obj)));
125                }
126                else
127                    result.append(f.get(obj).toString());
128                result.append("\n");
129            }
130        }
131        return result.toString();
132    }
133
134    public class TypeLiterals
135    {
136        public static class Sample
137        {
138            ArrayList<Integer> nums;
139            ArrayList<Character> chars;
140            ArrayList<String> strings;
141            public Sample()
142            {
143                nums = new ArrayList<>();
144                nums.add(42); nums.add(1729);
145                chars = new ArrayList<>();
146                chars.add('H'); chars.add('i');
147                strings = new ArrayList<>();
148                strings.add("Hello"); strings.add("World");
149            }
150        }
151
152        private static <T> String join(String separator,
153            ArrayList<T> elements)

```

```
154 {
155     var result = new StringBuilder();
156     for (T e : elements)
157     {
158         if (result.length() > 0)
159             result.append(separator);
160         result.append(e.toString());
161     }
162     return result.toString();
163 }
164
165 public static void main(String[] args)
166     throws Exception
167 {
168     var formatter = new Formatter();
169     formatter.forType(new
170         TypeLiteral<ArrayList<Integer>>() {},
171         lst -> join(" ", lst));
172     formatter.forType(new
173         TypeLiteral<ArrayList<Character>>() {},
174         lst -> "\"" + join("", lst) + "\"");
175     System.out.println(formatter.formatFields(
176         new Sample()));
177 }
178 }
```

java.lang.Class<T> 1.0

- **TypeVariable[] getTypeParameters() 5**
Получает переменные обобщенного типа, если этот тип был объявлен как обобщенный, а иначе — массив нулевой длины.
- **Type getGenericSuperclass() 5**
Получает обобщенный тип суперкласса, который был объявлен для этого типа, или же пустое значение `null`, если это тип `Object` или вообще не тип класса.
- **Type[] getGenericInterfaces() 5**
Получает обобщенные типы интерфейсов, которые были объявлены для этого типа, в порядке объявления, или массив нулевой длины, если данный тип не реализует интерфейсы.

java.lang.reflect.Method 1.1

- **TypeVariable[] getTypeParameters() 5**
- Получает переменные обобщенного типа, если этот тип был объявлен как обобщенный, а иначе — массив нулевой длины.
- **Type getGenericReturnType() 5**
Получает обобщенный возвращаемый тип, с которым был объявлен данный метод.
- **Type[] getGenericParameterTypes() 5**
Получает обобщенные типы параметров, с которыми был объявлен данный метод. Если у метода отсутствуют параметры, возвращается массив нулевой длины.

java.lang.reflect.TypeVariable 5

- **String getName()**
Получает имя переменной типа.
- **Type[] getBounds()**
Получает ограничения на подкласс для данной переменной типа или массив нулевой длины, если ограничения на переменную отсутствуют.

java.lang.reflect.WildcardType 5

- **Type[] getUpperBounds()**
Получает для данной переменной типа ограничения на подкласс, определяемые оператором **extends**, или массив нулевой длины, если ограничения на подкласс отсутствуют.
- **Type[] getLowerBounds()**
Получает для данной переменной типа ограничения на суперкласс, определяемые оператором **super**, а если ограничения на суперкласс отсутствуют, то возвращает массив нулевой длины.

java.lang.reflect.ParameterizedType 5

- **Type getRawType()**
Получает базовый тип для данного параметризованного типа.
- **Type[] getActualTypeArguments()**
Получает параметр типа, с которым был объявлен данный параметризованный тип.
- **Type getOwnerType()**
Получает тип внешнего класса, если данный тип — внутренний, а если это тип верхнего уровня, то возвращает пустое значение **null**.

java.lang.reflect.GenericArrayType 5

- **Type getGenericComponentType()**
Получает обобщенный тип компонента, с которым связан тип данного массива.

Теперь вы знаете, как пользоваться обобщенными классами и как программировать собственные обобщенные классы и методы, если возникнет такая потребность. Не менее важно и то, что вы знаете, как трактовать объявления обобщенных типов, которые вы можете встретить в документации на прикладной интерфейс API и в сообщениях об ошибках. А для того чтобы найти исчерпывающие ответы на вопросы, которые могут возникнуть у вас в связи с применением на практике механизма обобщений в Java, обратитесь к замечательному списку часто (и не очень часто) задаваемых вопросов по адресу <http://angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html>. В следующей главе будет показано, каким образом механизм обобщений применяется в каркасе коллекций в Java.

Коллекции

В этой главе...

- ▶ Каркас коллекций в Java
- ▶ Интерфейсы в каркасе коллекций
- ▶ Конкретные коллекции
- ▶ Отображения
- ▶ Представления и оболочки
- ▶ Алгоритмы
- ▶ Унаследованные коллекции

Выбираемые структуры данных могут заметно отличаться как в отношении реализации методов в “естественном стиле”, так и производительности. При этом возникают следующие вопросы.

- Требуется ли быстрый поиск среди тысяч (или даже миллионов) отсортированных элементов?
- Нужен ли быстрый ввод и удаление элементов в середине упорядоченной последовательности?
- Требуется ли устанавливать связи между ключами и значениями?

В этой главе будет показано, как реализовать средствами библиотеки Java традиционные структуры данных, без которых немислимо серьезное программирование. На специальностях, связанных с вычислительной техникой, как правило, предусмотрен курс по структурам данных, поэтому имеется немало литературы по этому важному предмету. Наше изложение отличается от такого курса. В частности, мы опустим теорию и перейдем непосредственно к практике, поясняя на конкретных примерах, как используются классы коллекций из стандартной библиотеки в прикладном коде.

9.1. Каркас коллекций в Java

В первоначальной версии Java предлагался лишь небольшой набор классов для наиболее употребительных структур данных: `Vector`, `Stack`, `Hashtable`, `BitSet`,

а также интерфейс `Enumeration`, предоставлявший абстрактный механизм для обращения к элементам, находящимся в произвольном контейнере. Безусловно, это было мудрое решение, ведь для реализации всеобъемлющей библиотеки классов коллекций требуется время и опыт.

После выпуска версии Java 1.2 разработчики осознали, что настало время создать полноценный набор структур данных. При этом они столкнулись со многими противоречивыми требованиями. Они стремились к тому, чтобы библиотека была компактной и простой в освоении. Для этого нужно было избежать сложности стандартной библиотеки шаблонов (STL) в C++, но в то же время позаимствовать обобщенные алгоритмы, впервые появившиеся в библиотеке STL. Кроме того, нужно было обеспечить совместимость унаследованных классов коллекций с новой архитектурой. И, как это случается со всеми разработчиками библиотек коллекций, им приходилось не раз делать нелегкий выбор, находя попутно немало оригинальных решений. В этом разделе мы рассмотрим основную структуру каркаса коллекций в Java, покажем, как применять их на практике, а также разъясним наиболее противоречивые их особенности.

9.1.1. Разделение интерфейсов и реализаций коллекций

Как это принято в современных библиотеках структур данных, в рассматриваемой здесь библиотеке коллекций Java *интерфейсы* и *реализации* разделены. Покажем, каким образом происходит это разделение, на примере хорошо известной структуры данных — *очереди*.

Интерфейс очереди определяет, что элементы можно добавлять в хвост очереди, удалять их в ее голове, а также выяснять, сколько элементов находится в очереди в данный момент. Очереди применяются в тех случаях, когда требуется накапливать объекты и извлекать их по принципу “первым пришел — первым обслужен” (рис. 9.1).

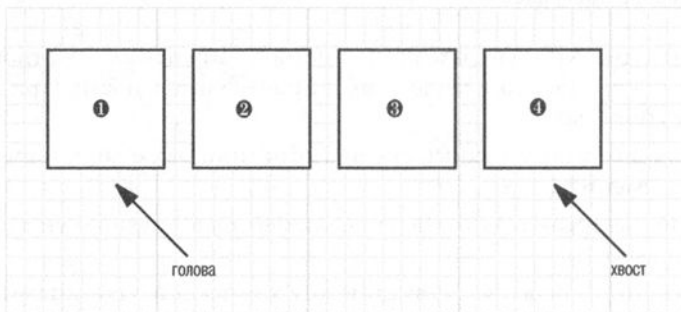


Рис. 9.1. Очередь

Самая элементарная форма интерфейса очереди может выглядеть так:

```
interface Queue<E> // простейшая форма интерфейса очереди
// из стандартной библиотеки
{
    void add(E element);
    E remove();
    int size();
}
```

Из интерфейса нельзя ничего узнать, каким образом реализована очередь. В одной из широко распространенных реализаций очереди применяется циклический массив, а в другой — связный список (рис. 9.2).

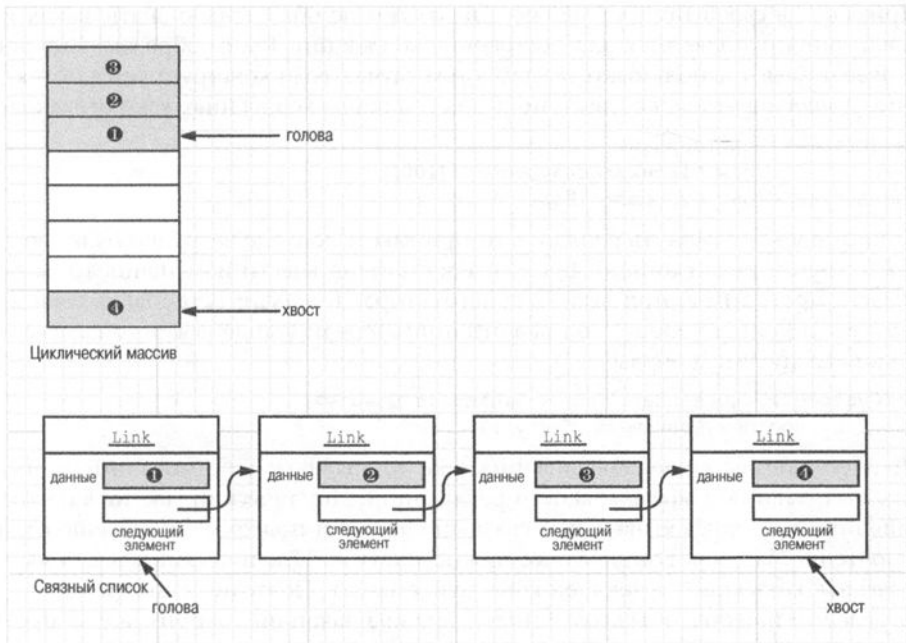


Рис. 9.2. Разные реализации очереди

Каждая реализация может быть выражена классом, реализующим интерфейс `Queue`, как показано ниже.

```
class CircularArrayQueue<E> implements Queue<E>
    // этот класс не из библиотеки
```

```
{
    CircularArrayQueue(int capacity) { . . . }
    public void add(E element) { . . . }
    public E remove() { . . . }
    public int size() { . . . }
    private E[] elements;
    private int head;
    private int tail;
}
```

```
class LinkedListQueue<E> implements Queue<E>
    // и этот класс не из библиотеки
```

```
{
    LinkedListQueue() { . . . }
    public void add(E element) { . . . }
    public E remove() { . . . }
    public int size() { . . . }
    private Link head;
    private Link tail;
}
```



НА ЗАМЕТКУ! Библиотека Java на самом деле не содержит классы `CircularArrayQueue` и `LinkedListQueue`. Они служат здесь лишь в качестве примера, чтобы пояснить принципиальное отличие интерфейса от реализации. Если же требуется организовать очередь на основе циклического массива, воспользуйтесь классом `ArrayDeque`, а для организации очереди в виде связанного списка — классом `LinkedList`, реализующим интерфейс `Queue`.

Применяя в своей программе очередь, совсем не обязательно знать, какая именно реализация используется для построения коллекции. Таким образом, конкретный класс имеет смысл использовать *только* в том случае, если конструируется объект коллекции. А *тип интерфейса* служит лишь для ссылки на коллекцию, как показано ниже.

```
Queue<Customer> expressLane =  
    new CircularArrayQueue<>(100);  
expressLane.add(new Customer("Harry"));
```

Если придется изменить решение, то при таком подходе нетрудно будет воспользоваться другой реализацией. Для этого достаточно внести изменения только в одном месте программы: при вызове конструктора. Так, если остановить свой выбор на классе `LinkedListQueue`, код реализации очереди в виде связанного списка будет выглядеть следующим образом:

```
Queue<Customer> expressLane = new LinkedListQueue<>();  
expressLane.add(new Customer("Harry"));
```

Почему выбирается одна реализация, а не другая? Ведь из самого интерфейса ничего нельзя узнать об эффективности реализации. Циклический массив в некотором отношении более эффективен, чем связный список, и поэтому ему обычно отдается предпочтение. Но, как всегда, за все нужно платить. Циклический массив является *ограниченной коллекцией*, имеющей конечную емкость. Поэтому если неизвестен верхний предел количества объектов, которые должна накапливать прикладная программа, то имеет смысл выбрать реализацию очереди на основе связанного списка.

Изучая документацию на прикладной интерфейс API, вы обнаружите другой набор классов, имена которых начинаются на **Abstract**, как, например, `AbstractQueue`. Эти классы предназначены для разработчиков библиотек. В том редком случае, когда вам потребуется реализовать свой собственный класс очереди, вы обнаружите, что сделать это проще, расширив класс `AbstractQueue`, чем реализовывать все методы из интерфейса `Queue`.

9.1.2. Интерфейс `Collection`

Основополагающим для классов коллекций в библиотеке Java является интерфейс `Collection`. В его состав входят два основных метода:

```
public interface Collection<E>  
{  
    boolean add(E element);  
    Iterator<E> iterator();  
    . . .  
}
```

В дополнение к ним имеется еще несколько методов, обсуждаемых далее в этой главе. Метод `add()` добавляет элемент в коллекцию. Он возвращает логическое значение `true`, если добавление элемента в действительности изменило коллекцию, а если коллекция осталась без изменения — логическое значение `false`. Так, если попытаться добавить объект в коллекцию, где такой объект уже имеется, вызов метода `add()` не даст желаемого результата, поскольку коллекция не допускает дублирование объектов. А метод `iterator()` возвращает объект класса, реализующего интерфейс `Iterator`. Объект итератора можно выбрать для обращения ко всем элементам коллекции по очереди. Подробнее итераторы обсуждаются в следующем разделе.

9.1.3. Итераторы

В состав интерфейса `Iterator` входят три метода:

```
public interface Iterator<E>
{
    E next();
    boolean hasNext();
    void remove();
}
```

Множественно вызывая метод `next()`, можно обратиться к каждому элементу коллекции по очереди. Но если будет достигнут конец коллекции, то метод `next()` сгенерирует исключение типа `NoSuchElementException`. Поэтому перед вызовом метода `next()` следует вызывать метод `hasNext()`. Этот метод возвращает логическое значение `true`, если у объекта итератора все еще имеются объекты, к которым можно обратиться. Если же требуется перебрать все элементы коллекции, то следует запросить итератор, продолжая вызывать метод `next()` до тех пор, пока метод `hasNext()` возвращает логическое значение `true`. В приведенном ниже примере показано, как все это реализуется непосредственно в коде.

```
Collection<String> c = . . .;
Iterator<String> iter = c.iterator();
while (iter.hasNext())
{
    String element = iter.next();
    сделать что-нибудь с элементом element
}
```

Тот же самый код можно написать более компактно, организовав цикл в стиле `for each` следующим образом:

```
for (String element : c)
{
    сделать что-нибудь с элементом element
}
```

Компилятор просто преобразует цикл в стиле `for each` в цикл с итератором. Цикл в стиле `for each` подходит для любых объектов, класс которых реализует интерфейс `Iterable` со следующим единственным методом:

```
public interface Iterable<E>
{
    Iterator<E> iterator();
}
```

Интерфейс `Collection` расширяет интерфейс `Iterable`. Поэтому цикл в стиле `for each` подходит для обращения к элементам любой коллекции из стандартной библиотеки.

Чтобы перебрать элементы коллекции, можно даже не организовывать цикл. Для этого достаточно вызвать метод `forEachRemaining()` с лямбда-выражением, где употребляется элемент коллекции. Это лямбда-выражение вызывается с каждым из элементов до тех пор, пока их больше не останется в коллекции:

```
iterator.forEachRemaining(element ->
    сделать что-нибудь с элементом element);
```


Порядок, в котором перебираются элементы, зависит от типа коллекции. Так, если осуществляется перебор элементов коллекции типа `ArrayList`, итератор начинает его с нулевого индекса, увеличивая последнее значение на каждом шаге итерации. Но если осуществляется перебор элементов коллекции типа `HashSet`, то они получаются в совершенно случайном порядке. Можно быть уверенным лишь в том, что за время итерации будут перебраны все элементы коллекции, хотя нельзя сделать никаких предположений относительно порядка их следования. Обычно это не особенно важно, потому что порядок не имеет значения при таких вычислениях, как, например, подсчет суммы или количества совпадений.



НА ЗАМЕТКУ! Программирующие на Java со стажем заметят, что методы `next()` и `hasNext()` из интерфейса `Iterator` служат той же цели, что и методы `nextElement()` и `hasMoreElements()` из интерфейса `Enumeration`. Разработчики библиотеки коллекций в Java могли бы отдать предпочтение интерфейсу `Enumeration`. Но им не понравились громоздкие имена методов, и поэтому они разработали новый интерфейс с более короткими именами.

Имеется принципиальное отличие между итераторами из библиотеки коллекций в Java и других библиотек. В традиционных библиотеках коллекций, как, например, `Standard Template Library` в C++, итераторы моделируются по индексам массива. Имея такой итератор, можно найти элемент, находящийся на данной позиции в массиве, подобно тому, как находится элемент массива `a[i]`, если имеется индекс `i`. Независимо от способа поиска элементов коллекции, итератор можно передвинуть на следующую позицию. Эта операция аналогична приращению индекса `i++` без поиска. Но итераторы в Java действуют иначе. Поиск элементов в коллекции и изменение их позиции тесно взаимосвязаны. Единственный способ найти элемент — вызвать метод `next()`, а в ходе поиска происходит переход на следующую позицию.

Напротив, итераторы в Java следует представлять себе так, как будто они находятся между элементами коллекции. Когда вызывается метод `next()`, итератор *перескакивает* следующий элемент и возвращает ссылку на тот элемент, который он только что прошел (рис. 9.3).

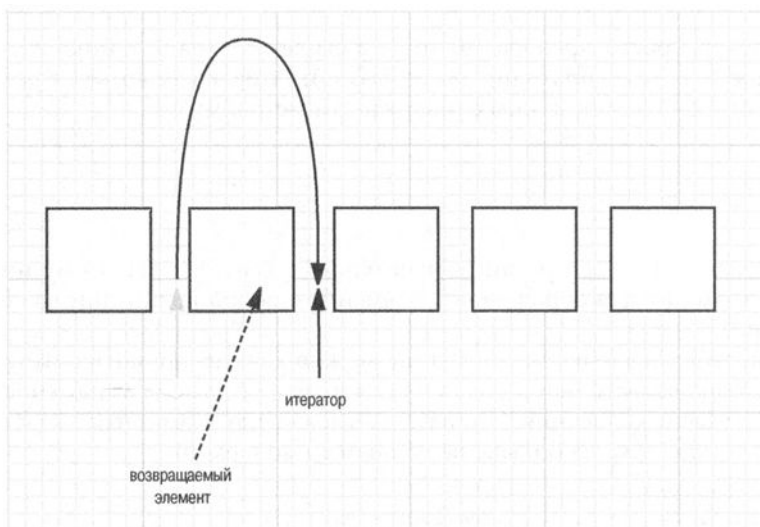


Рис. 9.3. Продвижение итератора по элементам коллекции



НА ЗАМЕТКУ! Можно прибегнуть к еще одной удобной аналогии, рассматривая объект `Iterator`. `next` в качестве эквивалента объекта `InputStream.read`. При вводе байта из потока этот байт автоматически потребляется. А при последующем вызове метода `read()` потребляется и возвращается следующий байт из потока ввода. Аналогично повторяющиеся вызовы метода `next()` позволяют ввести все элементы из коллекции.

Метод `remove()` из интерфейса `Iterator` удаляет элемент, который был возвращен при последнем вызове метода `next()`. Во многих случаях это имеет смысл, поскольку нужно проанализировать элемент, прежде чем решаться на его удаление. Но если требуется удалить элемент, находящийся на определенной позиции, то сначала придется его пройти. В приведенном ниже примере показано, как удалить первый элемент из коллекции символьных строк.

```
Iterator<String> it = c.iterator();
it.next(); // пройти первый элемент коллекции
it.remove(); // а теперь удалить его
```

Но важнее то, что между вызовами методов `next()` и `remove()` существует определенная связь. В частности, вызывать метод `remove()` не разрешается, если перед ним не был вызван метод `next()`. Если же попытаться сделать это, будет сгенерировано исключение типа `IllegalStateException`. А если из коллекции требуется удалить два соседних элемента, то нельзя просто вызвать метод `remove()` два раза подряд, как показано ниже.

```
it.remove();
it.remove(); // ОШИБКА!
```

Вместо этого нужно сначала вызвать метод `next()`, чтобы пройти удаляемый элемент, а затем удалить его, как следует из приведенного ниже примера кода.

```
it.remove();
it.next();
it.remove(); // Допустимо!
```

9.1.4. Обобщенные служебные методы

Интерфейсы `Collection` и `Iterator` являются обобщенными, а следовательно, можно написать служебные методы для обращения к разнотипным коллекциям. В качестве примера ниже приведен обобщенный служебный метод, проверяющий, содержит ли произвольная коллекция заданный элемент.

```
public static <E> boolean contains(Collection<E> c,
                                   Object obj)
{
    for (E element : c)
        if (element.equals(obj))
            return true;
    return false;
}
```

Разработчики библиотеки Java решили, что некоторые из этих служебных методов настолько полезны, что их следует сделать доступными из самой библиотеки. Таким образом, пользователи библиотеки избавлены от необходимости заново изобретать колесо. По существу, в интерфейсе `Collection` объявляется немало полезных методов, которые должны использоваться во всех реализующих его классах. К числу служебных методов относятся следующие:

```
int size()
boolean isEmpty()
boolean contains(Object obj)
boolean containsAll(Collection<?> c)
boolean equals(Object other)
boolean addAll(Collection<? extends E> from)
boolean remove(Object obj)
boolean removeAll(Collection<?> c)
void clear()
boolean retainAll(Collection<?> c)
Object[] toArray()
<T> T[] toArray(T[] arrayToFill)
```

Многие из этих методов самоочевидны, а подробнее о них можно узнать из короткого описания прикладного интерфейса API в конце этого раздела. Безусловно, было бы неудобно, если бы в каждом классе, реализующем интерфейс `Collection`, пришлось реализовывать такое количество служебных методов. Чтобы облегчить жизнь разработчикам, в библиотеке коллекций Java предоставляется класс `AbstractCollection`, где основополагающие методы `size()` и `iterator()` оставлены абстрактными, но реализованы служебные методы:

```
public abstract class AbstractCollection<E>
    implements Collection<E>
{
    . . .
    public abstract Iterator<E> iterator();

    public boolean contains(Object obj)
    {
        for (E element : this) // вызвать метод iterator()
            if (element.equals(obj))
                return = true;
        return false;
    }
    . . .
}
```

Конкретный класс коллекции теперь может расширить класс `AbstractCollection`. И тогда именно этот конкретный класс отвечает за реализацию метода `iterator()`, тогда как о служебном методе `contains()` уже позаботились в суперклассе `AbstractCollection`. Но если в его подклассе можно предложить более эффективный способ реализации служебного метода `contains()`, то ничто не мешает сделать это именно в нем.

Такой подход считается немного устаревшим. Было бы намного лучше, если бы упомянутые выше методы были объявлены по умолчанию в интерфейсе `Collection`, но этого, к сожалению, не произошло. Тем не менее в этот интерфейс было введено несколько методов по умолчанию. Большинство из них предназначено для обработки потоков данных, рассматриваемых во втором томе настоящего издания. Кроме того, для удаления элементов из коллекции по заданному условию имеется следующий удобный метод:

```
default boolean removeIf(Predicate<? super E> filter)
```

java.util.Collection<E> 1.2

- **Iterator<E> iterator()**
Возвращает итератор, который можно использовать для перебора элементов коллекции.
- **int size()**
Возвращает количество элементов, хранящихся в коллекции на данный момент.
- **boolean isEmpty()**
Возвращает логическое значение **true**, если коллекция не содержит ни одного из элементов.
- **boolean contains(Object obj)**
Возвращает логическое значение **true**, если коллекция содержит объект, равный заданному объекту **obj**.
- **boolean containsAll(Collection<?> other)**
Возвращает логическое значение **true**, если коллекция содержит все элементы из другой коллекции.
- **boolean add(Object element)**
Добавляет элемент в коллекцию. Возвращает логическое значение **true**, если в результате вызова этого метода коллекция изменилась.
- **boolean addAll(Collection<? extends E> other)**
Добавляет все элементы из другой коллекции в данную. Возвращает логическое значение **true**, если в результате вызова этого метода коллекция изменилась.
- **boolean remove(Object obj)**
Удаляет из коллекции объект, равный заданному объекту **obj**. Возвращает логическое значение **true**, если в результате вызова этого метода коллекция изменилась.
- **boolean removeAll(Collection<?> other)**
Удаляет из данной коллекции все элементы другой коллекции. Возвращает логическое значение **true**, если в результате вызова этого метода коллекция изменилась.
- **default boolean removeIf(Predicate<? super E> filter) 8**
Удаляет из данной коллекции все элементы, для которых по заданному условию **filter** возвращается логическое значение **true**. Возвращает логическое значение **true**, если в результате вызова этого метода коллекция изменилась.
- **void clear()**
Удаляет все элементы из данной коллекции.
- **boolean retainAll(Collection<?> other)**
Удаляет из данной коллекции все элементы, которые не равны ни одному из элементов другой коллекции. Возвращает логическое значение **true**, если в результате вызова этого метода коллекция изменилась.
- **Object[] toArray()**
Возвращает из коллекции массив объектов.
- **<T> T[] toArray(T[] arrayToFill)**
Возвращает из коллекции массив объектов. Если заполняемый массив **arrayToFill** имеет достаточную длину, он заполняется элементами данной коллекции. Если же остается место, добавляются пустые элементы **null**. В противном случае выделяется и заполняется новый массив с тем же типом элементов и длиной, что и у заданного массива **arrayToFill**.

java.util.Iterator<E> 1.2

- **boolean hasNext()**

Возвращает логическое значение **true**, если в коллекции еще имеются элементы, к которым можно обратиться.

- **E next()**

Возвращает следующий перебираемый объект. Генерирует исключение типа **NoSuchElementException**, если достигнут конец коллекции.

- **void remove()**

Удаляет последний перебираемый объект. Этот метод должен вызываться сразу же после обращения к удаляемому элементу. Если коллекция была видоизменена после обращения к последнему ее элементу, этот метод генерирует исключение типа **IllegalStateException**.

- **default void forEachRemaining(Consumer<? super E> action) 8**

Обходит элементы и передает их заданному действию до тех пор, пока в коллекции не останется ни одного элемента, а иначе в этом действии будет сгенерировано исключение.

9.2. Интерфейсы в каркасе коллекций Java

В каркасе коллекций Java определен целый ряд интерфейсов для различных типов коллекций, как показано на рис. 9.4.

В рассматриваемом здесь каркасе имеются два основополагающих интерфейса коллекций: **Collection** и **Map**. Для ввода элементов в коллекцию служит следующий метод:

```
boolean add(E element)
```

Но отображения содержат пары “ключ–значение”, поэтому для ввода этих пар вызывается метод **put()**:

```
V put(K key, V value)
```

Для извлечения элементов из коллекции служит итератор, организующий обращение к ним. А для извлечения значений из отображения вызывается метод **get()**:

```
V get(K key)
```

Список представляет собой *упорядоченную коллекцию*. Элементы добавляются на определенной позиции в контейнере. Доступ к элементу списка осуществляется с помощью итератора или по целочисленному индексу. В последнем случае доступ оказывается *произвольным*, поскольку к элементам можно обращаться в любом порядке. А если используется итератор, то обращаться к элементам списка можно только по очереди.

Для произвольного доступа к элементам списка в интерфейсе **List** определяются следующие методы:

```
void add(int index, E element)
```

```
void remove(int index)
```

```
E get(int index)
```

```
E set(int index, E element)
```

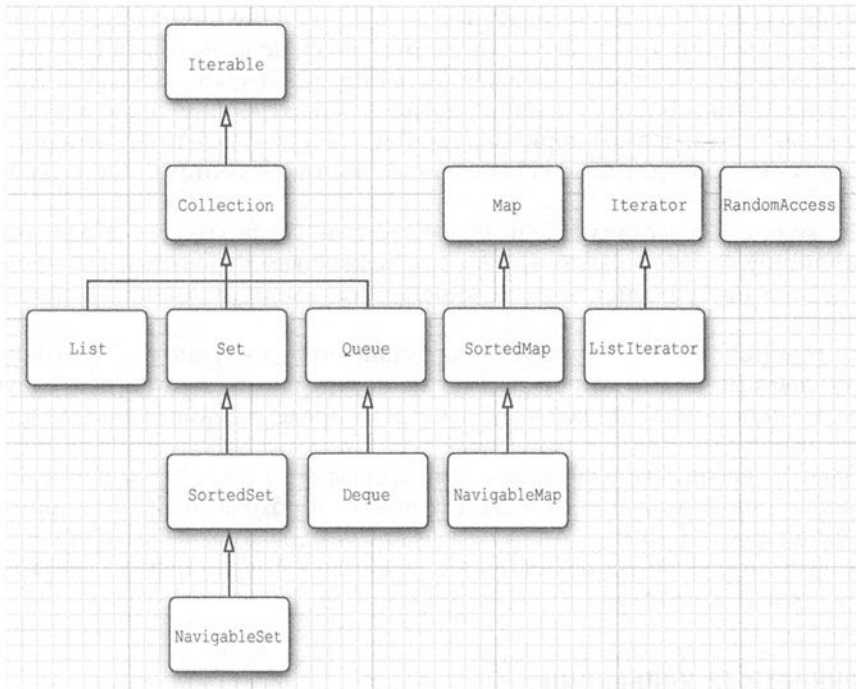


Рис. 9.4. Интерфейсы в каркасе коллекций Java

В интерфейсе `ListIterator` определяется следующий метод для ввода элемента до позиции итератора:

```
void add(E element)
```

Откровенно говоря, этот аспект каркаса коллекций спроектирован неудачно. На практике имеются две разновидности упорядоченных коллекций с совершенно разными показателями производительности. Упорядоченная коллекция на основе массива отличается быстрым произвольным доступом, и поэтому методы из интерфейса `List` целесообразно вызывать с целочисленными индексами. А связный список отличается медленным произвольным доступом, хотя он и относится к категории упорядоченных коллекций. Поэтому перебирать его элементы лучше с помощью итератора. И для этой цели следовало бы предоставить два разных интерфейса.



НА ЗАМЕТКУ! Чтобы избежать дорогостоящих (с точки зрения потребляемых вычислительных ресурсов) операций произвольного доступа, в версии Java 1.4 появился новый интерфейс `RandomAccess`. В этом интерфейсе отсутствуют методы, но с его помощью можно проверить, поддерживается ли в конкретной коллекции эффективный произвольный доступ, как показано ниже.

```

if (c instanceof RandomAccess)
{
    использовать алгоритм произвольного доступа
}
else
{
    использовать алгоритм последовательного доступа
}
  
```

Интерфейс `Set` подобен интерфейсу `Collection`, но поведение его методов определено более строго. В частности, метод `add()` должен отвергать дубликаты во множестве. Метод `equals()` должен быть определен таким образом, чтобы два множества считались одинаковыми, если они содержат одни и те же элементы, но не обязательно в одинаковом порядке. А метод `hashCode()` должен быть определен таким образом, чтобы два множества с одинаковыми элементами порождали один и тот же хеш-код.

Зачем же объявлять отдельные интерфейсы, если сигнатуры их методов совпадают? В принципе не все коллекции являются множествами. Поэтому наличие интерфейса `Set` дает программистам возможность писать методы, принимающие только множества.

В интерфейсах `SortedSet` и `SortedMap` становится доступным объект-компаратор, применяемый для сортировки, а также определяются методы для получения представлений подмножеств коллекций. Об этих представлениях речь пойдет в разделе 9.5.

Наконец, в версии Java 6 внедрены интерфейсы `NavigableSet` и `NavigableMap`, содержащие дополнительные методы для поиска и обхода отсортированных множеств и отображений. (В идеальном случае эти методы должны быть просто включены в состав интерфейсов `SortedSet` и `SortedMap`.) Эти интерфейсы реализуются в классах `TreeSet` и `TreeMap`.

9.3. Конкретные коллекции

В табл. 9.1 перечислены коллекции из библиотеки Java вместе с кратким описанием назначения каждого из их классов. (Ради простоты в ней не указаны классы потокобезопасных коллекций, о которых речь пойдет в главе 12.) Все классы из табл. 9.1 реализуют интерфейс `Collection`, за исключением классов, имена которых оканчиваются на **Map**, поскольку эти классы реализуют интерфейс `Map`, подробнее рассматриваемый в разделе 9.4.

Таблица 9.1. Конкретные коллекции из библиотеки Java

Тип коллекции	Описание
ArrayList	Индексированная динамически расширяющаяся и сокращающаяся последовательность
LinkedList	Упорядоченная последовательность, допускающая эффективную вставку и удаление на любой позиции
ArrayDeque	Двунаправленная очередь, реализуемая в виде циклического массива
HashSet	Неупорядоченная коллекция, исключающая дубликаты
TreeSet	Отсортированное множество
EnumSet	Множество значений перечислимого типа
LinkedHashSet	Множество, запоминающее порядок ввода элементов
PriorityQueue	Коллекция, позволяющая эффективно удалять наименьший элемент
HashMap	Структура данных для хранения связанных вместе пар “ключ–значение”
TreeMap	Отображение с отсортированными ключами
EnumMap	Отображение с ключами, относящимися к перечислимому типу

Окончание табл. 9.1

Тип коллекции	Описание
LinkedHashMap	Отображение с запоминанием порядка, в котором добавлялись элементы
WeakHashMap	Отображение со значениями, которые могут удаляться системой сборки "мусора", если они нигде больше не используются
IdentityHashMap	Отображение с ключами, сравниваемыми с помощью операции <code>==</code> , а не вызова метода <code>equals()</code>

Отношения между этими классами показаны на рис. 9.5.

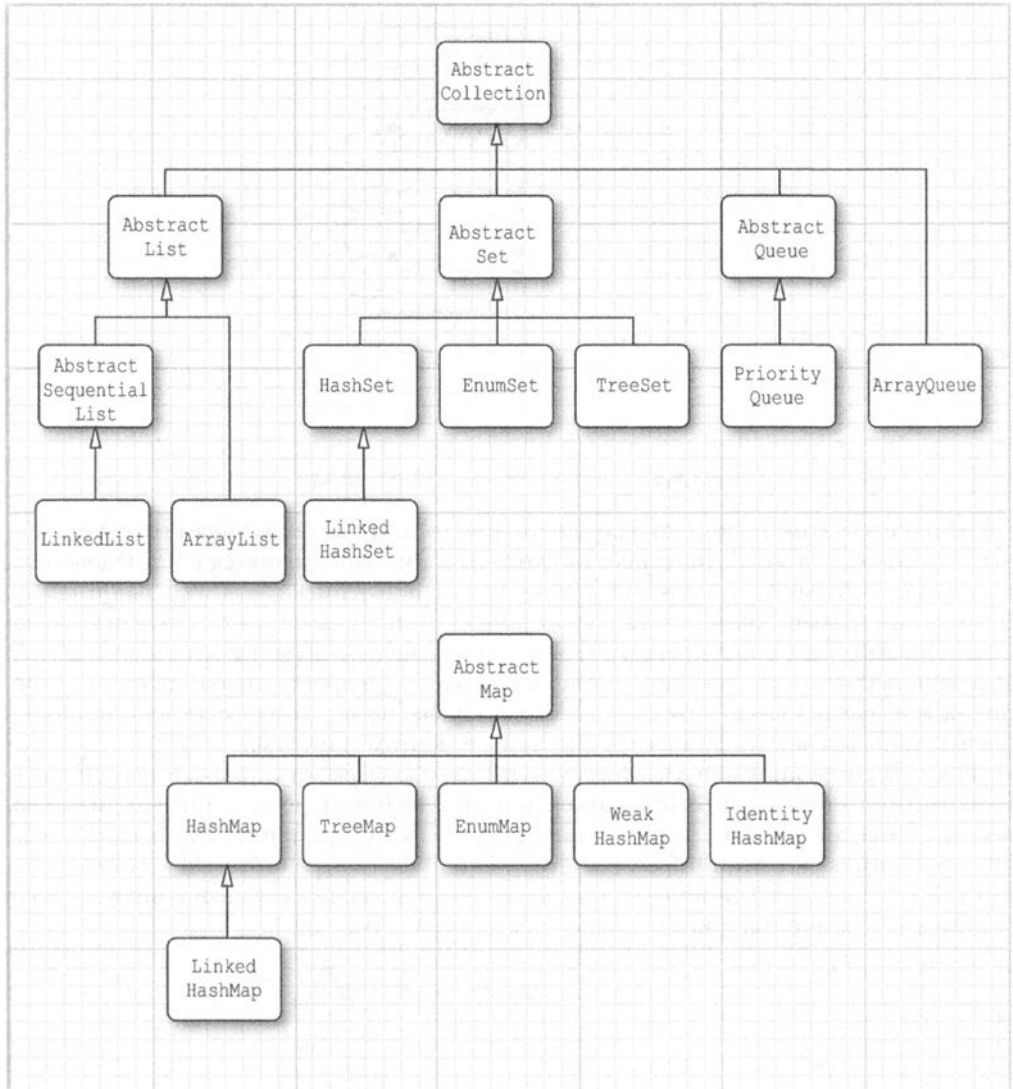


Рис. 9.5. Классы из каркаса коллекций в Java

9.3.1. Связные списки

Во многих примерах программ, приведенных ранее в этой книге, уже не раз использовались массивы и родственный им класс `ArrayList` динамического списочного массива. Но обычные и списочные массивы страдают существенным недостатком. Удаление элемента из середины массива обходится дорого с точки зрения потребляемых вычислительных ресурсов, потому что все элементы, следующие за удаляемым, приходится перемещать к началу массива (рис. 9.6). Это же справедливо и для ввода элементов в середине массива.

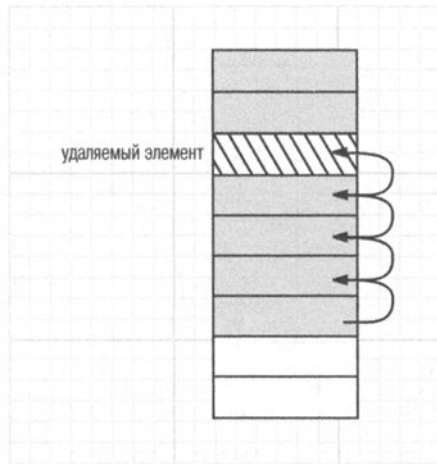


Рис. 9.6. Удаление элемента из массива

Этот недостаток позволяет устранить другая широко известная структура данных — *связный список*. Если ссылки на объекты из массива хранятся в последовательных областях памяти, то каждый объект из связного списка — в отдельной *связке*. В языке программирования Java все связные списки на самом деле являются двуправленными, т.е. в каждой связке хранится ссылка на ее предшественника (рис. 9.7). Удаление элемента из середины связного списка — недорогая операция с точки зрения потребляемых вычислительных ресурсов, поскольку в этом случае достаточно обновить лишь связки, соседние с удаляемым элементом (рис. 9.8).

Если вы когда-нибудь изучали структуры данных и реализацию связных списков, то, возможно, еще помните, насколько хлопотно соединять связки при вводе или удалении элементов связного списка. В таком случае вы будете приятно удивлены, узнав, что в библиотеке коллекций Java для этой цели предусмотрен готовый к вашим услугам класс `LinkedList`. В приведенном ниже примере кода в связный список сначала вводятся три элемента, а затем удаляется второй из них.

```
var staff = new LinkedList<String>();  
    // Объект типа LinkedList, реализующий связный список  
staff.add("Amy");  
staff.add("Bob");  
staff.add("Carl");  
Iterator<String> iter = staff.iterator();  
String first = iter.next(); // обратиться к  
                           // первому элементу
```

```
String second = iter.next(); // обратиться ко  
                             // второму элементу  
iter.remove(); // удалить последний перебираемый  
               // элемент списка
```

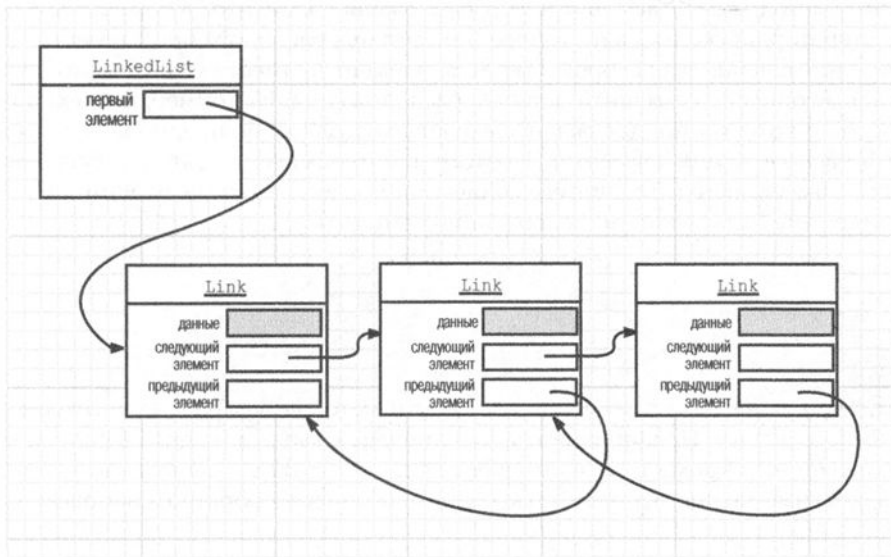


Рис. 9.7. Двухнаправленный связный список

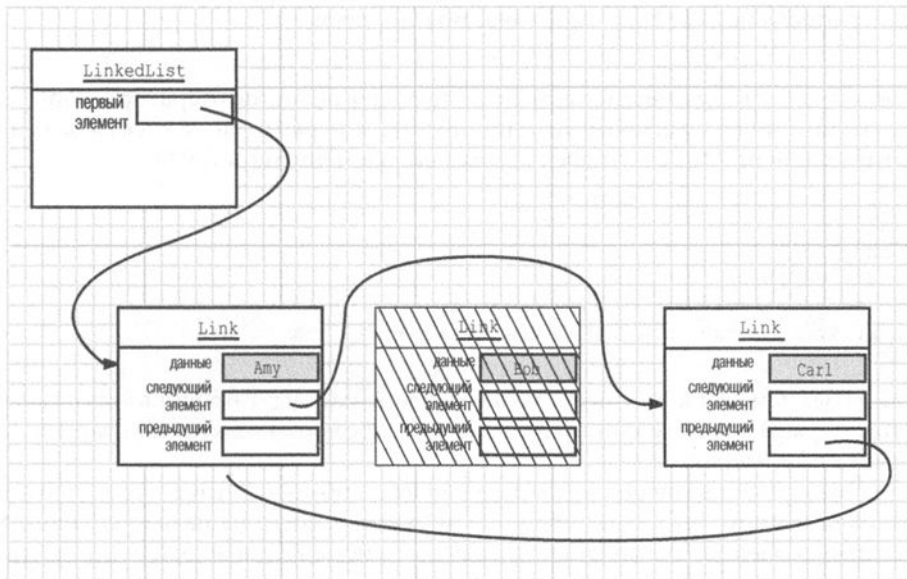


Рис. 9.8. Удаление элемента из связного списка

Но связанные списки существенно отличаются от обобщенных коллекций. Связный список — это *упорядоченная коллекция*, в которой имеет значение расположение объектов. Метод `LinkedList.add()` вводит объект в конце списка. Но объекты зачастую требуется вводить где-то в середине списка.

За этот метод `add()`, зависящий от расположения элементов в связанном списке, отвечает итератор, поскольку в итераторе описывается расположение элементов в коллекции. Применение итераторов для ввода элементов имеет смысл только для коллекций, имеющих естественный порядок расположения. Например, коллекция типа *множества*, о которой пойдет речь в следующем разделе, не предполагает никакого порядка расположения элементов. Поэтому в интерфейсе `Iterator` отсутствует метод `add()`. Вместо него в библиотеке коллекций предусмотрен следующий подчиненный интерфейс `ListIterator`, содержащий метод `add()`:

```
interface ListIterator<E> extends Iterator<E>
{
    void add(E element);
    . . .
}
```

В отличие от метода `Collection.add()`, этот метод не возвращает логическое значение типа `boolean`. Предполагается, что операция ввода элемента в список всегда видоизменяет его. Кроме того, в интерфейсе `ListIterator` имеются следующие два метода, которые можно использовать для обхода списка в обратном направлении:

```
E previous()
boolean hasPrevious()
```

Как и метод `next()`, метод `previous()` возвращает объект, который он прошел. А метод `listIterator()` из класса `LinkedList` возвращает объект итератора, реализующего интерфейс `ListIterator`, как показано ниже.

```
listIterator<String> iter = staff.listIterator()
```

Метод `add()` вводит новый элемент *до* текущей позиции итератора. Например, в приведенном ниже фрагменте кода пропускается первый элемент связанного списка и вводится элемент "Juliet" перед вторым его элементом (рис. 9.9).

```
var staff = new LinkedList<String>();
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");
ListIterator<String> iter = staff.listIterator();
iter.next(); // пропустить первый элемент списка
iter.add("Juliet");
```

Если вызвать метод `add()` несколько раз, элементы будут просто вводиться в список в том порядке, в каком они предоставляются. Все они вводятся по очереди до текущей позиции итератора.

Когда выполняется операция ввода элемента с помощью итератора, только что возвращенного методом `listIterator()` и указывающего на начало связанного списка, вводимый элемент располагается в начале списка. Когда же итератор достигает последнего элемента списка (т.е. метод `hasNext()` возвращает логическое значение `false`), вводимый элемент располагается в конце списка. Если связный список содержит n элементов, тогда для ввода нового элемента в нем имеется $n + 1$ доступных мест. Эти места соответствуют $n + 1$ возможным позициям итератора. Так, если

связный список содержит три элемента, A, B и C, для ввода нового элемента в нем имеются четыре возможные позиции, обозначенные ниже знаком курсора |.

| ABC
A | BC
AB | C
ABC |

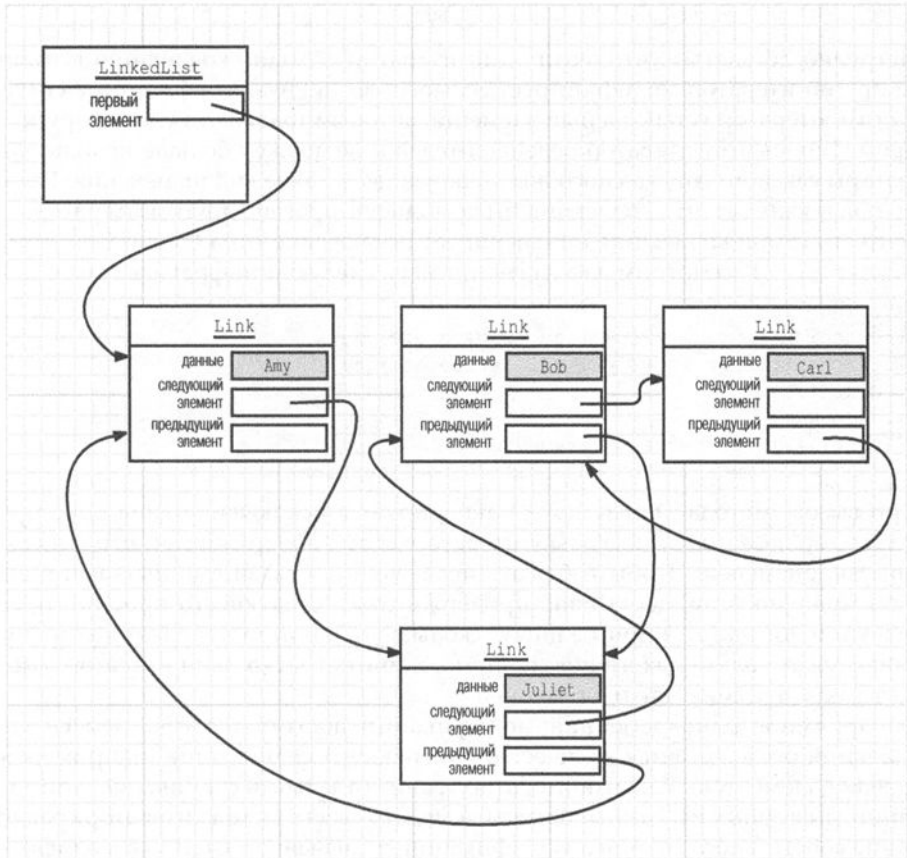


Рис. 9.9. Добавление элемента в связный список



НА ЗАМЕТКУ! Впрочем, аналогия с курсором | не совсем точна. Операция удаления элемента из списка выполняется не совсем так, как при нажатии клавиши «Backspace». Если метод `remove()` вызывается сразу же после метода `next()`, то он действительно удаляет элемент, расположенный слева от итератора, как это и происходит при нажатии клавиши «Backspace». Но если он вызывается сразу же после метода `previous()`, то удаляется элемент, расположенный справа от итератора. К тому же метод `remove()` нельзя вызывать два раза подряд. В отличие от метода `add()`, действие которого зависит только от позиции итератора, действие метода `remove()` зависит и от состояния итератора.

И, наконец, метод `set()` заменяет новым элементом последний элемент, возвращаемый при вызове метода `next()` или `previous()`. Например, в следующем фрагменте кода первый элемент списка заменяется новым значением:

```
ListIterator<String> iter = list.listIterator();
String oldValue = iter.next(); // возвращает первый
                               // элемент списка
iter.set(newValue); // устанавливает в первом элементе
                    // новое значение newValue
```

Нетрудно догадаться, что, если один итератор обходит коллекцию в то время, когда другой итератор модифицирует ее, могут возникнуть конфликтные ситуации. Допустим, итератор установлен до элемента, который только что удален другим итератором. Теперь этот итератор недействителен и не должен больше использоваться. Итераторы связного списка способны обнаруживать такие видоизменения. Если итератор обнаруживает, что коллекция была модифицирована другим итератором или же методом самой коллекции, он генерирует исключение типа `ConcurrentModificationException`. Рассмотрим в качестве примера следующий фрагмент кода:

```
List<String> list = . . .;
ListIterator<String> iter1 = list.listIterator();
ListIterator<String> iter2 = list.listIterator();
iter1.next();
iter1.remove();
iter2.next(); // генерирует исключение
              // типа ConcurrentModificationException
```

При вызове метода `iter2.next()` генерируется исключение типа `ConcurrentModificationException`, поскольку итератор `iter2` обнаруживает, что список был внешне видоизменен. Чтобы избежать исключений в связи с попытками одновременной модификации, достаточно придерживаться следующего простого правила: к коллекции допускается присоединять сколько угодно итераторов, при условии, что все они служат только для чтения, но присоединить только один итератор, который служит как для чтения, так и для записи.

Обнаружение одновременной модификации достигается простым способом. В коллекции отслеживается количество изменяющих ее операций (например, ввода и удаления элементов). Каждый итератор хранит отдельный счетчик операций модификации, вызванных им самим. В начале каждого своего метода итератор сравнивает значение собственного счетчика модификаций со значением счетчика модификаций в коллекции. Если эти значения не равны, генерируется исключение типа `ConcurrentModificationException`.



НА ЗАМЕТКУ! Но из приведенного выше правила обнаружения одновременных видоизменений имеется одно любопытное исключение. Связный список отслеживает лишь структурные модификации вроде ввода и удаления связей. А действие метода `set()` не считается структурной модификацией. К связному списку можно присоединить несколько итераторов, способных вызывать метод `set()` для изменения содержимого списка. Наличие такой возможности требуется во многих алгоритмах, применяемых в классе `Collections`, о котором речь пойдет далее в этой главе.

Итак, мы рассмотрели основные методы из класса `LinkedList`. А интерфейс `ListIterator` служит как для обхода элементов связного списка в любом направлении, так и для ввода и удаления элементов из списка.

Как было показано в разделе 9.2, в интерфейсе `Collection` объявлено немало других полезных методов для операций со связными списками. Они реализованы главным образом в суперклассе `AbstractCollection` класса `LinkedList`. Например, метод `toString()` вызывает одноименный метод для всех элементов списка и формирует одну длинную строку в формате `[A, B, C]`, что удобно для отладки. А метод `contains()` удобен для проверки наличия конкретного элемента в связном списке. Так, в результате вызова `staff.contains("Harry")` возвращается логическое значение `true`, если связный список уже содержит символьную строку "Harry".

В библиотеке коллекций предусмотрен также ряд методов, которые с теоретической точки зрения кажутся избыточными. Ведь в связных списках не поддерживается быстрый произвольный доступ. Если требуется проанализировать n -й элемент связного списка, то начинать придется с самого начала и перебрать первые $n - 1$ элементов списка, причем без всяких пропусков. Именно по этой причине программисты обычно не применяют связные списки в тех случаях, когда к элементам требуется обращаться по целочисленному индексу. Тем не менее в классе `LinkedList` предусмотрен метод `get()`, который позволяет обратиться к определенному элементу следующим образом:

```
LinkedList<String> list = . . . ;  
String obj = list.get(n);
```

Безусловно, этот метод не очень эффективен. И если он все же применяется, то, скорее всего, к неподходящей для этого структуре данных. Столь обманчивый произвольный доступ не годится для обращения к элементам связного списка. Например, приведенный ниже фрагмент кода совершенно неэффективен.

```
for (int i = 0; i < list.size(); i++)  
    сделать что-нибудь с результатом вызова list.get(i);
```

Всякий раз, когда требуется обратиться к другому элементу с помощью метода `get()`, поиск начинается с самого начала списка. В объекте типа `LinkedList` не предпринимается никаких попыток буферизовать данные о расположении элементов в списке.



НА ЗАМЕТКУ! В методе `get()` предусмотрена единственная незначительная оптимизация: если указанный индекс больше величины `size()/2`, то поиск элемента начинается с конца списка.

В интерфейсе итератора списка имеется также метод, предоставляющий индекс текущей позиции в списке. Но поскольку итераторы в Java устроены таким образом, что они обозначают позицию *между* элементами коллекции, то таких методов на самом деле два. Так, метод `nextIndex()` возвращает целочисленный индекс того элемента, который должен быть возвращен при последующем вызове метода `next()`. А метод `previousIndex()` возвращает индекс того элемента, который был бы возвращен при последующем вызове метода `previous()`. И этот индекс будет, конечно, на единицу меньше, чем `nextIndex`. Оба метода действуют эффективно, поскольку в итераторе запоминается текущая позиция. И, наконец, если имеется целочисленный индекс n , в результате вызова метода `list.listIterator(n)` будет возвращен итератор, установленный до элемента с индексом n . Это означает, что при вызове метода `next()` получается тот же самый элемент, что и при вызове метода `list.get(n)`. Следовательно, получение такого итератора малоэффективно.

Если связный список содержит немного элементов, то вряд ли стоит беспокоиться об издержках, связанных с применением методов `set()` и `get()`. Но в таком случае

зачем вообще пользоваться связным списком? Единственная причина для его применения — минимизация издержек на ввод и удаление в середине списка. Если в коллекции предполагается лишь несколько элементов, то для ее составления лучше воспользоваться списочным массивом типа `ArrayList`.

Рекомендуется держаться подальше от всех методов, в которых целочисленный индекс служит для обозначения позиции в связном списке. Если требуется произвольный доступ к коллекции, лучше воспользоваться обычным или списочным массивом типа `ArrayList`, а не связным списком.

В примере программы из листинга 9.1 демонстрируется практическое применение связного списка. В этой программе сначала составляются два списка, затем они объединяются и удаляется каждый второй элемент из второго списка, а в завершение проверяется метод `removeAll()`. Рекомендуется тщательно проанализировать порядок выполнения операций в этой программе, уделив особое внимание итераторам. Для этого, возможно, будет полезно составить схематическое представление позиций итератора аналогично следующему:

```
|ACE |BDFG
A|CE |BDFG
AB|CE B|DFG
. . .
```

Следует, однако, иметь в виду, что в результате приведенного ниже вызова выведутся все элементы связного списка. Для этой цели вызывается метод `toString()` из класса `AbstractCollection`.

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

Листинг 9.1. Исходный код из файла `linkedList/LinkedListTest.java`

```
1 package linkedList;
2
3 import java.util.*;
4
5 /**
6  * В этой программе демонстрируются операции
7  * со связными списками
8  * @version 1.12 2018-04-10
9  * @author Cay Horstmann
10  */
11 public class LinkedListTest
12 {
13     public static void main(String[] args)
14     {
15         var a = new LinkedList<String>();
16         a.add("Amy");
17         a.add("Carl");
18         a.add("Erica");
19
20         var b = new LinkedList<String>();
21         b.add("Bob");
22         b.add("Doug");
23         b.add("Frances");
24         b.add("Gloria");
25     }
26 }
```

```
26 // объединить слова из связанных списков b и a
27
28 ListIterator<String> aIter = a.listIterator();
29 Iterator<String> bIter = b.iterator();
30
31 while (bIter.hasNext())
32 {
33     if (aIter.hasNext()) aIter.next();
34     aIter.add(bIter.next());
35 }
36
37 System.out.println(a);
38
39 // удалить каждое второе слово из связанного списка b
40
41 bIter = b.iterator();
42 while (bIter.hasNext())
43 {
44     bIter.next(); // пропустить один элемент
45     if (bIter.hasNext())
46     {
47         bIter.next(); // пропустить следующий элемент
48         bIter.remove(); // удалить данный элемент
49     }
50 }
51
52 System.out.println(b);
53
54 // групповая операция удаления из связанного списка a
55 // всех слов, составляющих связанный список b
56
57 a.removeAll(b);
58
59 System.out.println(a);
60 }
61 }
```

java.util.List<E> 1.2

- **ListIterator<E> listIterator()**
Возвращает итератор списка, который можно использовать для перебора элементов списка.
- **ListIterator<E> listIterator(int index)**
Возвращает итератор списка для обращения к элементам списка, если в результате вызова метода **next()** возвращается элемент этого списка с заданным индексом.
- **void add(int i, E element)**
Вводит элемент на указанной позиции в списке.
- **void addAll(int i, Collection<? extends E> elements)**
Вводит все элементы из коллекции на указанной позиции в списке.
- **E remove(int i)**
Удаляет и возвращает элемент на указанной позиции в списке.

java.util.List<E> 1.2 (окончание)

- **E get(int i)**
Получает элемент на указанной позиции в списке.
- **E set(int i, E element)**
Заменяет элемент на указанной позиции в списке новым элементом и возвращает прежний элемент.
- **int indexOf(Object element)**
Возвращает позицию первого вхождения искомого элемента в списке или значение `-1`, если искомый элемент не найден.
- **int lastIndexOf(Object element)**
Возвращает позицию последнего вхождения искомого элемента в списке или значение `-1`, если искомый элемент не найден.

java.util.ListIterator<E> 1.2

- **void add(E newElement)**
Вводит новый элемент до текущей позиции в списке.
- **void set(E newElement)**
Заменяет новым элементом последний элемент, обращение к которому было сделано при вызове метода `next()` или `previous()`. Генерирует исключение типа `IllegalStateException`, если структура списка была видоизменена в результате последнего вызова метода `next()` или `previous()`.
- **boolean hasPrevious()**
Возвращает логическое значение `true`, если имеется еще один элемент для обращения при итерации по списку в обратном направлении.
- **E previous()**
Возвращает предыдущий объект. Генерирует исключение типа `NoSuchElementException`, если достигнуто начало списка.
- **int nextIndex()**
Возвращает индекс элемента, который должен быть возвращен при последующем вызове метода `next()`.
- **int previousIndex()**
Возвращает индекс элемента, который должен быть возвращен при последующем вызове метода `previous()`.

9.3.2. Списочные массивы

В предыдущем разделе обсуждались интерфейс `List` и реализующий его класс `LinkedList`. Интерфейс `List` описывает упорядоченную коллекцию, в которой имеет значение расположение элемента. Существуют два способа для обхода элементов: посредством итератора и произвольного доступа с помощью методов `get()` и `set()`. Второй способ не совсем подходит для связных списков, но применение методов

`get()` и `set()` совершенно оправдано для массивов. В библиотеке коллекций предоставляется уже не раз упоминавшийся здесь класс `ArrayList`, также реализующий интерфейс `List`. Класс `ArrayList` инкапсулирует динамически выделяемый массив объектов.



НА ЗАМЕТКУ! Программирующим на Java со стажем, скорее всего, уже приходилось пользоваться классом `Vector` всякий раз, когда возникала потребность в динамическом массиве. Почему же вместо класса `Vector` для подобных целей следует применять класс `ArrayList`? По одной простой причине: все методы из класса `Vector` синхронизированы. К объекту типа `Vector` можно безопасно обращаться одновременно из двух потоков исполнения. Но если обращаться к такому объекту только из одного потока исполнения, что случается намного чаще, то в прикладном коде впустую тратится время на синхронизацию. В отличие от этого, методы из класса `ArrayList` не синхронизированы. Поэтому рекомендуется пользоваться классом `ArrayList` вместо класса `Vector`, если только нет особой необходимости в синхронизации.

9.3.3. Хеш-множества

Связные списки и массивы позволяют указывать порядок, в котором должны следовать элементы. Но если вам нужно найти конкретный элемент, а вы не помните его позицию в коллекции, то придется перебирать все элементы до тех пор, пока не будет обнаружено совпадение по критерию поиска. На это может потребоваться некоторое время, если коллекция содержит достаточно много элементов. Если же порядок расположения элементов не имеет особого значения, то для подобных случаев предусмотрены структуры данных, которые позволяют намного быстрее находить элементы в коллекции. Но недостаток таких структур данных заключается в том, что они не обеспечивают никакого контроля над порядком расположения элементов в коллекции. Эти структуры данных организуют элементы в том порядке, который удобен для их собственных целей.

К числу широко известных структур данных для быстрого нахождения объектов относится так называемая *хеш-таблица*, которая вычисляет для каждого объекта целочисленное значение, называемое *хеш-кодом*. Хеш-код — это целочисленное значение, которое выводится определенным образом из данных в полях экземпляра объекта, причем предполагается, что объекты с разными данными порождают разные хеш-коды. В табл. 9.2 приведено несколько примеров хеш-кодов, получаемых в результате вызова метода `hashCode()` из класса `String`.

Таблица 9.2. Хеш-коды, возвращаемые методом `hashCode()`

Символьная строка	Хеш-код
"Lee"	76268
"lee"	107020
"ee1"	100300

Если вы определяете собственные классы, то на вас ложится ответственность за самостоятельную реализацию метода `hashCode()` (подробнее об этом см. в главе 5). Ваша реализация данного метода должна быть совместима с методом `equals()`. Так, если в результате вызова `a.equals(b)` возвращается логическое значение `true`, то объекты `a` и `b` должны иметь одинаковые хеш-коды. Но самое главное, чтобы хеш-коды вычислялись быстро, а вычисление зависело только от состояния хешируемого объекта, а не от других объектов в хеш-таблице.

В языке Java хеш-таблицы реализованы в виде массивов связанных списков, каждый из которых называется *группой* (рис. 9.10). Чтобы найти место объекта в таблице, вычисляется его хеш-код и уменьшается его модуль до общего количества групп. Полученное в итоге числовое значение и будет индексом группы, содержащей искомый элемент. Так, если хеш-код объекта равен 76268, а всего имеется 128 групп, объект должен быть размещен в группе под номером 108 (поскольку остаток от целочисленного деления 76268 на 128 равен 108). Если повезет, то в этой группе не окажется других элементов и элемент просто вводится в группу. Безусловно, рано или поздно встретится непустая группа. Такое явление называется *хеш-конфликтом*. В таком случае новый объект сравнивается со всеми объектами в группе, чтобы проверить, находится ли он уже в группе. Если учесть сравнительно равномерное распределение хеш-кодов при достаточно большом количестве групп, то в конечном итоге потребуется лишь несколько сравнений.

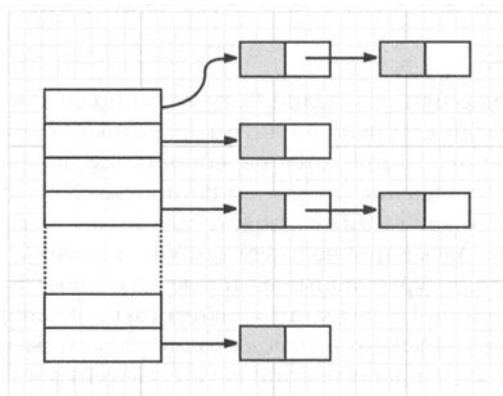


Рис. 9.10. Хеш-таблица



НА ЗАМЕТКУ! Начиная с версии Java 8, вместо связанных списков в заполненных группах применяются сбалансированные двоичные деревья. Благодаря этому повышается производительность, если в результате неудачно выбранной хеш-функции возникает немало конфликтов или же если в злонамеренном коде предпринимается попытка заполнить хеш-таблицу многими значениями с одинаковыми хеш-кодами.

Если требуется более полный контроль над производительностью хеш-таблицы, то можно указать первоначальное количество групп. Оно определяет количество групп, используемых для накопления объектов с одинаковыми хеш-значениями. Если же в хеш-таблицу вводится слишком много элементов, количество конфликтов возрастает, что отрицательно сказывается на производительности извлечения элементов из хеш-таблицы.

Если приблизительно известно, сколько элементов в конечном итоге окажется в хеш-таблице, можно установить количество групп. Обычно это количество устанавливается в пределах от 75 до 150% от ожидаемого числа элементов. Некоторые исследователи полагают, что количество групп должно быть выражено простым числом, чтобы предотвратить группирование ключей. Но на этот счет нет общего мнения. В стандартной библиотеке используются количества групп, выражаемые числом, являющимся степенью 2, по умолчанию — 16. (Любое указываемое значение размеров хеш-таблицы автоматически округляется до следующей степени 2.)

Безусловно, далеко не всегда известно, сколько элементов придется хранить в хеш-таблице, а кроме того, первоначально предполагаемое их количество может оказаться заниженным. Если хеш-таблица становится чрезмерно заполненной, ее следует хешировать *повторно*. Для повторного хеширования создается новая хеш-таблица с большим количеством групп, все элементы старой таблицы вводятся в новую, а старая хеш-таблица отвергается. *Коэффициент загрузки* определяет момент повторного хеширования хеш-таблицы. Так, если коэффициент загрузки равен 0.75 (по умолчанию), а хеш-таблица заполнена более чем на 75%, она автоматически хешируется повторно с увеличенным вдвое количеством групп. Для большинства приложений целесообразно оставить коэффициент загрузки равным 0.75.

Хеш-таблицы можно использовать для реализации ряда важных структур данных. Простейшая из них относится к типу множества. *Множество* — это совокупность элементов, не содержащая дубликатов. Так, метод `add()` сначала пытается найти вводимый объект и вводит его только в том случае, если он отсутствует в множестве.

В библиотеке коллекций Java предоставляется класс `HashSet`, реализующий множество на основе хеш-таблицы. Элементы вводятся в такое множество методом `add()`. А метод `contains()` переопределяется, чтобы осуществлять быстрый поиск элементов в множестве. Он проверяет элементы только одной группы, а не все элементы коллекции.

Итератор хеш-множества перебирает все группы по очереди. В результате хеширования элементы распределяются по таблице, и создается впечатление, будто обращение к ним происходит в случайном порядке. Поэтому классом `HashSet` следует пользоваться только в том случае, если порядок расположения элементов в коллекции не имеет особого значения.

В примере программы из листинга 9.2 отдельные слова текста вводятся из стандартного потока `System.in` в хеш-множество, а затем выводятся из него. Например, данной программе можно направить англоязычный текст книги “Алиса в стране чудес” (доступный по адресу <http://www.gutenberg.net>), запустив ее из командной строки следующим образом:

```
java SetTest < alice30.txt
```

Программа введет все слова из стандартного потока ввода в хеш-множество, а затем переберет все неповторяющиеся слова в хеш-множестве и выведет их количество. (Так, текст книги “Алиса в стране чудес” содержит 5909 неповторяющихся слов, включая уведомление об авторском праве в самом начале.) Слова извлекаются из хеш-множества в случайном порядке.



ВНИМАНИЕ! Будьте внимательны и аккуратны, изменяя элементы хеш-множества. Если хеш-код элемента изменится, этот элемент уже не будет находиться на правильной позиции в структуре данных.

Листинг 9.2. Исходный код из файла `set/SetTest.java`

```
1 package set;
2
3 import java.util.*;
4
5 /**
6  * В этой программе выводятся все неповторяющиеся
7  * слова, введенные в множество из стандартного
```

```
8  * потока System.in
9  * @version 1.12 2015-06-21
10 * @author Cay Horstmann
11 */
12 public class SetTest
13 {
14     public static void main(String[] args)
15     {
16         Set<String> words = new HashSet<>();
17         // объект типа HashSet, реализующий хеш-множество
18         long totalTime = 0;
19
20         try (Scanner in = new Scanner(System.in))
21         {
22             while (in.hasNext())
23             {
24                 String word = in.next();
25                 long callTime = System.currentTimeMillis();
26                 words.add(word);
27                 callTime = System.currentTimeMillis()
28                     - callTime;
29                 totalTime += callTime;
30             }
31         }
32
33         Iterator<String> iter = words.iterator();
34         for (int i = 1; i <= 20 && iter.hasNext(); i++)
35             System.out.println(iter.next());
36         System.out.println(". . .");
37         System.out.println(words.size()
38             + " distinct words. "
39             + totalTime + " milliseconds.");
40     }
41 }
```

java.util.HashSet<E> 1.2

- **HashSet()**
Конструирует пустое хеш-множество.
- **HashSet(Collection<? extends E> elements)**
Конструирует хеш-множество и вводит в него все элементы из коллекции.
- **HashSet(int initialCapacity)**
Конструирует пустое хеш-множество заданной емкости (количество групп).
- **HashSet(int initialCapacity, float loadFactor)**
Конструирует пустое хеш-множество заданной емкости и с указанным коэффициентом загрузки (числовым значением в пределах от 0.0 до 1.0, определяющим предельный процент заполнения хеш-таблицы, по достижении которого происходит повторное хеширование).

```
java.lang.Object 1.0
```

- `int hashCode()`

Возвращает хеш-код данного объекта. Хеш-код может быть любым целым значением (положительным или отрицательным). Определения методов `equals()` и `hashCode()` должны быть согласованы: если в результате вызова `x.equals(y)` возвращается логическое значение `true`, то в результате вызова `x.hashCode()` должно возвращаться то же самое значение, что и в результате вызова `y.hashCode()`.

9.3.4. Древовидные множества

Класс `TreeSet` реализует древовидное множество, подобное хеш-множеству, но с одним дополнительным усовершенствованием: древовидное множество представляет собой *отсортированную коллекцию*, в которую можно вводить элементы в любом порядке. Когда же выполняется перебор ее элементов, извлекаемые из нее значения оказываются автоматически отсортированными. Допустим, что в такую коллекцию сначала введены три символьные строки, а затем перебраны все введенные в нее элементы:

```
var sorter = new TreeSet<String>();
sorter.add("Bob");
sorter.add("Amy");
sorter.add("Carl");
for (String s : sorter) System.println(s);
```

Полученные в итоге значения выводятся в отсортированном порядке: Amy, Bob, Carl. Как следует из имени класса `TreeSet`, сортировка обеспечивается древовидной структурой данных. (В текущей реализации используется структура так называемого *красно-черного дерева*. Подробное описание древовидных структур данных приведено в книге *Алгоритмы: построение и анализ. 3-е издание*. Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн (ИД "Вильямс", 2016 г.) Всякий раз при вводе элемента в древовидное множество он размещается на правильно отсортированной позиции. Таким образом, итератор всегда перебирает элементы такого множества в отсортированном порядке.

Ввод элемента в древовидное множество происходит медленнее, чем в хеш-таблицу (табл. 9.3), но все же намного быстрее, чем в требуемое место массива или связанного списка. Если древовидное множество состоит из n элементов, то в среднем требуется $\log_2 n$ сравнений, чтобы найти правильное расположение нового элемента. Так, если древовидное множество уже содержит 1000 элементов, для ввода нового элемента потребуется около 10 сравнений.



НА ЗАМЕТКУ! Чтобы пользоваться древовидным множеством, необходимо иметь возможность сравнивать его элементы. Для этого элементы должны относиться к классу, реализующему интерфейс `Comparable`, а иначе при построении множества придется предоставить объект класса, реализующего интерфейс `Comparator`. (Интерфейсы `Comparable` и `Comparator` были представлены в главе 6.)

Таблица 9.3. Сравнение операций ввода элементов в древовидное и хеш-множество

Документ	Всего слов	Количество неповторяющихся слов	Коллекция типа HashSet	Коллекция типа TreeSet
“Алиса в стране чудес”	28195	5909	5 с	7 с
“Граф Монте-Кристо”	466300	37545	75 с	98 с

Если обратиться к табл. 9.3, то можно прийти к выводу, что вместо хеш-множества всегда следует пользоваться древовидным множеством. Ведь для ввода элементов в такое множество, по-видимому, не требуется много времени, а его элементы сортируются автоматически. Выбор одной из этих разновидностей множеств зависит от характера накапливаемых данных. Так, если данные не нужно сортировать, то и нет никаких оснований в излишних затратах на их сортировку. Но важнее другое: отсортировать некоторые данные в нужном порядке намного сложнее, чем с помощью хеш-функции. Хеш-функция должна лишь достаточно равномерно распределять объекты, тогда как функция сравнения — различать объекты с абсолютной точностью.

Чтобы сделать такое различие конкретным, рассмотрим задачу составления множества прямоугольников. Если воспользоваться для этой цели древовидным множеством типа `TreeSet`, то придется предоставить компаратор типа `Comparator<Rectangle>`. Как же сравнить два прямоугольника? Сравнить их по площади нельзя, поскольку могут оказаться два прямоугольника с разными координатами, но одинаковой площадью. Порядок сортировки древовидного множества должен быть *общим*. Два любых элемента должны быть сравнимы, и результат сравнения может быть нулевым лишь в том случае, если сравниваемые элементы равны. Для прямоугольников имеется способ лексикографического упорядочения по координатам, но он кажется неестественным и сложным для вычисления. С другой стороны, хеш-функция уже определена для класса `Rectangle` и просто хеширует координаты.



НА ЗАМЕТКУ! Начиная с версии Java 6, класс `TreeSet` реализует интерфейс `NavigableSet`, в который введены удобные методы для обнаружения элементов древовидного множества и его обхода в обратном порядке. Подробнее об этом см. в документации на прикладной интерфейс API.

В примере программы из листинга 9.3 строятся два древовидных множества объектов типа `Item`. Первое из них сортируется по номеру изделия в каталоге, т.е. в порядке сортировки, выбираемом по умолчанию для объектов типа `Item`, а второе сортируется по описанию изделия с помощью специального компаратора. Само же изделие и его номер описываются в классе `Item` из листинга 9.4.

Листинг 9.3. Исходный код из файла `treeSet/TreeSetTest.java`

```
1 package treeSet;
2
3 import java.util.*;
4
5 /**
6  * В этой программе множество изделий путем
7  * сравнения их описаний
8  * @version 1.13 2018-04-10
9  * @author Cay Horstmann
```

```
10 */
11 public class TreeSetTest
12 {
13     public static void main(String[] args)
14     {
15         var parts = new TreeSet<Item>();
16         parts.add(new Item("Toaster", 1234));
17         parts.add(new Item("Widget", 4562));
18         parts.add(new Item("Modem", 9912));
19         System.out.println(parts);
20
21         var sortByDescription = new TreeSet<Item>
22             (Comparator.comparing(Item::getDescription));
23
24         sortByDescription.addAll(parts);
25         System.out.println(sortByDescription);
26     }
27 }
```

Листинг 9.4. Исходный код из файла `treeSet/Item.java`

```
1 package treeSet;
2
3 import java.util.*;
4
5 /**
6  * Описание изделия и его номер по каталогу
7  */
8 public class Item implements Comparable<Item>
9 {
10     private String description;
11     private int partNumber;
12
13     /**
14      * Конструирует объект изделия
15      *
16      * @param aDescription Описание изделия
17      * @param aPartNumber Номер изделия по каталогу
18      */
19     public Item(String aDescription, int aPartNumber)
20     {
21         description = aDescription;
22         partNumber = aPartNumber;
23     }
24
25     /**
26      * Получает описание данного изделия
27      *
28      * @return Описание изделия
29      */
30     public String getDescription()
31     {
32         return description;
33     }
34 }
```



```
35 public String toString()
36 {
37     return "[description=" + description + ", partNumber="
38         + partNumber + "]";
39 }
40
41 public boolean equals(Object otherObject)
42 {
43     if (this == otherObject) return true;
44     if (otherObject == null) return false;
45     if (getClass() != otherObject.getClass())
46         return false;
47     Item other = (Item) otherObject;
48     return Objects.equals(description, other.description)
49         && partNumber == other.partNumber;
50 }
51
52 public int hashCode()
53 {
54     return Objects.hash(description, partNumber);
55 }
56
57 public int compareTo(Item other)
58 {
59     int diff = Integer.compare(partNumber,
60                               other.partNumber);
61     return diff != 0 ? diff :
62         description.compareTo(other.description);
63 }
64 }
```

java.util.TreeSet<E> 1.2

- **TreeSet()**
- **TreeSet(Comparator<? super E> comparator)**
Конструируют пустое древовидное множество.
- **TreeSet(Collection<? extends E> elements)**
- **TreeSet(SortedSet<E> s)**
Конструируют древовидное множество и вводят в него все элементы из коллекции.

java.util.SortedSet<E> 1.2

- **Comparator<? super E> comparator()**
Возвращает компаратор для сортировки элементов или пустое значение **null**, если элементы сравниваются методом **compareTo()** из интерфейса **Comparable**.
- **E first()**
- **E last()**
Возвращают наименьший и наибольший элементы из отсортированного множества.

java.util.NavigableSet<E> 6

- **E higher(E value)**

- **E lower(E value)**

Возвращают наименьший элемент, который больше указанного значения **value**, или наибольший элемент, который меньше указанного значения **value**, а если такой элемент не обнаружен — пустое значение **null**.

- **E ceiling(E value)**

- **E floor(E value)**

Возвращают наименьший элемент, который больше или равен указанному значению **value**, или наибольший элемент, который меньше или равен указанному значению **value**, а если такой элемент не обнаружен — пустое значение **null**.

- **E pollFirst()**

- **E pollLast()**

Удаляют и возвращают наименьший или наибольший элемент во множестве или же пустое значение **null**, если множество оказывается пустым.

- **Iterator<E> descendingIterator()**

Возвращает итератор, обходящий данное множество в обратном порядке.

9.3.5. Одно- и двухсторонние очереди

Как упоминалось выше, обычная (односторонняя) очередь позволяет эффективно вводить элементы в свой хвост и удалять элементы из своей головы, а *двухсторонняя очередь* — вводить и удалять элементы на обоих своих концах, хотя ввод элементов в середине очереди не поддерживается. В версии Java 6 появился интерфейс **Deque**, реализуемый классами **ArrayDeque** и **LinkedList**, причем оба класса предоставляют двухстороннюю очередь, которая может расти по мере надобности. В главе 12 будут приведены примеры применения ограниченных одно- и двухсторонних очередей.

java.util.Queue<E> 5.0

- **boolean add(E element)**

- **boolean offer(E element)**

Вводят заданный элемент в конце очереди и возвращают логическое значение **true**, если очередь не заполнена. Если же очередь заполнена, первый метод генерирует исключение типа **IllegalStateException**, тогда как второй возвращает логическое значение **false**.

- **E remove()**

- **E poll()**

Удаляют и возвращают элемент из головы очереди, если очередь не пуста. Если же очередь пуста, то первый метод генерирует исключение типа **NoSuchElementException**, тогда как второй возвращает пустое значение **null**.

- **E element()**

- **E peek()**

Возвращают элемент из головы очереди, не удаляя его, если очередь не пуста. Если же очередь пуста, то первый метод генерирует исключение типа **NoSuchElementException**, тогда как второй возвращает пустое значение **null**.

java.util.Deque<E> 6

- **void addFirst(E element)**
- **void addLast(E element)**
- **boolean offerFirst(E element)**
- **boolean offerLast(E element)**

Вводят заданный элемент в голову или в хвосте двухсторонней очереди. Если очередь заполнена, первые два метода генерируют исключение типа **IllegalStateException**, тогда как два последние возвращают логическое значение **false**.

- **E removeFirst()**
- **E removeLast()**
- **E pollFirst()**
- **E pollLast()**

Удаляют и возвращают элемент из головы очереди, если очередь не пуста. Если же она пуста, то первые два метода генерируют исключение типа **NoSuchElementException**, тогда как последние два возвращают пустое значение **null**.

- **E getFirst()**
- **E getLast()**
- **E peekFirst()**
- **E peekLast()**

Возвращают элемент из головы очереди, не удаляя ее, если очередь не пуста. Если же она пуста, первые два метода генерируют исключение типа **NoSuchElementException**, тогда как последние два возвращают пустое значение **null**.

java.util.ArrayDeque<E> 6

- **ArrayDeque()**
- **ArrayDeque(int initialCapacity)**

Конструируют неограниченные двунаправленные очереди с начальной емкостью 16 элементов или заданной начальной емкостью.

9.3.6. Очереди по приоритету

В очередях по приоритету элементы извлекаются в отсортированном порядке после того, как они были введены в произвольном порядке. Следовательно, в результате каждого вызова метода **remove()** получается наименьший из элементов, находящихся в очереди. Но в очереди по приоритету сортируются не все ее элементы. Если выполняется перебор элементов такой очереди, они совсем не обязательно оказываются отсортированными. В очереди по приоритету применяется изящная и эффективная структура данных — так называемая “куча” — самоорганизующееся двоичное дерево, в котором операции ввода и удаления вызывают перемещение наименьшего элемента в корень, не тратя времени на сортировку всех элементов очереди.

Подобно древовидному множеству, очередь по приоритету может содержать элементы класса, реализующего интерфейс **Comparable**, или же принимать объект типа **Comparator**, предоставляемый конструктору ее класса. Как правило, очередь

по приоритету применяется для планирования заданий на выполнение. У каждого задания имеется свой приоритет. Задания вводятся в очередь в случайном порядке. Когда новое задание запускается на выполнение, наиболее высокоприоритетное задание удаляется из очереди. (По традиции приоритет 1 считается наивысшим, поэтому в результате операции удаления из очереди извлекается элемент с наименьшим приоритетом.)

В примере программы из листинга 9.5 демонстрируется применение очереди по приоритету непосредственно в коде. В отличие от перебора элементов древовидного множества, в данном примере элементы очереди не перебираются в отсортированном порядке. Но удаление из очереди по приоритету всегда касается ее наименьшего элемента.

Листинг 9.5. Исходный код из файла `priorityQueue/PriorityQueueTest.java`

```
1 package priorityQueue;
2
3 import java.util.*;
4 import java.time.*;
5
6 /**
7  * В этой программе демонстрируется применение
8  * очереди по приоритету
9  * @version 1.02 2015-06-20
10 * @author Cay Horstmann
11 */
12 public class PriorityQueueTest
13 {
14     public static void main(String[] args)
15     {
16         var pq = new PriorityQueue<LocalDate>();
17         pq.add(LocalDate.of(1906, 12, 9)); // G. Hopper
18         pq.add(LocalDate.of(1815, 12, 10)); // A. Lovelace
19         pq.add(LocalDate.of(1903, 12, 3)); // J. von Neumann
20         pq.add(LocalDate.of(1910, 6, 22)); // K. Zuse
21
22         System.out.println("Iterating over elements . . .");
23         for (LocalDate date : pq)
24             System.out.println(date);
25         System.out.println("Removing elements . . .");
26         while (!pq.isEmpty())
27             System.out.println(pq.remove());
28     }
29 }
```

java.util.PriorityQueue 5

- **PriorityQueue()**
- **PriorityQueue(int initialCapacity)**
Конструируют очередь по приоритету для хранения объектов типа `Comparable`.
- **PriorityQueue(int initialCapacity, Comparator<? super E> c)**
Конструирует очередь по приоритету и использует заданный компаратор для сортировки ее элементов.

9.4. Отображения

Множество — это коллекция, которая позволяет быстро находить существующий элемент. Но для того, чтобы найти такой элемент, нужно иметь его точную копию. Это не слишком распространенная операция поиска, поскольку, как правило, имеется некоторая ключевая информация, по которой требуется найти соответствующий элемент. Для этой цели предназначена структура данных типа отображения. В *отображении* хранятся пары “ключ–значение”. Следовательно, значение можно найти, если предоставить связанный с ним ключ. Например, можно составить и сохранить таблицу записей о работниках, где ключами служат идентификаторы работников, а значениями — объекты типа `Employee`. В последующих разделах поясняется, как обращаться с отображениями.

9.4.1. Основные операции над отображениями

В библиотеке коллекций Java предоставляются две реализации отображений общего назначения: классы `HashMap` и `TreeMap`, реализующие интерфейс `Map`. Хеш-отображение типа `HashMap` хеширует ключи, а древовидное отображение типа `TreeMap` использует общий порядок ключей для организации поискового дерева. Функции хеширования или сравнения применяются *только* к ключам. Значения, связанные с ключами, не хешируются и не сравниваются.

Когда же следует применять хеш-отображение, а когда — древовидное отображение? Как и во множествах, хеширование выполняется немного быстрее, и поэтому хеш-отображение оказывается более предпочтительным, если не требуется перебирать ключи в отсортированном порядке. В приведенном ниже примере кода показано, как организовать хеш-отображение для хранения записей о работниках.

```
var staff = new HashMap<String, Employee>();  
    // объект класса HashMap, реализующего интерфейс Map  
var harry = new Employee("Harry Hacker");  
staff.put("987-98-9996", harry);  
. . .
```

Всякий раз, когда объект вводится в отображение, следует указать и его ключ. В данном случае ключом является символьная строка, а соответствующим значением — объект типа `Employee`. Чтобы извлечь объект, нужно использовать (а значит, запомнить) ключ, как показано ниже.

```
var id = "987-98-9996";  
Employee e = staff.get(id); // получить объект harry
```

Если в отображении отсутствуют данные по указанному ключу, то метод `get()` возвращает пустое значение `null`. Обработать возвращаемое пустое значение не совсем удобно. Иногда для ключей, отсутствующих в отображении, вполне подходит значение по умолчанию, и тогда можно воспользоваться методом `getOrDefault()` следующим образом:

```
Map<String, Integer> scores = . . .;  
int score = scores.getOrDefault(id, 0); // получить нулевое  
    // значение, если идентификатор отсутствует
```

Ключи должны быть однозначными. Нельзя сохранить два значения по одинаковым ключам. Если дважды вызвать метод `put()` с одним и тем же ключом, то второе значение заменит первое. По существу, метод `put()` возвращает предыдущее значение, сохраненное по ключу, указанному в качестве его параметра.

Метод `remove()` удаляет элемент из отображения по заданному ключу, а метод `size()` возвращает количество элементов в отображении.

Перебрать элементы отображения по ключам и значениям проще всего методом `forEach()`, предоставив ему лямбда-выражение, получающее ключ и значение. Это выражение вызывается по очереди для каждой записи в отображении:

```
scores.forEach((k, v) ->
    System.out.println("key=" + k + ", value=" + v));
```

В примере программы из листинга 9.6 демонстрируется применение отображения непосредственно в коде. Сначала в отображение вводится пара “ключ–значение”, затем из него удаляется один ключ, а следовательно, и связанное с ним значение. Далее изменяется значение, связанное с ключом, вызывается метод `get()` для нахождения значения и выполняется перебор множества элементов отображения.

Листинг 9.6. Исходный код из файла `map/MapTest.java`

```
1 package map;
2
3 import java.util.*;
4
5 /**
6  * В этой программе демонстрируется применение
7  * отображения с ключами типа String и значениями
8  * типа Employee
9  * @version 1.12 2015-06-21
10 * @author Cay Horstmann
11 */
12 public class MapTest
13 {
14     public static void main(String[] args)
15     {
16         var staff = new HashMap<String, Employee>();
17         staff.put("144-25-5464", new Employee("Amy Lee"));
18         staff.put("567-24-2546",
19             new Employee("Harry Hacker"));
20         staff.put("157-62-7935",
21             new Employee("Gary Cooper"));
22         staff.put("456-62-5527",
23             new Employee("Francesca Cruz"));
24
25         // вывести все элементы из отображения
26
27         System.out.println(staff);
28
29         // удалить элемент из отображения
30
31         staff.remove("567-24-2546");
32
33         // заменить элемент в отображении
34
35         staff.put("456-62-5527",
36             new Employee("Francesca Miller"));
37
38         // найти значение в отображении
39
```

```
40     System.out.println(staff.get("157-62-7935"));
41
42     // перебрать все элементы в отображении
43
44     staff.forEach((k, v) ->
45         System.out.println("key=" + k + ", value=" + v));
46 }
47 }
```

java.util.Map<K, V> 1.2

- **V get(Object key)**

Получает значение, связанное с ключом, а возвращает объект, связанный с ключом, или же пустое значение **null**, если ключ не найден в отображении. В реализующих классах могут быть запрещены пустые ключи типа **null**.

- **default V getOrDefault(Object key, V defaultValue) 8**

Получает значение, связанное с ключом, а возвращает объект, связанный с ключом, или же указанное значение по умолчанию **defaultValue**, если ключ не найден в отображении.

- **V put(K key, V value)**

Размещает в отображении связь ключа со значением. Если ключ уже присутствует, новый объект заменяет старый, ранее связанный с тем же самым ключом. Этот метод возвращает старое значение ключа или же пустое значение **null**, если ключ ранее отсутствовал. В реализующих классах могут быть запрещены пустые ключи или значения **null**.

- **void putAll(Map<? extends K, ? extends V> entries)**

Вводит все элементы из указанного отображения в данное отображение.

- **boolean containsKey(Object key)**

Возвращает логическое значение **true**, если ключ присутствует в отображении.

- **default void forEach(BiConsumer<? super K, ? super V> action) 8**

Выполняет действие над всеми парами "ключ-значение" в данном отображении.

java.util.HashMap<K, V> 1.2

- **HashMap()**

- **HashMap(int initialCapacity)**

- **HashMap(int initialCapacity, float loadFactor)**

Конструируют пустое хеш-отображение указанной емкости и с заданным коэффициентом загрузки (числовым значением в пределах от 0,0 до 1,0, определяющим процент заполнения хеш-таблицы, по достижении которого происходит повторное хеширование). Коэффициент загрузки по умолчанию равен 0,75.

java.util.TreeMap<K,V> 1.2

- **TreeMap()**

Конструирует древовидное отображение по ключам, относящимся к типу, реализующему интерфейс **Comparable**.

```
java.util.TreeMap<K,V> 1.2 (окончание)
```

- **TreeMap(Comparator<? super K> c)**
Конструирует древовидное отображение, используя указанный компаратор для сортировки ключей.
- **TreeMap(Map<? extends K, ? extends V> entries)**
Конструирует древовидное отображение и вводит все элементы из указанного отображения.
- **TreeMap(SortedMap<? extends K, ? extends V> entries)**
Конструирует древовидное отображение и вводит все элементы из отсортированного отображения, используя тот же самый компаратор элементов, что и для отсортированного отображения.

```
java.util.SortedMap<K, V> 1.2
```

- **Comparator<? super K> comparator()**
Возвращает компаратор, используемый для сортировки ключей, или пустое значение `null`, если ключи сравниваются методом `compareTo()` из интерфейса `Comparable`.
- **K firstKey()**
- **K lastKey()**
Возвращают наименьший и наибольший ключи в отображении.

9.4.2. Обновление записей в отображении

Самое трудное в обращении с отображениями — обновить записи в них. Как правило, с этой целью сначала получают значение, связанное с заданным ключом, затем обновляют его и вводят обновленное значение в отображение. Следует, однако, иметь в виду особый случай первого вхождения ключа. Рассмотрим в качестве примера применение отображения для подсчета частоты появления слова в текстовом файле. При обнаружении искомого слова следует инкрементировать счетчик, как показано ниже.

```
counts.put(word, counts.get(word) + 1);
```

Такой прием вполне пригоден, за исключением того случая, когда искомое слово `word` встречается в текстовом файле в первый раз. В таком случае метод `get()` возвращает пустое значение `null` и возникает исключение типа `NullPointerException`. Поэтому в качестве выхода из этого положения можно воспользоваться методом `getOrDefault()` следующим образом:

```
counts.put(word, counts.getOrDefault(word, 0) + 1);
```

Кроме того, можно вызвать сначала метод `putIfAbsent()`, как показано ниже. Этот метод вводит значение в отображение только в том случае, если соответствующий ключ в нем ранее отсутствовал.

```
counts.putIfAbsent(word, 0);  
counts.put(word, counts.get(word) + 1); // Теперь точно  
// известно, что операция будет выполнена успешно
```

Но можно поступить еще лучше, вызвав метод `merge()`, упрощающий эту типичную операцию. Так, в результате следующего вызова:

```
counts.merge(word, 1, Integer::sum);
```


заданное слово `word` связывается со значением 1, если ключ ранее отсутствовал, а иначе предыдущее значение соединяется со значением 1 по ссылке на метод `Integer::sum`.

Другие, менее употребительные методы обновления записей в отображении вкратце описаны ниже.

java.util.Map<K, V> 1.2

- **default V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction) 8**

Если указанный ключ `key` связан с непустым значением `v`, то данный метод применяет к значению `v` и `value` заданную функцию, а затем связывает указанный ключ `key` с получаемым результатом или удаляет этот ключ, если в итоге получается пустое значение `null`. Возвращает результат вызова `get(key)`.

- **default V compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction) 8**

Применяет заданную функцию к указанному ключу `key` или к результату вызова `get(key)`. Связывает указанный ключ `key` с получаемым результатом или удаляет этот ключ, если в итоге получается пустое значение `null`. Возвращает результат вызова `get(key)`.

- **default V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction) 8**

- Если указанный ключ `key` связан с непустым значением `v`, то данный метод применяет к этому ключу и значению `v` заданную функцию, а затем связывает указанный ключ `key` с получаемым результатом или удаляет этот ключ, если в итоге получается пустое значение `null`. Возвращает результат вызова `get(key)`.

- **default V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction) 8**

- Применяет заданную функцию к указанному ключу `key`, если только этот ключ не связан с непустым значением. Связывает указанный ключ `key` с получаемым результатом или удаляет этот ключ, если в итоге получается пустое значение `null`. Возвращает результат вызова `get(key)`.

- **default V putIfAbsent(K key, V value) 8**

- Если указанный ключ `key` отсутствует или связан с пустым значением `null`, то данный метод связывает его с заданным значением `value` и возвращает пустое значение `null`, а иначе — связанное значение.

- **default void replaceAll(BiFunction<? super K, ? super V, ? extends V> function) 8**

Вызывает заданную функцию для всех записей в отображении. Связывает указанный ключ `key` с непустыми результатами и удаляет ключи с пустыми результатами.

9.4.3. Представления отображений

В каркасе коллекций само отображение не рассматривается в качестве коллекции. (В других каркасах для построения структур данных отображение рассматривается в качестве коллекции пар “ключ–значение” или коллекции значений, индексированных ключами.) Тем не менее можно получить *представления* отдельного отображения — объекты класса, реализующего интерфейс `Collection` или один из его подчиненных интерфейсов.

Имеются три таких представления: множество ключей, коллекция (не множество) значений и множество пар “ключ–значение”. Ключи и пары “ключ–значение” образуют множество, потому что в отображении может присутствовать только по одной копии каждого ключа. Приведенные ниже методы возвращают эти три представления. (Элементы последнего множества являются объектами статического внутреннего класса `Map.Entry`.)

```
Set<K> keySet()
Collection<K> values()
Set<Map.Entry<K, V>> entrySet()
```

Следует также иметь в виду, что `keySet` — это объект не класса `HashSet` или `TreeSet`, а некоторого другого класса, реализующего интерфейс `Set`, который расширяет интерфейс `Collection`. Следовательно, объектом `keySet` можно пользоваться как любой другой коллекцией.

Например, все ключи в отображении можно перечислить следующим образом:

```
Set<String> keys = map.keySet();
for (String key : keys)
{
    сделать что-нибудь с ключом key
}
```

Если требуется просмотреть ключи и значения, то можно избежать поиска значений, перечисляя элементы в отображении. Для этой цели служит следующий скелетный код:

```
for (Map.Entry<String, Employee> entry : staff.entrySet())
{
    String key = entry.getKey();
    Employee value = entry.getValue();
    сделать что-нибудь с ключом key и значением value
}
```



СОВЕТ. Чтобы не оперировать неудобными ссылками типа `Map.Entry` в исходном коде, переменные можно объявлять с помощью ключевого слова `var`, как показано ниже.

```
for (var entry : map.entrySet())
{
    сделать что-нибудь с результатами вызовов
    методов entry.getKey(), entry.getValue()
}
```

А с другой стороны, можно вызвать метод `forEach()` следующим образом:

```
map.forEach((k, v) -> {
    сделать что-нибудь с ключом key и значением v
});
```

Если вызывается метод `remove()` итератора, то на самом деле из отображения удаляется ключ и связанное с ним значение. Но ввести элемент в представление множества ключей нельзя. Ведь нет никакого смысла вводить ключ, не вводя связанное с ним значение. Если попытаться вызвать метод `add()`, он сгенерирует исключение типа `UnsupportedOperationException`. На представление множества элементов отображения накладывается такое же ограничение, даже если операция ввода новой пары “ключ–значение” имеет принципиальный смысл.

java.util.Map<K, V> 1.2

- **Set<Map.Entry<K, V>> entrySet()**

Возвращает представление множества объектов типа **Map.Entry**, т.е. пар “ключ–значение” в отображении. Из этого множества можно удалять имеющиеся в нем элементы, но в него нельзя вводить новые элементы.

- **Set<K> keySet()**

Возвращает представление множества всех ключей в отображении. Из этого множества можно удалять имеющиеся в нем элементы, и в этом случае будут удалены ключи и связанные с ними значения, но в него нельзя вводить новые элементы.

- **Collection<V> values()**

Возвращает представление множества всех значений в отображении. Из этого множества можно удалять имеющиеся в нем элементы, и в этом случае будут удалены значения и связанные с ними ключи, но в него нельзя вводить новые элементы.

java.util.Map.Entry<K, V> 1.2

- **K getKey()**

- **V getValue()**

Возвращают ключ или значение из данного элемента отображения.

- **V setValue(V newValue)**

Заменяет прежнее значение новым в связанном с ним отображении и возвращает прежнее значение.

9.4.4. Слабые хеш-отображения

В состав библиотеки коллекций Java входит ряд классов отображений для специальных нужд. Они будут вкратце описаны в последующих разделах.

Класс **WeakHashMap** был разработан для решения одной интересной задачи. Что произойдет со значением, ключ которого не используется нигде больше в программе? Допустим, что последняя ссылка на ключ исчезла. Следовательно, не остается никакого способа сослаться на объект-значение. Но поскольку ни одна часть программы больше не содержит обращения к данному ключу, то и соответствующая пара “ключ–значение” не может быть удалена из отображения. Почему бы системе сборки “мусора” не удалить эту пару? Разве это не ее задача — удалять неиспользуемые объекты?

К сожалению, все не так просто. Система сборки “мусора” отслеживает *действующие* объекты. До тех пор, пока действует объект хеш-отображения, *все* группы в нем активны и не могут быть освобождены из памяти. Поэтому прикладная программа должна позаботиться об удалении неиспользуемых значений из долгосрочных отображений. С другой стороны, можно воспользоваться структурой данных типа **WeakHashMap**, которая взаимодействует с системой сборки “мусора” для удаления пар “ключ–значение”, когда единственной ссылкой на ключ остается ссылка из элемента хеш-таблицы.

Поясним принцип действия этого механизма. В хеш-отображении типа **WeakHashMap** используются *слабые ссылки* для хранения ключей. Объект типа **WeakHashMap** содержит ссылку на другой объект (в данном случае ключ из

хеш-таблицы). Объекты этого типа интерпретируются системой сборки “мусора” особым образом. Если система сборки “мусора” обнаруживает отсутствие ссылок на конкретный объект, то она, как правило, освобождает занятую им память. А если объект доступен только из хеш-отображения типа `WeakHashMap`, то система сборки “мусора” освобождает и его, но размещает в очереди слабую ссылку на него. В операциях, выполняемых над хеш-отображением типа `WeakHashMap`, эта очередь периодически проверяется на предмет появления новых слабых ссылок. Появление такой ссылки в очереди свидетельствует о том, что ключ больше не используется нигде, но по-прежнему хранится в коллекции. В таком случае связанный с ним элемент удаляется из хеш-отображения типа `WeakHashMap`.

9.4.5. Связные хеш-множества и отображения

Классы `LinkedHashSet` и `LinkedHashMap` запоминают порядок ввода в них элементов. Таким образом, можно избежать кажущегося случайным порядка расположения элементов в хеш-таблице. По мере ввода элементов в таблицу они присоединяются к двунаправленному связному списку (рис. 9.11).

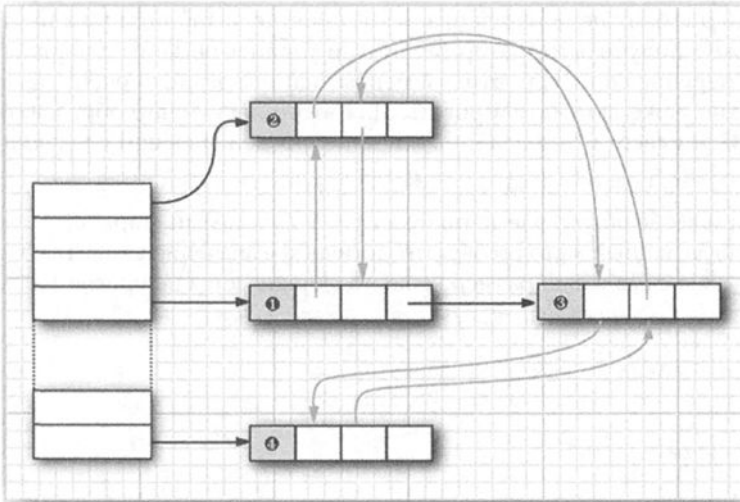


Рис. 9.11. Связная хеш-таблица

В качестве примера рассмотрим следующие элементы, введенные в отображение из листинга 9.6, но теперь это отображение типа `LinkedHashMap`:

```
var staff = new LinkedHashMap<String, Employee>();  
staff.put("144-25-5464", new Employee("Amy Lee"));  
staff.put("567-24-2546", new Employee("Harry Hacker"));  
staff.put("157-62-7935", new Employee("Gary Cooper"));  
staff.put("456-62-5527", new Employee("Francesca Cruz"));
```

Затем вызываемый итератор `staff.keySet().iterator()` перечисляет ключи в следующем порядке:

```
144-25-5464  
567-24-2546  
157-62-7935  
456-62-5527
```

А вызываемый итератор `staff.values().iterator()` перечисляет значения в таком порядке:

Amy Lee
Harry Hacker
Gary Cooper
Francesca Cruz

Связное хеш-отображение позволяет сменить порядок ввода на *порядок доступа* для перебора его элементов. При каждом вызове метода `get()` или `put()` затрагиваемый элемент удаляется из его текущей позиции и вводится в *конец* связанного списка элементов. (Затрагивается только позиция в связанном списке элементов, а не в группах хеш-таблицы. Элемент всегда остается на том месте, которое соответствует хеш-коду его ключа.) Чтобы сконструировать такое хеш-отображение, достаточно сделать следующий вызов:

```
LinkedHashMap<K, V>(initialCapacity, loadFactor, true)
```

Порядок доступа удобен для реализации дисциплины кэширования с так называемым “наиболее давним использованием”. Допустим, требуется сохранять в памяти часто используемые элементы, а менее часто используемые вводить из базы данных. Если нужный элемент не обнаруживается в таблице, а таблица уже почти заполнена, тогда можно получить итератор таблицы и удалить несколько первых элементов, которые он перечисляет. Ведь эти элементы использовались очень давно. Данный процесс можно даже автоматизировать. Для этого достаточно образовать подкласс, производный от класса `LinkedHashMap`, и переопределить в нем следующий метод:

```
protected boolean removeEldestEntry(Map.Entry<K, V> eldest)
```

И тогда при вводе нового элемента будет удаляться самый старый (`eldest`) элемент всякий раз, когда данный метод возвратит логическое значение `true`. Так, в следующем примере кода максимальный размер кеша поддерживается на уровне 100 элементов:

```
var cache = new LinkedHashMap<K, V>(128, 0.75F, true)
{
    protected boolean removeEldestEntry(
        Map.Entry<K, V> eldest)
    {
        return size() > 100;
    }
};
```

А с другой стороны, можно проанализировать элемент `eldest`, чтобы решить, стоит ли его удалять. Например, можно проверить отметку времени, хранящуюся в данном элементе.

9.4.6. Перечислимые множества и отображения

В классе `EnumSet` эффективно реализуется множество элементов, относящихся к перечислимому типу. А поскольку у перечислимого типа ограниченное количество экземпляров, то класс `EnumSet` реализован внутренним образом в виде битовой последовательности. Каждый бит устанавливается, если соответствующее значение перечисления присутствует в множестве.

У класса `EnumSet` отсутствуют открытые конструкторы. Для конструирования перечислимого множества используется статический фабричный метод, как показано ниже. А для видоизменения перечислимого множества типа `EnumSet` можно использовать обычные методы из интерфейса `Set`.

```
enum Weekday { MONDAY, TUESDAY, WEDNESDAY,
               THURSDAY, FRIDAY, SATURDAY, SUNDAY };
EnumSet<Weekday> always = EnumSet.allOf(Weekday.class);
EnumSet<Weekday> never = EnumSet.noneOf(Weekday.class);
EnumSet<Weekday> workday = EnumSet.range(Weekday.MONDAY,
                                         Weekday.FRIDAY);
EnumSet<Weekday> mwf = EnumSet.of(Weekday.MONDAY,
                                  Weekday.WEDNESDAY, Weekday.FRIDAY);
```

Класс `EnumMap` реализует отображение с ключами, относящимися к перечислимому типу. Такое отображение реализуется просто и эффективно в виде массива значений. Для этого достаточно указать тип ключа в конструкторе следующим образом:

```
EnumMap<Weekday, Employee> personInCharge =
    new EnumMap<>(Weekday.class);
```



НА ЗАМЕТКУ! В документации на прикладной интерфейс API класса `EnumSet` можно обнаружить необычные параметры вроде `E extends Enum<E>`. Такое обозначение просто означает, что “`E` относится к перечислимому типу”. Перечислимые типы расширяют обобщенный класс `Enum`. Например, перечислимый тип `Weekday` расширяет класс `Enum<Weekday>`.

9.4.7. Хеш-отображения идентичности

Класс `IdentityHashMap` предназначен для построения хеш-отображения идентичности, преследующего особые цели, когда хеш-значения ключей должны вычисляться не методом `hashCode()`, а методом `System.identityHashCode()`. В этом методе для вычисления хеш-кода, исходя из адреса объекта в памяти, используется метод `Object.hashCode()`. Кроме того, для сравнения объектов типа `IdentityHashMap` применяется операция `==`, а не вызывается метод `equals()`.

Иными словами, разные объекты-ключи рассматриваются как отличающиеся, даже если они имеют одинаковое содержимое. Этот класс удобен для реализации алгоритмов обхода объектов (например, сериализации объектов), когда требуется отслеживать уже пройденные объекты.

```
java.util.WeakHashMap<K, V> 1.2
```

- `WeakHashMap()`
- `WeakHashMap(int initialCapacity)`
- `WeakHashMap(int initialCapacity, float loadFactor)`

Конструируют пустое хеш-отображение заданной емкости с указанным коэффициентом загрузки.

```
java.util.LinkedHashSet<E> 1.4
```

- `LinkedHashSet()`
- `LinkedHashSet(int initialCapacity)`
- `LinkedHashSet(int initialCapacity, float loadFactor)`

Конструируют пустое связанное хеш-множество заданной емкости с указанным коэффициентом загрузки.

java.util.LinkedHashMap<K, V> 1.4

- **LinkedHashMap()**
- **LinkedHashMap(int initialCapacity)**
- **LinkedHashMap(int initialCapacity, float loadFactor)**
- **LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)**

Конструируют пустое связанное хеш-отображение заданной емкости с указанным коэффициентом загрузки и упорядочением. Логическое значение **true** параметра **accessOrder** задает порядок доступа, а его логическое значение **false** — порядок ввода.

- **protected boolean removeEldestEntry(Map.Entry<K, V> eldest)**

Этот метод должен быть переопределен, чтобы возвращать логическое значение **true**, если требуется удалить самый старый элемент. Параметр **eldest** обозначает самый старый элемент, который предполагается удалить. Данный метод вызывается после того, как в отображение введен элемент. В его реализации по умолчанию возвращается логическое значение **false**, т.е. старые элементы по умолчанию не удаляются. Но этот метод можно переопределить для выборочного возврата логического значения **true**, если, например, самый старый элемент удовлетворяет определенным условиям или размеры отображения достигают определенной величины.

java.util.EnumSet<E extends Enum<E>> 5

- **static <E extends Enum<E>> EnumSet<E> allOf(Class<E> enumType)**
Возвращает множество, содержащее все значения заданного перечислимого типа.
- **static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> enumType)**
Возвращает пустое множество, способное хранить значения заданного перечислимого типа.
- **static <E extends Enum<E>> EnumSet<E> range(E from, E to)**
Возвращает множество, содержащее все значения от **from** до **to** включительно.
- **static <E extends Enum<E>> EnumSet<E> of(E value)**
- **static <E extends Enum<E>> EnumSet<E> of(E value, E... values)**
Возвращают множество, содержащее заданные значения.

java.util.EnumMap<K extends Enum<K>, V> 5

- **EnumMap(Class<K> keyType)**
Конструирует пустое отображение с ключами заданного типа.

java.util.IdentityHashMap<K, V> 1.4

- **IdentityHashMap()**
- **IdentityHashMap(int expectedMaxSize)**

Конструируют пустое хеш-отображение идентичности, емкость которого равна минимальной степени 2, превышающей величину $1.5 \times \text{expectedMaxSize}$ (по умолчанию значение параметра **expectedMaxSize** равно 21).

```
java.lang.System 1.0
```

- `static int identityHashCode(Object obj) 1.1`

Возвращает хеш-код, вычисляемый методом `Object.hashCode()`, исходя из адреса памяти, по которому хранится объект, даже если в классе, к которому относится заданный объект `obj`, переопределяется метод `hashCode()`.

9.5. Представления и оболочки

Глядя на рис. 9.4 и 9.5, можно подумать, что излишне иметь так много интерфейсов и абстрактных классов для реализации столь скромного количества конкретных классов коллекций. Но эти рисунки не отражают всей картины. Используя *представления*, можно получить объекты других классов, реализующих интерфейс `Collection` или `Map`. Характерным тому примером служит упоминавшийся ранее метод `keySet()` из классов отображений. На первый взгляд данный метод создает новое множество, заполняет его всеми ключами из отображения и возвращает его. Но это не совсем так. Напротив, метод `keySet()` возвращает объект класса, который реализует интерфейс `Set` и методы которого манипулируют исходным отображением. Такая коллекция называется *представлением*. У методики представлений имеется ряд полезных применений в каркасе коллекций. Эти применения будут обсуждаться в последующих подразделах.

9.5.1. Мелкие коллекции

В версии Java 9 внедрены статические методы, выдающие множество или список заданных элементов, а также отображение с заданными парами “ключ–значение”. Так, в следующем примере кода получается список и множество трех элементов:

```
List<String> names = List.of("Peter", "Paul", "Mary");  
Set<Integer> numbers = Set.of(2, 3, 5);
```

А для того чтобы получить отображение, достаточно указать ключи и значения, как показано ниже. Элементы, ключи или значения не должны быть пустыми.

```
Map<String, Integer> scores = Map.of("Peter", 2,  
                                     "Paul", 3, "Mary", 5);
```

В интерфейсах `List` и `Set` насчитывается одиннадцать методов типа `of`, принимающих от нуля до десяти аргументов, а также метод `of()` с переменным числом аргументов. Такое разнообразие методов типа `of` предоставляется из соображений эффективности.

А в интерфейсе `Map` невозможно предоставить вариант метода `of()` с переменным числом аргументов, поскольку типы аргументов сменяются на типы ключей и значений. Но в то же время имеется статический метод `ofEntries()`, принимающий произвольное количество объектов типа `Map.Entry<K, V>`, которые можно создать с помощью статического метода `entry()`, как демонстрируется в следующем примере кода:

```
import static java.util.Map.*;  
...  
Map<String, Integer> scores = ofEntries(  
    entry("Peter", 2),  
    entry("Paul", 3),  
    entry("Mary", 5);
```



```
entry("Peter", 2),  
entry("Paul", 3),  
entry("Mary", 5));
```

Методы `of()` и `ofEntries()` производят объекты тех классов, в которых каждому элементу коллекции предоставляется переменная экземпляра или поддержка со стороны массива. Объекты таких коллекций *неизменяемы*. Любая попытка изменить их содержимое приводит к исключению типа `UnsupportedOperationException`.

Если же потребуется изменяемая коллекция, для этого достаточно передать неизменяемую коллекцию конструктору соответствующего класса, как показано ниже.

```
var names = new ArrayList<>(  
    List.of("Peter", "Paul", "Mary"));
```

В результате вызова метода

```
Collections.nCopies(n, anObject)
```

возвращается неизменяемый объект, реализующий интерфейс `List` и создающий впечатление, будто коллекция состоит из `n` элементов, каждый из которых оказывается объектом `anObject`.

Например, в результате приведенного ниже вызова создается список из 100 символьных строк `"DEFAULT"`. И делается это с минимальными затратами оперативной памяти, поскольку соответствующий объект сохраняется лишь один раз.

```
List<String> settings = Collections.nCopies(100, "DEFAULT");
```



НА ЗАМЕТКУ! Методы типа `of` были внедрены в версии Java 9. А прежде имелся статический метод `Arrays.asList()`, возвращавший изменяемый список постоянной длины. Это означает, что для такого списка можно вызвать метод `set()`, но не метод `add()` или `remove()`. В классе `Collections` имеются также унаследованные методы `emptySet()` `singleton()`.



НА ЗАМЕТКУ! В состав класса `Collections` входит ряд служебных методов, принимающих параметры или возвращающих значения, которые являются коллекциями. Этот класс не следует путать с интерфейсом `Collection`.



СОВЕТ. В стандартной библиотеке классов Java отсутствует класс `Pair`, и поэтому некоторые программисты пользуются внутренним классом `Map.Entry` в качестве дешевой замены пары. До версии Java 9 это было очень неудобно, поскольку такую пару приходилось строить с помощью операции `new AbstractMap.SimpleImmutableEntry<>(first, second)`. А теперь для этого достаточно сделать вызов `Map.entry(first, second)`.

9.5.2. Поддиапазоны

Для ряда коллекций можно сформировать представления поддиапазонов. Допустим, имеется список `staff`, из которого требуется извлечь с 10-го по 19-й элемент. Чтобы получить представление этого поддиапазона элементов списка, достаточно вызвать метод `subList()`, как показано ниже. Первый индекс поддиапазона указывается включительно, а второй индекс — исключительно аналогично параметрам метода `substring()` из класса `String`.

```
List<Employee> group2 = staff.subList(10, 20);
```

Над поддиапазонами можно выполнять любые операции, которые автоматически отражают целый список. Например, весь поддиапазон можно стереть приведенным

ниже способом. В итоге элементы автоматически удаляются из списка `staff`, и поддиапазон `group2` опорожняется.

```
group2.clear(); // сокращение штатов
```

Что касается отсортированных множеств и отображений, то для формирования из них поддиапазонов можно воспользоваться порядком сортировки, а не расположением элементов. Так, в интерфейсе `SortedSet` объявляются три метода:

```
SortedSet<E> subSet(E from, E to)
SortedSet<E> headSet(E to)
SortedSet<E> tailSet(E from)
```

Эти методы возвращают подмножества всех элементов, которые оказываются больше или равными параметру `from`, но строго меньше параметра `to`. Для отсортированных отображений имеются аналогичные методы, возвращающие представления, состоящие из всех элементов, где ключи находятся в указанных пределах.

```
SortedMap<K, V> subMap(K from, K to)
SortedMap<K, V> headMap(K to)
SortedMap<K, V> tailMap(K from)
```

В версии Java 6 был внедрен интерфейс `NavigableSet`, обеспечивающий более полный контроль операций над поддиапазонами. Указать включаемые границы поддиапазонов можно следующим образом:

```
NavigableSet<E> subSet(E from, boolean fromInclusive,
                      E to, boolean toInclusive)
NavigableSet<E> headSet(E to, boolean toInclusive)
NavigableSet<E> tailSet(E from, boolean fromInclusive)
```

9.5.3. Немодифицируемые представления

В состав класса `Collections` входят методы, производящие *немодифицируемые представления* коллекций. Такие представления вводят динамическую проверку в существующие коллекции. Если в ходе такой проверки обнаруживается попытка видоизменить коллекцию, генерируется исключение и коллекция остается невредимой. Получить немодифицируемые представления можно следующими восемью методами:

```
Collections.unmodifiableCollection
Collections.unmodifiableList
Collections.unmodifiableSet
Collections.unmodifiableSortedSet
Collections.unmodifiableNavigableSet
Collections.unmodifiableMap
Collections.unmodifiableSortedMap
Collections.unmodifiableNavigableMap
```

Каждый из этих методов определен для работы с интерфейсом. Так, метод `Collections.unmodifiableList()` работает с классом `ArrayList`, `LinkedList` или любым другим, реализующим интерфейс `List`. Допустим, что некоторой части прикладной программы требуется предоставить возможность просматривать, но не изменять содержимое коллекции. В приведенном ниже фрагменте кода показано, как это можно сделать.

```
var staff = new LinkedList<String>();
...
lookAt(Collections.unmodifiableList(staff));
```

Метод `Collections.unmodifiableList()` возвращает объект класса, реализующего интерфейс `List`. Его методы доступа извлекают значения из коллекции `staff`. Безусловно, метод `lookAt()` может вызывать все методы из интерфейса `List`, а не только методы доступа. Но все методы, изменяющие коллекцию, например `add()`, переопределены таким образом, чтобы немедленно генерировать исключение типа `UnsupportedOperationException` вместо передачи вызова базовой коллекции.

Немодифицируемое представление не делает саму коллекцию немодифицируемой. Коллекцию можно по-прежнему видоизменить по ее исходной ссылке (в данном случае — `stuff`). А модифицирующие методы можно вызывать для отдельных элементов коллекции.

Представления заключают в оболочку *интерфейс*, а не конкретный объект коллекции, и поэтому доступ требуется лишь к тем методам, которые определены в интерфейсе. Например, в состав класса `LinkedList` входят служебные методы `addFirst()` и `addLast()`, не являющиеся частью интерфейса `List`, но они недоступны через немодифицируемое представление.



ВНИМАНИЕ! Метод `unmodifiableCollection()` (как, впрочем, и методы `synchronizedCollection()` и `checkedCollection()`, о которых речь пойдет далее) возвращает коллекцию, в которой метод `equals()` не вызывает одноименный метод из базовой коллекции. Вместо этого он наследует метод `equals()` из класса `Object`, который проверяет объекты на равенство. Если просто преобразовать множество или список в коллекцию, то проверить ее содержимое на равенство не удастся.

Представление действует подобным образом, потому что проверка на равенство четко определена на данном уровне иерархии. Аналогичным образом в представлениях интерпретируется и метод `hashCode()`. Но в классах объектов, возвращаемых методами `unmodifiableSet()` и `unmodifiableList()`, используются методы `equals()` и `hashCode()` из базовой коллекции.

9.5.4. Синхронизированные представления

Если обращение к коллекции происходит из нескольких потоков исполнения, то нужно каким-то образом исключить ее непреднамеренное повреждение. Было бы, например, губительно, если бы в одном потоке исполнения была предпринята попытка ввести элемент в хеш-таблицу в тот момент, когда в другом потоке повторно хешировались ее элементы.

Вместо реализации потокобезопасных классов разработчики библиотеки коллекций воспользовались механизмом представлений, чтобы сделать потокобезопасными обычные коллекции. Например, статический метод `synchronizedMap()` из класса `Collections` может превратить любое отображение в объект типа `Map` с синхронизированными методами доступа следующим образом:

```
var map = Collections.synchronizedMap(  
    new HashMap<String, Employee>());
```

После этого к объекту `map` можно обращаться из нескольких потоков исполнения. Такие методы, как `get()` и `put()`, сериализованы. Это означает, что каждый вызов метода должен полностью завершаться до того, как другой поток сможет его вызвать. Вопросы синхронизированного доступа к структурам данных подробнее обсуждаются в главе 12.

9.5.5. Проверяемые представления

Проверяемые представления предназначены для поддержки отладки ошибок, сопутствующих применению обобщенных типов в прикладном коде. Как пояснялось в главе 8, существует возможность незаконно внедрить в обобщенную коллекцию элементы неверного типа. Например, в приведенном ниже фрагменте кода ошибочный вызов метода `add()` не обнаруживается.

```
var strings = new ArrayList<String>();
ArrayList rawList = strings; // ради совместимости с
    // унаследованным кодом при компиляции этой строки кода
    // выдается только предупреждение, но не ошибка
rawList.add(new Date()); // теперь символьные строки
    // содержат объект типа Date!
```

Вместо этого возникнет исключение, когда при последующем вызове метода `get()` будет сделана попытка привести результат к типу `String`. Проверяемое представление позволяет обнаружить этот недостаток в коде. Для этого сначала определяется безопасный список:

```
List<String> safeStrings = Collections.checkedList(
    strings, String.class);
```

Затем в методе представления `add()` проверяется принадлежность объекта, вводимого в коллекцию, заданному классу. И если обнаружится несоответствие, то немедленно сгенерируется исключение типа `ClassCastException`, как показано ниже. Преимущество такого подхода заключается в том, что ошибка произойдет в том месте кода, где ее можно обнаружить и обработать.

```
ArrayList rawList = safeStrings;
rawList.add(new Date()); // проверяемый список
    // генерирует исключение типа ClassCastException
```



ВНИМАНИЕ! Проверяемые представления ограничиваются динамическими проверками, которые способна выполнить виртуальная машина. Так, если имеется списочный массив типа `ArrayList<Pair<String>>`, его нельзя защитить от ввода элемента типа `Pair<Date>`, поскольку виртуальной машине ничего неизвестно о единственном базовом классе `Pair`.

9.5.6. О необязательных операциях

Обычно на представление накладывается определенное ограничение: оно может быть доступно только для чтения, может не допускать изменений размера или поддерживать удаление, но запрещать ввод элементов, как это имеет место для представления ключей в отображении. Ограниченное таким образом представление генерирует исключение типа `UnsupportedOperationException`, если попытаться выполнить неразрешенную операцию.

В документации на прикладной интерфейс API для интерфейсов коллекций и итераторов многие методы описаны как “необязательные операции”. На первый взгляд, это противоречит самому понятию интерфейса. Разве не в том назначение интерфейсов, чтобы объявлять методы, которые класс *обязан* реализовать? На самом деле такая организация неудовлетворительна с теоретической точки зрения. Возможно, лучше было бы разработать отдельные интерфейсы для представлений “только для чтения” и представлений, которые не могут изменять размер коллекции. Но это привело бы к значительному увеличению количества интерфейсов, что разработчики библиотеки сочли неприемлемым.

Следует ли распространять методику “необязательных” методов на собственные проекты? Вряд ли. Даже если коллекции используются часто, стиль программирования для их реализации нетипичен для других предметных областей. Разработчикам библиотеки классов коллекций пришлось удовлетворить ряд жестких и противоречивых требований. Пользователи хотели бы, чтобы библиотеку было легко усвоить и удобно использовать, чтобы она была полностью обобщенной, защищенной от неумелого обращения и такой же эффективной, как и алгоритмы, разработанные вручную. Удовлетворить всем этим требованиям одновременно или даже приблизиться к этой цели практически невозможно. Но в своей практике программирования на Java вы редко столкнетесь с таким суровым рядом ограничений. Тем не менее вы должны быть готовы находить решения, которые не опираются на такую крайнюю меру, как применение “необязательных” интерфейсных операций.

`java.util.List<E>` 1.2

- `static <E> List<E> of() 9`
- `static <E> List<E> of(E e1) 9`
-
- `static <E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10) 9`
- `static <E> Set<E> of(E... elements) 9`

Возвращают неизменяемый список заданных элементов, которые не должны содержать пустое значение `null`.

`java.util.Set` 1.2

- `static <E> Set<E> of() 9`
- `static <E> Set<E> of(E e1) 9`
- ...
- `static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10) 9`

Возвращают неизменяемое множество заданных элементов, которые не должны содержать пустое значение `null`.

`java.util.Map` 1.2

- `static <K, V> Map<K, V> of() 9`
- `static <K, V> Map<K, V> of(K k1, V v1) 9`
- ...
- `static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8, K k9, V v9, K k10, V v10) 9`

Возвращают неизменяемое отображение заданных ключей и значений, которые не должны быть пустыми `[null]`.

java.util.Map 1.2 (окончание)

- **static <K,V> Map.Entry<K,V> entry(K k, V v)** 9

Возвращают из неизменяемого отображения запись с заданным ключом и значением, которые не должны быть пустыми [null].

- **static <K,V> Map<K,V> ofEntries(Map.Entry<? extends K,? extends V>... entries)** 9

Возвращают неизменяемое отображение заданных записей.

java.util.Collections 1.2

- **static <E> Collection unmodifiableCollection(Collection<E> c)**
- **static <E> List unmodifiableList(List<E> c)**
- **static <E> Set unmodifiableSet(Set<E> c)**
- **static <E> SortedSet unmodifiableSortedSet(SortedSet<E> c)**
- **static <E> NavigableSet synchronizedNavigableSet(NavigableSet<E> c)** 8
- **static <K, V> Map unmodifiableMap(Map<K, V> c)**
- **static <K, V> SortedMap unmodifiableSortedMap(SortedMap<K, V> c)**
- **static <K, V> NavigableMap<K, V> synchronizedNavigableMap(NavigableMap<K, V> c)** 8

Конструируют представления коллекций, в которых модифицирующие методы генерируют исключение типа `UnsupportedOperationException`.

- **static <E> Collection<E> synchronizedCollection(Collection<E> c)**
- **static <E> List synchronizedList(List<E> c)**
- **static <E> Set synchronizedSet(Set<E> c)**
- **static <E> SortedSet synchronizedSortedSet(SortedSet<E> c)**
- **static <E> NavigableSet synchronizedNavigableSet(NavigableSet<E> c)** 8
- **static <K, V> Map<K, V> synchronizedMap(Map<K, V> c)**
- **static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> c)**
- **static <K, V> NavigableMap<K, V> synchronizedNavigableMap(NavigableMap<K, V> c)** 8

Конструируют представления коллекций, методы которых синхронизированы.

- **static <E> Collection<E> checkedCollection(Collection<E> c, Class<E> elementType)**
- **static <E> List checkedList(List<E> c)**
- **static <E> Set checkedSet(Set<E> c, Class<E> elementType)**
- **static <E> SortedSet checkedSortedSet(SortedSet<E> c, Class<E> elementType)**
- **static <E> NavigableSet checkedNavigableSet(NavigableSet<E> c, Class<E> elementType)** 8
- **static <K, V> Map<K, V> checkedMap(Map<K, V> c, Class<K> keyType, Class<V> valueType)**

java.util.Collections 1.2 (окончание)

- `static <K, V> SortedMap<K, V> checkedSortedMap(SortedMap<K, V> c, Class<K> keyType, Class<V> valueType)`
- `static <K, V> NavigableMap checkedNavigableMap(NavigableMap<K, V> c, Class<K> keyType, Class<V> valueType)` 8
- `static <E> Queue<E> checkedQueue(Queue<E> queue, Class<E> elementType)` 8
Конструируют представления коллекций с методами, генерирующими исключение типа `ClassCastException`, если вводится элемент неверного типа.
- `static <E> List<E> nCopies(int n, E value)`
Возвращает неизменяемый список, состоящий из *n* одинаковых значений.
- `static <E> List<E> singletonList(E value)`
- `static <E> Set<E> singleton(E value)`
- `static <K, V> Map<K, V> singletonMap(K key, V value)`
Возвращают список, множество или отображение одиночных объектов. Начиная с версии Java 9, следует пользоваться одним из этих методов.
- `static <E> List<E> emptyList()`
- `static <T> Set<T> emptySet()`
- `static <E> SortedSet<E> emptySortedSet()`
- `static NavigableSet<E> emptyNavigableSet()`
- `static <K,V> Map<K,V> emptyMap()`
- `static <K,V> SortedMap<K,V> emptySortedMap()`
- `static <K,V> NavigableMap<K,V> emptyNavigableMap()`
- `static <T> Enumeration<T> emptyEnumeration()`
- `static <T> Iterator<T> emptyIterator()`
- `static <T> ListIterator<T> emptyListIterator()`
Выдают пустую коллекцию, отображение или итератор.

java.util.Arrays 1.2

- `static <E> List<E> asList(E... array)`
Возвращает представление списка элементов массива, который является модифицируемым, но с неизменяемым размером.

java.util.List<E> 1.2

- `List<E> subList(int firstIncluded, int firstExcluded)`
Возвращает представление списка элементов в заданном диапазоне позиций.

java.util.SortedSet<E> 1.2

- `SortedSet<E> subSet(E firstIncluded, E firstExcluded)`
- `SortedSet<E> headSet(E firstExcluded)`
- `SortedSet<E> tailSet(E firstIncluded)`

Возвращают представление элементов отсортированного множества в заданном диапазоне.

java.util.NavigableSet<E> 6

- `NavigableSet<E> subSet(E from, boolean fromIncluded, E to, boolean toIncluded)`
- `NavigableSet<E> headSet(E to, boolean toIncluded)`
- `NavigableSet<E> tailSet(E from, boolean fromIncluded)`

Возвращают представление элементов множества в заданном диапазоне. Параметры типа `boolean` определяют, следует ли включать в представление заданные границы диапазона.

java.util.SortedMap<K, V> 1.2

- `SortedMap<K, V> subMap(K firstIncluded, K firstExcluded)`
- `SortedMap<K, V> headMap(K firstExcluded)`
- `SortedMap<K, V> tailMap(K firstIncluded)`

Возвращают представление элементов отсортированного отображения в заданном диапазоне.

java.util.NavigableMap<K, V> 6

- `NavigableMap<K, V> subMap(K from, boolean fromIncluded, K to, boolean toIncluded)`
- `NavigableMap<K, V> headMap(K from, boolean fromIncluded)`
- `NavigableMap<K, V> tailMap(K to, boolean toIncluded)`

Возвращают представление элементов отображения в заданном диапазоне. Параметры типа `boolean` определяют, следует ли включать в представление заданные границы диапазона.

9.6. Алгоритмы

Помимо реализации классов коллекций, в каркасе коллекций Java предоставляется целый ряд полезных алгоритмов. В последующих разделах поясняется, как пользоваться этими алгоритмами и составлять собственные алгоритмы, вполне пригодные для каркаса коллекций.

9.6.1. Назначение обобщенных алгоритмов

Обобщенные интерфейсы коллекций дают огромное преимущество: конкретный алгоритм достаточно реализовать лишь один раз. В качестве примера рассмотрим простой алгоритм вычисления наибольшего элемента в коллекции. Традиционно он реализуется в цикле. Ниже показано, как обнаружить самый большой элемент в массиве традиционным способом.

```
if (a.length == 0) throw new NoSuchElementException();
T largest = a[0];
for (int i = 1; i < a.length; i++)
    if (largest.compareTo(a[i]) < 0)
        largest = a[i];
```

Разумеется, чтобы найти наибольший элемент в списочном массиве, придется написать код немного иначе:

```
if (v.size() == 0) throw new NoSuchElementException();
T largest = v.get(0);
for (int i = 1; i < v.size(); i++)
    if (largest.compareTo(v.get(i)) < 0)
        largest = v.get(i);
```

А как насчет связанного списка? Для связанного списка не существует эффективного произвольного доступа, но можно воспользоваться итератором:

```
if (l.isEmpty()) throw new NoSuchElementException();
Iterator<T> iter = l.iterator();
T largest = iter.next();
while (iter.hasNext())
{
    T next = iter.next();
    if (largest.compareTo(next) < 0)
        largest = next;
}
```

Все эти циклы писать довольно утомительно, а кроме того, они чреваты ошибками. Нужно постоянно проверять себя: не допущена ли ошибка выхода за допустимые пределы, правильно ли будут выполняться эти циклы с пустыми коллекциями, а как насчет контейнеров с одним элементом? Вряд ли захочется тестировать и отлаживать такой код каждый раз, но и желания реализовывать уйму методов вроде приведенных ниже также не возникнет.

```
static <T extends Comparable> T max(T[] a)
static <T extends Comparable> T max(ArrayList<T> v)
static <T extends Comparable> T max(LinkedList<T> l)
```

И здесь на помощь приходят интерфейсы коллекций. Подумайте о минимальном интерфейсе коллекции, который понадобится для эффективной реализации алгоритма. Произвольный доступ с помощью методов `get()` и `set()` стоит рангом выше, чем простая итерация. Как следует из приведенного выше примера вычисления наибольшего элемента в связанном списке, произвольный доступ совсем не обязателен для решения подобной задачи. Вычисление максимума может быть выполнено путем простой итерации по элементам коллекции. Следовательно, метод `max()` можно воплотить в коде таким образом, чтобы он принимал объект *любого* класса, реализующего интерфейс `Collection`:

```
public static <T extends Comparable> T max(Collection<T> c)
{
    if (c.isEmpty()) throw new NoSuchElementException();
    Iterator<T> iter = c.iterator();
    T largest = iter.next();
    while (iter.hasNext())
    {
        T next = iter.next();
        if (largest.compareTo(next) < 0)
            largest = next;
    }
    return largest;
}
```

Теперь единственным обобщенным методом можно вычислить максимум в связанном списке, в списочном или простом массиве. Это весьма эффективный алгоритм. На самом деле в стандартной библиотеке C++ имеются десятки полезных алгоритмов, каждый из которых оперирует обобщенной коллекцией. Библиотека Java не настолько богата, но в ней все же имеются самые основные алгоритмы: сортировки, двоичного поиска, а также некоторые служебные алгоритмы.

9.6.2. Сортировка и перетасовка

Ветераны программирования иногда вспоминают о том, как им приходилось пользоваться перфокартами и программировать алгоритмы сортировки вручную. Ныне алгоритмы сортировки уже вошли в состав стандартных библиотек большинства языков программирования, и в этом отношении Java не является исключением. Так, метод `sort()` из класса `Collections` сортирует коллекцию, реализующую интерфейс `List`, следующим образом:

```
List<String> staff = new LinkedList<>();
заполнить коллекцию
Collections.sort(staff);
```

В этом методе предполагается, что элементы списка реализуют интерфейс `Comparable`. Если же требуется отсортировать список каким-то другим способом, можно вызвать метод `sort()` из интерфейса `List`, передав ему объект типа `Comparator` в качестве параметра. Ниже показано, как отсортировать список элементов.

```
staff.sort(Comparator.comparingDouble(
    Employee::getSalary));
```

Если требуется отсортировать список по *убывающей*, следует воспользоваться служебным статическим методом `Comparator.reverseOrder()`. Он возвращает компаратор, который, в свою очередь, возвращает результат вызова `b.compareTo(a)`. Например, в приведенной ниже строке кода элементы списка `staff` сортируются в обратном порядке, который задается методом `compareTo()` для типа элементов списка.

```
staff.sort(Comparator.reverseOrder())
```

Аналогично в следующей строке кода порядок сортировки изменяется на обратный:

```
staff.sort(Comparator.comparingDouble(
    Employee::getSalary).reversed())
```

Вас может заинтересовать, каким образом метод `sort()` сортирует список. Если проанализировать алгоритм сортировки, рассматриваемый в специальной

литературе по алгоритмам, то он обычно поясняется на примере обычных массивов с произвольным доступом к элементам. Но ведь произвольный доступ к элементам списка неэффективен. Списки лучше сортировать, используя алгоритм сортировки слиянием. Но в реализации на Java этого не делается. Напротив, все элементы выводятся в массив, который затем сортируется с помощью другой разновидности сортировки слиянием, после чего отсортированная последовательность копируется обратно в список.

Алгоритм сортировки слиянием, применяемый в библиотеке коллекций, немного медленнее быстрой сортировки, традиционно выбираемой для алгоритмов сортировки общего назначения. Но у него имеется следующее важное преимущество: он *устойчив*, т.е. не меняет местами равнозначные элементы. А зачем вообще беспокоиться о порядке следования равнозначных элементов? Рассмотрим распространенный случай. Допустим, имеется список работников, который уже отсортирован по их Ф.И.О., а теперь их нужно отсортировать по зарплате. Как же будут отсортированы работники с одинаковой зарплатой? При устойчивой сортировке упорядочение по Ф.И.О. сохраняется. Иными словами, в конечном итоге получится список, отсортированный сначала по зарплате, а затем по Ф.И.О. работников.

В коллекциях не нужно реализовывать все “необязательные” методы, поэтому все методы, принимающие коллекции в качестве параметров, должны указывать, когда безопасно передавать коллекцию алгоритму. Например, совершенно очевидно, что вряд ли стоит передавать немодифицируемый список алгоритму сортировки. Какие же списки можно передавать? Согласно документации, список должен быть модифицируемым, но его размер не должен быть изменяемым. Ниже поясняется, что все это означает.

- Список является *модифицируемым*, если он поддерживает метод `set()`.
- Список имеет *изменяемый размер*, если он поддерживает методы `add()` и `remove()`.

В классе `Collections` реализован алгоритм перетасовки и соответствующий метод `shuffle()`, который выполняет задачу, противоположную сортировке, изменяя случайным образом порядок расположения элементов в списке, как показано в приведенном ниже примере кода.

```
ArrayList<Card> cards = . . . ;  
Collections.shuffle(cards);
```

Если предоставить список, который не реализует интерфейс `RandomAccess`, то метод `shuffle()` скопирует все его элементы в массив, перетасует его, после чего скопирует перетасованные элементы обратно в список.

В примере программы из листинга 9.7 списочный массив заполняется 49 объектами типа `Integer`, содержащими числа от 1 до 49. Затем они перетасовываются случайным образом в списке, откуда далее выбираются первые 6 значений. И, наконец, выбранные значения сортируются и выводятся.

Листинг 9.7. Исходный код из файла `shuffle/ShuffleTest.java`

```
1 package shuffle;  
2  
3 import java.util.*;  
4  
5 /**
```

```

6  * В этой программе демонстрируются алгоритмы
7  * произвольной перетасовки и сортировки
8  * @version 1.12 2018-04-10
9  * @author Cay Horstmann
10 */
11 public class ShuffleTest
12 {
13     public static void main(String[] args)
14     {
15         var numbers = new ArrayList<Integer>();
16         for (int i = 1; i <= 49; i++)
17             numbers.add(i);
18         Collections.shuffle(numbers);
19         List<Integer> winningCombination =
20             numbers.subList(0, 6);
21         Collections.sort(winningCombination);
22         System.out.println(winningCombination);
23     }
24 }

```

java.util.Collections 1.2

- **static <T extends Comparable<? super T>> void sort(List<T> elements)**
Сортируют элементы в списке, используя алгоритм устойчивой сортировки. Выполнение алгоритма гарантируется за время $O(n \log n)$, где n — длина списка.
- **static void shuffle(List<?> elements)**
- **static void shuffle(List<?> elements, Random r)**
Случайно перетасовывают элементы в списке. Этот алгоритм выполняется за время $O(n a(n))$, где n — длина списка, тогда как $a(n)$ — среднее время доступа к элементу списка.

java.util.List<E> 1.2

- **default void sort(Comparator<? super T> comparator)** 8
Сортирует данный список, используя указанный компаратор.

java.util.Comparator<T> 1.2

- **static <T extends Comparable<? super T>> Comparator<T> reverseOrder()** 8
Выдает компаратор, обращающий порядок, обеспечиваемый интерфейсом `Comparable`.
- **default Comparator<T> reversed()** 8
Выдает компаратор, обращающий порядок, обеспечиваемый данным компаратором.

9.6.3. Двоичный поиск

Для поиска объекта в массиве все его элементы обычно перебираются до тех пор, пока не будет найден тот, который соответствует критерию поиска. Но если массив отсортирован, то можно сразу же перейти к среднему элементу и сравнить, больше ли

он искомого элемента. Если он больше, то поиск нужно продолжить в первой половине массива, а иначе — во второй половине. Благодаря этому задача поиска сокращается наполовину. Дальше поиск продолжается в том же духе. Так, если массив содержит 1024 элемента, то совпадение с искомым элементом (или его отсутствие в массиве) будет обнаружено после 10 шагов, тогда как для линейного поиска потребуется в среднем 512 шагов, если элемент присутствует в массиве, и 1024 шага, чтобы убедиться в его отсутствии.

Такой алгоритм двоичного поиска реализуется в методе `binarySearch()` из класса `Collections`. Следует, однако, иметь в виду, что коллекция уже должна быть отсортирована, иначе алгоритм даст неверный результат. Чтобы найти нужный элемент в коллекции, следует передать ее, при условии, что она реализует интерфейс `List`, а также элемент, который требуется найти. Если коллекция не отсортирована методом `compareTo()` из интерфейса `Comparable`, то необходимо предоставить также объект компаратора, как показано ниже.

```
i = Collections.binarySearch(c, element);  
i = Collections.binarySearch(c, element, comparator);
```

Неотрицательное числовое значение, возвращаемое методом `binarySearch()`, обозначает индекс найденного объекта. Следовательно, элемент `c.get(i)` равнозначен элементу `element` по порядку сравнения. Если же возвращается отрицательное числовое значение, это означает, что элемент не найден. Но возвращаемое значение можно использовать для определения места, где *следует* ввести элемент `element` в коллекцию, чтобы сохранить порядок сортировки. Место ввода элемента определяется следующим образом:

```
insertionPoint = -i - 1;
```

Это не просто значение `-i`, потому что нулевое значение было бы неоднозначным. Иными словами, приведенная ниже операция вводит элемент на нужном месте в коллекции.

```
if (i < 0)  
    c.add(-i - 1, element);
```

Алгоритм двоичного поиска нуждается в произвольном доступе, чтобы быть эффективным. Так, если итерацию приходится выполнять поэлементно на половине связанного списка, чтобы найти средний элемент, то теряются все преимущества двоичного поиска. Поэтому данный алгоритм в методе `binarySearch()` превращается в линейный поиск, если этому методу передается связный список.

`java.util.Collections 1.2`

- `static <T extends Comparable<? super T>> int binarySearch(List<T> elements, T key)`
- `static <T> int binarySearch(List<T> elements, T key, Comparator<? super T> c)`

Осуществляют поиск по указанному ключу *key* в отсортированном списке, используя линейный поиск, если заданный объект *elements* расширяет класс `AbstractSequentialList`, а иначе — двоичный поиск. Выполнение алгоритма поиска гарантируется за время $O(a(n) \log n)$, где n — длина списка, тогда как $a(n)$ — среднее время доступа к элементу. Возвращают индекс указанного ключа *key* в списке или отрицательное значение *i*, если ключ не найден в списке. В таком случае указанный ключ *key* должен быть введен на позиции с индексом `-i-1`, чтобы список остался отсортированным.

9.6.4. Простые алгоритмы

В состав класса `Collections` входит ряд простых, но полезных алгоритмов. Один из них был приведен ранее в примере кода, где обнаруживается наибольший элемент в коллекции. К числу других алгоритмов относится копирование элементов из одного списка в другой, заполнение контейнера постоянным значением и обращение списка.

Зачем же включать такие простые алгоритмы в стандартную библиотеку? Ведь их нетрудно реализовать в простых циклах. Простые алгоритмы удобны тем, что они упрощают чтение исходного кода. Когда вы читаете цикл, реализованный кем-то другим, вам приходится расшифровывать намерения автора этого кода. Рассмотрим в качестве примера следующий цикл:

```
for (int i = 0; i < words.size(); i++)
    if (words.get(i).equals("C++")) words.set(i, "Java");
```

А теперь сравним этот цикл с приведенным ниже вызовом. Обнаружив вызов метода в прикладном коде, можно сразу же выяснить назначение такого кода. В конце этого раздела дается краткое описание простых алгоритмов, реализованных в классе `Collections`.

```
Collections.replaceAll(words, "C++", "Java");
```

Более сложными являются методы с реализацией по умолчанию `Collection.removeIf()` и `List.replaceAll()`. Для проверки или преобразования элементов коллекции в качестве параметра этим методам предоставляется лямбда-выражение. Например, в следующем фрагменте кода из коллекции удаляются все краткие слова, а оставшиеся слова приводятся к нижнему регистру букв:

```
words.removeIf(w -> w.length() <= 3);
words.replaceAll(String::toLowerCase);
```

java.util.Collections 1.2

- `static <T extends Comparable<? super T>> T min(Collection<T> elements)`
- `static <T extends Comparable<? super T>> T max(Collection<T> elements)`
- `static <T> min(Collection<T> elements, Comparator<? super T> c)`
- `static <T> max(Collection<T> elements, Comparator<? super T> c)`

Возвращают наименьший или наибольший элемент из коллекции (границы параметров упрощены для ясности).

- `static <T> void copy(List<? super T> to, List<T> from)`

Копирует все элементы из исходного списка на те же позиции целевого списка. Целевой список должен быть не короче исходного.

- `static <T> void fill(List<? super T> l, T value)`

Устанавливает на всех позициях списка одно и то же значение.

- `static <T> boolean addAll(Collection<? super T> c, T... values) 5`

Вводит все значения в заданную коллекцию и возвращает логическое значение `true`, если в результате этого коллекция изменилась.

java.util.Collections 1.2 (окончание)

- **static <T> boolean replaceAll(List<T> l, T oldValue, T newValue) 1.4**
Заменяет на *newValue* все элементы, равные *oldValue*.
- **static int indexOfSubList(List<?> l, List<?> s) 1.4**
- **static int lastIndexOfSubList(List<?> l, List<?> s) 1.4**
Возвращают индекс первого и последнего подсписка *l*, равных списку *s*, или значение -1, если ни один из подсписков *l* не равен списку *s*. Так, если список *l* содержит элементы [*s*, *t*, *a*, *r*], а список *s* — элементы [*t*, *a*, *r*], то оба метода возвращают индекс, равный 1.
- **static void swap(List<?> l, int i, int j) 1.4**
Меняет местами элементы списка на указанных позициях.
- **static void reverse(List<?> l)**
Меняет порядок следования элементов в списке. Например, в результате обращения списка [*t*, *a*, *r*] порождается список [*r*, *a*, *t*]. Этот метод выполняется за время $O(n)$, где *n* — длина списка.
- **static void rotate(List<?> l, int d) 1.4**
Циклически сдвигает элементы в списке, перемещая элемент по индексу *i* на позицию $(i+d) \% l.size()$. Например, в результате циклического сдвига списка [*t*, *a*, *r*] на 2 позиции порождается список [*a*, *r*, *t*]. Этот метод выполняется за время $O(n)$, где *n* — длина списка.
- **static int frequency(Collection<?> c, Object o) 5**
Возвращает количество элементов в коллекции *c*, равных заданному объекту *o*.
- **boolean disjoint(Collection<?> c1, Collection<?> c2) 5**
Возвращает логическое значение **true**, если у коллекций отсутствуют общие элементы.

java.util.Collection<T> 1.2

- **default boolean removeAll(Predicate<? super E> filter) 8**
Удаляет из коллекции все совпавшие элементы.

java.util.List<E> 1.2

- **default void replaceAll(UnaryOperator<E> op) 8**
Выполняет указанную операцию над всеми элементами данного списка.

9.6.5. Групповые операции

Имеется ряд операций, предназначенных для группового копирования или удаления элементов из коллекции. Так, в результате вызова

```
coll1.removeAll(coll2);
```

из коллекции *coll1* удаляются все элементы, присутствующие в коллекции *coll2*. А в результате приведенного ниже вызова из коллекции *coll1* удаляются все элементы, отсутствующие в коллекции *coll2*.

```
coll1.retainAll(coll2);
```

Рассмотрим типичный пример применения групповых операций. Допустим, требуется найти *пересечение* двух множеств, т.е. элементы, общие для обоих множеств. Для хранения результата данной операции сначала создается новое множество:

```
var result = new HashSet<String>(firstSet);
```

В данном случае используется тот факт, что у каждой коллекции имеется свой конструктор, параметром которого является другая коллекция, содержащая неинициализированные значения. Затем вызывается метод `retainAll()`:

```
result.retainAll(b);
```

Он оставляет в текущей коллекции только те элементы, которые имеются в коллекции `b`. В итоге пересечение двух множеств получается без всякого программирования цикла их поочередного обхода.

Этот принцип можно распространить и на групповые операции над *представлениями*. Допустим, имеется отображение, связывающее идентификаторы работников с объектами, описывающими работников, а также множество идентификаторов работников, которые должны быть уволены, как показано ниже.

```
Map<String, Employee> staffMap = . . . ;  
Set<String> terminatedIDs = . . . ;
```

Достаточно сформировать множество ключей и удалить идентификаторы всех увольняемых работников, как показано в приведенной ниже строке кода. Это множество ключей является представлением отображения, и поэтому ключи и имена соответствующих работников автоматически удаляются из отображения.

```
staffMap.keySet().removeAll(terminatedIDs);
```

Используя представление поддиапазона, можно ограничить групповые операции подписками и подмножествами. Допустим, требуется ввести первые 10 элементов списка в другой контейнер. Для этого сначала формируется подсписок из первых 10 элементов:

```
relocated.addAll(staff.subList(0, 10));
```

Поддиапазон также может стать целью модифицирующей групповой операции, как показано в следующей строке кода:

```
staff.subList(0, 10).clear();
```

9.6.6. Взаимное преобразование коллекций и массивов

Значительная часть прикладного интерфейса API на платформе Java была разработана до появления каркаса коллекций, поэтому рано или поздно традиционные массивы придется преобразовать в более современные коллекции. Так, если имеется массив, который требуется преобразовать в коллекцию, для этой цели служит обертка метода `List.of()`, как показано в приведенном ниже примере кода.

```
String[] values = . . . ;  
var staff = new HashSet<>(List.of(values));
```

Получить массив из коллекции немного сложнее. Безусловно, для этого можно воспользоваться методом `toArray()` следующим образом:

```
Object[] values = staff.toArray();
```

Но в итоге будет получен массив *объектов*. Даже если известно, что коллекция содержит объекты определенного типа, их все равно нельзя привести к нужному типу:

```
String[] values = (String[]) staff.toArray(); // ОШИБКА!
```


Массив, возвращаемый методом `toArray()`, создан как массив типа `Object[]`, а этот тип изменить нельзя. Вместо этого следует воспользоваться приведенным ниже вариантом метода `toArray()`, передав ему массив нулевой длины и требуемого типа. Возвращаемый в итоге массив окажется *того же самого типа*.

```
String[] values = staff.toArray(new String[0]);
```

Если есть желание, можно сразу сконструировать массив нужного размера следующим образом:

```
staff.toArray(new String[staff.size()]);
```

В этом случае новый массив не создается.



НА ЗАМЕТКУ! У вас может возникнуть следующий вопрос: почему бы просто не передать объект типа `Class` (например, `String.class`) методу `toArray()`? Но ведь этот метод несет на себе “двойную нагрузку”, заполняя существующий массив (если он достаточно длинный) и создавая новый.

9.6.7. Написание собственных алгоритмов

Собираясь писать свой собственный алгоритм (по существу, любой метод, принимающий коллекцию в качестве параметра), старайтесь по возможности оперировать *интерфейсами*, а не конкретными их реализациями. Допустим, требуется обработать элементы коллекции. Для этого можно, конечно, реализовать метод, аналогичный следующему:

```
public void processItems(ArrayList<Item> items)
{
    for (Item item : items)
        сделать что-нибудь с элементом item
}
```

Но в этом случае ограничиваются возможности той части кода, где вызывается приведенный выше метод, поскольку ему должны передаваться пункты в списочном массиве типа `ArrayList`. Если это окажется другой контейнер, его придется сначала перепаковать. Намного лучше принимать в качестве параметра обобщенную коллекцию.

Следует задать себе такой вопрос: какой интерфейс коллекции является наиболее обобщенным, чтобы справиться с поставленной задачей, и насколько это вообще важно? Для ответа на этот вопрос реализуемый метод должен принимать в качестве параметра коллекцию типа `List`. Но если порядок не имеет значения, то можно принять коллекцию любого типа, как показано ниже. Теперь всякий может вызвать этот метод, передав ему коллекцию типа `ArrayList`, `LinkedList` или даже массив, заключенный в оболочку метода `List.of()`.

```
public void processItems(Collection<Item> items)
{
    for (Item item : items)
        сделать что-нибудь с элементом item
}
```



СОВЕТ. В данном случае можно поступить еще лучше, приняв объект типа `Iterable<Item>`. В интерфейсе `Iterable` объявляется единственный абстрактный метод `iterator()`, применяемый в усовершенствованном цикле `for` подспудно. Интерфейс `Collection` расширяет интерфейс `Iterable`.

А если метод возвращает несколько элементов, то вряд ли стоит ограничивать его реализацию ради последующих усовершенствований. Рассмотрим в качестве примера следующий метод:

```
public ArrayList<Item> lookupItems(. . .)
{
    var result = new ArrayList<Item>();
    . . .
    return result;
}
```

Этот метод обещает вернуть списочный массив типа `ArrayList`, несмотря на то, что вызывающий код, вероятнее всего, не интересуется конкретным типом возвращаемого списка. А если этот метод возвращает список типа `List`, то в его тело можно в любой момент ввести ветвь для возврата пустого или одиночного списка, вызвав метод `List.of()`.



НА ЗАМЕТКУ! Если использовать интерфейс коллекции в качестве параметра метода настолько удобно, то почему же в библиотеке Java не так часто соблюдается этот замечательный принцип? Например, в классе `JComboBox` имеются следующие конструкторы:

```
JComboBox(Object[] items)
JComboBox(Vector<?> items)
```

Причина проста: библиотека `Swing` была создана до каркаса коллекций.

9.7. Унаследованные коллекции

В первом выпуске Java появился целый ряд унаследованных контейнерных классов, применявшихся до появления каркаса коллекций. Они были внедрены в каркас коллекций (рис. 9.12) и вкратце рассматриваются в последующих разделах.

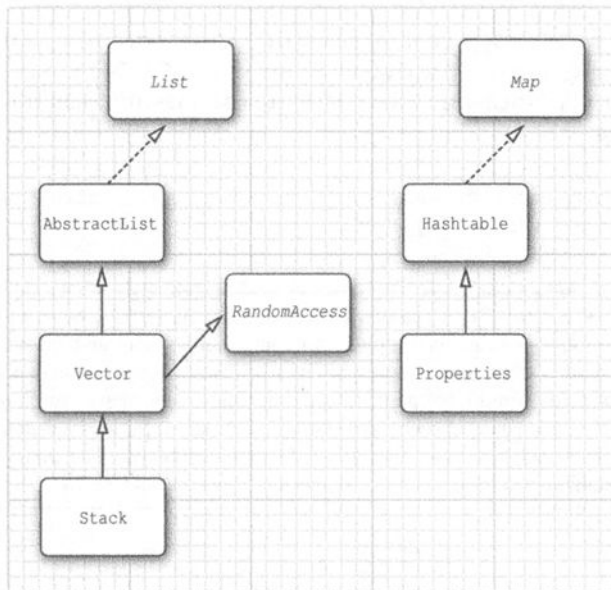


Рис. 9.12. Унаследованные контейнерные классы в каркасе коллекций

9.7.1. Класс Hashtable

Традиционный класс `Hashtable` служит той же цели, что и класс `HashMap`, и по существу имеет тот же интерфейс. Как и методы класса `Vector`, методы класса `Hashtable` синхронизированы. Если же не нужна синхронизация или совместимость с унаследованным кодом, то вместо него следует использовать класс `HashMap`. А если требуется параллельный доступ к коллекции, то рекомендуется воспользоваться классом `ConcurrentHashMap`, как поясняется в главе 12.

9.7.2. Перечисления

В унаследованных коллекциях применяется интерфейс `Enumeration` для обхода последовательностей элементов. У интерфейса `Enumeration` имеются два метода: `hasMoreElements()` и `nextElement()`, которые являются полными аналогами методов `hasNext()` и `next()` из интерфейса `Iterator`. Например, метод `elements()` из класса `Hashtable` порождает объект для перечисления значений из таблицы, как показано ниже.

Если этот интерфейс обнаруживается в унаследованных классах, то для собирания элементов в коллекцию типа `ArrayList` можно вызвать метод `Collections.list()`. Например, единственное назначение класса `LogManager` состоит в том, чтобы выявить имена регистраторов в виде перечисления типа `Enumeration`. Ниже показано, как получить все эти имена.

```
ArrayList<String> loggerNames =  
    Collections.list(LogManager.getLoggerNames());
```

А с другой стороны, начиная с версии Java 9, перечисление можно преобразовать в итератор следующим образом:

```
LogManager.getLoggerNames().asIterator()  
    .forEachRemaining(n -> { . . . });
```

Иногда может встретиться унаследованный метод, который ожидает перечисление в качестве своего параметра. Статический метод `Collections.enumeration()` порождает объект перечисления всех элементов коллекции, как показано в следующем примере кода:

```
List<InputStream> streams = . . .;  
var in = new SequenceInputStream(Collections.enumeration(streams));  
// в конструкторе класса SequenceInputStream  
// ожидается перечисление
```



НА ЗАМЕТКУ C++! В языке C++ итераторы очень часто используются в качестве параметров. Правда, при программировании на платформе Java лишь очень немногие пользуются этим приемом. Разумнее передавать коллекцию, чем итератор, поскольку объект коллекции удобнее. Код, принимающий параметры, может всегда получить итератор из коллекции, когда потребуется. К тому же он получает в свое распоряжение все методы коллекции. Но в унаследованном коде можно встретить перечисления, поскольку раньше они были единственным доступным механизмом обобщенных коллекций до появления каркаса коллекций в версии Java SE 1.2.

java.util.Enumeration<E> 1.0

- **boolean hasMoreElements()**
Возвращает логическое значение **true**, если в коллекции еще остались элементы для просмотра.
- **E nextElement()**
Возвращает следующий элемент для просмотра. Этот метод не следует вызывать, если метод **hasMoreElements()** возвратит логическое значение **false**.
- **default Iterator<E> asIterator()** 9
Выдает итератор для обхода перечисляемых элементов в коллекции.

java.util.Collections 1.2

- **static <T> Enumeration<T> enumeration(Collection<T> c)**
Возвращает перечисление элементов из заданной коллекции **c**.
- **public static <T> ArrayList<T> list(Enumeration<T> e)**
Возвращает списочный массив, содержащий элементы из заданного перечисления **e**.

9.7.3. Таблицы свойств

Таблица свойств — это структура отображения особенного типа, обладающая следующими особенностями.

- Ключи и значения являются символьными строками.
- Таблица может быть сохранена в файле и загружена из файла.
- Используется вторичная таблица для установок по умолчанию.

Класс, реализующий таблицу свойств на платформе Java, называется **Properties**. Таблицы свойств широко применяются для описания глобальных параметров настройки программ, как демонстрируется в следующем примере кода:

```
var settings = new Properties();
settings.setProperty("width", "600.0");
settings.setProperty("filename",
    "/home/cay/books/cj11/code/vlch11/raven.html");
```

Для сохранения таблицы свойств списком в файле следует вызвать метод **store()**. В данном случае достаточно сохранить таблицу свойств в файле **program.properties**. А в качестве второго аргумента можно указать комментарий, включаемый в данный файл.

```
var out = new FileOutputStream("program.properties");
settings.store(out, "Program Properties");
```

Обработка образцового множества свойств программы дает следующий результат:

```
# Свойства программы
# Sun Dec 31 12:54:19 PST 2017
top=227.0
left=1286.0
```

```
width=423.0
height=547.0
filename=/home/cay/books/cj11/code/vlch11/raven.html
```

Чтобы загрузить свойства из файла, достаточно воспользоваться следующим фрагментом кода:

```
var in = new FileInputStream("program.properties");
settings.load(in);
```

Метод `System.getProperties()` возвращает объект типа `Properties` для описания системной информации. Например, для начального каталога имеется ключ `"user.home"`. Его можно прочитать с помощью метода `getProperties()`, возвращающего данный ключ в виде символьной строки:

```
String userDir = System.getProperty("user.home");
```



ВНИМАНИЕ! По историческим причинам класс `Properties` реализует интерфейс `Map<Object, Object>`, что дает возможность вызывать методы `get()` и `put()` из интерфейса `Map`. Но метод `get()` возвращает объект типа `Object`, а метод `put()` позволяет ввести любой объект. Поэтому для обработки символьных строк, а не объектов лучше придерживаться методов `getProperty()` и `setProperty()`.

Чтобы получить текущую версию виртуальной машины Java, достаточно обратиться к свойству `"java.version"`. В итоге получается символьная строка вроде `"9.0.1"` (или `"1.8.0"` для версии Java 8).



СОВЕТ. Как видите, нумерация версий изменилась в версии Java 9. Это незначительное, на первый взгляд, изменение нарушило работу целого ряда инструментальных средств, опиравшихся на прежний формат. Поэтому, прежде чем анализировать символьную строку с версией Java, обратитесь к спецификации JEP 322, доступной по адресу <http://openjdk.java.net/jeps/322>, чтобы выяснить, каким образом будут в дальнейшем (или хотя бы до смены схемы нумерации) форматироваться символьные строки с версиями Java.

В классе `Properties` реализованы два механизма для предоставления устанавливаемых по умолчанию свойств. Прежде всего, всякий раз, когда требуется выявить значение символьной строки, можно указать значение, используемое по умолчанию, если соответствующий ключ отсутствует:

```
String filename = settings.getProperty("filename", "");
```

Если в таблице свойств имеется свойство `"filename"`, в переменной `filename` устанавливается данная символьная строка. В противном случае в переменной `filename` устанавливается пустая символьная строка.

Если указывать значение по умолчанию слишком обременительно, вызывая каждый раз метод `getProperty()`, все значения по умолчанию можно разместить во вспомогательной таблице свойств и предоставить ее в конструкторе основной таблицы свойств.

```
var defaultSettings = new Properties();
defaultSettings.setProperty("width", "600");
defaultSettings.setProperty("height", "400");
defaultSettings.setProperty("filename", "");
...
var settings = new Properties(defaultSettings);
```

Кроме того, можно задать стандартные значения для настроек по умолчанию, если предоставить еще одну таблицу свойств в качестве параметра конструктору объекта `defaultSettings`, хотя это, как правило, не делается.

В примерах исходного кода, прилагаемых к данной книге, имеется программа, демонстрирующая порядок применения свойств для хранения и загрузки состояния данной программы. В данной программе, в свою очередь, применяется программа `ImageViewer` из главы 2, запоминающая положение фрейма, размер и последний загруженный файл. Выполните данную программу, загрузите файл свойств, переместите окно и измените его размер. Можете также отредактировать содержимое файла свойств `.corejava/ImageViewer.properties`, находящегося в вашем начальном каталоге.



НА ЗАМЕТКУ! До версии Java 9 в файлах свойств применялась кодировка в 7-разрядном коде ASCII, теперь в них применяется кодировка UTF-8.

Свойства состоят из простых таблиц без иерархической структуры, поэтому принято внедрять фиктивную иерархию с такими именами ключей, как `window.main.color`, `window.main.title` и т.д. Но в классе `Properties` отсутствуют методы, помогающие организовать такую иерархию. Если же предполагается хранить сложную конфигурационную информацию, для этой цели лучше воспользоваться классом `Preferences`, как поясняется в главе 10.

`java.util.Properties 1.0`

- **`Properties()`**
Создает пустую таблицу свойств.
- **`Properties(Properties defaults)`**
Создает пустую таблицу свойств с рядом установок по умолчанию.
- **`String getProperty(String key)`**
Получает связь со свойством. Возвращает символьную строку, связанную с ключом, или аналогичную строку из таблицы установок по умолчанию, если ключ отсутствует в текущей таблице.
- **`String getProperty(String key, String defaultValue)`**
Получает свойство со значением по умолчанию, если ключ не найден. Возвращает символьную строку, связанную с ключом, или символьную строку по умолчанию, если ключ отсутствует в текущей таблице.
- **`void load(InputStream in)`**
Загружает таблицу свойств из потока ввода `InputStream`.
- **`Object setProperty(String key, String value)`**
Устанавливает свойство. Возвращает установленное ранее значение по заданному ключу.
- **`void store(OutputStream out, String commentString) 1.2`**
Выводит таблицу свойств в поток вывода `OutputStream`.

`java.lang.System 1.0`

- **`Properties getProperties()`**
Извлекает все системные свойства. У прикладной программы должно быть разрешение на извлечение всех свойств, а иначе генерируется исключение в связи с нарушением безопасности.

java.lang.System 1.0 (окончание)

- **String getProperty(String key)**

Извлекает системное свойство по заданному имени ключа. У прикладной программы должно быть разрешение на извлечение данного свойства, а иначе генерируется исключение в связи с нарушением безопасности. Данным методом всегда извлекаются следующие свойства:

```
java.version
java.vendor
java.vendor.url
java.home
java.class.path
java.library.path
java.class.version
os.name
os.version
os.arch
file.separator
path.separator
line.separator
java.io.tmpdir
user.name
user.home
user.dir
java.compiler
java.specification.version
java.specification.vendor
java.specification.name
java.vm.specification.version
java.vm.specification.vendor
java.vm.specification.name
java.vm.version
java.vm.vendor
java.vm.name
```

9.7.4. Стек

В версии 1.0 в стандартную библиотеку Java входит класс `Stack` с хорошо известными методами `push()` и `pop()`. Но класс `Stack` расширяет класс `Vector`, что неудовлетворительно с теоретической точки зрения, поскольку в нем допускаются операции, не характерные для стека. Например, вызывая методы `insert()` и `remove()`, можно вводить и удалять значения откуда угодно, а не только из вершины стека.

java.util.Stack<E> 1.0

- **E push(E item)**

Помещает заданный элемент `item` в стек и возвращает его.

- **E pop()**

Извлекает и возвращает элемент из вершины стека. Этот метод не следует вызывать, если стек пуст.

```
java.util.Stack<E> 1.0 (окончание)
```

- **E peek()**

Возвращает элемент, находящийся на вершине стека, не извлекая его. Этот метод не следует вызывать, если стек пуст.

9.7.5. Битовые множества

В классе `BitSet` на платформе Java хранятся последовательности битов. (Это не *множество* в математическом смысле. Битовый *вектор*, или битовый *массив*, — возможно, более подходящий для этого термин.) Битовое множество применяется в тех случаях, когда требуется эффективно хранить последовательность битов (например, признаков или флагов). А поскольку битовое множество упаковывает биты в байты, то пользоваться им намного эффективнее, чем списочным массивом типа `ArrayList` с объектами типа `Boolean`.

Класс `BitSet` предоставляет удобный интерфейс для чтения, установки или переустановки отдельных битов. Применяя этот интерфейс, можно избежать маскирования или других операций с битами, которые потребовались бы, если бы биты хранились в переменных типа `long`.

Например, в результате вызова `bucketOfBits.get(i)` для объекта `bucketOfBits` типа `BitSet` возвращается логическое значение `true`, если *i*-й бит установлен, а иначе — логическое значение `false`. Аналогично в результате вызова `bucketOfBits.set(i)` устанавливается *i*-й бит. И, наконец, в результате вызова `bucketOfBits.clear(i)` сбрасывается *i*-й бит.



НА ЗАМЕТКУ C++! Шаблон `bitset` в C++ обладает теми же функциональными возможностями, что и класс `BitSet` на платформе Java.

```
java.util.BitSet 1.0
```

- **BitSet(int initialCapacity)**

Конструирует битовое множество.

- **int length()**

Возвращает "логическую длину" битового множества: 1 + индекс самого старшего установленного бита.

- **boolean get(int bit)**

Получает бит.

- **void set(int bit)**

Устанавливает бит.

- **void clear(int bit)**

Сбрасывает бит.

- **void and(BitSet set)**

Выполняет логическую операцию И над данным и другим битовым множеством.


```
java.util.BitSet 1.0 (окончание)
```

- **void or(BitSet set)**
Выполняет логическую операцию ИЛИ над данным и другим битовым множеством.
- **void xor(BitSet set)**
Выполняет логическую операцию исключающее ИЛИ над данным и другим битовым множеством.
- **void andNot(BitSet set)**
Сбрасывает все биты данного битового множества, установленные в другом битовом множестве.

Продemonстрируем применение битовых множеств на примере реализации алгоритма “Решето Эратосфена”, служащего для нахождения простых чисел. (Простое число — это число вроде 2, 3 или 5, которое делится только на самое себя и на 1, а “Решето Эратосфена” было одним из первых методов, изобретенных для перечисления этих основополагающих математических единиц.) Это не самый лучший алгоритм для нахождения простых чисел, но по ряду причин он получил широкое распространение в качестве теста производительности компилятора. (Впрочем, это далеко не лучший тест, поскольку он тестирует в основном битовые операции.) Отдавая дань традиции, реализуем этот алгоритм в программе, подсчитывающей все простые числа от 2 до 2000000. (В этом диапазоне находится 148933 простых числа, так что вряд ли стоит выводить весь этот ряд чисел.)

Не слишком вдаваясь в подробности, поясним: ключ к достижению цели лежит в обходе битового множества, состоящего из 2 миллионов бит. Сначала все эти биты устанавливаются в 1. Затем сбрасываются те биты, которые кратны известным простым числам. Позиции битов, оставшихся после этого процесса, сами представляют простые числа. В листинге 9.8 приведен исходный код данной программы на Java, а в листинге 9.9 — ее исходный код на C++.

Листинг 9.8. Исходный код из файла `sieve/Sieve.java`

```
1 package sieve;
2
3 import java.util.*;
4
5 /**
6  * В этой программе выполняется тест по
7  * алгоритму "Решето Эратосфена" для нахождения
8  * всех простых чисел вплоть до 2000000
9  * @version 1.21 2004-08-03
10 * @author Cay Horstmann
11 */
12 public class Sieve
13 {
14     public static void main(String[] s)
15     {
16         int n = 2000000;
17         long start = System.currentTimeMillis();
18         BitSet b = new BitSet(n + 1);
19         int count = 0;
20         int i;
```

```
21     for (i = 2; i <= n; i++)
22         b.set(i);
23     i = 2;
24     while (i * i <= n)
25     {
26         if (b.get(i))
27         {
28             count++;
29             int k = 2 * i;
30             while (k <= n)
31             {
32                 b.clear(k);
33                 k += i;
34             }
35         }
36         i++;
37     }
38     while (i <= n)
39     {
40         if (b.get(i)) count++;
41         i++;
42     }
43     long end = System.currentTimeMillis();
44     System.out.println(count + " primes");
45     System.out.println((end - start) + " milliseconds");
46 }
47 }
```

Листинг 9.9. Исходный код из файла `sieve/sieve.cpp`

```
1  /**
2   * @version 1.21 2004-08-03
3   * @author Cay Horstmann
4   */
5
6  #include <bitset>
7  #include <iostream>
8  #include <ctime>
9  using namespace std;
10
11 int main()
12 {
13     const int N = 2000000;
14     clock_t cstart = clock();
15
16     bitset<N + 1> b;
17     int count = 0;
18     int i;
19     for (i = 2; i <= N; i++)
20         b.set(i);
21     i = 2;
22     while (i * i <= N)
23     {
24         if (b.test(i))
25         {
```

```
26     count++;
27     int k = 2 * i;
28     while (k <= N)
29     {
30         b.reset(k);
31         k += i;
32     }
33 }
34 i++;
35 }
36 while (i <= N)
37 {
38     if (b.test(i))
39         count++;
40     i++;
41 }
42
43 clock_t cend = clock();
44 double millis = 1000.0 * (cend - cstart)
45                 / CLOCKS_PER_SEC;
46
47 cout << count << " primes\n" << millis
48      << " milliseconds\n";
49
50 return 0;
51 }
```



НА ЗАМЕТКУ! Несмотря на то что “Решето Эратосфена” — не самый лучший тест, автор все же не преминул сравнить время работы обеих представленных выше реализаций данного алгоритма. Ниже приведены результаты прогона обеих реализаций теста на компьютере с четырехъядерным процессором i7-8550U на 4 ГГц и ОЗУ на 16 Гбайт, работающим под управлением ОС Ubuntu 17.10:

- C++ (g++ 7.2.0): 173 мс
- Java (Java 9.0.1): 41 мс
- Этот тест запускался для десяти изданий данной книги, и в шести последних язык Java легко одерживал верх над языком C++. Справедливости ради, следует сказать, что при повышении уровня оптимизации компилятора язык C++ обгонял Java на 34 мс. А язык Java мог сравниться с этим показателем только при достаточно долгой работе программы, чтобы вступил в действие динамический компилятор Hotspot.

На этом рассмотрение архитектуры коллекций Java завершается. Как вы сами могли убедиться, в библиотеке Java предоставляется широкое разнообразие классов коллекций для ваших потребностей в программировании. В следующей главе будут обсуждаться вопросы построения графических пользовательских интерфейсов.

Программирование графики

В этой главе...

- ▶ История развития инструментальных средств для разработки графических приложений на Java
- ▶ Отображение фреймов
- ▶ Отображение данных в компоненте
- ▶ Обработка событий
- ▶ Прикладной интерфейс Preferences API

Язык Java появился в то время, когда большинство пользователей компьютеров взаимодействовали с графическими настольными приложениями. В настоящее время намного более распространены приложения, ориентированные на браузеры и мобильные устройства. Тем не менее иногда полезно предоставить настольное приложение. В этой и последующей главах обсуждаются основы программирования графического пользовательского интерфейса (GUI) инструментальными средствами библиотеки Swing. Если же вы намерены пользоваться языком Java для программирования только серверных приложений и вас вообще не интересует написание графических программ, можете благополучно пропустить обе эти главы.

10.1. История развития инструментальных средств для разработки GUI на Java

В версию Java 1.0 входила библиотека классов Abstract Window Toolkit (AWT), предоставлявшая основные инструментальные средства программирования GUI. Создание элементов GUI на конкретной платформе (Windows, Linux, Macintosh и т.д.)

библиотека AWT поручала платформенно-ориентированным инструментальным средствам. Так, если с помощью библиотеки AWT на экран требовалось вывести окно с текстом, то оно фактически отображалось базовыми средствами конкретной платформы. Теоретически созданные таким образом программы должны были работать на любых платформах и иметь внешний вид, характерный для целевой платформы.

Методика, основанная на использовании базовых средств конкретных платформ, отлично подходила для простых приложений. Но вскоре стало ясно, что с ее помощью крайне трудно создавать высококачественные переносимые графические библиотеки, зависящие от платформенно-ориентированных интерфейсных элементов платформы. Элементы пользовательского интерфейса, например, меню, панели прокрутки и текстовые поля, на разных платформах могут вести себя по-разному. Следовательно, на основе этого подхода трудно создавать согласованные программы с предсказуемым поведением. Кроме того, некоторые графические среды (например, X11/Motif) не имеют такого богатого набора компонентов пользовательского интерфейса, как операционные системы Windows и Macintosh. Это, в свою очередь, делало библиотеки еще более зависимыми. В результате графические приложения, созданные с помощью библиотеки AWT, выглядели по сравнению с прикладными программами для Windows или Macintosh не так привлекательно и не имели таких функциональных возможностей. Хуже того, в библиотеке AWT на различных платформах обнаруживались *разные* ошибки. Разработчикам приходилось тестировать каждое приложение на каждой платформе, что на практике означало: “Написано однажды, отлаживается везде”.

В 1996 году компания Netscape создала библиотеку программ для разработки GUI, назвав ее IFC (Internet Foundation Classes). Эта библиотека основана на совершенно других принципах. Элементы пользовательского интерфейса вроде меню, экранных кнопок и тому подобных воспроизводились в пустом окне. А от оконной системы конкретной платформы требовалось лишь отображать окно и рисовать в нем графику. Таким образом, элементы GUI, созданные с помощью библиотеки IFC, выглядели и вели себя одинаково, но не зависели от той платформы, на которой запускалась программа. Компании Sun Microsystems и Netscape объединили свои усилия и усовершенствовали данную методику, создав библиотеку под кодовым названием Swing. С тех пор слово Swing стало официальным названием набора инструментальных средств для создания машинно-независимого пользовательского интерфейса. Средства Swing стали доступны как расширение Java 1.1 и вошли в состав стандартной версии Java 1.2.



НА ЗАМЕТКУ! Библиотека Swing не является полной заменой библиотеки AWT. Она построена на основе архитектуры AWT. Библиотека Swing просто дает больше возможностей для создания пользовательского интерфейса. При написании программы средствами Swing, по существу, используются основные ресурсы AWT. Здесь и далее под Swing подразумеваются платформенно-независимые классы для создания “рисованного” пользовательского интерфейса, а под AWT — базовые средства конкретной платформы для работы с окнами, включая обработку событий.

Библиотеке Swing приходится воспроизводить буквально каждый элемент изображения (так называемый пиксель) пользовательского интерфейса, поэтому впервые пользователи жаловались на ее медленную работу. (В этом можно все еще убедиться, выполняя Swing-приложения на таком вычислительном оборудовании, как Raspberry Pi.) Со временем быстродействие настольных компьютеров повысилось, и пользователи стали жаловаться уже на непривлекательный внешний вид компонентов Swing, которые, действительно, уступали в этом отношении платформенно-ориентированным виджетам, снабженным анимацией и спецэффектами. Хуже того, инструментальные средства Adobe Flash стали все чаще применяться для создания пользовательских

интерфейсов со все более ориентированными на Flash эффектами и вообще без платформенно-ориентированных элементов управления.

В 2007 году компания Sun Microsystems внедрила совершенно новую библиотеку для разработки GUI под названием JavaFX в качестве конкурента Adobe Flash. Эта библиотека работала на виртуальной машине Java, но имела свой язык программирования JavaFX Script, оптимизированный для создания анимации и спецэффектов. Но программисты жаловались на необходимость изучать новый язык и воздерживались в основной своей массе от применения данной библиотеки на практике. В 2011 году компания Oracle выпустила новую версию JavaFX 2.0 данной библиотеки, где имелся прикладной интерфейс Java API и больше не требовался отдельный язык программирования. Начиная с обновления 6 версии Java 7, библиотека JavaFX входит в состав комплекта JDK и платформы JRE. Но когда писалась эта книга, компания Oracle объявила, что в версии Java 11 библиотека JavaFX больше не будет входить в один комплект с Java.

Этот том настоящего издания посвящен основам языка Java и его прикладным интерфейсам API, поэтому основное внимание здесь уделяется программированию GUI средствами Swing. Тем не менее в главе 13 представлено введение в библиотеку JavaFX для тех, кого она может заинтересовать.

10.2. Отображение фреймов

Окно верхнего уровня (т.е. такое, которое не содержится в другом окне) в Java называется *фреймом*. В библиотеке AWT для такого окна предусмотрен класс `Frame`, а в библиотеке Swing его аналогом является класс `JFrame`. Класс `JFrame` расширяет класс `Frame` и представляет собой один из немногих компонентов библиотеки Swing, которые не воспроизводятся на холсте. Экранные кнопки, строка заголовков, пиктограммы и другие элементы оформления GUI реализуются с помощью пользовательской оконной системы, а не библиотеки Swing.



ВНИМАНИЕ! Большинство имен компонентов из библиотеки Swing начинаются с буквы **Ж**. В качестве примера можно привести классы **JButton** и **JFrame** — аналоги соответствующих компонентов библиотеки AWT (например, **Button** и **Frame**). Если пропустить букву **Ж** в имени применяемого компонента, программа скомпилируется и будет работать, но сочетание компонентов Swing и AWT в одном окне приведет к несогласованности внешнего вида и поведения элементов GUI.

10.2.1. Создание фрейма

В этом разделе рассмотрены самые распространенные приемы работы с компонентом `JFrame` библиотеки Swing. В листинге 10.1 приведен исходный код простой программы, отображающей на экране пустой фрейм (рис. 10.1).

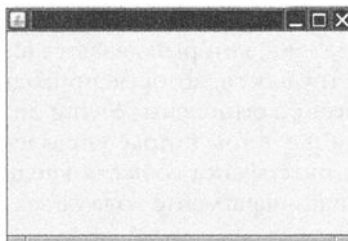


Рис. 10.1. Простейший отображаемый фрейм

Листинг 10.1. Исходный код из файла `simpleframe/SimpleFrameTest.java`

```
1 package simpleFrame;
2
3 import java.awt.*;
4 import javax.swing.*;
5
6 /**
7  * @version 1.34 2018-04-10
8  * @author Cay Horstmann
9  */
10 public class SimpleFrameTest
11 {
12     public static void main(String[] args)
13     {
14         EventQueue.invokeLater(() ->
15         {
16             var frame = new SimpleFrame();
17             frame.setDefaultCloseOperation(
18                 JFrame.EXIT_ON_CLOSE);
19             frame.setVisible(true);
20         });
21     }
22 }
23
24 class SimpleFrame extends JFrame
25 {
26     private static final int DEFAULT_WIDTH = 300;
27     private static final int DEFAULT_HEIGHT = 200;
28
29     public SimpleFrame()
30     {
31         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
32     }
33 }
```

Проанализируем эту программу построчно. Классы библиотеки Swing находятся в пакете `javax.swing`. Имя пакета `javax` означает, что он является расширением Java и не входит в число основных пакетов. По ряду причин исторического характера библиотека Swing считается расширением с версии Java 1.1, но она присутствует в каждой реализации Java, начиная с версии 1.2.

По умолчанию фрейм имеет совершенно бесполезные размеры: 0×0 пикселей. В данном примере определяется подкласс `SimpleFrame`, в конструкторе которого устанавливаются размеры фрейма 300×200 пикселей. Это единственное отличие подкласса `SimpleFrame` от класса `JFrame`. В методе `main()` из класса `SimpleFrameTest` создается объект типа `SimpleFrame`, который делается видимым.

Имеются две технические трудности, которые приходится преодолевать в каждой Swing-программе. Прежде всего, компоненты Swing должны быть настроены в *потоке диспетчеризации событий*, т.е. в том потоке управления, который передает компонентам пользовательского интерфейса события вроде щелчков кнопками мыши и нажатий клавиш. В следующем фрагменте кода операторы выполняются в потоке диспетчеризации событий:

```
EventQueue.invokeLater(() ->
{
    операторы
});
```



НА ЗАМЕТКУ! Вам встретится еще немало Swing-программ, где пользовательский интерфейс не инициализируется в потоке диспетчеризации событий. Раньше инициализацию допускалось выполнять в главном потоке исполнения. К сожалению, в связи с усложнением компонентов Swing разработчики JDK не смогли больше гарантировать безопасность такого подхода. Вероятность ошибки чрезвычайно низка, но вам вряд ли захочется стать одним из тех немногих, кого угораздит столкнуться с такой прерывистой ошибкой. Лучше сделать все правильно, даже если код покажется поначалу и не совсем понятным.

Далее в рассматриваемом здесь примере определяется, что именно должно произойти, если пользователь закроет фрейм приложения. В данном случае программа должна завершить свою работу. Для этого служит следующая строка кода:

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Если бы в программе использовалось несколько фреймов, завершать работу только потому, что пользователь закрыл один из них, было бы не обязательно. По умолчанию закрытый фрейм исчезает с экрана, а программа продолжает свою работу. (Было бы, конечно, неплохо, если бы программа завершалась после исчезновения *последнего* фрейма с экрана, но, к сожалению, Swing действует иначе.)

Простое создание фрейма не приводит к его автоматическому появлению на экране. В начале своего существования все фреймы невидимы. Это дает возможность добавлять во фрейм компоненты еще до того, как он впервые появится на экране. Для отображения фрейма на экране в методе `main()` вызывается метод `setVisible()`.

После диспетчеризации операторов инициализации происходит выход из метода `main()`. Обратите внимание на то, что это не приводит к прекращению работы программы. Завершается лишь ее основной поток. Поток диспетчеризации событий, обеспечивающий нормальную работу программы, продолжает действовать до тех пор, пока она не завершится закрытием фрейма или вызовом метода `System.exit()`.

Окно выполняющейся программы показано на рис. 10.1. Как видите, такие элементы, как строка заголовка и пиктограммы для изменения размеров окна, отображаются операционной системой, а не компонентами библиотеки Swing. Все, что находится во фрейме, отображается средствами Swing. В данной программе фрейм просто заполняется фоном, цвет которого задается по умолчанию.

10.2.2. Свойства фрейма

В классе `JFrame` имеется лишь несколько методов, позволяющих изменить внешний вид фрейма. Разумеется, благодаря наследованию в классе `JFrame` можно использовать методы из его суперклассов, задающие размеры и расположение фрейма. К наиболее важным из них относятся следующие.

- Методы `setLocation()` и `setBounds()`, устанавливающие положение фрейма.
- Метод `dispose()`, закрывающий окно и освобождающий все системные ресурсы, использованные при его создании.
- Метод `setIconImage()`, сообщающий оконной системе, какая пиктограмма должна отображаться в строке заголовка, окне переключателя задач и т.п.

- Метод `setTitle()`, позволяющий изменить текст в строке заголовка.
- Метод `setResizable()`, получающий в качестве параметра логическое значение и определяющий, имеет ли пользователь право изменять размеры фрейма.

Иерархия наследования для класса `JFrame` показана на рис. 10.2.

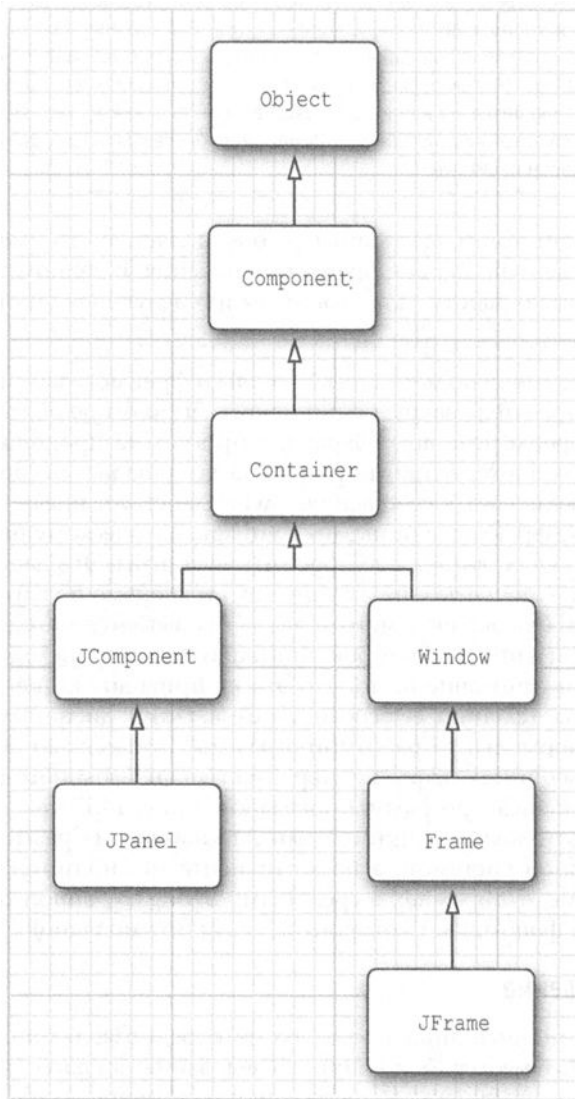


Рис. 10.2. Иерархия наследования классов фреймов и компонентов AWT и Swing

Как указано в документации на прикладной интерфейс API, методы для изменения размеров и формы фреймов следует искать в классе `Component`, который является предшественником всех объектов GUI, а также в классе `Window`, который является суперклассом для класса `Frame`. Например, метод `show()`, который служит

для отображения фрейма на экране, находится в классе `Window`, а в классе `Component` имеется метод `setLocation()`, позволяющий изменить расположение компонента. В приведенной ниже строке кода левый верхний угол фрейма размещается в точке, находящейся на расстоянии `x` пикселей вправо и `y` пикселей вниз от точки начала отсчета (0,0) в левом верхнем углу экрана.

```
setLocation(x, y)
```

Аналогично метод `setBounds()` из класса `Component` позволяет одновременно изменить размер и расположение компонента (в частности, объекта типа `JFrame`) с помощью следующего вызова:

```
setBounds(x, y, width, height)
```

Многие методы из классов компонентов объединены в пары для получения и установки соответствующих *свойств*. Примером тому служат следующие методы из класса `Frame`:

```
public String getTitle()
public void setTitle(String title)
```

У каждого свойства имеется свое имя и тип. Имя свойства получается путем изменения первой буквы на строчную после слова **get** или **set** в имени соответствующего метода доступа. Например, в классе `Frame` имеется свойство `title` типа `String`. По существу, `title` является свойством фрейма. При его установке предполагается, что заголовок окна на пользовательском экране изменится. А при получении данного свойства предполагается, что будет возвращено установленное в нем значение.

Из правила получения и установки свойств имеется единственное исключение: для свойств типа `boolean` имя метода получения начинается со слова `is`. Так, в приведенных ниже строках кода определяется свойство `resizable`.

```
public boolean isResizable()
public void setResizable(boolean resizable)
```

Чтобы определить соответствующие размеры фрейма, необходимо выяснить сначала размеры экрана. Для этого вызывается статический метод `getDefaultToolkit()` из класса `Toolkit`, который возвращает объект типа `Toolkit`. (Класс `Toolkit` содержит много методов, предназначенных для взаимодействия с оконной системой конкретной платформы.) Затем вызывается метод `getScreenSize()`, который возвращает размеры экрана в виде объекта типа `Dimension`. Этот объект содержит ширину и высоту в открытых (!) переменных `width` и `height` соответственно. Затем можно указать подходящую долю размеров экрана в процентах для задания размеров фрейма, как демонстрируется в следующем примере кода:

```
Toolkit kit = Toolkit.getDefaultToolkit();
Dimension screenSize = kit.getScreenSize();
int screenWidth = screenSize.width;
int screenHeight = screenSize.height;
setSize(screenWidth / 2, screenHeight / 2);
```

Кроме того, для фрейма можно предоставить пиктограмму, как показано ниже.

```
Image img = new ImageIcon("icon.gif").getImage();
setIconImage(img);
```

java.awt.Component 1.0

- **boolean isVisible()**

- **void setVisible(boolean b)**

Получают или устанавливают свойство видимости **visible**. Компоненты являются видимыми изначально, кроме компонентов верхнего уровня типа **JFrame**.

- **void setSize(int width, int height) 1.1**

Устанавливает текущую ширину и высоту компонента.

- **void setLocation(int x, int y) 1.1**

Перемещает компонент в новую точку. Если компонент не относится к верхнему уровню, то его координаты **x** и **y** отсчитываются относительно контейнера, а иначе используется экранная система координат (например, для объектов типа **JFrame**).

- **void setBounds(int x, int y, int width, int height) 1.1**

Перемещает текущий компонент и изменяет его размеры. Расположение левого верхнего угла задают параметры **x** и **y**, а новый размер — параметры **width** и **height**.

- **Dimension getSize() 1.1**

- **void setSize(Dimension d) 1.1**

Получают или устанавливают свойство **size**, задающее размеры текущего компонента.

java.awt.Window 1.0

- **void setLocationByPlatform(boolean b) 5**

Получают или устанавливают свойство **locationByPlatform**. Если это свойство установлено до того, как отобразилось данное окно, то подходящее расположение выбирает платформа.

java.awt.Frame 1.0

- **boolean isResizable()**

- **void setResizable(boolean b)**

Получают или устанавливают свойство **resizable**. Если это свойство установлено, то пользователь может изменять размеры фрейма.

- **String getTitle()**

- **void setTitle(String s)**

Получают или устанавливают свойство **title**, определяющее текст в строке заголовка фрейма.

- **Image getIconImage()**

- **void setIconImage(Image image)**

Получают или устанавливают свойство **iconImage**, определяющее пиктограмму фрейма. Оконная система может отображать пиктограмму как часть оформления фрейма или в каком-нибудь другом месте.

java.awt.Toolkit 1.0

- **static Toolkit getDefaultToolkit()**
Возвращает объект типа `Toolkit`, т.е. выбираемый по умолчанию набор инструментов.
- **Dimension getScreenSize()**
Получает размеры пользовательского экрана.

javax.swing.ImageIcon 1.2

- **ImageIcon(String имя_файла)**
Конструирует пиктограмму, изображение которой хранится в файле.
- **Image getImage()**
Получает изображение данной пиктограммы.

10.3. Отображение данных в компоненте

В этом разделе будет показано, как выводить данные во фрейме (рис. 10.3). Строку сообщения можно вывести непосредственно во фрейм, но на практике никто так не поступает. В языке Java фреймы предназначены именно для того, чтобы служить контейнерами для компонентов (например, меню или других элементов пользовательского интерфейса). Как правило, рисунки выводятся в другом компоненте, который добавляется во фрейм.



Рис. 10.3. Фрейм, в который выводятся данные

Оказывается, что структура компонента `JFrame` довольно сложная (рис. 10.4). Как видите, компонент `JFrame` состоит из четырех областей, каждая из которых представляет собой отдельную панель. Корневая, многослойная и прозрачная панели не представляют особого интереса и нужны лишь для оформления меню и панели содержимого в определенном стиле. Наиболее интересной для применения библиотеки Swing является *панель содержимого*. Любые компоненты, вводимые во фрейм, автоматически располагаются на панели содержимого:

```
Component c = . . . ;  
frame.add(c); // вводится на панели содержимого
```

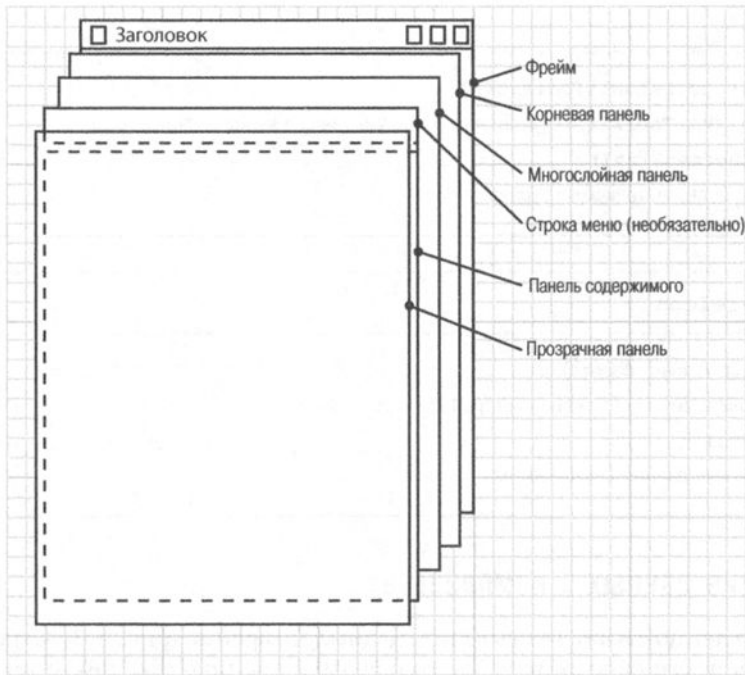


Рис. 10.4. Внутреннее строение компонента **JFrame**

В данном случае требуется добавить во фрейм единственный компонент, где будет выводиться сообщение. Чтобы отобразить компонент, следует сначала определить класс, расширяющий класс `JComponent`, а затем переопределить метод `paintComponent()` этого класса. Метод `paintComponent()` получает в качестве параметра объект типа `Graphics`, который содержит набор установок для отображения рисунков и текста. В нем, например, задается шрифт и цвет текста. Все операции рисования графики в Java выполняются с помощью объектов класса `Graphics`. В этом классе предусмотрены методы для рисования узоров, воспроизведения графических изображений и текста.

В приведенном ниже фрагменте кода показано в общих чертах, каким образом создается компонент для рисования графики.

```
class MyComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        код для рисования
    }
}
```

Всякий раз, когда возникает потребность перерисовать окно независимо от конкретной причины, обработчик событий уведомляет об этом соответствующий компонент. В итоге метод `paintComponent()` выполняется для всех компонентов. Метод `paintComponent()` вообще не следует вызывать вручную. Когда требуется перерисовать окно приложения, он вызывается автоматически, и вмешиваться в этот процесс не рекомендуется.

В каких случаях требуется автоматическая перерисовка? Она требуется, например, при увеличении или уменьшении размеров окна. Если пользователь открыл новое окно, перекрыв им уже существующее окно приложения, а затем закрыл его, оставшееся на экране окно оказывается поврежденным и должно быть перерисовано. (Графическая система не сохраняет в памяти пиксели нижележащего окна.) И, наконец, когда окно выводится на экран впервые, следует выполнить код, указывающий, как и где должны отображаться его исходные элементы.



СОВЕТ. Если требуется принудительно перерисовать экран, вместо метода `paintComponent()` следует вызвать метод `repaint()`. Этот метод, в свою очередь, обратится к методу `paintComponent()` каждого компонента и передаст ему настроенный должным образом объект типа `Graphics`.

Как следует из приведенного выше фрагмента кода, у метода `paintComponent()` имеется один параметр типа `Graphics`. При выводе на экран размеры, сохраняемые в объекте типа `Graphics`, указываются в пикселях. Координаты (0,0) соответствуют левому верхнему углу компонента, на поверхности которого выполняется рисование.

В классе `Graphics` имеются разные методы рисования, а вывод текста на экран считается особой разновидностью рисования. В частности, метод `paintComponent()` выглядит приблизительно так, как показано ниже.

```
class NotHelloWorldComponent extends JComponent
{
    public static final int MESSAGE_X = 75;
    public static final int MESSAGE_Y = 100;

    public void paintComponent(Graphics g)
    {
        g.drawString("Not a Hello, World program",
                     MESSAGE_X, MESSAGE_Y);
    }
    . . .
}
```

И, наконец, компонент должен сообщить своим пользователям, насколько большим он должен быть. Для этого переопределяется метод `getPreferredSize()`, возвращающий объект класса `Dimension` с предпочтительными размерами по ширине и по высоте:

```
class NotHelloWorldComponent extends JComponent
{
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;
    . . .
    public Dimension getPreferredSize()
    { return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT); }
}
```

Если при заполнении фрейма одним или несколькими компонентами требуется лишь воспользоваться их предпочтительными размерами, то вместо метода `setSize()` вызывается метод `pack()`:

```
class NotHelloWorldFrame extends JFrame
{
    public NotHelloWorldFrame()
    {
```

```
        add(new NotHelloWorldComponent());  
        pack();  
    }  
}
```

Весь исходный код рассмотренной здесь программы приведен в листинге 10.2.

Листинг 10.2. Исходный код из файла `notHelloWorld/NotHelloWorld.java`

```
1  package notHelloWorld;  
2  
3  import javax.swing.*;  
4  import java.awt.*;  
5  
6  /**  
7   * @version 1.34 2018-04-10  
8   * @author Cay Horstmann  
9   */  
10 public class NotHelloWorld  
11 {  
12     public static void main(String[] args)  
13     {  
14         EventQueue.invokeLater(() ->  
15             {  
16                 var frame = new NotHelloWorldFrame();  
17                 frame.setTitle("NotHelloWorld");  
18                 frame.setDefaultCloseOperation(  
19                     JFrame.EXIT_ON_CLOSE);  
20                 frame.setVisible(true);  
21             });  
22     }  
23 }  
24  
25 /**  
26  * Фрейм, содержащий панель сообщений  
27  */  
28 class NotHelloWorldFrame extends JFrame  
29 {  
30     public NotHelloWorldFrame()  
31     {  
32         add(new NotHelloWorldComponent());  
33         pack();  
34     }  
35 }  
36  
37 /**  
38  * Компонент, выводящий сообщение  
39  */  
40 class NotHelloWorldComponent extends JComponent  
41 {  
42     public static final int MESSAGE_X = 75;  
43     public static final int MESSAGE_Y = 100;  
44  
45     private static final int DEFAULT_WIDTH = 300;  
46     private static final int DEFAULT_HEIGHT = 200;
```

```
47
48 public void paintComponent(Graphics g)
49 {
50     g.drawString("Not a Hello, World program",
51                 MESSAGE_X, MESSAGE_Y);
52 }
53
54 public Dimension getPreferredSize()
55 {
56     return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT);
57 }
58 }
```

javax.swing.JFrame 1.2

- **Component add(Component c)**
Возвращает указанный компонент для ввода на панели содержимого данного фрейма.

java.awt.Component 1.0

- **void repaint()**
Вызывает перерисовку компонента. Перерисовка выполняется сразу же после того, как возникнут условия, позволяющие это сделать.
- **Dimension getPreferredSize()**
Этот метод переопределяется для возврата предпочтительных размеров данного компонента.

javax.swing.JComponent 1.2

- **void paintComponent(Graphics g)**
Этот метод переопределяется для описания способа рисования заданного компонента.

java.awt.Window 1.0

- **void pack()**
Изменяет размеры данного окна, принимая во внимание предпочтительные размеры его компонентов.

10.3.1. Двухмерные формы

В версии Java 1.0 в классе `Graphics` появились методы рисования линий, прямоугольников, эллипсов и прочих двухмерных форм. Но эти операции рисования предоставляют слишком ограниченный набор функциональных возможностей. Вместо этого можно воспользоваться классами двухмерных форм из библиотеки *Java 2D*.

Чтобы воспользоваться данной библиотекой, необходимо получить объект класса `Graphics2D`, который является подклассом, производным от класса `Graphics`. Начиная еще с версии Java 1.2, такие методы, как `paintComponent()`, автоматически получают объект класса `Graphics2D`. Для этого достаточно выполнить приведение типов, как показано ниже.

```
public void paintComponent(Graphics g)
{
    Graphics2D g2 = (Graphics2D) g;
    . . .
}
```

При создании геометрических форм в библиотеке Java 2D применяется объектно-ориентированный подход. В частности, перечисленные ниже классы, предназначенные для рисования линий, прямоугольников и эллипсов, реализуют интерфейс `Shape`. В библиотеке Java 2D поддерживаются и более сложные формы, в том числе дуги, квадратичные и кубические кривые и общие пути, которые не рассматриваются в этой главе.

```
Line2D
Rectangle2D
Ellipse2D
```

Чтобы нарисовать двухмерную форму, необходимо создать сначала объект соответствующего класса, реализующего интерфейс `Shape`, а затем вызвать метод `draw()` из класса `Graphics2D`, как показано в следующем примере кода:

```
Rectangle2D rect = . . . ;
g2.draw(rect);
```

Внутренние вычисления в библиотеке Java 2D выполняются в формате чисел с плавающей точкой и одинарной точностью. Этого вполне достаточно, поскольку главной целью вычислений геометрических форм является их вывод в пикселях на экран или на принтер. Все округления в ходе вычислений не выходят за пределы одного пикселя, что совершенно не влияет на внешний вид геометрической формы.

И все же манипулировать числовыми значениями типа `float` иногда бывает неудобно, поскольку в вопросах приведения типов язык Java непреклонен и требует преобразовывать числовые значения типа `double` в числовые значения типа `float` явным образом. Рассмотрим для примера следующее выражение:

```
float f = 1.2; // ОШИБКА! Возможна потеря точности
```

Это выражение не будет скомпилировано, поскольку константа `1.2` имеет тип `double`, а компилятор зафиксирует потерю точности. В таком случае при формировании константы с плавающей точкой придется явно указать суффикс **F** следующим образом:

```
float f = 1.2F; // Верно!
```

А теперь рассмотрим следующий оператор:

```
float f = r.getWidth(); // ОШИБКА!
```

Этот оператор не скомпилируется по тем же причинам. Метод `getWidth()` возвращает число, имеющее тип `double`. На этот раз нужно выполнить приведение типов:

```
float f = (float)r.getWidth(); // Верно!
```

Указание суффиксов и приведение типов доставляет немало хлопот, поэтому разработчики библиотеки Java 2D решили предусмотреть две версии каждого класса для рисования двухмерных форм: в первой версии все координаты выражаются

числовыми значениями типа `float` (для дисциплинированных программистов), а во второй — числовыми значениями типа `double` (для ленивых). (Относя себя ко второй группе, мы указываем координаты числовыми значениями типа `double` во всех примерах рисования двумерных форм, приведенных в данной книге.)

Разработчики данной библиотеки выбрали необычный и запутанный способ создания пакетов, соответствующих этим двум версиям. Рассмотрим в качестве примера класс `Rectangle2D`. Это абстрактный класс, имеющий два следующих конкретных подкласса, каждый из которых является внутренним и статическим:

```
Rectangle2D.Float  
Rectangle2D.Double
```

На рис. 10.5 приведена блок-схема наследования этих классов.

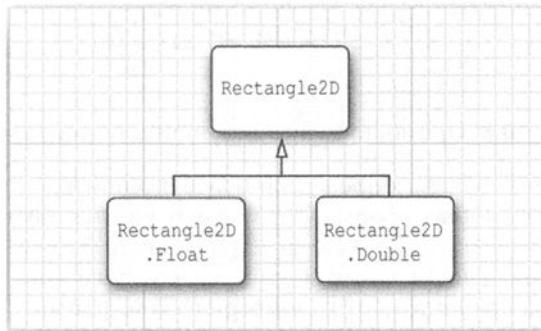


Рис. 10.5. Классы для рисования прямоугольников

Лучше совсем пренебречь тем, что эти два конкретных класса являются внутренними и статическими. Ведь это всего лишь уловка, чтобы избежать употребления таких имен, как `FloatRectangle2D` и `DoubleRectangle2D`. При построении объекта типа `Rectangle2D.Float` используются координаты, представленные числовыми значениями типа `float`, а в объектах типа `Rectangle2D.Double` они выражаются числовыми значениями типа `double`, как показано ниже, где параметры конструктора задают координаты верхнего левого угла, ширину и высоту прямоугольника.

```
var floatRect = new Rectangle2D.Float(10.0F,  
    25.0F, 22.5F, 20.0F);  
var doubleRect = new Rectangle2D.Double(10.0, 25.0,  
    22.5, 20.0);
```

Параметры и значения, возвращаемые методами из класса `Rectangle2D`, относятся к типу `double`. Например, метод `getWidth()` возвращает значение типа `double`, даже если ширина хранится в виде числового значения типа `float` в объекте типа `Rectangle2D.Float`.



СОВЕТ. Чтобы не оперировать числовыми значениями типа `float`, пользуйтесь классами, в которых координаты выражаются числовыми значениями типа `double`. Но если требуется создать тысячи объектов двумерных форм, то следует подумать об экономии памяти. И в этом помогут классы, где координаты задаются числовыми значениями типа `float`.

Все, что было сказано выше об иерархии классов `Rectangle2D`, относится к любым другим классам, предназначенным для рисования двумерных форм. Кроме

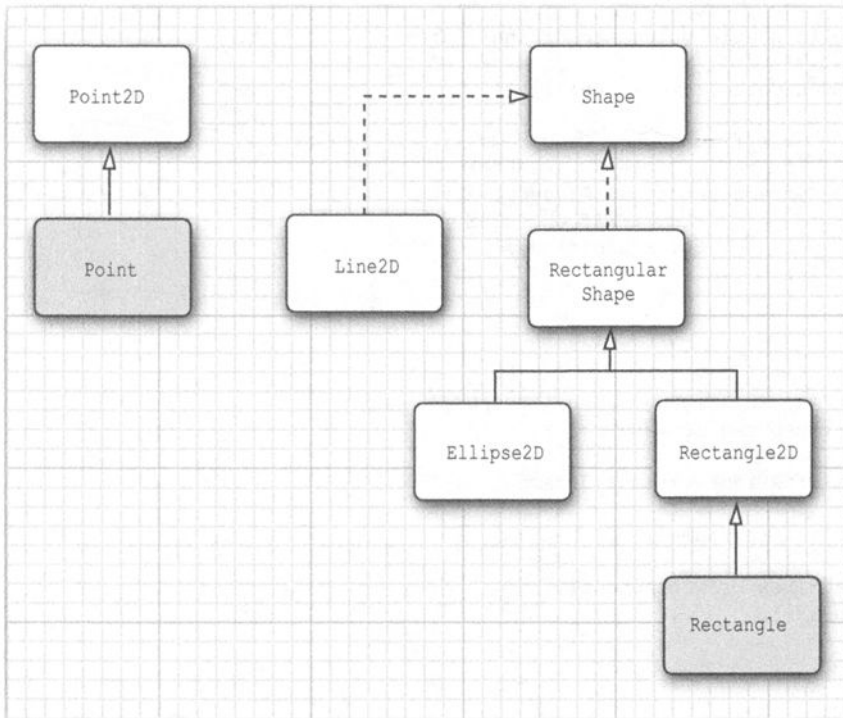


Рис. 10.7. Отношения между классами, представляющими двухмерные формы

Чтобы нарисовать линию, следует задать ее начальную и конечную точки в виде объектов типа `Point2D` или в виде пары чисел:

```
var line = new Line2D.Double(start, end);
```

или

```
var line = new Line2D.Double(startX, startY, endX, endY);
```

В примере программы, исходный код которой приведен в листинге 10.3, рисуются прямоугольник, эллипс, вписанный в прямоугольник, диагональ прямоугольника, а также окружность, центр которой совпадает с центром прямоугольника. Результат выполнения этой программы показан на рис. 10.8.

Листинг 10.3. Исходный код из файла `draw/DrawTest.java`

```

1 package draw;
2
3 import java.awt.*;
4 import java.awt.geom.*;
5 import javax.swing.*;
6
7 /**
8  * @version 1.34 2018-04-10
9  * @author Cay Horstmann
10 */
11 public class DrawTest
  
```

```
12 {
13     public static void main(String[] args)
14     {
15         EventQueue.invokeLater(() ->
16         {
17             var frame = new DrawFrame();
18             frame.setTitle("DrawTest");
19             frame.setDefaultCloseOperation(
20                 JFrame.EXIT_ON_CLOSE);
21             frame.setVisible(true);
22         });
23     }
24 }
25
26 /**
27  * Фрейм, содержащий панель с нарисованными
28  * двумерными формами
29  */
30 class DrawFrame extends JFrame
31 {
32     public DrawFrame()
33     {
34         add(new DrawComponent());
35         pack();
36     }
37 }
38
39 /**
40  * Компонент, отображающий прямоугольники и эллипсы
41  */
42 class DrawComponent extends JComponent
43 {
44     private static final int DEFAULT_WIDTH = 400;
45     private static final int DEFAULT_HEIGHT = 400;
46
47     public void paintComponent(Graphics g)
48     {
49         var g2 = (Graphics2D) g;
50
51         // нарисовать прямоугольник
52
53         double leftX = 100;
54         double topY = 100;
55         double width = 200;
56         double height = 150;
57
58         var rect = new Rectangle2D.Double(leftX, topY,
59                                           width, height);
60         g2.draw(rect);
61
62         // нарисовать вписанный эллипс
63
64
65         var ellipse = new Ellipse2D.Double();
66         ellipse.setFrame(rect);
67         g2.draw(ellipse);
68     }
```

```
69 // нарисовать диагональную линию
70
71 g2.draw(new Line2D.Double(leftX, topY, leftX + width,
72                           topY + height));
73
74 // нарисовать окружность с тем же самым центром
75
76 double centerX = rect.getCenterX();
77 double centerY = rect.getCenterY();
78 double radius = 150;
79
80 var circle = new Ellipse2D.Double();
81 circle.setFrameFromCenter(centerX, centerY,
82                           centerX + radius, centerY + radius);
83 g2.draw(circle);
84 }
85
86 public Dimension getPreferredSize()
87 {
88     return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT);
89 }
90 }
```

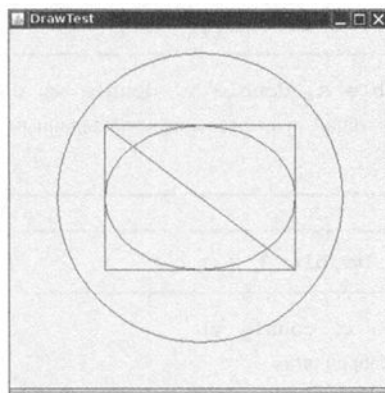


Рис. 10.8. Рисование геометрических двухмерных форм

java.awt.geom.RectangularShape 1.2

- `double getCenterX()`
- `double getCenterY()`
- `double getMinX()`
- `double getMinY()`
- `double getMaxX()`
- `double getMaxY()`

Возвращают координаты центра, наименьшую и наибольшую координату *x* и *y* описанного прямоугольника.

`java.awt.geom.RectangularShape 1.2 (окончание)`

- `double getWidth()`
- `double getHeight()`

Возвращают ширину и высоту описанного прямоугольника.

- `double getX()`
- `double getY()`

Возвращают координаты *x* и *y* левого верхнего угла описанного прямоугольника.

`java.awt.geom.Rectangle2D.Double 1.2`

- `Rectangle2D.Double(double x, double y, double w, double h)`

Строит прямоугольник по заданным координатам верхнего левого угла, ширине и высоте.

`java.awt.geom.Ellipse2D.Double 1.2`

- `Ellipse2D.Double(double x, double y, double w, double h)`

Строит эллипс с ограничивающим прямоугольником по заданным координатам верхнего левого угла, ширине и высоте.

`java.awt.geom.Point2D.Double 1.2`

- `Point2D.Double(double x, double y)`

Рисует точку по заданным координатам.

`java.awt.geom.Line2D.Double 1.2`

- `Line2D.Double(Point2D start, Point2D end)`
- `Line2D.Double(double startX, double startY, double endX, double endY)`

Рисует линию по заданным координатам начальной и конечной точек.

10.3.2. Окрашивание цветом

Метод `setPaint()` из класса `Graphics2D` позволяет выбрать цвет, который будет применяться при всех дальнейших операциях рисования в графическом контексте. Ниже приведен пример применения этого метода в прикладном коде.

```
g2.setPaint(Color.RED);  
g2.drawString("Warning!", 100, 100);
```

Окрашивать цветом можно внутренние участки замкнутых двухмерных геометрических форм вроде прямоугольников или эллипсов. Для этого вместо метода `draw()` достаточно вызвать метод `fill()` следующим образом:

```
Rectangle2D rect = . . . ;
g2.setPaint(Color.RED);
g2.fill(rect); // заполнить прямоугольник красным цветом
```

Для окрашивания разными цветами следует выбрать определенный цвет, нарисовать или заполнить одну форму, затем выбрать новый цвет, нарисовать или заполнить другую форму и т.д.



НА ЗАМЕТКУ! В методе `fill()` окрашивание цветом осуществляется на один пиксель меньше вправо и вниз. Так, если сделать вызов `new Rectangle2D.Double(0, 0, 10, 20)`, чтобы нарисовать новый прямоугольник, то в нарисованную форму войдут пиксели с координатами $x = 10$ и $y = 20$. Если же заполнить ту же самую прямоугольную форму выбранным цветом, то пиксели с этими координатами не будут окрашены.

Цвет определяется с помощью класса `Color`. Класс `java.awt.Color` содержит следующие константы, соответствующие 13 стандартным цветам: `BLACK` (ЧЕРНЫЙ), `BLUE` (синий), `CYAN` (голубой), `DARK_GRAY` (темно-серый), `GRAY` (серый), `GREEN` (зеленый), `LIGHT_GRAY` (светло-серый), `MAGENTA` (пурпурный), `ORANGE` (оранжевый), `PINK` (розовый), `RED` (красный), `WHITE` (белый), `YELLOW` (желтый).

Имеется также возможность указать произвольный цвет по его красной, зеленой и синей составляющим, создав объект класса `Color` и указав их значения в пределах от 0 до 255, как показано ниже.

```
g2.setPaint(new Color(0, 128, 128));
// скучный сине-зеленый цвет
g2.drawString("Welcome!", 75, 125);
```



НА ЗАМЕТКУ! Кроме выбора сплошного цвета, можно рисовать градиенты и текстуры, вызвав метод `Paint()` с экземплярами классов, реализующих интерфейс `Paint`.

Чтобы установить *цвет фона*, следует вызвать метод `setBackground()` из класса `Component`, предшественника класса `JComponent`:

```
var component = new MyComponent();
component.setBackground(Color.PINK);
```

Имеется также метод `setForeground()`. Он задает цвет переднего плана, который используется при рисовании компонента.

`java.awt.Color 1.0`

- `Color(int r, int g, int b)`

Создает объект цвета с заданными значениями красной, зеленой и синей составляющих в пределах от 0 до 255.

java.awt.Graphics2D 1.2

- **Paint** `getPaint()`
- **void** `setPaint(Paint p)`

Получают или устанавливают атрибуты рисования для данного графического контекста. Класс **Color** реализует интерфейс **Paint**. Этот метод можно использовать для задания сплошного цвета при рисовании.

- **void** `fill(Shape s)`
Заполняет текущую нарисованную форму.

java.awt.Component 1.0

- **Color** `getBackground()`
- **void** `setBackground(Color c)`
- **Color** `getForeground()`
- **void** `setForeground(Color c)`

Получают или устанавливают цвет переднего и заднего плана.

10.3.3. Применение шрифтов

Программа, приведенная в начале этой главы, выводила на экран текстовую строку, выделенную шрифтом, выбираемым по умолчанию. Но нередко требуется, чтобы текст отображался разными шрифтами. Шрифты определяются начертанием символов. Название начертания состоит из названия *гарнитуры* шрифтов (например, Helvetica) и необязательного суффикса (например, Bold для полужирного начертания). Так, названия Helvetica и Helvetica Bold считаются частью одной и той же гарнитуры шрифта Helvetica.

Чтобы выяснить, какие шрифты доступны на отдельном компьютере, следует вызвать метод `getAvailableFamilyNames()` из класса `GraphicsEnvironment`. Этот метод возвращает массив строк, состоящих из названий всех доступных в системе шрифтов. Чтобы создать экземпляр класса `GraphicsEnvironment`, описывающего графическую среду, вызывается статический метод `getLocalGraphicsEnvironment()`. Таким образом, приведенная ниже краткая программа выводит названия всех шрифтов, доступных в отдельной системе.

```
import java.awt.*;

public class ListFonts
{
    public static void main(String[] args)
    {
        String[] fontNames = GraphicsEnvironment
            .getLocalGraphicsEnvironment()
            .getAvailableFontFamilyNames();
        for (String fontName : fontNames)
            System.out.println(fontName);
    }
}
```

```
}  
}
```

В качестве общего основания в библиотеке AWT приняты следующие пять *логических названий* шрифтов:

```
SansSerif  
Serif  
Monospaced  
Dialog  
DialogInput
```

Эти названия всегда приводятся к шрифтам, фактически существующим на отдельной клиентской машине. Например, в Windows шрифт SanSerif приводится к шрифту Arial. Кроме того, в комплект JDK компании Oracle всегда входят три гарнитуры шрифтов: "Lucida Sans", "Lucida Bright" и "Lucida Sans Typewriter".

Чтобы воспроизвести букву заданным шрифтом, сначала нужно создать объект класса Font, а затем указать название шрифта, его стиль и размер. Ниже показано, каким образом создается объект класса Font.

```
var sansbold14 = new Font("SansSerif", Font.BOLD, 14);
```

В качестве третьего параметра в конструкторе класса Font задается размер шрифта. Для обозначения размера шрифта служит единица измерения, называемая *пунктом*. В одном дюйме содержится 72 пункта. В конструкторе класса Font вместо фактического названия начертания можно использовать логическое название шрифта. Затем нужно указать стиль (т.е. простой, **полужирный**, *курсив* или **полужирный курсив**), задав для второго параметра конструктора одно из следующих значений:

```
Font.PLAIN  
Font.BOLD  
Font.ITALIC  
Font.BOLD + Font.ITALIC
```

Данный шрифт имеет простое начертание и размер 1 пункт. Чтобы получить шрифт нужного размера, следует вызвать метод `deriveFont()`, как показано в приведенной ниже строке кода.

```
Font df = f.deriveFont(14.0F);
```



ВНИМАНИЕ! Имеются две перегружаемые версии метода `deriveFont()`. В одной из них (с параметром типа `float`) задается размер шрифта, а в другой (с параметром типа `int`) — стиль шрифта. Таким образом, при вызове `f.deriveFont(14)` задается стиль, а не размер шрифта! (В итоге будет установлено наклонное начертание, т.е. курсив, поскольку двоичное представление числа 14 содержит единицу в разряде, соответствующем константе `ITALIC`, но не константе `BOLD`.)

Ниже приведен фрагмент кода для вывода на экран символьной строки "Hello, World", набранной полужирным шрифтом SanSerif размером 14 пунктов.

```
var sansbold14 = new Font("SansSerif", Font.BOLD, 14);  
g2.setFont(sansbold14);  
String message = "Hello, World!";  
g2.drawString(message, 75, 100);
```

А теперь требуется выровнять текстовую строку по *центру* компонента. Для этого нужно знать ширину и высоту строки в пикселях. Процесс выравнивания зависит от следующих факторов.

- Используемый шрифт (в данном случае полужирный шрифт sans serif размером 14 пунктов).
- Символы строки (в данном случае "Hello, World!").
- Устройство, на котором будет воспроизводиться строка (в данном случае экран монитора пользователя).

Чтобы получить объект, представляющий характеристики устройства вывода, следует вызвать метод `getFontRenderContext()` из класса `Graphics2D`. Он возвращает объект класса `FontRenderContext`. Этот объект нужно передать методу `getStringBounds()` из класса `Font`, как показано ниже. А метод `getStringBounds()` возвращает прямоугольник, ограничивающий текстовую строку.

```
FontRenderContext context = g2.getFontRenderContext();
Rectangle2D bounds = sansbold14.getStringBounds(
    message, context);
```

Чтобы выяснить, от чего зависят размеры этого прямоугольника, следует рассмотреть некоторые основные термины, применяемые при типографском наборе текста (рис. 10.9). *Базовая линия* — это воображаемая линия, которая касается снизу таких символов, как **e**. *Подъем* — максимальное расстояние от базовой линии до верхушек надстрочных элементов, например, верхнего края буквы **b** или **k**. *Спуск* — это расстояние от базовой линии до подстрочного элемента, например, нижнего края буквы **p** или **g**.

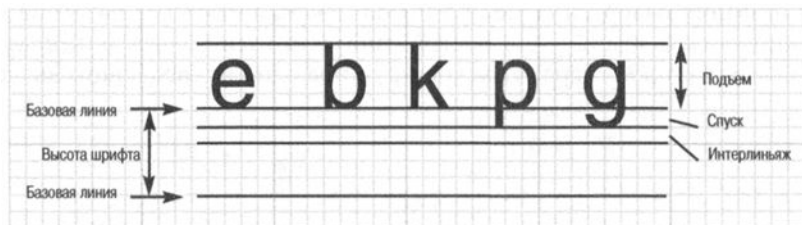


Рис. 10.9. Основные термины, применяемые при формировании текстовой строки

Интерлиньяж — это разность между спуском предыдущей строки и подъемом следующей. *Высота шрифта* — это расстояние между соседними базовыми линиями, равное сумме спуска, подъема и интерлиньяжа.

Ширина прямоугольника, возвращаемого методом `getStringBounds()`, задает протяженность строки по горизонтали. Высота прямоугольника равна сумме спуска, подъема и интерлиньяжа. Начало отсчета прямоугольника находится на базовой линии строки. Координата *y* отсчитывается от базовой линии. Для верхней части прямоугольника она является отрицательной. Таким образом, ширину, высоту и подъем строки можно вычислить следующим образом:

```
double stringWidth = bounds.getWidth();
double stringHeight = bounds.getHeight();
double ascent = -bounds.getY();
```

Если же требуется выяснить, чему равен интерлиньяж или спуск, то следует вызвать метод `getLineMetrics()` из класса `Font`, как показано ниже. Этот метод возвращает объект класса `LineMetrics`, где имеются методы для определения указанных выше типографских характеристик.

```
LineMetrics metrics = f.getLineMetrics(message, context);  
float descent = metrics.getDescent();  
float leading = metrics.getLeading();
```

Чтобы показать, насколько правильно расположена текстовая строка, в примере программы из листинга 10.4 отображаются базовая линия и ограничивающий прямоугольник. На рис. 10.10 приведен результат выполнения этой программы, выводимый на экран, а ее исходный код — в листинге 10.4.



Рис. 10.10. Тестовая строка, отображаемая на экране с базовой линией и ограничивающим прямоугольником

Листинг 10.4. Исходный код из файла `font/FontTest.java`

```
1 package font;  
2  
3 import java.awt.*;  
4 import java.awt.font.*;  
5 import java.awt.geom.*;  
6 import javax.swing.*;  
7  
8 /**  
9  * @version 1.35 2018-04-10  
10 * @author Cay Horstmann  
11 */  
12 public class FontTest  
13 {  
14     public static void main(String[] args)  
15     {  
16         EventQueue.invokeLater(() ->  
17             {  
18                 var frame = new FontFrame();  
19                 frame.setTitle("FontTest");  
20                 frame.setDefaultCloseOperation(  
21                     JFrame.EXIT_ON_CLOSE);  
22                 frame.setVisible(true);  
23             });  
24     }  
25 }  
26  
27 /**  
28 * Фрейм с компонентом текстового сообщения  
29 */  
30 class FontFrame extends JFrame  
31 {
```

```
32 public FontFrame()
33 {
34     add(new FontComponent());
35     pack();
36 }
37 }
38
39 /**
40  * А Компонент, отображающий текстовое сообщение,
41  * выровненное по центру в прямоугольной рамке
42  */
43 class FontComponent extends JComponent
44 {
45     private static final int DEFAULT_WIDTH = 300;
46     private static final int DEFAULT_HEIGHT = 200;
47
48     public void paintComponent(Graphics g)
49     {
50         var g2 = (Graphics2D) g;
51
52         var message = "Hello, World!";
53
54         var f = new Font("Serif", Font.BOLD, 36);
55         g2.setFont(f);
56
57         // определить размеры текстового сообщения
58
59         FontRenderContext context =
60             g2.getFontRenderContext();
61         Rectangle2D bounds = f.getStringBounds(message,
62             context);
63
64         // определить координаты (x,y)
65         // верхнего левого угла текста
66
67         double x = (getWidth() - bounds.getWidth()) / 2;
68         double y = (getHeight() - bounds.getHeight()) / 2;
69
70         // сложить подъем с координатой y,
71         // чтобы достичь базовой линии
72
73         double ascent = -bounds.getY();
74         double baseY = y + ascent;
75
76         // воспроизвести текстовое сообщение
77
78         g2.drawString(message, (int) x, (int) baseY);
79
80         g2.setPaint(Color.LIGHT_GRAY);
81
82         // нарисовать базовую линию
83
84         g2.draw(new Line2D.Double(x, baseY,
85             x + bounds.getWidth(), baseY));
86
87         // нарисовать ограничивающий прямоугольник
```

```
88
89     var rect = new Rectangle2D.Double(x, y,
90         bounds.getWidth(), bounds.getHeight());
91     g2.draw(rect);
92 }
93
94 public Dimension getPreferredSize()
95 {
96     return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT);
97 }
98 }
```

`java.awt.Font 1.0`

- **Font(String name, int style, int size)**
Создает новый объект типа **Font** для описания шрифта.
- **String getFontName()**
Возвращает название начертания шрифта (например, **Helvetica Bold**).
- **String getFamily()**
Возвращает название гарнитуры шрифта (например, **Helvetica**).
- **String getName()**
Возвращает логическое название шрифта (например, **SansSerif**), если оно присвоено шрифту при его создании, а иначе — название начертания шрифта.
- **Rectangle2D getStringBounds(String s, FontRenderContext context) 1.2**
Возвращает прямоугольник, ограничивающий данную строку. Координата **y** прямоугольника отсчитывается от базовой линии. Координата **y** верхней части прямоугольника равна отрицательному подъему. Высота прямоугольника равна сумме подъема, спуска и интерлиньяжа. Ширина прямоугольника равна ширине строки.
- **LineMetrics getLineMetrics(String s, FontRenderContext context) 1.2**
Возвращает объект типа **LineMetrics**, описывающий типографские характеристики текстовой строки, чтобы определить ее протяженность.
- **Font deriveFont(int style) 1.2**
- **Font deriveFont(float size) 1.2**
- **Font deriveFont(int style, float size) 1.2**
Возвращают новый объект типа **Font** для описания шрифта, совпадающего с текущим шрифтом, за исключением размера и стиля, задаваемых в качестве параметров.

`java.awt.font.LineMetrics 1.2`

- **float getAscent()**
Получает подъем шрифта — расстояние от базовой линии до вершук прописных букв.
- **float getDescent()**
Получает спуск — расстояние от базовой линии до подстрочных элементов букв.

java.awt.font.LineMetrics 1.2 (окончание)

- **float getLeading()**
Получает интерлиньяж — расстояние от нижнего края предыдущей строки до верхнего края следующей строки.
- **float getHeight()**
Получает общую высоту шрифта — расстояние между двумя базовыми линиями текста, равное сумме спуска, интерлиньяжа и подъема.

java.awt.Graphics2D 1.2

- **FontRenderContext getFontRenderContext()**
Получает контекст воспроизведения, в котором задаются характеристики шрифта для текущего графического контекста.
- **void drawString(String str, float x, float y)**
Выводит текстовую строку, выделенную текущим шрифтом и цветом.

javax.swing.JComponent 1.2

- **FontMetrics getFontMetrics(Font f)** 5
Получает типографские характеристики заданного шрифта. Класс **FontMetrics** является предшественником класса **LineMetrics**.

java.awt.FontMetrics 1.0

- **FontRenderContext getFontRenderContext()** 1.2
Получает контекст воспроизведения для шрифта.

10.3.4. Воспроизведение изображений

Чтобы прочитать изображение из файла, можно воспользоваться классом **ImageIcon** следующим образом:

```
Image image = new ImageIcon(filename).getImage();
```

Теперь переменная **image** содержит ссылку на объект, инкапсулирующий данные изображения. Используя этот объект, изображение можно далее вывести на экран с помощью метода **drawImage()** из класса **Graphics**, как показано ниже.

```
public void paintComponent(Graphics g)
{
    . . .
    g.drawImage(image, x, y, null);
}
```

Можно пойти еще дальше, чтобы вывести указанное изображение в окне рядами. Получаемый в итоге результат приведен на рис. 10.11. Вывод изображения рядами осуществляется с помощью метода `paintComponent()`. Сначала одна копия изображения воспроизводится в левом верхнем углу окна, а затем вызывается метод `copyArea()`, который копирует его по всему окну:

```
for (int i = 0; i * imageWidth <= getWidth(); i++)  
    for (int j = 0; j * imageHeight <= getHeight(); j++)  
        if (i + j > 0)  
            g.copyArea(0, 0, imageWidth, imageHeight,  
                        i * imageWidth, j * imageHeight);
```

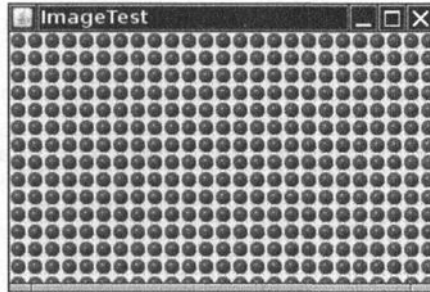


Рис. 10.11. Окно с графическим изображением, воспроизводимым рядами

java.awt.Graphics 1.0

- `boolean drawImage(Image img, int x, int y, ImageObserver observer)`
- `boolean drawImage(Image img, int x, int y, int width, int height, ImageObserver observer)`
- Выводят немасштабированное изображение. *Примечание:* эти методы могут вернуть управление до того, как изображение будет выведено полностью. Объект типа `ImageObserver` уведомляется о процессе воспроизведения. Это приносило пользу в далеком прошлом, теперь достаточно передать наблюдателю пустое значение `null`.

10.4. Обработка событий

В любой операционной среде, поддерживающей GUI, непрерывно отслеживаются такие события, как нажатие клавиш или щелчки кнопками мыши, а затем о них сообщается исполняемой программе, которая сама решает, как реагировать на них.

10.4.1. Общее представление об обработке событий

В библиотеке AWT у *источников событий* (например, экранных кнопок или полос прокрутки) имеются методы, позволяющие регистрировать *приемники событий* — объекты, несущие требующуюся реакцию на события. Когда приемник событий уведомляется о наступившем событии, сведения об этом событии инкапсулируются в *объекте события*. Все события в Java описываются подклассами, производными

от класса `java.util.EventObject`. Разумеется, существуют подклассы и для каждого вида событий, например `ActionEvent` и `WindowEvent`.

Различные источники могут порождать разные виды событий. Например, экранная кнопка может посылать объекты типа `ActionEvent`, а окно — объекты типа `WindowEvent`. Кратко механизм обработки событий в библиотеке AWT можно описать следующим образом.

- Объект приемника событий — это экземпляр класса, реализующего специальный интерфейс, называемый (естественно) *интерфейсом приемника событий*.
- Источник событий — это объект, который может регистрировать приемники событий и посылать им объекты событий.
- При наступлении события источник посылает объекты событий всем зарегистрированным приемникам.
- Приемники используют данные, инкапсулированные в объекте события, чтобы решить, как реагировать на это событие.

На рис. 10.12 схематически показаны отношения между классами обработки событий и интерфейсами.



Рис. 10.12. Отношения между источниками и приемниками событий

В приведенном ниже примере кода показано, каким образом указывается приемник событий.

```
ActionListener listener = . . . ;
var = new JButton("Ok");
button.addActionListener(listener);
```

Теперь объект `listener` оповещается о наступлении "события действия" в экранной кнопке. Для экранной кнопки, как и следовало ожидать, таким событием действия является щелчок на ней кнопкой мыши. Чтобы реализовать интерфейс `ActionListener`, в классе приемника событий должен присутствовать метод `actionPerformed()`, принимающий в качестве параметра объект типа `ActionEvent`:

```
class MyListener implements ActionListener
{
    . . .
    public void actionPerformed(ActionEvent event)
    {
        // здесь следует код, реагирующий на щелчок
    }
}
```

```
// на экранной кнопке  
...  
}  
}
```

Когда пользователь щелкает на экранной кнопке, объект типа `JButton` создает объект типа `ActionEvent` и вызывает метод `listener.actionPerformed(event)`, передавая ему объект события. У источника событий может быть несколько приемников. В этом случае после щелчка на экранной кнопке объект типа `JButton` вызовет метод `actionPerformed()` для всех зарегистрированных приемников событий. На рис. 10.13 схематически показано взаимодействие источника, приемника и объекта события.

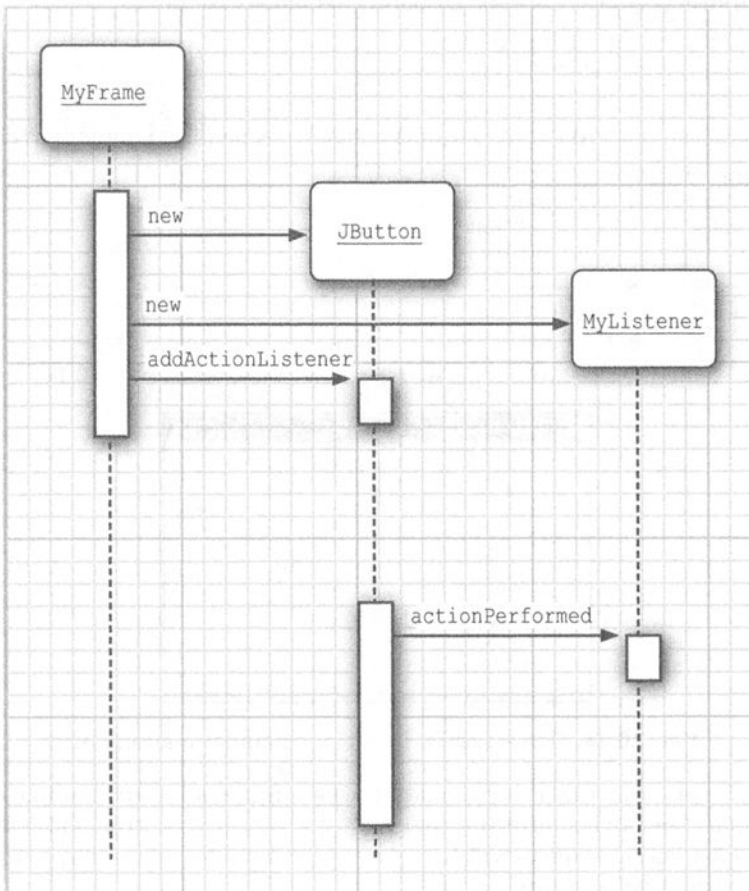


Рис. 10.13. Уведомление о событии

10.4.2. Пример обработки событий от щелчков на экранных кнопках

Чтобы стал понятнее принцип действия модели делегирования событий, рассмотрим подробно, как обрабатываются события от щелчков на экранных кнопках. Для этого потребуются три экранные кнопки, расположенные на панели, а также три

объекта приемников событий, добавляемые к экранным кнопкам в качестве приемников действий над ними.

Всякий раз, когда пользователь щелкает кнопкой мыши на какой-нибудь экранной кнопке, находящейся на панели, соответствующий приемник получает объект типа `ActionEvent`, указывающий на факт щелчка. В рассматриваемом здесь примере программы объект приемника событий, реагируя на щелчок, будет изменять цвет фона панели.

Но прежде чем демонстрировать пример программы, реагирующей на щелчки на экранных кнопках, необходимо пояснить, каким образом эти кнопки создаются и вводятся на панели. Чтобы создать экранную кнопку, необходимо указать в конструкторе ее класса символьную строку метки, пиктограмму или оба атрибута вместе. Ниже приведены два примера создания экранных кнопок.

```
var yellowButton = new JButton("Yellow");  
var blueButton = new JButton(new ImageIcon(  
    "blue-ball.gif"));
```

Затем вызывается метод `add()`, чтобы добавить экранные кнопки на панели:

```
var yellowButton = new JButton("Yellow");  
var blueButton = new JButton("Blue");  
var redButton = new JButton("Red");
```

```
buttonPanel.add(yellowButton);  
buttonPanel.add(blueButton);  
buttonPanel.add(redButton);
```

Результат выполнения этого фрагмента кода приведен на рис. 10.14.

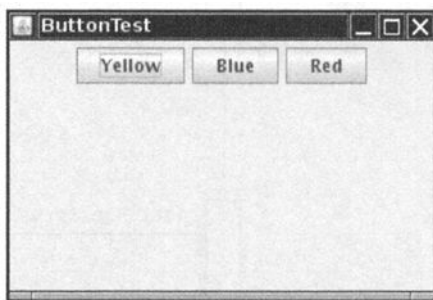


Рис. 10.14. Панель, заполненная экранными кнопками

Далее следует ввести код, позволяющий реагировать на эти кнопки. Для этого требуется класс, реализующий интерфейс `ActionListener`, где, как упоминалось выше, объявлен единственный метод `actionPerformed()`. Сигнатура этого метода выглядит следующим образом:

```
public void actionPerformed(ActionEvent event)
```

В любом случае интерфейс `ActionListener` применяется совершенно одинаково: метод `actionPerformed()` — единственный в интерфейсе `ActionListener` — принимает в качестве параметра объект типа `ActionEvent`. Этот объект несет в себе сведения о наступившем событии.

Допустим, что после щелчка на экранной кнопке требуется изменить цвет фона панели. Новый цвет указывается в классе приемника событий следующим образом:

```
class ColorAction implements ActionListener
{
    private Color backgroundColor;

    public ColorAction(Color c)
    {
        backgroundColor = c;
    }

    public void actionPerformed(ActionEvent event)
    {
        // установить цвет фона панели
        . . .
    }
}
```

Затем для каждого цвета создается один объект. Все эти объекты устанавливаются в качестве приемников событий от соответствующих кнопок:

```
var yellowAction = new ColorAction(Color.YELLOW);
var blueAction = new ColorAction(Color.BLUE);
var redAction = new ColorAction(Color.RED);

yellowButton.addActionListener(yellowAction);
blueButton.addActionListener(blueAction);
redButton.addActionListener(redAction);
```

Так, если пользователь щелкнет на экранной кнопке с меткой Yellow, вызывается метод `actionPerformed()` из объекта `yellowAction`. В поле экземпляра `backgroundColor` хранится значение `Color.YELLOW`, поэтому выполняющийся метод может установить требуемый (в данном случае желтый) цвет фона панели.

Осталось разрешить еще одно небольшое затруднение. Объект типа `ColorAction` не имеет доступа к переменной `buttonPanel`. Это затруднение можно разрешить разными путями. В частности, переменную `buttonPanel` можно указать в конструкторе класса `ColorAction`. Но удобнее сделать класс `ColorAction` внутренним по отношению к классу `ButtonFrame`. В этом случае его методы получают доступ к внешним переменным автоматически.

В листинге 10.5 приведен весь исходный код класса фрейма, реализующего обработку событий от экранных кнопок. Как только пользователь щелкнет на какой-нибудь экранной кнопке, соответствующий приемник событий изменит цвет фона панели.

Листинг 10.5. Исходный код из файла `button/ButtonFrame.java`

```
1 package button;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 /**
8  * Фрейм с панелью экранных кнопок
9  */
10 public class ButtonFrame extends JFrame
11 {
```

```
12 private JPanel buttonPanel;
13 private static final int DEFAULT_WIDTH = 300;
14 private static final int DEFAULT_HEIGHT = 200;
15
16 public ButtonFrame()
17 {
18     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
19
20     // создать экранные кнопки
21     var = new JButton("Yellow");
22     var = new JButton("Blue");
23     var = new JButton("Red");
24
25     buttonPanel = new JPanel();
26
27     // ввести экранные кнопки на панели
28     buttonPanel.add(yellowButton);
29     buttonPanel.add(blueButton);
30     buttonPanel.add(redButton);
31
32     // ввести панель во фрейм
33     add(buttonPanel);
34
35     // сформировать действия экранных кнопок
36     var = new ColorAction(Color.YELLOW);
37     var = new ColorAction(Color.BLUE);
38     var = new ColorAction(Color.RED);
39
40     // связать действия с экранными кнопками
41     yellowButton.addActionListener(yellowAction);
42     blueButton.addActionListener(blueAction);
43     redButton.addActionListener(redAction);
44 }
45
46 /**
47  * Приемник действий, устанавливающий цвет фона панели
48  */
49 private class ColorAction implements ActionListener
50 {
51     private Color backgroundColor;
52
53     public ColorAction(Color c)
54     {
55         backgroundColor = c;
56     }
57
58     public void actionPerformed(ActionEvent event)
59     {
60         buttonPanel.setBackground(backgroundColor);
61     }
62 }
63 }
```

javax.swing.JButton 1.2

- `JButton(String label)`
- `JButton(Icon icon)`
- `JButton(String label, Icon icon)`

Создают экранную кнопку. Символьная строка, передаваемая в качестве параметра, может содержать текст или HTML-разметку, например, `<HTML>Ок</HTML>`.

java.awt.Container 1.0

- `Component add(Component c)`

Добавляет заданный компонент `c` в контейнер.

10.4.3. Краткое обозначение приемников событий

В предыдущем разделе был определен класс для приемника событий и построены три объекта этого класса. Наличие нескольких экземпляров класса приемника событий требуется редко. Чаще всего каждый приемник событий выполняет отдельное действие. И в таком случае отпадает необходимость создавать отдельный класс для приемника событий. А вместо этого проще воспользоваться лямбда-выражением следующим образом:

```
exitButton.addActionListener(event -> System.exit(0));
```

А теперь рассмотрим случай, когда имеется несколько связанных вместе действий (например, при выборе экранных кнопок в программе из предыдущего раздела). В этом случае необходимо реализовать следующий вспомогательный метод:

```
public void makeButton(String name, Color backgroundColor)
{
    var = new JButton(name);
    buttonPanel.add(button);
    button.addActionListener(event ->
        buttonPanel.setBackground(backgroundColor));
}
```

Обратите внимание на то, что в приведенном выше лямбда-выражении делается ссылка на переменную параметра `backgroundColor`. И тогда вспомогательный метод вызывается следующим образом:

```
makeButton("yellow", Color.YELLOW);
makeButton("blue", Color.BLUE);
makeButton("red", Color.RED);
```

В данном случае три объекта приемников (по одному на каждый цвет) конструируются без явного определения класса. Всякий раз, когда вызывается вспомогательный метод, он создает экземпляр класса, реализующего интерфейс `ActionListener`. Его метод `actionPerformed()`, реализующий выполняемое действие, обращается к значению параметра `backgroundColor`, которое, по существу, сохраняется объектом приемника событий. Но все это происходит без явного определения классов приемников событий, переменных экземпляра или конструкторов, которые их устанавливают.



НА ЗАМЕТКУ! В прежнем коде можно нередко встретить употребление анонимных классов следующим образом:

```
exitButton.addActionListener(new ActionListener()
{
    public void actionPerformed(new ActionEvent)
    {
        System.exit(0);
    }
});
```

Безусловно, такой код оказывается довольно многословным. Но этого больше не требуется благодаря более простым и понятным лямбда-выражениям.

10.4.4. Классы адаптеров

Не все события обрабатываются так же просто, как и события от экранных кнопок. Допустим, требуется постоянно контролировать, когда пользователь пытается закрыть главный фрейм, чтобы свернуть диалоговое окно и выйти из программы, но только с согласия пользователя.

Когда пользователь пытается закрыть фрейм, объект типа `JFrame` становится источником события типа `WindowEvent`. Чтобы перехватить это событие, требуется соответствующий объект приемника событий, который следует добавить в список приемников событий в окне, как показано ниже.

```
WindowListener listener = . . . ;
frame.addWindowListener(listener);
```

Приемник событий должен быть объектом класса, реализующего интерфейс `WindowListener`. В интерфейсе `WindowListener` имеется семь методов. Фрейм вызывает их в ответ на семь разных событий, которые могут произойти в окне. Имена этих методов отражают их назначение. Исключением может быть лишь слово **Iconified**, которое в Windows означает “свернутое” окно. Ниже показано, как выглядит весь интерфейс `WindowListener`.

```
public interface WindowListener
{
    void windowOpened(WindowEvent e);
    void windowClosing(WindowEvent e);
    void windowClosed(WindowEvent e);
    void windowIconified(WindowEvent e);
    void windowDeiconified(WindowEvent e);
    void windowActivated(WindowEvent e);
    void windowDeactivated(WindowEvent e);
}
```

Безусловно, можно определить класс, реализующий интерфейс `WindowListener`, введя в тело его метода `windowClosing()` вызов `System.exit(0)`, а тела остальных шести методов оставить пустыми. Но набирать код для шести методов, которые ничего не делают, — неблагодарное занятие. Чтобы упростить эту задачу, для каждого из интерфейсов приемников событий в AWT, у которых имеется несколько методов, создается сопутствующий класс адаптера, реализующий все эти методы, причем тела их остаются пустыми. Например, класс `WindowAdapter` содержит семь методов, не выполняющих никаких действий. Следовательно, класс адаптера автоматически

удовлетворяет требованиям, предъявляемым к реализации соответствующего интерфейса. Класс адаптера можно расширить и уточнить нужные виды реакции на некоторые, но не на все виды событий в интерфейсе. (Обратите внимание на то, что интерфейс `ActionListener` содержит только один метод, поэтому для него класс адаптера не нужен.)

Ниже показано, как определить приемник событий в окне, переопределяющий метод `windowClosing()`.

```
class Terminator extends WindowAdapter
{
    public void windowClosing(WindowEvent e)
    {
        if (пользователь согласен)
            System.exit(0);
    }
}
```

Теперь объект класса `Terminator` можно зарегистрировать в качестве приемника событий, как показано ниже.

```
var listener = new Terminator();
frame.addWindowListener(listener);
```



НА ЗАМЕТКУ! Теперь методы из интерфейса `WindowListener`, которые ничего не делают, можно было бы определить как методы с реализацией по умолчанию. Но ведь библиотека `Swing` была разработана задолго до внедрения в `Java` методов с реализацией по умолчанию.

`java.awt.event.WindowListener 1.1`

- `void windowOpened(WindowEvent e)`
Вызывается после открытия окна.
- `void windowClosing(WindowEvent e)`
Вызывается, когда пользователь выдает диспетчеру окон команду закрыть окно. Следует, однако, иметь в виду, что окно закроется только в том случае, если для него будет вызван метод `hide()` или `dispose()`.
- `void windowClosed(WindowEvent e)`
Вызывается после закрытия окна.
- `void windowIconified(WindowEvent e)`
Вызывается после свертывания окна.
- `void windowDeiconified(WindowEvent e)`
Вызывается после развертывания окна.
- `void windowActivated(WindowEvent e)`
Вызывается после активизации окна. Активным может быть только фрейм или диалоговое окно. Обычно диспетчер окон специально выделяет активное окно, например, подсвечивает его заголовок.
- `void windowDeactivated(WindowEvent e)`
Вызывается после того, как окно становится неактивным.


```
java.awt.event.WindowStateListener 1.4
```

- `void windowStateChanged(WindowEvent event) 1.4`

Вызывается после того, как окно было полностью развернуто, свернуто или восстановлено до нормальных размеров.

10.4.5. Действия

Одну и ту же команду можно выполнить разными способами. В частности, пользователь может выбрать пункт меню, нажать соответствующую клавишу или щелкнуть на экранной кнопке, находящейся на панели инструментов. В этом случае очень удобна рассматриваемая здесь модель делегирования событий в библиотеке AWT: достаточно связать все эти события с одним и тем же приемником. Допустим, что `blueAction` — это приемник действий, в методе `actionPerformed()` которого цвет фона изменяется на синий. Один и тот же объект можно связать с разными источниками событий, перечисленными ниже.

- Кнопка Blue (Синий) панели инструментов.
- Пункт меню Blue.
- Нажатие комбинации клавиш <Ctrl+B>.

Команда, изменяющая цвет фона, выполняется одинаково, независимо от того, что именно привело к ее выполнению — щелчок на экранной кнопке, выбор пункта меню или нажатие комбинации клавиш. В библиотеке Swing предусмотрен очень полезный механизм, позволяющий инкапсулировать команды и связывать их с несколькими источниками событий. Этим механизмом служит интерфейс *Action*. *Действие* представляет собой объект, инкапсулирующий следующее.

- Описание команды (в виде текстовой строки или пиктограммы).
- Параметры, необходимые для выполнения команды (в данном случае для выбора нужного цвета).

Интерфейс *Action* содержит следующие методы:

```
void actionPerformed(ActionEvent event)
void setEnabled(boolean b)
boolean isEnabled()
void putValue(String key, Object value)
Object getValue(String key)
void addPropertyChangeListener(
    PropertyChangeListener listener)
void removePropertyChangeListener(
    PropertyChangeListener listener)
```

Первый метод похож на аналогичный метод из интерфейса *ActionListener*. На самом деле интерфейс *Action* расширяет интерфейс *ActionListener*. Следовательно, вместо объекта класса, реализующего интерфейс *ActionListener*, можно использовать объект класса, реализующего интерфейс *Action*.

Следующие два метода позволяют активизировать или запретить действие, а также проверить, активизировано ли указанное действие в настоящий момент. Если пункт меню или кнопка панели инструментов связаны с запрещенным действием, они выделяются светло-серым цветом, как недоступные.

Методы `putValue()` и `getValue()` позволяют записывать и извлекать из памяти произвольные пары “имя–значение” из объектов действий класса, реализующего интерфейс `Action`. Так, в паре предопределенных строк `Action.NAME` и `Action.SMALL_ICON` имена и пиктограммы действий в объекте действия сохраняются следующим образом:

```
action.putValue(Action.NAME, "Blue");
action.putValue(Action.SMALL_ICON,
                new ImageIcon("blue-ball.gif"));
```

В табл. 10.1 перечислены предопределенные имена действий.

Таблица 10.1. Предопределенные имена действий

Имя	Значение
NAME	Имя действия. Отображается на экранной кнопке и в названии пункта меню
SMALL_ICON	Место для хранения пиктограммы, которая отображается на экранной кнопке, в пункте меню или на панели инструментов
SHORT_DESCRIPTION	Краткое описание пиктограммы, отображаемое во всплывающей подсказке
LONG_DESCRIPTION	Подробное описание пиктограммы. Может использоваться для подсказки. Не применяется ни в одном из компонентов библиотеки Swing
MNEMONIC_KEY	Мнемоническое сокращение. Отображается в пункте меню
ACCELERATOR_KEY	Место для хранения комбинации клавиш. Не применяется ни в одном из компонентов библиотеки Swing
ACTION_COMMAND_KEY	Применялось раньше в устаревшем теперь методе <code>registeredKeyboardAction()</code>
DEFAULT	Может быть полезным для хранения разнообразных объектов. Не применяется ни в одном из компонентов библиотеки Swing

Если в меню или на панели инструментов добавляется какое-то действие, его имя и пиктограмма автоматически извлекаются из памяти и отображаются в меню и на панели. Значение `SHORT_DESCRIPTION` выводится во всплывающей подсказке.

Последние два метода из интерфейса `Action` позволяют уведомить другие объекты, в частности, меню и панели инструментов, об изменении свойств объекта действия. Так, если меню введено в качестве приемника для изменений свойств в объекте действия и это действие впоследствии было запрещено, то при отображении меню на экране соответствующее имя действия может быть выделено светло-серым, как недоступное.

Следует, однако, иметь в виду, что `Action` является *интерфейсом*, а не классом. Любой класс, реализующий этот интерфейс, должен реализовать семь только что рассмотренных методов. Правда, сделать это совсем не трудно, поскольку все они, кроме первого метода, содержатся в классе `AbstractAction`, который предназначен для хранения пар “имя–значение”, а также для управления приемниками изменений свойств. На практике для этого достаточно создать соответствующий подкласс и ввести в него метод `actionPerformed()`.

В качестве примера попробуем создать объект действия, изменяющего цвет фона. Для этого в памяти размещаются имя команды, соответствующая пиктограмма и требующийся цвет. Код цвета записывается в таблицу, состоящую из пар “имя–значение”, предусмотренных в классе `AbstractAction`. Ниже приведен исходный код класса `ColorAction`, в котором выполняются все эти операции. В конструкторе этого класса задаются пары “имя–значение”, а в методе `ActionPerformed()` изменяется цвет фона.

```
public class ColorAction extends AbstractAction
{
    public ColorAction(String name, Icon icon, Color c)
    {
        putValue(Action.NAME, name);
        putValue(Action.SMALL_ICON, icon);
        putValue("color", c);
        putValue(Action.SHORT_DESCRIPTION, "Set panel color to "
            + name.toLowerCase());
    }

    public void actionPerformed(ActionEvent event)
    {
        Color c = (Color) getValue("color");
        buttonPanel.setBackground(c);
    }
}
```

В тестовой программе, рассматриваемой здесь в качестве примера, сначала создаются три объекта данного класса:

```
var = new ColorAction("Blue",
    new ImageIcon("blue-ball.gif"), Color.BLUE);
```

Затем действие по изменению цвета связывается с соответствующей кнопкой. Для этой цели служит конструктор класса `JButton`, получающий объект типа `ColorAction` в качестве параметра, как показано ниже.

```
var blueButton = new JButton(blueAction);
```

Этот конструктор считывает имя и пиктограмму действия, размещает его краткое описание во всплывающей подсказке и регистрирует объект типа `ColorAction` в качестве приемника действий. Пиктограмма и всплывающая подсказка приведены на рис. 10.15. Как будет показано в следующей главе, действия можно так же просто внедрять и в меню.

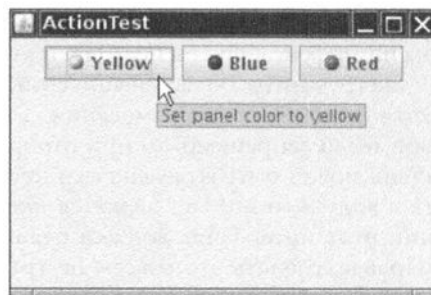


Рис. 10.15. Экранные кнопки с пиктограммами из объектов действий

И, наконец, объекты действий нужно связать с клавишами, чтобы эти действия выполнялись, когда пользователь нажимает соответствующие клавиши. Для того чтобы связать действия с нажатием клавиш, сначала нужно создать объект класса `KeyStroke`. Это удобный класс, инкапсулирующий описание клавиши следующим образом:

```
KeyStroke ctrlBKey = KeyStroke.getKeyStroke("ctrl B");
```

Прежде чем сделать следующий шаг, необходимо разъяснить понятие *фокуса ввода с клавиатуры*. Пользовательский интерфейс может состоять из многих экранных кнопок, меню, полос прокрутки и других компонентов. Нажатия кнопок передаются компонентам, обладающим фокусом ввода. Такой компонент обычно (но не всегда) выделяется для большей наглядности. Например, в визуальном стиле Java кнопка с фокусом ввода имеет тонкую прямоугольную рамку вокруг текста надписи. Для перемещения фокуса ввода между компонентами пользовательского интерфейса можно нажимать клавишу <Tab>. Когда же нажимается клавиша пробела, экранная кнопка, обладающая в данный момент фокусом ввода, оказывается выбранной. Другие клавиши вызывают иные действия; например, клавиши со стрелками служат для управления полосами прокрутки.

Но в данном случае нажатия клавиш не нужно посылать компоненту, владеющему фокусом ввода. Вместо этого каждая из экранных кнопок должна обрабатывать события, связанные с нажатием клавиш, и реагировать на комбинации клавиш <Ctrl+Y>, <Ctrl+B> и <Ctrl+R>.

Это часто встречающееся затруднение, и поэтому разработчики библиотеки Swing предложили удобный способ его разрешения. Каждый объект класса `JComponent` содержит три *привязки ввода*, связывающие объекты класса `KeyStroke` с действиями. Эти привязки ввода соответствуют разным условиям, как следует из табл. 10.2.

Таблица 10.2. Условия для привязки ввода

Условие	Вызываемое действие
<code>WHEN_FOCUSED</code>	Когда данный компонент находится в фокусе ввода с клавиатуры
<code>WHEN_ANCESTOR_OF_FOCUSED_COMPONENT</code>	Когда данный компонент содержит другой компонент, находящийся в фокусе ввода с клавиатуры
<code>WHEN_IN_FOCUSED_WINDOW</code>	Когда данный компонент содержится в том же окне, что и компонент, находящийся в фокусе ввода с клавиатуры

При нажатии клавиши условия привязки ввода проверяются в следующем порядке.

1. Проверяется условие привязки ввода `WHEN_FOCUSED` компонента, владеющего фокусом ввода. Если предусмотрена реакция на нажатие клавиши, выполняется соответствующее действие. И если действие разрешено, то обработка прекращается.
2. Начиная с компонента, обладающего фокусом ввода, проверяется условие привязки ввода `WHEN_ANCESTOR_OF_FOCUSED_COMPONENT` его родительского компонента. Как только обнаруживается привязка ввода с клавиатуры, выполняется соответствующее действие. И если действие разрешено, то обработка прекращается.
3. В окне, обладающем фокусом ввода, проверяются все *видимые и активизированные* компоненты с зарегистрированными нажатиями клавиш по условию привязки ввода `WHEN_IN_FOCUSED_WINDOW`. Каждый из этих компонентов (в порядке регистрации нажатий клавиш) получает возможность выполнить соответствующее действие. Как только будет выполнено первое разрешенное действие, обработка прекратится.

Привязку ввода можно получить из компонента с помощью метода `getInputMap()` следующим образом:

```
InputMap imap = panel.getInputMap(JComponent.WHEN_FOCUSED);
```

Условие привязки ввода `WHEN_FOCUSED` означает, что оно проверяется, если компонент обладает фокусом ввода. В данном случае это условие не проверяется, поскольку фокусом ввода владеет одна кнопка, а не панель в целом. Каждое из оставшихся двух условий привязки ввода также позволяет очень легко изменить цвет фона в ответ на нажатия клавиш. В рассматриваемом здесь примере программы проверяется условие привязки ввода `WHEN_ANCESTOR_OF_FOCUSED_COMPONENT`.

Класс `InputMap` не связывает напрямую объекты типа `KeyStroke` с объектами класса `ColorAction`, реализующего интерфейс `Action`. Вместо этого он выполняет первую привязку к произвольным объектам, а вторую привязку, реализованную в классе `ActionMap`, объектов — к действиям. Благодаря этому упрощается выполнение одних и тех же действий при нажатии клавиш, зарегистрированных по разным условиям привязки ввода.

Таким образом, у каждого компонента имеются три привязки ввода и одна привязка действия. Чтобы связать их вместе, каждому действию нужно присвоить соответствующее имя. Ниже показано, каким образом комбинация клавиш связывается с нужным действием.

```
imap.put(KeyStroke.getKeyStroke("ctrl Y"), "panel.yellow");  
ActionMap amap = panel.getActionMap();  
amap.put("panel.yellow", yellowAction);
```

Если требуется задать отсутствие действия, то обычно указывается символьная строка `"none"` (отсутствует). Это позволяет легко запретить реагирование на нажатие определенной комбинации клавиш, как показано ниже.

```
imap.put(KeyStroke.getKeyStroke("ctrl C"), "none");
```



ВНИМАНИЕ! В документации на комплект JDK предполагается, что названия клавиш и соответствующего действия совпадают. Такое решение вряд ли можно считать оптимальным. Название действия отображается на кнопке и в пункте меню, но оно может изменяться в процессе разработки. Это, в частности, неизбежно при интернационализации пользовательского интерфейса на разных языках. Поэтому действиям рекомендуется присваивать имена независимо от названий, отображаемых на экране.

Итак, чтобы одно и то же действие выполнялось в ответ на щелчок на экранной кнопке, выбор пункта меню или нажатие клавиши, следует предпринять описанные ниже шаги.

1. Реализовать класс, расширяющий класс `AbstractAction`. Один класс можно будет использовать для программирования разных взаимосвязанных действий.
2. Создать объект класса действия.
3. Сконструировать экранную кнопку или пункт меню из объекта действия. Конструктор прочтет текст метки и пиктограмму из объекта действия.
4. Для действий, которые выполняются в ответ на нажатие клавиш, нужно предпринять дополнительные шаги.
 - Сначала обнаружить в окне компонент верхнего уровня, например, панель, содержащую все остальные компоненты.
 - Затем проверить условие привязки ввода `WHEN_ANCESTOR_OF_FOCUSED_COMPONENT` компонента верхнего уровня.

- Создать объект типа `KeyStroke` для нужного нажатия клавиш.
- Создать объект, соответствующий нажатию клавиш, например, символьную строку, описывающую нужное действие.
- Добавить пару (нажатие клавиш, ответное действие) к привязке ввода.
- И, наконец, получить привязку действия для компонента верхнего уровня, а затем добавить пару (ответное действие, объект действия) к привязке действия.

`javax.swing.Action` 1.2

- `boolean isEnabled()`
Получают или устанавливают свойство `enabled` объекта данного действия.
- `void setEnabled(boolean b)`
Получают или устанавливают свойство `enabled` объекта данного действия.
- `void putValue(String key, Object value)`
Размещает пару "имя-значение" в объекте действия.
- `Object getValue(String key)`
Возвращает значение из сохраненной пары "имя-значение".

`javax.swing.KeyStroke` 1.2

- `static KeyStroke getKeyStroke(String description)`
Конструирует объект типа `KeyStroke` из удобочитаемого описания (последовательности символьных строк, разделяемых пробелами). Описание начинается с нулевого или большего количества модификаторов (`shift`, `control`, `ctrl`, `meta`, `alt`, `altGraf`) и оканчивается строкой со словом `typed` и последующим символом (например, "`typed a`") или необязательным спецификатором события (по умолчанию — `pressed` или `released`) и последующим кодом клавиши. Код клавиши, снабженный префиксом `VK_`, должен соответствовать константе `KeyEvent`; например, код клавиши `<INSERT>` соответствует константе `KeyEvent.VK_INSERT`.

`javax.swing.JComponent` 1.2

- `ActionMap getActionMap()` 1.3
Возвращает привязку действия, связывающую назначенные клавиши действий, которые могут быть произвольными объектами, с объектами класса, реализующего интерфейс `Action`.
- `InputMap getInputMap(int flag)` 1.3
Получает привязку ввода, связывающую нажатия клавиш с клавишами действий. В качестве параметра `flag` указывается одно из условий привязки, перечисленных в табл. 10.2.

10.4.6. События от мыши

Чтобы предоставить пользователю возможность щелкнуть на экранной кнопке или выбрать пункт меню, совсем не обязательно обрабатывать явным образом события от мыши. Операции с мышью автоматически обрабатываются компонентами

пользовательского интерфейса. Но если пользователь должен иметь возможность рисовать мышью, то придется отслеживать ее перемещения, щелчки и события, наступающие при перетаскивании объектов на экране.

В этом разделе будет рассмотрен пример простого графического редактора, позволяющего создавать, перемещать и стирать квадраты на холсте (рис. 10.16).

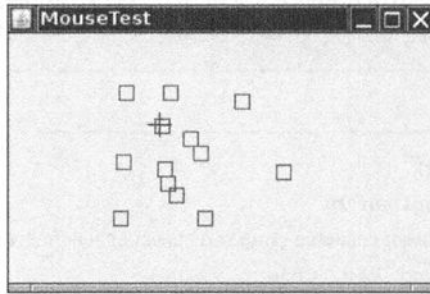


Рис. 10.16. Программа, демонстрирующая обработку событий от мыши

Как только пользователь щелкнет кнопкой мыши, вызываются следующие три метода объекта приемника событий: `mousePressed()`, если кнопка мыши нажата; `mouseReleased()`, если кнопка мыши отпущена; а также `mouseClicked()`, если произведен щелчок. Если же требуется отследить только сам щелчок, применять первые два метода совсем не обязательно. Вызывая методы `getX()` и `getY()` для объекта типа `MouseEvent`, передаваемого в качестве параметра, можно определить координаты положения курсора мыши в момент щелчка. А если требуется отличить обычный щелчок от двойного и тройного (!), то вызывается метод `getClickCount()`.

В программе, рассматриваемой здесь в качестве примера, применяются методы `mousePressed()` и `mouseClicked()`. Если щелкнуть кнопкой мыши на пикселе, не принадлежащем ни одному из нарисованных квадратов, на экране появится новый квадрат. Эта процедура реализована в методе `mousePressed()`, поэтому реакция на щелчок кнопкой мыши произойдет немедленно, не дожидаясь отпускания кнопки мыши. Двойной щелчок в каком-нибудь квадрате приведет к его стиранию. Эта процедура реализована в методе `mouseClicked()`, поскольку щелчки кнопкой мыши нужно подсчитывать. Ниже приведен исходный код обоих методов.

```
public void mousePressed(MouseEvent event)
{
    current = find(event.getPoint());
    if (current == null) // за пределами квадрата
        add(event.getPoint());
}















public void mouseClicked(MouseEvent event)
{
    current = find(event.getPoint());
    if (current != null && event.getClickCount() >= 2)
        remove(current);
}
```

При перемещении мыши по окну последнее получает постоянный поток событий, связанных с движением мыши. Для их отслеживания служат отдельные интерфейсы

MouseListener и MouseMotionListener. Это сделано из соображений эффективности, чтобы приемник событий от щелчков кнопкой мыши игнорировал события, наступающие при перемещении мыши.

Рассматриваемая здесь прикладная программа отслеживает события от перемещений мыши, чтобы изменить вид курсора (на крестообразный), как только он выйдет за пределы квадрата. Эта процедура реализована в методе `getPredefinedCursor()` из класса `Cursor`. В табл. 10.3 перечислены константы, используемые в этом методе, а также приведены виды курсоров для Windows.

Таблица 10.3. Виды курсоров и соответствующие константы

Пиктограмма	Константа	Пиктограмма	Константа
	DEFAULT_CURSOR		NE_RESIZE_CURSOR
	CROSSHAIR_CURSOR		E_RESIZE_CURSOR
	HAND_CURSOR		SE_RESIZE_CURSOR
	MOVE_CURSOR		S_RESIZE_CURSOR
	TEXT_CURSOR		SW_RESIZE_CURSOR
	WAIT_CURSOR		W_RESIZE_CURSOR
	N_RESIZE_CURSOR		NW_RESIZE_CURSOR

Ниже приведен исходный код метода `mouseMoved()`, объявленного в интерфейсе `MouseMotionListener` и реализованного в рассматриваемой здесь программе для отслеживания перемещений мыши и установки соответствующего курсора.

```
public void mouseMoved(MouseEvent event)
{
    if (find(event.getPoint()) == null)
        setCursor(Cursor.getDefaultCursor());
    else
        setCursor(Cursor.getPredefinedCursor(
            Cursor.CROSSHAIR_CURSOR));
}
```

Если перемещение мыши осуществляется при ее нажатой кнопке, вместо метода `mouseClicked()` вызывается метод `mouseDragged()`. В данном примере квадраты можно перетаскивать по экрану. Квадрат перемещается таким образом, чтобы его центр располагался в той точке, где находится указатель мыши. Содержимое холста перерисовывается, чтобы отобразить новое положение указателя мыши. Ниже приведен исходный код метода `mouseClicked()`.

```
public void mouseDragged(MouseEvent event)
{
    if (current != null)
    {
        int x = event.getX();
        int y = event.getY();
        current.setFrame(x - SIDELENGTH / 2,
            y - SIDELENGTH / 2, SIDELENGTH, SIDELENGTH);
    }
}
```



```

        repaint();
    }
}

```



НА ЗАМЕТКУ! Метод `mouseMoved()` вызывается только в том случае, если указатель мыши находится в пределах компонента. Но метод `mouseDragged()` вызывается даже тогда, когда указатель мыши находится за пределами компонента.

Имеются еще два метода, обрабатывающих события от мыши: `mouseEntered()` и `mouseExited()`. Они вызываются в тех случаях, когда указатель мыши входит в пределы компонента и выходит из его пределов.

И, наконец, поясним, каким образом отслеживаются и обрабатываются события от мыши. В ответ на щелчок кнопкой мыши вызывается метод `mouseClicked()`, входящий в состав интерфейса `MouseListener`. Во многих приложениях отслеживаются только щелчки кнопкой мыши, а перемещения мыши происходят слишком часто, поэтому события, связанные с перемещением мыши и перетаскиванием объектов, определяются в отдельном интерфейсе `MouseMotionListener`.

В рассматриваемой здесь программе отслеживаются и обрабатываются оба упомянутых выше вида событий от мыши. Для этого в ней определены два внутренних класса: `MouseHandler` и `MouseMotionHandler`. Класс `MouseHandler` расширяет класс `MouseAdapter`, поскольку в нем определяются только два из пяти методов интерфейса `MouseListener`. А класс `MouseMotionHandler` реализует интерфейс `MouseMotionListener` и определяет оба его метода. Весь исходный код данной программы приведен в листинге 10.6.

Листинг 10.6. Исходный код из файла `mouse/MouseComponent.java`

```

1  package mouse;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.awt.geom.*;
6  import java.util.*;
7  import javax.swing.*;
8
9  /**
10   * Компонент для операций с мышью по
11   * добавлению и удалению квадратов
12   */
13  public class MouseComponent extends JComponent
14  {
15      private static final int DEFAULT_WIDTH = 300;
16      private static final int DEFAULT_HEIGHT = 200;
17
18      private static final int SIDELENGTH = 10;
19      private ArrayList<Rectangle2D> squares;
20      private Rectangle2D current; // квадрат, содержащий
21                                   // курсор мыши
22
23      public MouseComponent()
24      {
25          squares = new ArrayList<>();

```

```
26     current = null;
27
28     addMouseListener(new MouseHandler());
29     addMouseMotionListener(new MouseMotionHandler());
30 }
31
32 public Dimension getPreferredSize()
33 {
34     return new Dimension(DEFAULT_WIDTH,
35                           DEFAULT_HEIGHT);
36 }
37
38 public void paintComponent(Graphics g)
39 {
40     var = (Graphics2D) g;
41
42     // нарисовать все квадраты
43     for (Rectangle2D r : squares)
44         g2.draw(r);
45 }
46
47 /**
48  * Обнаруживает первый квадрат, содержащий
49  * заданную точку
50  * @param p а Точка
51  * @return the Первый квадрат, содержащий точку p
52  */
53 public Rectangle2D find(Point2D p)
54 {
55     for (Rectangle2D r : squares)
56     {
57         if (r.contains(p)) return r;
58     }
59     return null;
60 }
61
62 /**
63  * Вводит квадрат в коллекцию
64  * @param p Центр квадрата
65  */
66 public void add(Point2D p)
67 {
68     double x = p.getX();
69     double y = p.getY();
70
71     current = new Rectangle2D.Double(
72         x - SIDELENGTH / 2,
73         y - SIDELENGTH / 2,
74         SIDELENGTH, SIDELENGTH);
75     squares.add(current);
76     repaint();
77 }
78
79 /**
80  * Удаляет квадрат из коллекции
81  * @param s Удаляемый квадрат
```

```
82     */
83     public void remove(Rectangle2D s)
84     {
85         if (s == null) return;
86         if (s == current) current = null;
87         squares.remove(s);
88         repaint();
89     }
90
91     private class MouseHandler extends MouseAdapter
92     {
93         public void mousePressed(MouseEvent event)
94         {
95             // добавить новый квадрат, если курсор
96             // находится за пределами квадрата
97             current = find(event.getPoint());
98             if (current == null) add(event.getPoint());
99         }
100
101         public void mouseClicked(MouseEvent event)
102         {
103             // удалить текущий квадрат, если на нем
104             // произведен двойной щелчок
105             current = find(event.getPoint());
106             if (current != null
107                 && event.getClickCount() >= 2)
108                 remove(current);
109         }
110     }
111
112     private class MouseMotionHandler
113         implements MouseMotionListener
114     {
115         public void mouseMoved(MouseEvent event)
116         {
117             // задать курсор в виде перекрестья,
118             // если он находится внутри квадрата
119
120             if (find(event.getPoint()) == null)
121                 setCursor(Cursor.getDefaultCursor());
122             else setCursor(Cursor.getPredefinedCursor(
123                 Cursor.CROSSHAIR_CURSOR));
124         }
125
126         public void mouseDragged(MouseEvent event)
127         {
128             if (current != null)
129             {
130                 int x = event.getX();
131                 int y = event.getY();
132
133                 // перетащить текущий квадрат,
134                 // чтобы отцентровать его в точке
135                 // с координатами (x,y)
136                 current.setFrame(x - SIDELENGTH / 2,
137                                 y - SIDELENGTH / 2,
```

```

138             SIDELENGTH, SIDELENGTH);
139         repaint();
140     }
141 }
142 }
143 }

```

java.awt.event.MouseEvent 1.1

- **int getX()**
- **int getY()**
- **Point getPoint()**

Возвращают горизонтальную (**x**) и вертикальную (**y**) координаты или точку, в которой наступило событие. Отсчет производится от левого верхнего угла компонента, являющегося источником события.

- **int getClickCount()**

Возвращает количество последовательных щелчков кнопкой мыши, связанных с данным событием. (Промежуток времени, в пределах которого подсчитываются щелчки, зависит от операционной системы.)

java.awt.Component 1.0

- **public void setCursor(Cursor cursor) 1.1**

Изменяет внешний вид курсора.

10.4.7. Иерархия событий в библиотеке AWT

У класса `EventObject` имеется подкласс `AWTEvent`, являющийся родительским для всех классов событий из библиотеки AWT. На рис. 10.17 схематически показана иерархия наследования классов событий в библиотеке AWT. Некоторые компоненты из библиотеки Swing формируют объекты других видов событий и непосредственно расширяют класс `EventObject`.

Объекты событий инкапсулируют данные, которые источник событий передает приемникам. По мере необходимости объекты событий, переданные объекту приемника событий, могут быть проанализированы с помощью методов `getSource()` и `getActionCommand()`.

Некоторые классы событий из библиотеки AWT не имеют никакой практической ценности для программирующих на Java. Например, объекты типа `PaintEvent` вводятся из библиотеки AWT в очередь событий, но эти объекты не доставляются приемнику событий. Программисты должны сами переопределить метод `paintComponent()`, чтобы управлять перерисовкой. В библиотеке AWT иницируется также ряд событий, интересующих лишь системных программистов. Такие события могут служить для поддержки иероглифических языков, автоматизации тестирования роботов и решения прочих задач.

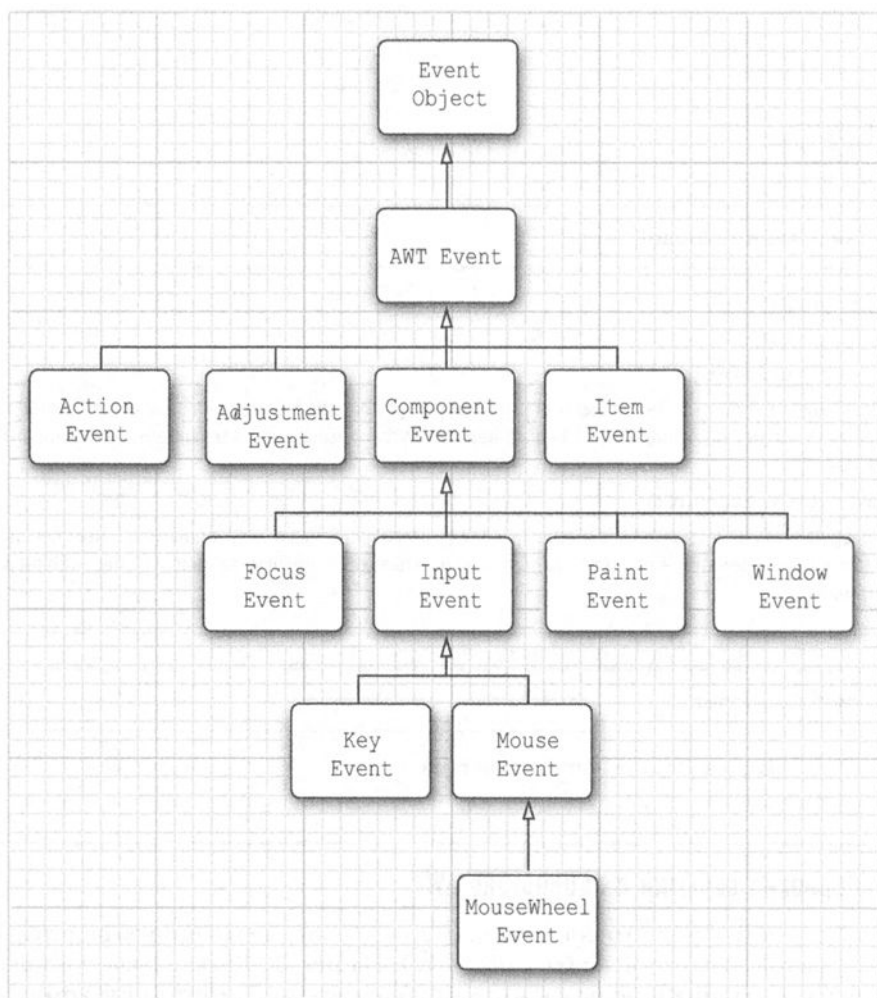


Рис. 10.17. Схематическое представление иерархии наследования классов событий в библиотеке AWT

События делятся в библиотеке AWT на *низкоуровневые* и *семантические*. Семантические события описывают действия пользователя, например щелчок на экранной кнопке, поэтому событие типа `ActionEvent` является семантическим. А низкоуровневые события обеспечивают возможность подобных действий. Если пользователь щелкнул на экранной кнопке, значит, он нажал кнопку мыши, возможно, переместил курсор по экрану и отпустил кнопку мыши (причем курсор мыши должен находиться в пределах выбираемой кнопки). Семантические события могут быть также инициированы нажатием клавиш, например, для перемещения по кнопкам на панели с помощью клавиши `<Tab>`. Аналогично перемещение бегунка по полосе прокрутки относится к семантическим событиям, тогда как перетаскивание объекта мышью — к низкоуровневым.

Из классов, входящих в пакет `java.awt.event` и описывающих семантические события, чаще всего применяются следующие.

- Класс `ActionEvent` (щелчок на кнопке, выбор пункта меню, выбор элемента из списка, нажатие клавиши `<Enter>` при вводе текста в поле).
- Класс `AdjustmentEvent` (перемещение бегунка на полосе прокрутки).
- Класс `ItemEvent` (выбор одной из кнопок-переключателей, установка одного из флажков или выбор элемента из списка).

Из классов низкоуровневых событий чаще всего применяются следующие.

- Класс `KeyEvent` (нажатие или отпускание клавиши).
- Класс `MouseEvent` (нажатие и отпускание кнопки мыши, перемещение курсора мыши, перетаскивание курсора, т.е. его перемещение при нажатой кнопке мыши).
- Класс `MouseEvent` (вращение колесика мыши).
- Класс `FocusEvent` (получение или потеря фокуса ввода).
- Класс `WindowEvent` (изменение состояния окна).

Эти события отслеживаются и обрабатываются с помощью компонентов библиотеки AWT, перечисленных в табл. 10.4.

Таблица 10.4. Компоненты из библиотеки AWT для обработки событий

Интерфейс	Методы	Параметр/методы доступа	Источник событий
ActionListener	actionPerformed	ActionEvent	AbstractButton
		getActionCommand	JComboBox
		getModifiers	TextField
			Timer
AdjustmentListener	adjustmentValueChanged	AdjustmentEvent	JScrollbar
		getAdjustable	
		getAdjustmentType	
		getValue	
ItemListener	itemStateChanged	ItemEvent	AbstractButton
		getItem	JComboBox
		getItemSelectable	
		getStateChange	
FocusListener	focusGained focusLost	FocusEvent	Component
		isTemporary	
KeyListener	keyPressed keyReleased keyTyped	KeyEvent	Component
		getKeyChar	
		getKeyCode	
		getKeyModifiersText	
		getKeyText	
		isActionKey	

Окончание табл. 10.4

Интерфейс	Методы	Параметр/методы доступа	Источник событий
MouseListener	mousePressed	MouseEvent	Component
	mouseReleased	getClickCount	
	mouseEntered	getX	
	mouseExited	getY	
	mouseClicked	getPoint	
MouseMotionListener	mouseDragged mouseMoved	translatePoint	Component
		MouseEvent	
MouseWheelListener	mouseWheelMoved	MouseWheelEvent	Component
		getWheelRotation	
		getScrollAmount	
WindowListener	windowClosing	WindowEvent	Window
	windowOpened		
	windowIconified		
	windowDeiconified		
	windowClosed		
	windowActivated		
	windowDeactivated		
WindowFocusListener	windowGainedFocus	WindowEvent	Window
	windowLostFocus	getOppositeWindow	
WindowStateListener	windowStateChanged	WindowEvent	Window
		getOldState	
		getNewState	

10.5. Прикладной интерфейс Preferences API

В завершение этой главы рассмотрим прикладной интерфейс Preferences API из пакета `java.util.preferences`. В настольной программе нередко требуется сохранять пользовательские глобальные параметры настройки, в том числе последний файл, с которым работал пользователь, местоположение последнего рабочего окна и т.д.

Как пояснялось в главе 9, класс `Properties` позволяет легко загружать и сохранять информацию о конфигурации прикладной программы. Но файлам свойств присущи следующие недостатки.

- В некоторых операционных системах вообще не поддерживаются начальные каталоги, что затрудняет выбор единого места для хранения конфигурационных файлов.
- Стандартные условные обозначения имен конфигурационных файлов отсутствуют, что может привести к конфликтам имен, если пользователь установит несколько приложений.

В некоторых операционных системах конфигурационные данные хранятся в центральном хранилище. Наиболее известным примером такого хранилища является системный реестр Windows. Подобное центральное хранилище поддерживается в классе Preferences независимо от операционной системы. Так, в Windows таким хранилищем конфигурационных данных для класса Preferences служит системный реестр, а в Linux — локальный системный файл. Разумеется, реализация центрального хранилища абсолютно прозрачна для разработчиков, пользующихся классом Preferences.

Хранилище, организуемое в классе Preferences, имеет древовидную структуру, в узлах которой содержатся имена путей, например `/com/mycompany/myapp`. Как и при составлении пакетов, во избежание конфликтов имена рекомендуется формировать на основе доменных имен, записывая их в обратном порядке. При разработке прикладного интерфейса API для сохранения глобальных параметров настройки предполагалось, что пути к узлам должны совпадать с именами пакетов, используемых в прикладной программе.

Каждый узел в хранилище содержит отдельную таблицу пар “ключ–значение”, которую можно использовать для записи чисел, символьных строк или байтовых массивов. Возможность хранить сериализованные объекты не предусмотрена. Разработчики прикладного интерфейса API посчитали, что формат сериализации слишком хрупок, и обеспечить долговременное хранение данных в нем совсем не просто. Разумеется, если вы не согласны с этим, можете хранить сериализованные объекты в виде байтовых массивов.

Для дополнительного удобства предусмотрено несколько параллельных деревьев. В каждой программе используется одно дерево. Кроме того, существует дополнительное дерево, называемое системным и предназначенное для хранения настроек, общих для всех пользователей. Для доступа к соответствующему пользовательскому дереву в классе Preferences используется понятие “текущего пользователя”.

Поиск требуемого узла в дереве начинается с пользовательского или системного корня:

```
Preferences root = Preferences.userRoot();
```

или

```
Preferences root = Preferences.systemRoot();
```

И тогда для доступа к узлу достаточно указать соответствующий путь:

```
Preferences node = root.node("/com/mycompany/myapp");
```

Для быстрого и удобного доступа к узлу дерева путь к нему приравняется к имени пакета его класса. Для этого достаточно взять объект данного класса и сделать вызов одним из двух приведенных ниже способов. Как правило, ссылка `obj` означает ссылку `this`:

```
Preferences node = Preferences.userNodeForPackage(  
    obj.getClass());
```

или

```
Preferences node = Preferences.systemNodeForPackage(  
    obj.getClass());
```

Получив доступ к узлу, можно обратиться к таблице с парами “ключ–значение”, используя перечисленные ниже методы.


```
String get(String key, String defval)
int getInt(String key, int defval)
long getLong(String key, long defval)
float getFloat(String key, float defval)
double getDouble(String key, double defval)
boolean getBoolean(String key, boolean defval)
byte[] getByteArray(String key, byte[] defval)
```

Но если хранилище недоступно, то при чтении конфигурационных данных придется указывать значения, устанавливаемые по умолчанию. Эти значения требуются по нескольким причинам. Во-первых, данные могут быть неполными из-за того, что пользователи не всегда указывают все параметры. Во-вторых, на некоторых платформах хранилища могут не создаваться из-за нехватки ресурсов. И в-третьих, мобильные устройства могут быть временно отсоединены от хранилища.

А с другой стороны, данные в хранилище можно записать, используя методы

```
put(String key, String value)
putInt(String key, int value)
```

и так далее, в зависимости от типа данных. Перечислить все ключи, сохраненные в узле, можно с помощью метода `String[] keys()`. В настоящее время нет способа, позволяющего определить тип значения, связанного с конкретным ключом.



НА ЗАМЕТКУ! Имена узлов и ключей не должны превышать 80 символов, а строки — 8192 символов.

Центральным хранилищам, например системному реестру в Windows, традиционно присущи два недостатка.

- Во-первых, в них постепенно накапливается устаревшая информация.
- Во-вторых, данные о конфигурации системы становятся все более запутанными, что затрудняет перенос глобальных параметров настройки на новую платформу.

Класс `Preferences` позволяет устранить второй недостаток. Данные можно экспортировать в поддерево, а в некоторых случаях — в отдельный узел, вызвав один из перечисленных ниже методов.

```
void exportSubtree(OutputStream out)
void exportNode(OutputStream out)
```

Данные хранятся в формате XML. Их можно импортировать в другое хранилище, вызвав следующий метод:

```
void importPreferences(InputStream in)
```

В качестве примера ниже приведены конфигурационные данные, размеченные в формате XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE preferences SYSTEM "http://java.sun.com/dtd/preferences.dtd">
<preferences EXTERNAL_XML_VERSION="1.0">
  <root type="user">
    <map/>
    <node name="com">
      <map/>
      <node name="horstmann">
        <map/>
        <node name="corejava">
```

```

    <map>
      <entry key="left" value="11"/>
      <entry key="top" value="9"/>
      <entry key="width" value="453"/>
      <entry key="height" value="365"/>
      <entry key="title" value="Hello, World!"/>
    </map>
  </node>
</node>
</node>
</root>
</preferences>

```

Если в прикладной программе используются глобальные параметры настройки, ее пользователям можно предоставить возможность экспортировать и импортировать эти глобальные параметры, чтобы упростить перенос программы с одного компьютера на другой. Такой подход демонстрируется в примере программы, исходный код которой приведен в листинге 10.7. В этой программе сохраняются координаты и размеры главного окна. Попробуйте изменить размеры окна, выйти из программы и запустить ее снова на выполнение. Размеры окна должны совпадать с теми размерами, которые были заданы перед выходом из программы. Импортируйте свои глобальные параметры настройки, чтобы вернуть окно на прежнее место.

Листинг 10.7. Исходный код из файла `preferences/ImageViewer.java`

```

1  package preferences;
2
3  import java.awt.EventQueue;
4  import java.awt.event.*;
5  import java.io.*;
6  import java.util.prefs.*;
7  import javax.swing.*;
8
9  /**
10   * В этой программе проверяются глобальные параметры
11   * настройки. В ней запоминаются положение и размеры
12   * фрейма, а также последний выбранный файл.
13   * @version 1.10 2018-04-10
14   * @author Cay Horstmann
15   */
16  public class ImageViewer
17  {
18      public static void main(String[] args)
19      {
20          EventQueue.invokeLater(() -> {
21              var frame = new ImageViewerFrame();
22              frame.setTitle("ImageViewer");
23              frame.setDefaultCloseOperation(
24                  JFrame.EXIT_ON_CLOSE);
25              frame.setVisible(true);
26          });
27      }
28  }
29
30  /**
31   * Средство просмотра изображений, восстанавливающее

```

```
32  * положение, размеры и просматриваемое изображение из
33  * пользовательских глобальных параметров настройки и
34  * обновляющее эти параметры по завершении работы
35  */
36  class ImageViewerFrame extends JFrame
37  {
38      private static final int DEFAULT_WIDTH = 300;
39      private static final int DEFAULT_HEIGHT = 200;
40      private String image;
41
42      public ImageViewerFrame()
43      {
44          Preferences root = Preferences.userRoot();
45          Preferences node = root.node(
46              "/com/horstmann/corejava/ImageViewer");
47          // получить положение, размеры и заглавие из свойств
48          int left = node.getInt("left", 0);
49          int top = node.getInt("top", 0);
50          int width = node.getInt("width", DEFAULT_WIDTH);
51          int height = node.getInt("height", DEFAULT_HEIGHT);
52          setBounds(left, top, width, height);
53          image = node.get("image", null);
54          var label = new JLabel();
55          if (image != null)
56              label.setIcon(new ImageIcon(image));
57
58          addWindowListener(new WindowAdapter()
59          {
60              public void windowClosing(WindowEvent event)
61              {
62                  node.putInt("left", getX());
63                  node.putInt("top", getY());
64                  node.putInt("width", getWidth());
65                  node.putInt("height", getHeight());
66                  node.put("image", image);
67              }
68          });
69
70          // воспользоваться меткой для воспроизведения
71          // изображений
72          add(label);
73
74          // установить селектор файлов
75          var chooser = new JFileChooser();
76          chooser.setCurrentDirectory(new File("."));
77
78          // установить строку меню
79          var menuBar = new JMenuBar();
80          setJMenuBar(menuBar);
81
82          var menu = new JMenu("File");
83          menuBar.add(menu);
84
85          var openItem = new JMenuItem("Open");
86          menu.add(openItem);
87          openItem.addActionListener(event -> {
88
```

```

89      // отобразить диалоговое окно селектора файлов
90      int result = chooser.showOpenDialog(null);
91
92      // если файл выбран, установить его в виде
93      // пиктограммы метки
94      if (result == JFileChooser.APPROVE_OPTION)
95      {
96          image = chooser.getSelectedFile().getPath();
97          label.setIcon(new ImageIcon(image));
98      }
99  });
100
101  var exitItem = new JMenuItem("Exit");
102  menu.add(exitItem);
103  exitItem.addActionListener(event -> System.exit(0));
104  }
105  }

```

java.util.prefs.Preferences 1.4

- **Preferences userRoot()**

Возвращает корневой узел из дерева глобальных параметров настройки для пользователя вызывающей программы.

- **Preferences systemRoot()**

Возвращает системный корневой узел из дерева глобальных параметров настройки.

- **Preferences node(String path)**

Возвращает узел, доступный из текущего узла по заданному пути. Если в качестве параметра **path** указан абсолютный путь, который обычно начинается со знака косой черты [/], то узел доступен из корня дерева глобальных параметров настройки. Если же узел отсутствует по заданному пути, он создается.

- **Preferences userNodeForPackage(Class c1)**

- **Preferences systemNodeForPackage(Class c1)**

Возвращают узел из дерева текущего пользователя или системного дерева, абсолютный путь к которому соответствует имени пакета, содержащего заданный класс **c1**.

- **String[] keys()**

- Возвращает все ключи, принадлежащие данному узлу.

- **String get(String key, String defval)**

- **int getInt(String key, int defval)**

- **long getLong(String key, long defval)**

- **float getFloat(String key, float defval)**

- **double getDouble(String key, double defval)**

- **boolean getBoolean(String key, boolean defval)**

- **byte[] getByteArray(String key, byte[] defval)**

Возвращают значение, связанное с заданным ключом. Если значение отсутствует в хранилище глобальных параметров настройки, имеет неверный тип или же само хранилище недоступно, возвращается значение, предусмотренное по умолчанию.

java.util.prefs.Preferences 1.4 (окончание)

- **void put(String key, String value)**
- **void putInt(String key, int value)**
- **void putLong(String key, long value)**
- **void putFloat(String key, float value)**
- **void putDouble(String key, double value)**
- **void putBoolean(String key, boolean value)**
- **void putByteArray(String key, byte[] value)**

Сохраняют пару “ключ–значение” в заданном узле дерева.

- **void exportSubtree(OutputStream out)**

Выводит в указанный поток глобальные параметры настройки, хранящиеся в заданном узле и производных от него узлах.

- **void exportNode(OutputStream out)**

Направляет в указанный поток вывода глобальные параметры настройки, хранящиеся в заданном узле, игнорируя производные от него узлы.

- **void importPreferences(InputStream in)**

Импортирует параметры глобальных настроек из указанного потока ввода.

На этом завершается введение в программирование GUI на Java. В следующей главе будет показано, как обращаться с наиболее употребительными компонентами библиотеки Swing.

Компоненты пользовательского интерфейса в Swing

В этой главе...

- ▶ Библиотека Swing и проектный шаблон “модель–представление–контроллер”
- ▶ Введение в компоновку пользовательского интерфейса
- ▶ Ввод текста
- ▶ Компоненты для выбора разных вариантов
- ▶ Меню
- ▶ Расширенные средства компоновки
- ▶ Диалоговые окна

Предыдущая глава была в основном посвящена рассмотрению модели обработки событий в Java. Проработав ее, вы приобрели знания и навыки, без которых невозможно создать приложение с графическим интерфейсом. В этой главе будут рассмотрены самые важные инструментальные средства, требующиеся для создания полнофункциональных графических интерфейсов.

Сначала в ней будут вкратце рассмотрены архитектурные принципы, положенные в основу библиотеки Swing. Чтобы эффективно пользоваться современными компонентами пользовательского интерфейса, нужно как следует разбираться в их функционировании. Затем будет показано, как применять обычные компоненты

пользовательского интерфейса из библиотеки Swing, включая текстовые поля, кнопки-переключатели и меню. Далее поясняется, как пользоваться возможностями диспетчеров компоновки в Java, чтобы размещать компоненты в окне независимо от визуального стиля GUI. И в заключение главы будет показано, каким образом диалоговые окна создаются средствами Swing.

В этой главе описываются основные компоненты библиотеки Swing, в том числе текстовые поля, экранные кнопки и полосы прокрутки. Это самые важные и наиболее употребительные компоненты пользовательского интерфейса. А более сложные компоненты Swing будут рассматриваться во втором томе настоящего издания.

11.1. Библиотека Swing и проектный шаблон “модель–представление–контроллер”

Напомним, из чего состоят компоненты GUI, например, экранная кнопка, флажок, текстовое поле или сложное окно управления древовидной структурой элементов. Каждый из этих компонентов обладает следующими характеристиками.

- *Содержимое*, например, состояние кнопки (нажата или отпущена) или текст в поле редактирования.
- *Внешний вид* (цвет, размер и т.д.).
- *Поведение* (реакция на события).

Даже у таких, на первый взгляд, простых компонентов, как экранные кнопки, эти характеристики тесно связаны между собой. Очевидно, что внешний вид экранной кнопки зависит от визуального стиля интерфейса в целом. Экранная кнопка в стиле Metal отличается от кнопки в стиле Windows или Motif. Кроме того, внешний вид зависит от состояния экранной кнопки: в нажатом состоянии кнопка должна выглядеть иначе, чем в отпущенном. Состояние, в свою очередь, зависит от событий. Если пользователь щелкнул на экранной кнопке, она считается нажатой.

Разумеется, когда вы используете экранную кнопку в своих программах, то рассматриваете ее как таковую, не особенно вдаваясь в подробности ее внутреннего устройства и характеристик. В конце концов, технические подробности — удел программиста, реализовавшего эту кнопку. Но программисты, реализующие экранные кнопки и прочие элементы GUI, должны тщательно обдумывать их внутреннее устройство и функционирование, чтобы все эти компоненты правильно работали независимо от выбранного визуального стиля.

Для решения подобных задач разработчики библиотеки Swing обратились к хорошо известному проектному шаблону “модель–представление–контроллер” (Model-View-Controller — MVC), который предписывает разработчикам предоставить три отдельных объекта, представляющие следующие его составляющие.

- *Модель*, где хранится содержимое.
- *Представление*, отображающее содержимое.
- *Контроллер*, обрабатывающий вводимые пользователем данные.

Проектный шаблон “модель–представление–контроллер” точно обозначает взаимодействие этих объектов. Модель хранит содержимое и не реализует пользовательский интерфейс. Содержимое экранной кнопки тривиально — это небольшой абор признаков, означающих, нажата кнопка или отпущена, активизирована или неактивизирована и т.д. Содержимым текстового поля является символьная строка, *не* совпадающая с представлением. Так, если содержимое превышает длину текстового поля, пользователь видит лишь часть отображаемого текста (рис. 11.1).

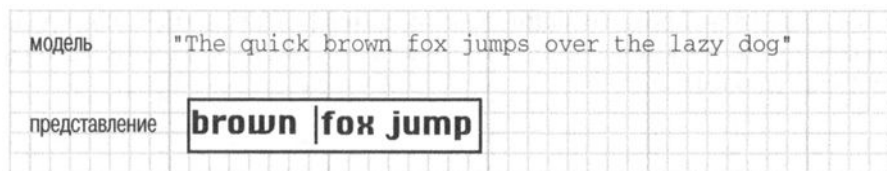


Рис. 11.1. Модель и представление текстового поля

Модель должна реализовывать методы, изменяющие содержимое и раскрывающие его смысл. Например, модель текстового поля имеет методы для ввода символов в строке, их удаления и возврата текста в виде строки. Следует, однако, иметь в виду, что модель совершенно невидима. Отображать данные, хранящиеся в модели, — обязанность представления.



НА ЗАМЕТКУ! Термин *модель*, по-видимому, выбран не совсем удачно, поскольку его часто связывают со способом представления абстрактного понятия. Например, авиаконструкторы и конструкторы автомобилей строят модели для того, чтобы имитировать настоящие самолеты и автомобили. Но эта аналогия не подходит к шаблону “модель–представление–контроллер”. В данном случае модель хранит содержимое, а представление отвечает за ее полное или неполное визуальное отображение. Намного более точной аналогией является натурщица, позирующая художнику. Художник должен смотреть на модель и создавать ее изображение. В зависимости от стиля, в котором работает художник, представление может оказаться классическим портретом, картиной в стиле импрессионизма или совокупностью фигур в стиле кубизма.

Одно из преимуществ шаблона “модель–представление–контроллер” состоит в том, что модель может иметь несколько представлений, каждое из которых отражает отдельный аспект ее содержимого. Например, редактор HTML-разметки документов часто предлагает одновременно *два* представления одних и тех же данных: WYSIWYG (Что видишь на экране, то и получишь при печати) и ряд дескрипторов (рис. 11.2). Когда контроллер обновляет модель, изменяются оба представления. Получив уведомление об изменении, представление обновляется автоматически. Разумеется, для таких простых компонентов пользовательского интерфейса, как кнопки, нет никакой необходимости предусматривать несколько представлений одной и той же модели.

Контроллер обрабатывает события, связанные с поступающей от пользователя информацией, например, щелчки кнопками мыши и нажатия клавиш, а затем решает, преобразовывать ли эти события в изменения модели или представления. Так, если пользователь нажмет клавишу символа при вводе в текстовом поле, контроллер вызовет из модели команду “вставить символ”. Затем модель уведомит представление обновить изображение. Представлению вообще неизвестно, почему

изменился текст. Но если пользователь нажал клавишу управления курсором, то контроллер может отдать представлению команду на прокрутку. Прокрутка не изменяет текст, поэтому модели ничего неизвестно об этом событии. Взаимодействие модели, представления и контроллера схематически показано на рис. 11.3.

Для большинства компонентов Swing классы модели реализуют интерфейсы, имена которых оканчиваются словом *Model*. В частности, для экранных кнопок используется интерфейс *ButtonModel*. Классы, реализующие этот интерфейс, могут определять состояние разнотипных экранных кнопок. Кнопки настолько просты, что для них в библиотеке Swing предусмотрен отдельный класс *DefaultButtonModel*, реализующий данный интерфейс. Понять, какого рода данные поддерживаются в модели кнопки, можно, рассмотрев свойства интерфейса *ButtonModel* (табл. 11.1).

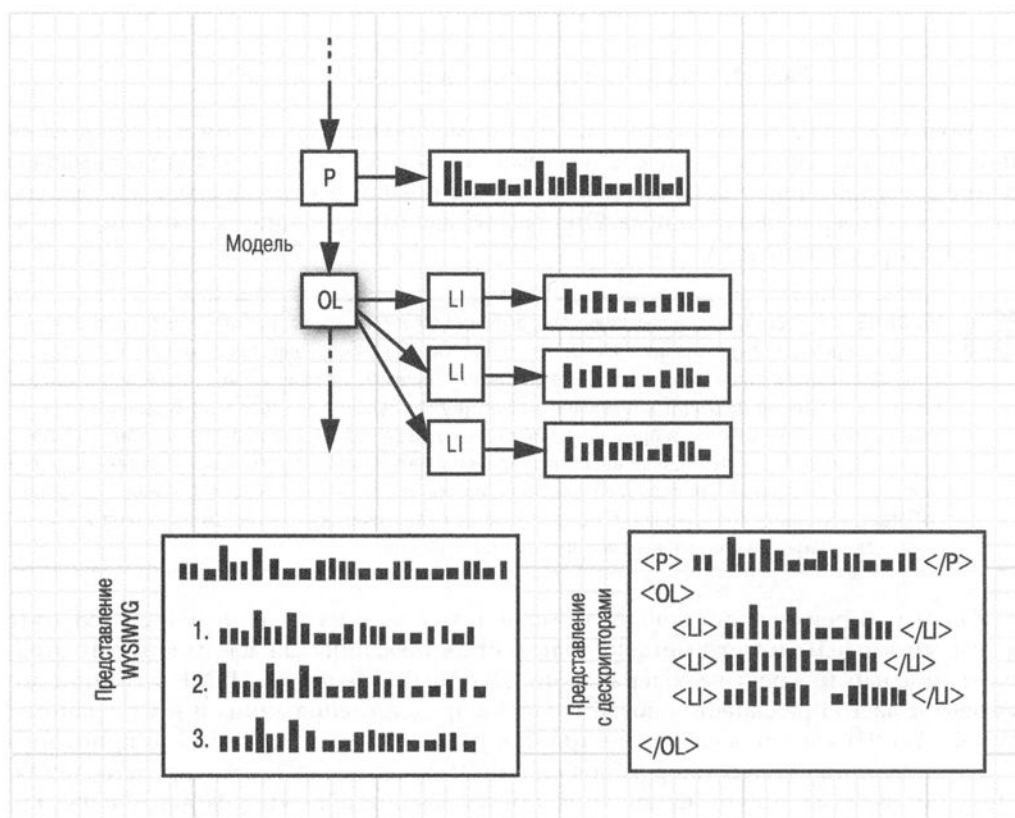


Рис. 11.2. Два разных представления одной и той же модели

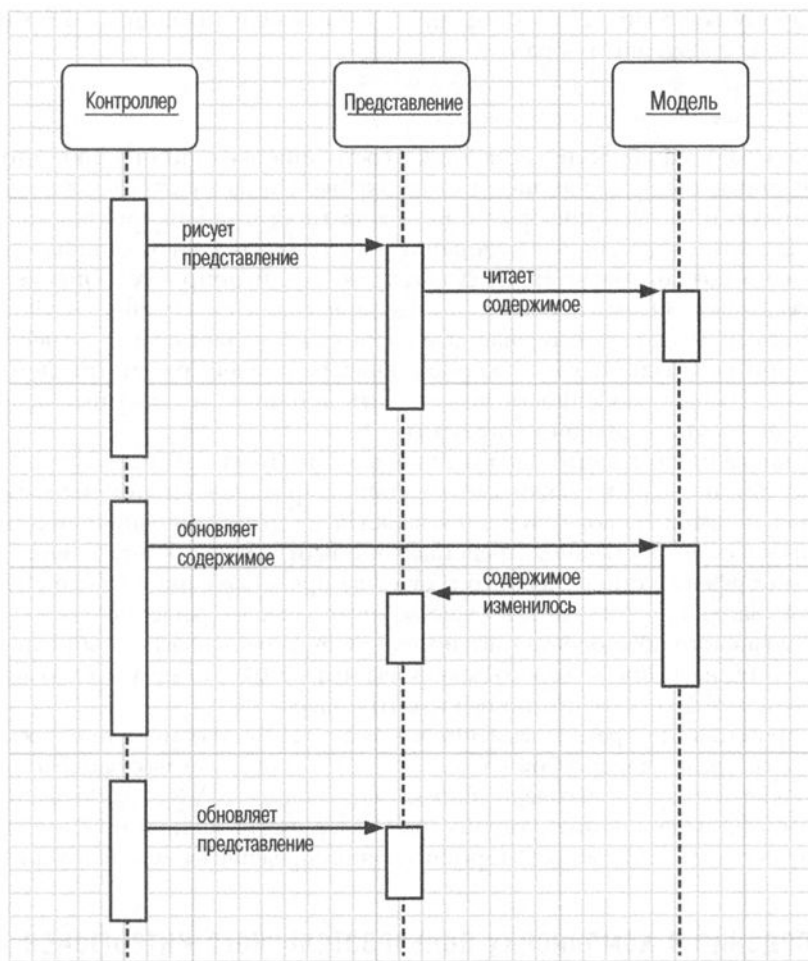


Рис. 11.3. Взаимодействие объектов модели, представления и контроллера

Таблица 11.1. Свойства интерфейса `ButtonModel`

Имя свойства	Значение
actionCommand	Символьная строка команды действия, связанного с экранной кнопкой
mnemonic	Мнемоническое обозначение экранной кнопки
armed	Логическое значение true , если экранная кнопка была нажата, а курсор мыши еще находится на кнопке
enabled	Логическое значение true , если экранная кнопка доступна
pressed	Логическое значение true , если экранная кнопка была нажата, а кнопка мыши еще не отпущена
rollover	Логическое значение true , если курсор мыши находится на экранной кнопке
selected	Логическое значение true , если экранная кнопка включена (используется для флажков и кнопок-переключателей)

В каждом объекте типа `JButton` хранится объект модели кнопки, который можно извлечь оттуда следующим образом:

```
var button = new JButton("Blue");  
ButtonModel model = button.getModel();
```

На практике подробности, касающиеся состояния экранной кнопки, интересуют лишь представление, которое рисует ее на экране. Но из класса `JButton` можно извлечь и другую полезную информацию — в частности, заблокирована ли экранная кнопка. (Для этого класс `JButton` запрашивает модель экранной кнопки.)

Обратимся еще раз к интерфейсу `ButtonModel` и попробуем определить, что в нем *отсутствует*. Оказывается, что в модели не хранится название экранной кнопки и ее пиктограмма. Поэтому анализ модели не позволяет судить о внешнем виде кнопки. (При реализации групп кнопок-переключателей, рассматриваемых далее в этой главе, данная особенность может стать источником серьезных осложнений для разработчика прикладной программы.)

Стоит также отметить, что одна и та же модель (типа `DefaultButtonModel`) используется для поддержки нажимаемых кнопок, флажков кнопок-переключателей и даже для пунктов меню. Разумеется, каждая из этих разновидностей экранных кнопок имеет свое собственное представление и отдельный контроллер. Если реализуется интерфейс в стиле `Metal`, то в качестве представления в классе `JButton` используется класс `BasicButtonUI`, а качестве контроллера — класс `ButtonUIListener`. В общем, у каждого компонента библиотеки `Swing` имеется связанный с ним объект представления, название которого заканчивается на `UI`. Но не у всех компонентов `Swing` имеется свой собственный объект контроллера.

Итак, прочитав это краткое введение в класс `JButton`, вы можете спросить: а что на самом деле представляет собой класс `JButton`? Это просто класс-оболочка, производный от класса `JComponent` и содержащий объект типа `DefaultButtonModel`, некоторые данные, необходимые для отображения (например, метку кнопки и ее пиктограмму), а также объект типа `BasicButtonUI`, реализующий представление экранной кнопки.

11.2. Введение в компоновку пользовательского интерфейса

Прежде чем перейти к обсуждению таких компонентов `Swing`, как текстовые поля и кнопки-переключатели, рассмотрим вкратце, каким образом они размещаются во фрейме. Разумеется, если вы программируете на Java в соответствующей IDE, то в этой среде, скорее всего, предусмотрены средства, автоматизирующие некоторые из задач построения GUI методом перетаскивания. Несмотря на это, вы обязаны ясно представлять, каким образом размещаются компоненты. Ведь даже те GUI, которые автоматически построены с помощью самых совершенных инструментальных средств, обычно нуждаются в ручной доработке.

11.2.1. Диспетчеры компоновки

Вернемся для начала к примеру программы из листинга 10.4, где экранные кнопки служили для изменения цвета фона во фрейме.

Экранные кнопки содержатся в объекте типа `JPanel` и управляются *диспетчером поточной компоновки* — стандартным диспетчером для компоновки панели. На рис. 11.4 показано, что происходит, когда на панели вводятся дополнительные экранные кнопки. Как видите, если кнопки не помещаются в текущем ряду, они переносятся

в новый ряд. Более того, экранные кнопки будут отцентрованы на панели, даже если пользователь изменит размеры фрейма (рис. 11.5).



Рис. 11.4. Панель с шестью экранными кнопками, расположенными подряд диспетчером поточной компоновки



Рис. 11.5. При изменении размеров панели автоматически изменяется расположение кнопок

В целом компоненты размещаются в контейнерах, а диспетчер компоновки определяет порядок расположения и размеры компонентов в контейнере. Классы экранных кнопок, текстовых полей и прочих элементов пользовательского интерфейса расширяют класс `Component`. Компоненты могут размещаться в таких контейнерах, как панели. А поскольку одни контейнеры могут размещаться в других контейнерах, то класс `Container` расширяет класс `Component`. На рис. 11.6 схематически показана иерархия наследования всех этих классов от класса `Component`.

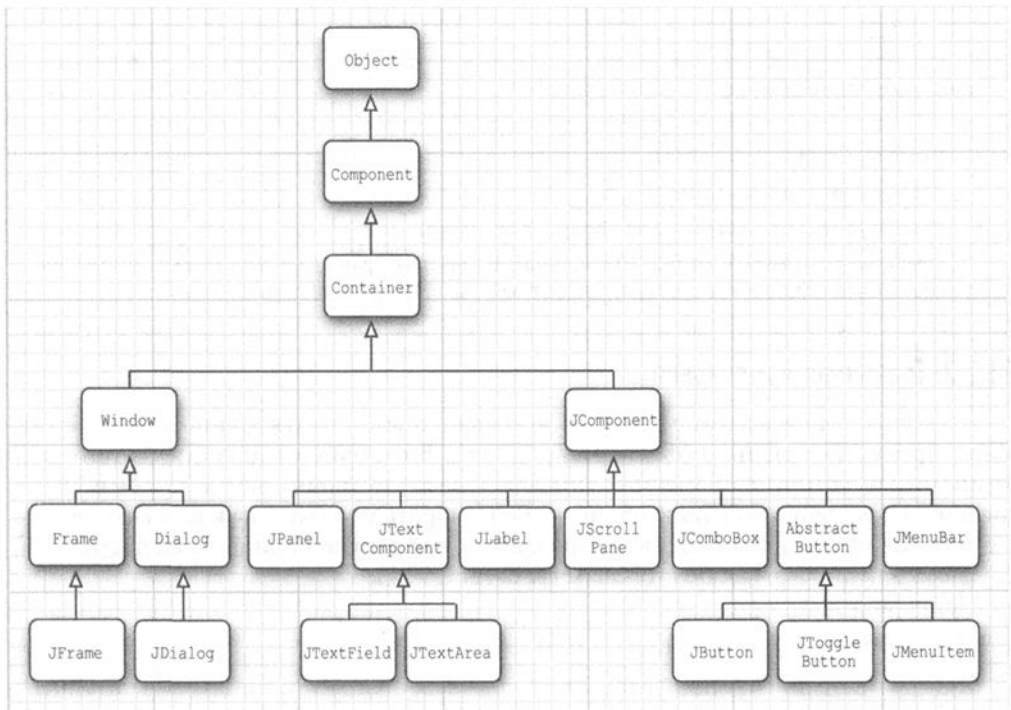


Рис. 11.6. Иерархия наследования от класса `Component`



НА ЗАМЕТКУ! К сожалению, иерархия наследования выглядит не совсем ясной по двум причинам. Во-первых, окна верхнего уровня, например типа `JFrame`, являются подклассами, производными от класса `Container`, а следовательно, и от класса `Component`, но их нельзя разместить в других контейнерах. Более того, класс `JComponent` является производным от класса `Container`, а не от класса `Component`, и поэтому другие компоненты можно добавлять в контейнер типа `JButton`. (Хотя эти компоненты и не будут отображаться.)

У каждого контейнера имеется свой диспетчер компоновки по умолчанию, но ничто не мешает установить свой собственный диспетчер компоновки. Например, в приведенной ниже строке кода класс `GridLayout` используется для размещения компонентов на панели. Когда компоненты вводятся в контейнер, метод `add()` контейнера принимает компонент и директивы по размещению, необходимые для диспетчера компоновки.

```
panel.setLayout(new GridLayout(4, 4));
```

`java.awt.Container 1.0`

- `void setLayout(LayoutManager m)`
Задает диспетчер компоновки для данного контейнера.
- `Component add(Component c)`
- `Component add(Component c, Object constraints) 1.1`
Вводят компонент в данный контейнер и возвращают ссылку на него.

`java.awt.FlowLayout 1.0`

- `FlowLayout()`
- `FlowLayout(int align)`
- `FlowLayout(int align, int hgap, int vgap)`
Конструируют новый объект типа `FlowLayout`. В качестве параметра `align` задается выравнивание по левому (`LEFT`) краю, правому (`RIGHT`) краю или по центру (`CENTER`).

11.2.2. Граничная компоновка

Диспетчер граничной компоновки по умолчанию выбирается для панели содержимого, присутствующей в объекте типа `JFrame`. В отличие от диспетчера поточной компоновки, который полностью управляет расположением каждого компонента, диспетчер граничной компоновки позволяет выбрать место для каждого компонента. Компонент можно разместить в центре панели, в ее верхней или нижней части, а также слева или справа, как по сторонам света (рис. 11.7).

Например:

```
frame.add(component, BorderLayout.SOUTH);
```

При размещении компонентов сначала выделяется место по краям контейнера, а оставшееся свободное пространство считается центральной областью. При изменении размеров контейнера размеры компонентов, располагаемых по краям, остаются

прежними, а изменяются лишь размеры центральной области. При вводе компонента на панели указываются константы `CENTER` (Центр), `NORTH` (Север), `SOUTH` (Юг), `EAST` (Восток) или `WEST` (Запад), определенные в классе `BorderLayout`. Занимать все места на панели совсем не обязательно. Если не указано никакого значения, то по умолчанию принимается константа `CENTER`, т.е. расположение по центру.

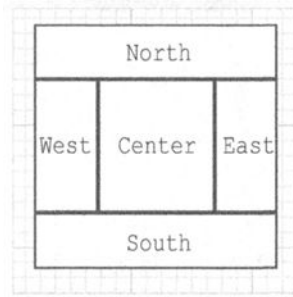


Рис. 11.7. Граничная компоновка



НА ЗАМЕТКУ! Константы в классе `BorderLayout` определены как символьные строки. Например, константа `BorderLayout.SOUTH` представляет собой символьную строку `"South"`. Пользоваться константами вместо символьных строк надежнее. Так, если вы случайно сделаете опечатку в символьной строке при вызове `frame.add(component, "South")`, компилятор не распознает ее как ошибку.

В отличие от поточной компоновки, при граничной компоновке все компоненты растягиваются, чтобы заполнить свободное пространство. (А при поточной компоновке предпочтительные размеры каждого компонента остаются без изменения.) Это может послужить препятствием к добавлению экранной кнопки, как показано ниже.

```
frame.add(yellowButton, BorderLayout.SOUTH);  
// Не рекомендуется!
```

На рис. 11.8 показано, что произойдет, если попытаться выполнить приведенную выше строку кода. Размеры экранной кнопки увеличатся, и она заполнит всю нижнюю часть фрейма. Если же попытаться вставить в нижнюю часть фрейма еще одну экранную кнопку, она просто заменит предыдущую.

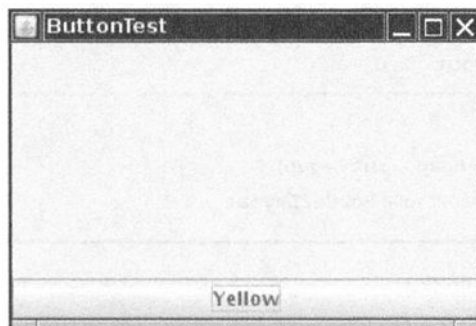


Рис. 11.8. Граничная компоновка
одиной экранной кнопки

В качестве выхода из этого затруднительного положения можно воспользоваться дополнительными панелями. Обратите внимание на пример компоновки, приведенный на рис. 11.9. Все три кнопки в нижней части экрана находятся на одной панели, которая, в свою очередь, располагается в южной области панели содержимого фрейма.

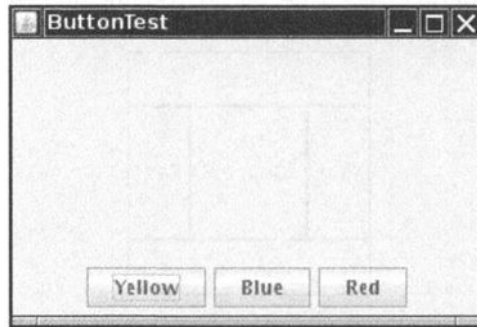


Рис. 11.9. Панель с тремя кнопками, располагаемая в южной области фрейма

Для такого расположения сначала создается новый объект типа `JPanel`, в который затем вводятся отдельные экранные кнопки. Как упоминалось ранее, с обычной панелью по умолчанию связывается диспетчер поточной компоновки типа `FlowLayout`. В данном случае он вполне подходит. С помощью метода `add()` на панели размещаются отдельные экранные кнопки, как было показано ранее. Расположение и размеры кнопок определяются диспетчером типа `FlowLayout`. Это означает, что кнопки будут выровнены по центру панели, а их размеры не будут увеличены для заполнения всего свободного пространства. И, наконец, панель с тремя экранными кнопками располагается в нижней части панели содержимого фрейма, как показано в приведенном ниже фрагменте кода. Граничная компоновка растягивает панель с тремя экранными кнопками, чтобы она заняла всю нижнюю (южную) область фрейма.

```
var panel = new JPanel();
panel.add(yellowButton);
panel.add(blueButton);
panel.add(redButton);
frame.add(panel, BorderLayout.SOUTH);
```

java.awt.BorderLayout 1.0

- `BorderLayout()`
- `BorderLayout(int hgap, int vgap)`
Конструирует новый объект типа `BorderLayout`.

11.2.3. Сеточная компоновка

При сеточной компоновке компоненты располагаются рядами и столбцами, как в таблице. Но в этом случае размеры всех компонентов оказываются одинаковыми. На рис. 11.10 показано окно, в котором для размещения кнопок калькулятора применяется сеточная компоновка. При изменении размеров окна экранные кнопки

автоматически увеличиваются или уменьшаются, причем размеры всех кнопок остаются одинаковыми.

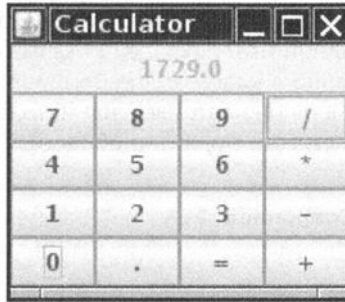


Рис. 11.10. Сеточная компоновка кнопок калькулятора

Требуемое количество рядов и столбцов указывается в конструкторе объекта типа `GridLayout` следующим образом:

```
panel.setLayout(new GridLayout(4, 4));
```

Компоненты вводятся построчно: сначала в первую ячейку первого ряда, затем во вторую ячейку первого ряда и так далее:

```
panel.add(new JButton("1"));  
panel.add(new JButton("2"));
```

На практике лишь в немногих программах применяется такая жесткая компоновка GUI, как на лицевой панели калькулятора, хотя небольшие сеточные компоновки (обычно из одного ряда или одного столбца) могут применяться для разбиения окна на равные части. Так, если в окне требуется разместить несколько экранных кнопок одинакового размера, их следует ввести на панели с сеточной компоновкой в один ряд. Очевидно, что в конструкторе диспетчера сеточной компоновки необходимо задать один ряд, а число столбцов должно равняться количеству экранных кнопок.

```
java.awt.GridLayout 1.0
```

- `GridLayout(int rows, int columns)`
- `GridLayout(int rows, int columns, int hgap, int vgap)`

Создают новый объект типа `GridLayout` с заданным расстоянием между рядами и столбцами по горизонтали и по вертикали. Один из параметров `rows` и `columns`, но не оба сразу, может принимать нулевое значение.

11.3. Ввод текста

Теперь рассмотрим компоненты пользовательского интерфейса, входящие в состав библиотеки Swing. Начнем с компонентов, дающих пользователю возможность вводить и править текст. Для этой цели предусмотрены два компонента: текстовое поле типа `JTextField` и текстовая область типа `JTextArea`. В текстовом поле можно ввести только одну текстовую строку, а в текстовой области — несколько строк. Поле типа `JPasswordField` принимает текстовую строку, не отображая ее содержимое.

Все три упомянутых выше класса для ввода текста расширяют класс `JTextComponent`. Создать объект этого класса нельзя, поскольку он является абстрактным. С другой стороны, как это часто бывает при программировании на Java, при просмотре документации на прикладной интерфейс API оказывается, что существуют методы, которые определены именно в классе `JTextComponent` и лишь наследуются его подклассами. В качестве примера ниже приведены методы, позволяющие установить текст или получить его из текстового поля или области.

```
javax.swing.text.JTextComponent 1.2
```

- `String getText()`
- `void setText(String text)`
Получают и устанавливают текст в данном текстовом компоненте.
- `boolean isEditable()`
- `void setEditable(boolean b)`
Получают и устанавливают свойство `editable`, определяющее, может ли пользователь редактировать содержимое данного текстового компонента.

11.3.1. Текстовые поля

Обычный способ ввести текстовое поле в окне состоит в том, чтобы разместить его на панели или в другом контейнере аналогично экранной кнопке. Ниже показано, как это делается непосредственно в коде.

```
var panel = new JPanel();  
var textField = new JTextField("Default input", 20);  
panel.add(textField);
```

В приведенном выше фрагменте кода вводится текстовое поле, инициализируемое текстовой строкой "Default input" (Ввод по умолчанию). Второй параметр конструктора задает длину текстовой строки. В данном случае длина строки равна 20 символам. К сожалению, символы — не очень точная единица измерения. Их ширина зависит от выбранного шрифта. Дело в том, что если ожидается ввод n или меньше символов, то n следует указать в качестве ширины столбца. На практике такая единица измерения не совсем пригодна, поэтому длину строки приходится завышать на один или два символа. Следует также учесть, что заданное количество символов считается в AWT, а следовательно, и в Swing, лишь *предпочтительной* длиной строки. Решив уменьшить или увеличить текстовое поле, диспетчер компоновки изменит длину строки. Длина строки, задаваемая параметром конструктора типа `JTextField`, не является максимально допустимым количеством символов, которое может ввести пользователь в данном текстовом поле. Пользователь может набирать и более длинные строки, а если набираемый текст выйдет за пределы текстового поля, то произойдет автоматическая прокрутка его содержимого. Но зачастую прокрутка текста плохо воспринимается пользователями, поэтому размеры текстового поля следует задавать с определенным запасом. Если же во время выполнения возникает необходимость изменить эти размеры, следует вызвать метод `setColumns()`.



СОВЕТ. После изменения размеров текстового поля методом `setColumns()` следует вызвать метод `revalidate()` из контейнера, содержащего данный компонент, как показано ниже.

```
textField.setColumns(10);  
panel.revalidate();
```

В методе **revalidate()** заново рассчитываются размеры и взаимное расположение всех компонентов в контейнере. Затем диспетчер компоновки перерисовывает контейнер, изменяя размеры текстового поля.

Метод **revalidate()** относится к классу **JComponent**. Его выполнение не приводит к немедленному изменению размеров компонента, который лишь помечается специальным образом. Такой подход исключает постоянные вычисления при запросе на изменение размеров нескольких компонентов. Если же требуется изменить размеры компонентов в контейнере типа **JFrame**, следует вызвать метод **validate()**, поскольку класс **JFrame** не является производным от класса **JComponent**.

Обычно пользователь программы должен иметь возможность вводить текст или редактировать содержимое текстового поля. Нередко в начале работы программы текстовые поля оказываются пустыми. Чтобы создать пустое текстовое поле, достаточно опустить соответствующий параметр в конструкторе класса **JTextField**, как показано ниже.

```
var textField = new JTextField(20);
```

Содержимое текстового поля можно изменить в любой момент, вызвав метод **setText()** из родительского класса **TextComponent** следующим образом:

```
textField.setText("Hello!");
```

Как упоминалось ранее, определить, какой именно текст содержится в текстовом поле, можно с помощью метода **getText()**, который возвращает текст, набранный пользователем. Чтобы отбросить лишние пробелы в начале и в конце текста, следует вызвать метод **trim()** по значению, возвращаемому методом **getText()**, как показано ниже. А для того чтобы задать шрифт, которым выделяется текст, следует вызвать метод **setFont()**.

```
String text = textField.getText().trim();
```

javax.swing.JTextField 1.2

- **JTextField(int cols)**
Создает пустое текстовое поле типа **JTextField** с заданным числом столбцов.
- **JTextField(String text, int cols)**
Создает текстовое поле указанных размеров с первоначальной символьной строкой и заданным числом столбцов.
- **int getColumns()**
- **void setColumns(int cols)**
Получают или устанавливают число столбцов для данного текстового поля.

javax.swing.JComponent 1.2

- **void revalidate()**
Обуславливает перерасчет местоположения и размеров компонента.
- **void setFont(Font f)**
Устанавливает шрифт для данного компонента.

```
java.awt.Component 1.0
```

- **void validate()**
Обуславливает перерасчет местоположения и размеров компонента. Если компонент является контейнером, местоположение и размеры содержащихся в нем компонентов должны быть также пересчитаны заново.
- **Font getFont()**
Получает шрифт данного компонента.

11.3.2. Метки и пометка компонентов

Метки являются компонентами, хранящими текст надписей. Они не имеют обрамления и других видимых элементов (например, границ), а также не реагируют на ввод данных пользователем. Метки могут использоваться для обозначения компонентов. Например, в отличие от экранных кнопок, текстовые компоненты не имеют меток, которые позволили бы их различать. Чтобы пометить компонент, не имеющий своего идентификатора, необходимо выполнить следующие действия.

1. Создать компонент типа `JLabel`, содержащий заданный текст.
2. Расположить его достаточно близко к компоненту, чтобы пользователь мог ясно видеть, что данная метка относится именно к этому компоненту.

Конструктор класса `JLabel` позволяет задать текст или пиктограмму, а если требуется, то и выровнять содержимое компонента. Для этой цели служат константы, объявленные в интерфейсе `SwingConstants`. В этом интерфейсе определено несколько полезных констант, в том числе `LEFT`, `RIGHT`, `CENTER`, `NORTH`, `EAST` и т.п. Класс `JLabel` является одним из нескольких классов из библиотеки `Swing`, реализующих этот интерфейс. В качестве примера ниже показаны два варианта задания метки, текст надписи в которой будет, например, выровнен по левому краю.

```
var label = new JLabel("User name: ", SwingConstants.RIGHT);
```

или

```
var label = new JLabel("User name: ", JLabel.RIGHT);
```

А с помощью методов `setText()` и `setIcon()` можно задать текст надписи и пиктограмму для метки во время выполнения.



СОВЕТ. В качестве надписей на экранных кнопках, метках и пунктах меню можно использовать как обычный текст, так и текст, размеченный в формате HTML. Тем не менее указывать текст надписей на экранных кнопках в формате HTML не рекомендуется, поскольку он нарушает общий стиль оформления пользовательского интерфейса. Впрочем, для меток такой текст может оказаться довольно эффективным. Для этого текстовую строку надписи на метке достаточно расположить между дескрипторами `<html>`...`</html>` следующим образом:

```
label = new JLabel("<html><b>Required</b> entry:</html>");
```

Но первый компонент с меткой, набранной текстом в формате HTML, отображается на экране с запаздыванием, поскольку для этого нужно загрузить довольно сложный код интерпретации и воспроизведения содержимого, размеченного в формате HTML.

Метки можно размещать в контейнере подобно любому другому компоненту пользовательского интерфейса. Это означает, что для их размещения применяются те же самые способы, что и рассмотренные ранее.

javax.swing.JLabel 1.2

- **JLabel(String text)**
- **JLabel(Icon icon)**
- **JLabel(String text, int align)**
- **JLabel(String text, Icon icon, int align)**

Создают метку с текстом и пиктограммой. В качестве параметра **align** указывается одна из следующих констант, определяемых в интерфейсе **SwingConstants**: **LEFT** (по умолчанию), **CENTER** или **RIGHT**.

- **String getText()**
- **void setText(String text)**
- **Icon getIcon()**
- **void setIcon(Icon icon)**

Получают или устанавливают текст данной метки.

Получают или устанавливают пиктограмму данной метки.

11.3.3. Поля для ввода пароля

Поля для ввода пароля представляют собой особый вид текстовых полей. Символы пароля не отображаются на экране, чтобы скрыть его от посторонних наблюдателей. Вместо этого каждый символ в пароле заменяется *эхо-символом*, обычно маркером (*). В библиотеке Swing предусмотрен класс **JPasswordField**, реализующий такое текстовое поле.

Поле для ввода пароля служит еще одним примером, наглядно демонстрирующим преимущества шаблона “модель–представление–контроллер”. В целях хранения данных в поле для ввода пароля применяется та же модель, что и для обычного текстового поля, но представление этого поля изменено, заменяя все символы пароля эхо-символами.

javax.swing.JPasswordField 1.2

- **JPasswordField(String text, int columns)**

Создает новое поле для ввода пароля.

- **void setEchoChar(char echo)**

Задаёт эхо-символ, который может зависеть от визуального стиля оформления пользовательского интерфейса. Если задано нулевое значение, выбирается эхо-символ по умолчанию.

- **char[] getPassword()**

Возвращает текст, содержащийся в поле для ввода пароля. Для обеспечения большей безопасности возвращаемый массив следует перезаписать после использования. Пароль возвращается как массив символов, а не как объект типа **String**. Причина такого решения заключается в том, что символьная строка может оставаться в виртуальной машине до тех пор, пока она не будет уничтожена системой сборки “мусора”.

11.3.4. Текстовые области

Иногда возникает потребность ввести несколько текстовых строк. Как указывалось ранее, для этого применяется компонент типа `JTextArea`. Внедрив этот компонент в свою программу, разработчик предоставляет пользователю возможность вводить сколько угодно текста, разделяя его строки нажатием клавиши `<Enter>`. Каждая текстовая строка завершается символом `'\n'`, как это предусмотрено в Java. Пример текстовой области в действии приведен на рис. 11.11.



Рис. 11.11. Текстовая область вместе с другими текстовыми компонентами

В конструкторе компонента типа `JTextArea` указывается количество строк и их длина, как в следующем примере кода:

```
textArea = new JTextArea(8, 40); // 8 строк по 40
                                // столбцов в каждой
```

Параметр `columns`, задающий количество столбцов (а по существу, символов) в строке, действует так же, как и для текстового поля; его значение рекомендуется немного завысить. Пользователь не ограничен количеством вводимых строк и их длиной. Если длина строки или число строк выйдет за пределы заданных параметров, текст будет прокручиваться в окне. Для изменения длины строк можно вызвать метод `setColumns()`, а для изменения их количества — метод `setRows()`. Эти параметры задают лишь рекомендуемые размеры, а диспетчер компоновки может самостоятельно увеличивать или уменьшать размеры текстовой области.

Если пользователь введет больше текста, чем умещается в текстовой области, остальной текст просто отсекается. Этого можно избежать, установив автоматический перенос строки следующим образом:

```
textArea.setLineWrap(true); // в длинных строках
                             // выполняется перенос
```

Автоматический перенос строки проявляется лишь визуально. Текст, хранящийся в документе, не изменяется — в него не вставляются символы '\n'.

11.3.5. Панели прокрутки

В библиотеке Swing текстовая область не снабжается полосами прокрутки. Если они требуются, текстовую область следует ввести на *панели прокрутки*, как показано ниже.

```
textArea = new JTextArea(8, 40);  
var scrollPane = new JScrollPane(textArea);
```

Теперь панель прокрутки управляет представлением текстовой области. Полосы прокрутки появляются автоматически, когда текст выходит за пределы отведенной для него области, и исчезают, когда оставшаяся часть текста удаляется. Сама прокрутка обеспечивается панелью прокрутки, а прикладная программа не должна обрабатывать события, связанные с прокруткой.

Это универсальный механизм, который пригоден для любого компонента, а не только для текстовых областей. Чтобы ввести полосы прокрутки в компонент, его достаточно разместить на панели прокрутки.

В программе из листинга 11.1 демонстрируются различные текстовые компоненты. Эта программа отображает текстовое поле, поле для ввода пароля и текстовую область с полосами прокрутки. Текстовое поле и поле для ввода пароля снабжено метками. Чтобы ввести предложение в конце текста, следует щелкнуть на кнопке Insert (Вставить).



НА ЗАМЕТКУ! Компонент типа `JTextArea` позволяет отображать только простой текст без форматирования и выделения специальными шрифтами. Для отображения отформатированного текста (например, в виде HTML-разметки) можно воспользоваться классом `JEditorPane`, подробнее рассматриваемым во втором томе настоящего издания.

Листинг 11.1. Исходный код из файла `text/TextComponentFrame.java`

```
1 package text;  
2  
3 import java.awt.BorderLayout;  
4 import java.awt.GridLayout;  
5  
6 import javax.swing.JButton;  
7 import javax.swing.JFrame;  
8 import javax.swing.JLabel;  
9 import javax.swing.JPanel;  
10 import javax.swing.JPasswordField;  
11 import javax.swing.JScrollPane;  
12 import javax.swing.JTextArea;  
13 import javax.swing.JTextField;  
14 import javax.swing.SwingConstants;  
15  
16 /**  
17  * Фрейм с образцами текстовых компонентов  
18  */  
19 public class TextComponentFrame extends JFrame  
20 {
```

```
21 public static final int TEXTAREA_ROWS = 8;
22 public static final int TEXTAREA_COLUMNS = 20;
23
24 public TextComponentFrame()
25 {
26     var textField = new JTextField();
27     var passwordField = new JPasswordField();
28
29     var northPanel = new JPanel();
30     northPanel.setLayout(new GridLayout(2, 2));
31     northPanel.add(new JLabel("User name: ",
32                               SwingConstants.RIGHT));
33     northPanel.add(textField);
34     northPanel.add(new JLabel("Password: ",
35                               SwingConstants.RIGHT));
36     northPanel.add(passwordField);
37
38     add(northPanel, BorderLayout.NORTH);
39
40     var textArea = new JTextArea(TEXTAREA_ROWS,
41                                  TEXTAREA_COLUMNS);
42     var scrollPane = new JScrollPane(textArea);
43
44     add(scrollPane, BorderLayout.CENTER);
45
46     // ввести кнопку для заполнения области текстом
47
48     JPanel southPanel = new JPanel();
49
50     JButton insertButton = new JButton("Insert");
51     southPanel.add(insertButton);
52     insertButton.addActionListener(event ->
53     textArea.append("User name: "
54                    + textField.getText()
55                    + " Password: "
56                    + new String(passwordField.getPassword())
57                    + "\n"));
58
59     add(southPanel, BorderLayout.SOUTH);
60     pack();
61 }
62 }
```

javax.swing.JTextArea 1.2

- **JTextArea()**
- **JTextArea(int rows, int cols)**
- **JTextArea(String text, int rows, int cols)**

Создают новую текстовую область.

- **void setColumns(int cols)**

Задаёт предпочтительное число столбцов, определяющее длину строк в текстовой области.

javax.swing.JTextArea 1.2 (окончание)

- **void setRows(int rows)**
Задаёт предпочтительное число строк в текстовой области.
- **void append(String newText)**
Добавляет заданный текст в конце содержимого текстовой области.
- **void setLineWrap(boolean wrap)**
Включает и отключает режим автоматического переноса строк.
- **void setWrapStyleWord(boolean word)**
Если параметр **word** принимает логическое значение **true**, перенос в длинных строках выполняется по границам слов, а иначе границы слов во внимание не принимаются.
- **void setTabSize(int c)**
Устанавливает позиции табуляции через каждые **c** символов. Следует, однако, иметь в виду, что символы табуляции не преобразуются в пробелы и лишь выравнивают текст по следующей позиции табуляции.

javax.swing.JScrollPane 1.2

- **JScrollPane(Component c)**
Создаёт панель прокрутки, которая отображает содержимое указанного компонента. Полоса прокрутки появляется лишь в том случае, если компонент крупнее представления.

11.4. Компоненты для выбора разных вариантов

Итак, мы рассмотрели, как принимать текстовые данные, вводимые пользователем. Но во многих случаях предпочтительнее ограничить действия пользователя выбором из конечного числа вариантов. Эти варианты могут быть представлены экранными кнопками или списком выбираемых элементов. (Как правило, такой подход освобождает от необходимости отслеживать ошибки ввода.) В этом разделе описывается порядок программирования таких компонентов пользовательского интерфейса, как флажки, кнопки-переключатели, списки и регулируемые ползунки.

11.4.1. Флажки

Если данные сводятся к двухзначной логике вроде положительного или отрицательного ответа, то для их ввода можно воспользоваться таким компонентом, как флажок. Чтобы установить флажок, достаточно щелкнуть кнопкой мыши на этом компоненте, а для того чтобы сбросить флажок — щелкнуть на нем еще раз. Установить или сбросить флажок можно также с помощью клавиши пробела, нажав ее в тот момент, когда на данном компоненте находится фокус ввода.

На рис. 11.12 показано простое окно прикладной программы с двумя флажками, один из которых включает и отключает курсивное, а другой — полужирное начертание шрифта. Обратите внимание на то, что первый флажок обладает фокусом ввода. Об этом свидетельствует прямоугольная рамка вокруг его метки. Всякий раз, когда

пользователь щелкает на флажке, содержимое окна обновляется с учетом нового начертания шрифта.

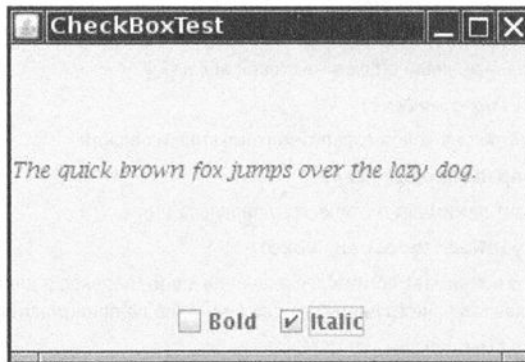


Рис. 11.12. Флажки

Флажки сопровождаются метками, указывающими их назначение. Текст метки задается в конструкторе следующим образом:

```
bold = new JCheckBox("Bold");
```

Для установки и сброса флажка вызывается метод `setSelected()`:

```
bold.setSelected(true);
```

Метод `isSelected()` позволяет определить текущее состояние каждого флажка. Если он возвращает логическое значение `false`, это означает, что флажок сброшен, а если логическое значение `true` — флажок установлен.

Щелкая на флажке, пользователь инициирует определенные события. Как всегда, с данным компонентом можно связать объект приемника событий. В рассматриваемом здесь примере программы для обоих флажков предусмотрен один и тот же приемник действий:

```
ActionListener listener = . . .  
bold.addActionListener(listener);  
italic.addActionListener(listener);
```

В приведенном ниже методе `actionPerformed()` обработки событий запрашивается текущее состояние флажков `bold` и `italic`, а затем устанавливается начертание шрифта, которым должен отображаться обычный текст: *полужирный*, *курсив* или *полужирный курсив*.

```
public void actionPerformed(ActionEvent event)  
{  
    int mode = 0;  
    if (bold.isSelected()) mode += Font.BOLD;  
    if (italic.isSelected()) mode += Font.ITALIC;  
    label.setFont(new Font("Serif", mode, FONTSIZE));  
}
```

В листинге 11.2 приведен весь исходный код программы, демонстрирующей манипулирование флажками при построении GUI.

Листинг 11.2. Исходный код из файла `checkBox/CheckBoxTest.java`

```
1 package checkBox;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 /**
8  * Фрейм с меткой образцового текста и
9  * флажками для выбора шрифта
10  * attributes.
11  */
12 public class CheckBoxFrame extends JFrame
13 {
14     private JLabel label;
15     private JCheckBox bold;
16     private JCheckBox italic;
17     private static final int FONTSIZE = 24;
18
19     public CheckBoxFrame()
20     {
21         // ввести метку образцового текста
22
23         label = new JLabel("The quick brown fox "
24                             + "jumps over the lazy dog.");
25         label.setFont(new Font("Serif", Font.BOLD, FONTSIZE));
26         add(label, BorderLayout.CENTER);
27
28         // В этом приемнике событий устанавливается
29         // атрибут шрифта для воспроизведения метки
30         // по состоянию флажка
31
32         ActionListener listener = event -> {
33             int mode = 0;
34             if (bold.isSelected()) mode += Font.BOLD;
35             if (italic.isSelected()) mode += Font.ITALIC;
36             label.setFont(new Font("Serif", mode, FONTSIZE));
37         };
38
39         // ввести флажки
40
41         JPanel buttonPanel = new JPanel();
42
43         bold = new JCheckBox("Bold");
44         bold.addActionListener(listener);
45         bold.setSelected(true);
46         buttonPanel.add(bold);
47
48         italic = new JCheckBox("Italic");
49         italic.addActionListener(listener);
50         buttonPanel.add(italic);
51
52         add(buttonPanel, BorderLayout.SOUTH);
53         pack();
54     }
55 }
```

```
javax.swing.JCheckBox 1.2
```

- `JCheckBox(String label)`
- `JCheckBox(String label, Icon icon)`
Создают флажок, который исходно сброшен.
- `JCheckBox(String label, boolean state)`
Создает флажок с указанной меткой и заданным исходным состоянием.
- `boolean isSelected()`
- `void setSelected(boolean state)`
Получают или устанавливают новое состояние флажка.

11.4.2. Кнопки-переключатели

В предыдущем примере программы пользователь мог установить оба флажка, один из них или ни одного. Но зачастую требуется выбрать только один из предлагаемых вариантов. Если пользователь установит другой флажок, то предыдущий флажок будет сброшен. Такую группу флажков часто называют *группой кнопок-переключателей*, поскольку они напоминают переключатели диапазонов на радиоприемниках — при нажатии одной из таких кнопок ранее нажатая кнопка возвращается в исходное состояние. На рис. 11.13 приведен типичный пример окна прикладной программы с группой кнопок-переключателей. Пользователь может выбрать размер шрифта — Small (Малый), Medium (Средний), Large (Крупный) и Extra large (Очень крупный). Разумеется, выбрать можно лишь один размер шрифта.



Рис. 11.13. Группа кнопок-переключателей

Библиотека Swing позволяет легко реализовать группы кнопок-переключателей. Для этого нужно создать по одному объекту типа `ButtonGroup` на каждую группу. Затем в группу кнопок-переключателей следует ввести объекты типа `JRadioButton`. Объект типа `ButtonGroup` предназначен для того, чтобы отключать выбранную ранее кнопку-переключатель, если пользователь щелкнет на новой кнопке. Ниже показано, каким образом все это воплощается непосредственно в коде.

```
var group = new ButtonGroup();
```

```
var smallButton = new JRadioButton("Small", false);  
group.add(smallButton);
```

```
var mediumButton = new JRadioButton("Medium", true);
group.add(mediumButton);
. . .
```

Второй параметр конструктора принимает логическое значение `true`, если изначально кнопка-переключатель должна быть включена, или логическое значение `false`, если она должна быть выключена. Следует, однако, иметь в виду, что объект типа `ButtonGroup` управляет лишь *поведением* кнопок-переключателей. Если нужно объединить несколько групп кнопок-переключателей, их следует разместить в контейнере, например, в объекте типа `JPanel`.

Обратите внимание на то, что на рис. 11.12 и 11.13 кнопки-переключатели отличаются по внешнему виду от флажков. Флажки изображаются в виде квадратов, причем на установленных флажках указывается галочка, в то время как кнопки-переключатели имеют круглую форму: включенные — с точкой внутри, а выключенные — пустые.

Механизм уведомления о наступлении событий от кнопок-переключателей точно такой же, как и для любых других видов экранных кнопок. Если пользователь выберет кнопку-переключатель, соответствующий объект инициирует событие. В рассматриваемом здесь примере программы установлен приемник событий, задающий конкретный размер шрифта, как показано ниже.

```
ActionListener listener = event -> label.setFont(
    new Font("Serif", Font.PLAIN, size));
```

Сравните этот приемник событий с приемником событий от флажка. Каждой кнопке-переключателю соответствует свой объект приемника событий. И каждому приемнику событий точно известно, что нужно делать — установить конкретный размер шрифта. Совсем иначе дело обстоит с флажками. Оба флажка в рассмотренном ранее примере программы были связаны с одним и тем же приемником событий, где вызывается метод, определяющий текущее состояние обоих флажков.

Можно ли применить такой же подход к кнопкам-переключателям? С этой целью можно было бы задать один приемник событий, устанавливающий конкретный размер шрифта, как показано ниже. Но все же предпочтительнее использовать отдельные объекты приемников событий, поскольку они более тесно связывают размер шрифта с конкретной кнопкой-переключателем.

```
if (smallButton.isSelected()) size = 8;
else if (mediumButton.isSelected()) size = 12;
. . .
```



НА ЗАМЕТКУ! В группе может быть выбрана только одна кнопка-переключатель. Хорошо бы заранее знать, какая именно, не проверяя каждую кнопку-переключатель в группе. Объект типа `ButtonGroup` управляет всеми кнопками-переключателями, и поэтому было бы удобно, если бы он предоставлял ссылку на выбранную кнопку-переключатель. В самом деле, в классе `ButtonGroup` имеется метод `getSelection()`, но он не возвращает ссылку на выбранную кнопку-переключатель. Вместо этого он возвращает ссылку типа `ButtonModel` на модель, связанную с этой кнопкой-переключателем. К сожалению, все методы из интерфейса `ButtonModel` не представляют собой ничего ценного в этом отношении.

Интерфейс `ButtonModel` наследует от интерфейса `ItemSelectable` метод `getSelectedObjects()`, возвращающий совершенно бесполезную пустую ссылку `null`. Метод `getActionCommand()` выглядит предпочтительнее, поскольку он позволяет определить текстовую строку с командой действия, а по существу, с текстовой меткой кнопки-переключателя. Но команда действия в модели этой кнопки-переключателя оказывается пустой [`null`]. И только в том случае, если явно задать команды действий для каждой кнопки-переключателя с помощью метода

`setActionCommand()`, в модели установятся значения, соответствующие каждой команде действия. А в дальнейшем команду действия для включенной кнопки-переключателя можно будет определить, сделав вызов `buttonGroup.getSelection().getActionCommand()`.

В листинге 11.3 представлен весь исходный код программы, в которой размер шрифта устанавливается с помощью кнопок-переключателей.

Листинг 11.3. Исходный код из файла `radioButton/RadioButtonFrame.java`

```
1 package radioButton;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 /**
8  * Фрейм с меткой образцового текста и
9  * кнопками-переключателями для выбора размера шрифта
10 */
11 public class RadioButtonFrame extends JFrame
12 {
13     private JPanel buttonPanel;
14     private ButtonGroup group;
15     private JLabel label;
16     private static final int DEFAULT_SIZE = 36;
17
18     public RadioButtonFrame()
19     {
20         // ввести метку с образцовым текстом
21
22         label = new JLabel("The quick brown fox jumps "
23                             + "over the lazy dog.");
24         label.setFont(new Font("Serif", Font.PLAIN,
25                                DEFAULT_SIZE));
26         add(label, BorderLayout.CENTER);
27
28         // ввести кнопки-переключатели
29
30         buttonPanel = new JPanel();
31         group = new ButtonGroup();
32
33         addRadioButton("Small", 8);
34         addRadioButton("Medium", 12);
35         addRadioButton("Large", 18);
36         addRadioButton("Extra large", 36);
37
38         add(buttonPanel, BorderLayout.SOUTH);
39         pack();
40     }
41
42     /**
43      * Вводит кнопку-переключатель, устанавливающую
44      * размер шрифта для выделения образцового текста
45      * @param name Строка надписи на кнопке
46      * @param size Размер шрифта, устанавливаемый
47      *              данной кнопкой
```

```
48     */
49     public void addRadioButton(String name, int size)
50     {
51         boolean selected = size == DEFAULT_SIZE;
52         JRadioButton button =
53             new JRadioButton(name, selected);
54         group.add(button);
55         buttonPanel.add(button);
56
57         // этот приемник событий устанавливает размер шрифта
58         // для образцового текста метки
59
60         ActionListener listener = event -> label.setFont(
61             new Font("Serif", Font.PLAIN, size));
62
63         button.addActionListener(listener);
64     }
65 }
```

javax.swing.JRadioButton 1.2

- **JRadioButton(String label, Icon icon)**
Создает кнопку-переключатель, которая исходно не выбрана.
- **JRadioButton(String label, boolean state)**
Создает кнопку-переключатель с заданной меткой и в указанном исходном состоянии.

javax.swing.ButtonGroup 1.2

- **void add(AbstractButton b)**
Вводит кнопку-переключатель в группу.
- **ButtonModel getSelection()**
Возвращает модель выбранной кнопки.

javax.swing.ButtonModel 1.2

- **String getActionCommand()**
Возвращает команду для модели данной экранной кнопки.

javax.swing.AbstractButton 1.2

- **void setActionCommand(String s)**
Задаёт команду для данной кнопки и ее модели.

11.4.3. Границы

Если в одном окне расположено несколько групп кнопок-переключателей, их нужно каким-то образом различать. Для этого в библиотеке Swing предусмотрен набор *границ*. Границу можно задать для каждого компонента, расширяющего класс `JComponent`. Обычно границей обрамляется панель, заполняемая элементами пользовательского интерфейса, например, кнопками-переключателями. Выбор границ невелик, и все они задаются с помощью одинаковых описываемых ниже действий.

1. Вызовите статический метод из класса `BorderFactory`, создающий границу в одном из следующих стилей (рис. 11.14):

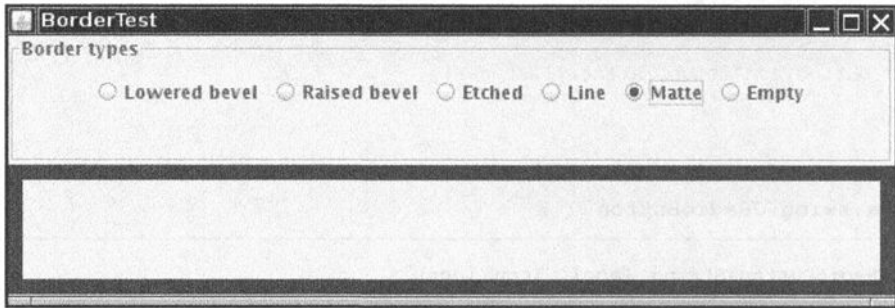


Рис. 11.14. Опробование различных видов границ

- Lowered bevel (Утопленная фаска)
 - Raised bevel (Приподнятая фаска)
 - Etched (Гравировка)
 - Line (Линия)
 - Matte (Кайма)
 - Empty (Пустая — создается пустое пространство, окружающее компонент)
2. Если требуется, дополните границу заголовком, сделав вызов `BorderFactory.createTitledBorder()`.
 3. Если требуется, объедините несколько границ в одну, сделав вызов `BorderFactory.createCompoundBorder()`.
 4. Добавьте полученную в итоге границу с помощью метода `setBorder()` из класса `JComponent`.

В приведенном ниже фрагменте кода на панели вводится граница в стиле гравировки с указанным заголовком.

```
Border etched = BorderFactory.createEtchedBorder();
Border titled = BorderFactory.createTitledBorder(
    etched, "A Title");
panel.setBorder(titled);
```

У различных границ имеются разные возможности для задания ширины и цвета. Подробнее об этом см. в документации на прикладной интерфейс API. Истинные любители пользоваться границами оценят по достоинству возможность сглаживать и скруглять углы границ, предоставляемую в классах `SoftBevelBorder` и `LineBorder`.

Такие границы можно создать только с помощью конструкторов этих классов, поскольку для них не предусмотрены соответствующие методы в классе `BorderFactory`.

`javax.swing.BorderFactory` 1.2

- `static Border createLineBorder(Color color)`
- `static Border createLineBorder(Color color, int thickness)`
Создают простую границу в стиле обычной линии.
- `static MatteBorder createMatteBorder(int top, int left, int bottom, int right, Color color)`
- `static MatteBorder createMatteBorder(int top, int left, int bottom, int right, Icon tileIcon)`
Создают широкую границу, заполняемую цветом или рисунком из повторяющихся пиктограмм.
- `static Border createEmptyBorder()`
- `static Border createEmptyBorder(int top, int left, int bottom, int right)`
Создают пустую границу.
- `static Border createEtchedBorder()`
- `static Border createEtchedBorder(Color highlight, Color shadow)`
- `static Border createEtchedBorder(int type)`
- `static Border createEtchedBorder(int type, Color highlight, Color shadow)`
Создают простую границу в стиле линии с трехмерным эффектом. В качестве параметра `type` указывается одна из констант `EtchedBorder.RAISED` или `EtchedBorder.LOWERED`.
- `static Border createBevelBorder(int type)`
- `static Border createBevelBorder(int type, Color highlight, Color shadow)`
- `static Border createLoweredBevelBorder()`
- `static Border createRaisedBevelBorder()`
Создают границу с эффектом утопленной или приподнятой поверхности. В качестве параметра `type` указывается одна из констант `BevelBorder.RAISED` или `BevelBorder.LOWERED`.
- `static TitledBorder createTitledBorder(String title)`
- `static TitledBorder createTitledBorder(Border border)`
- `static TitledBorder createTitledBorder(Border border, String title)`
- `static TitledBorder createTitledBorder(Border border, String title, int justification, int position)`
- `static TitledBorder createTitledBorder(Border border, String title, int justification, int position, Font font)`
- `static TitledBorder createTitledBorder(Border border, String title, int justification, int position, Font font, Color color)`
Создают границу с заданными свойствами и снабженную заголовком. В качестве параметра `justification` указывается одна из следующих констант, определяемых в классе `TitledBorder`: `LEFT`, `CENTER`, `RIGHT`, `LEADING`, `TRAILING` или `DEFAULT_JUSTIFICATION` (по левому краю), а в качестве параметра `position` — одна из констант `ABOVE_TOP`, `TOP`, `BELOW_TOP`, `ABOVE_BOTTOM`, `BOTTOM`, `BELOW_BOTTOM` или `DEFAULT_POSITION` (вверху).

javax.swing.BorderFactory 1.2 (окончание)

- **static CompoundBorder createCompoundBorder(Border outsideBorder, Border insideBorder)**

Объединяет две границы в одну новую границу.

javax.swing.border.SoftBevelBorder 1.2

- **SoftBevelBorder(int type)**
- **SoftBevelBorder(int type, Color highlight, Color shadow)**

Создают скошенную границу со сглаженными углами. В качестве параметра **type** указывается одна из следующих констант: **SoftBevelBorder.RAISED** или **SoftBevelBorder.LOWERED**.

javax.swing.border.LineBorder 1.2

- **public LineBorder(Color color, int thickness, boolean roundedCorners)**

Создает границу в стиле линии заданной толщины и цвета. Если параметр **roundedCorners** принимает логическое значение **true**, граница имеет скругленные углы.

javax.swing.JComponent 1.2

- **void setBorder(Border border)**

Задает границу для данного компонента.

11.4.4. Комбинированные списки

Если вариантов выбора слишком много, то кнопки-переключатели для этой цели не подойдут, поскольку для них не хватит места на экране. В таком случае следует воспользоваться *раскрывающимся списком*. Если пользователь щелкнет на этом компоненте, раскроется список, из которого он может выбрать один из его элементов (рис. 11.15).

Если раскрывающийся список является *редактируемым*, то выбранный из него элемент можно поправить так же, как и в обычном тестовом поле. Таким образом, редактируемый раскрывающийся список объединяет в себе удобства текстового поля и возможность выбора из предопределенного ряда вариантов, и в этом случае такой список называется *комбинированным*. Компоненты комбинированных списков создаются средствами класса `JComboBox`. Начиная с версии Java 7, класс `JComboBox` является обобщенным. Например, комбинированный список типа `JComboBox<String>` состоит из строковых объектов типа `String`, а комбинированный список типа `JComboBox<Integer>` — из целочисленных значений.

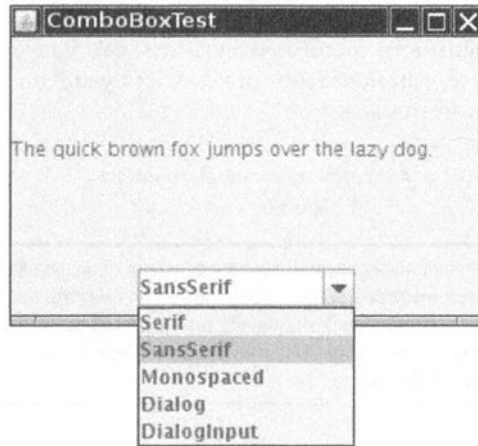


Рис. 11.15. Раскрывающийся список

Чтобы сделать раскрывающийся список редактируемым, т.е. комбинированным, достаточно вызвать метод `setEditable()`. Однако изменения вносятся только в текущий элемент списка. Перечень вариантов выбора все равно остается прежним.

Выбранный вариант в исходном или отредактированном виде можно получить с помощью метода `getSelectedItem()`. Но для комбинированного списка этот элемент может быть любого типа в зависимости от редактора, принимающего пользовательские правки и сохраняющего результат в соответствующем объекте. (Подробнее о редакторах речь пойдет в главе 6 второго тома настоящего издания.) Если же список является раскрывающимся и не допускает редактирования своих элементов, то для получения выбранного варианта требующегося типа лучше сделать следующий вызов:

```
combo.getItemAt(combo.getSelectedIndex())
```

В рассматриваемом здесь примере программы у пользователя имеется возможность выбрать стиль шрифта из предварительно заданного списка (*Serif* — с засечками, *SansSerif* — без засечек, *Monospaced* — моноширинный и т.д.). Кроме того, пользователь может ввести в список новый стиль шрифта, добавив в список соответствующий элемент, для чего служит метод `addItem()`. В данной программе метод `addItem()` вызывается только в конструкторе, как показано ниже, но при необходимости к нему можно обратиться из любой части программы.

```
var faceCombo = new JComboBox<String>();
faceCombo.addItem("Serif");
faceCombo.addItem("SansSerif");
...
```

Этот метод добавляет символьную строку в конце списка. Если же требуется вставить символьную строку в любом другом месте списка, нужно вызвать метод `insertItemAt()` следующим образом:

```
faceCombo.insertItemAt("Monospaced", 0);
// ввести элемент в начале списка
```

В список можно вводить элементы любого типа, а для их отображения вызывается метод `toString()`. Если во время выполнения возникает потребность

удалить элемент из списка, для этой цели вызывается метод `removeItem()` или `removeItemAt()`, в зависимости от того, что указать: сам удаляемый элемент или его местоположение в списке, как показано ниже. А для удаления сразу всех элементов из списка предусмотрен метод `removeAllItems()`.

```
faceCombo.removeItem("Monospaced");
faceCombo.removeItemAt(0); // удалить первый элемент
                           // из списка
```



СОВЕТ. Если в комбинированный список требуется включить большое количество объектов, применять для этой цели метод `addItem()` не следует, чтобы не снижать производительность программы. Вместо этого лучше сконструировать объект типа `DefaultComboBoxModel`, заполнить его элементами составляемого списка, вызывая метод `addElement()`, а затем обратиться к методу `setModel()` из класса `JComboBox`.

Когда пользователь выбирает нужный вариант из комбинированного списка, этот компонент инициирует событие. Чтобы определить вариант, выбранный из списка, следует вызвать метод `getSource()` с данным событием в качестве параметра. Этот метод возвращает ссылку на список, являющийся источником события. Затем следует вызвать метод `getSelectedItem()`, возвращающий вариант, выбранный из списка. Значение, возвращаемое этим методом, необходимо привести к соответствующему типу (как правило, к типу `String`). Но если возвращаемое значение передается в качестве параметра методу `getItemAt()`, то приведение типов не требуется, как выделено ниже полужирным.

```
ActionListener listener = event -> label.setFont(new Font(
    faceCombo.getItemAt(faceCombo.getSelectedIndex()),
    ont.PLAIN,
    DEFAULT_SIZE));
```

Весь исходный код программы, демонстрирующей применение комбинированного списка в пользовательском интерфейсе, приведен в листинге 11.4.

Листинг 11.4. Исходный код из файла `comboBox/ComboBoxFrame.java`

```
1 package comboBox;
2
3 import java.awt.BorderLayout;
4 import java.awt.Font;
5
6 import javax.swing.JComboBox;
7 import javax.swing.JFrame;
8 import javax.swing.JLabel;
9 import javax.swing.JPanel;
10
11 /**
12  * Фрейм с образцовым текстом метки и комбинированным
13  * списком для выбора начертаний шрифта
14  */
15 public class ComboBoxFrame extends JFrame
16 {
17     private JComboBox<String> faceCombo;
18     private JLabel label;
```

```
19 private static final int DEFAULT_SIZE = 24;
20
21 public ComboBoxFrame()
22 {
23     // ввести метку с образцовым текстом
24
25     label = new JLabel("The quick brown fox jumps "
26                        + "over the lazy dog.");
27     label.setFont(new Font("Serif", Font.PLAIN,
28                           DEFAULT_SIZE));
29     add(label, BorderLayout.CENTER);
30
31     // составить комбинированный список и ввести
32     // в него названия начертаний шрифта
33
34     faceCombo = new JComboBox<>();
35     faceCombo.addItem("Serif");
36     faceCombo.addItem("SansSerif");
37     faceCombo.addItem("Monospaced");
38     faceCombo.addItem("Dialog");
39     faceCombo.addItem("DialogInput");
40
41     // приемник событий от комбинированного списка
42     // изменяет на выбранное начертание шрифта,
43     // которым набран текст метки
44
45     faceCombo.addActionListener(event -> label.setFont(
46         (new Font(faceCombo.getItemAt(
47             faceCombo.getSelectedIndex()),
48             Font.PLAIN, DEFAULT_SIZE)));
49
50     // ввести комбинированный список на панели
51     // у южной границы фрейма
52
53     JPanel comboPanel = new JPanel();
54     comboPanel.add(faceCombo);
55     add(comboPanel, BorderLayout.SOUTH);
56     pack();
57 }
58 }
```

javax.swing.JComboBox 1.2

- **boolean isEditable()**
- **void setEditable(boolean b)**
Получают или устанавливают свойство **editable** данного комбинированного списка.
- **void addItem(Object item)**
Вводит новый элемент в список.
- **void insertItemAt(Object item, int index)**
Вводит заданный элемент в список по указанному индексу.

```
javax.swing.JComboBox 1.2 (окончание)
```

- `void removeItem(Object item)`
Удаляет заданный элемент из списка.
- `void removeItemAt(int index)`
Удаляет из списка заданный элемент по указанному индексу.
- `void removeAllItems()`
Удаляет из списка все элементы.
- `Object getSelectedItem()`
Возвращает выбранный элемент списка.

11.4.5. Регулируемые ползунки

Комбинированные списки дают пользователю возможность делать выбор из дискретного ряда вариантов. А регулируемые ползунки позволяют выбрать конкретное значение в заданных пределах, например, любое число от 1 до 100. Чаще всего регулируемые ползунки создаются следующим образом:

```
var slider = new JSlider(min, max, initialValue);
```

Если опустить минимальное, максимальное и начальное значения, то по умолчанию выбираются значения 0, 100 и 50 соответственно. А если регулируемый ползунок должен располагаться вертикально, то для этой цели служит следующий конструктор:

```
var slider = new JSlider(SwingConstants.VERTICAL, min,  
                        max, initialValue);
```

Каждый такой конструктор создает простой ползунок. В качестве примера можно привести самый верхний ползунок в окне, показанном на рис. 11.16. Далее будут рассмотрены более сложные разновидности регулируемых ползунков.

Когда пользователь перемещает ползунок, выбираемое значение в данном компоненте изменяется в пределах от минимального до максимального. При этом все приемники событий от регулируемого ползунка получают событие типа `ChangeEvent`. Чтобы получать уведомления об изменении выбираемого значения при перемещении ползунка, следует сначала создать объект класса, реализующего функциональный интерфейс `ChangeListener`, а затем вызвать метод `addChangeListener()`. При обратном вызове извлекается значение, на котором установлен ползунок:

```
ChangeListener listener = event -> {  
    JSlider slider = (JSlider) event.getSource();  
    int value = slider.getValue();  
    . . .  
};
```

Регулируемый ползунок можно дополнить *отметками*, как на шкале. Так, в программе, рассматриваемой здесь в качестве примера, для второго ползунка задаются следующие установки:

```
slider.setMajorTickSpacing(20);  
slider.setMinorTickSpacing(5);
```

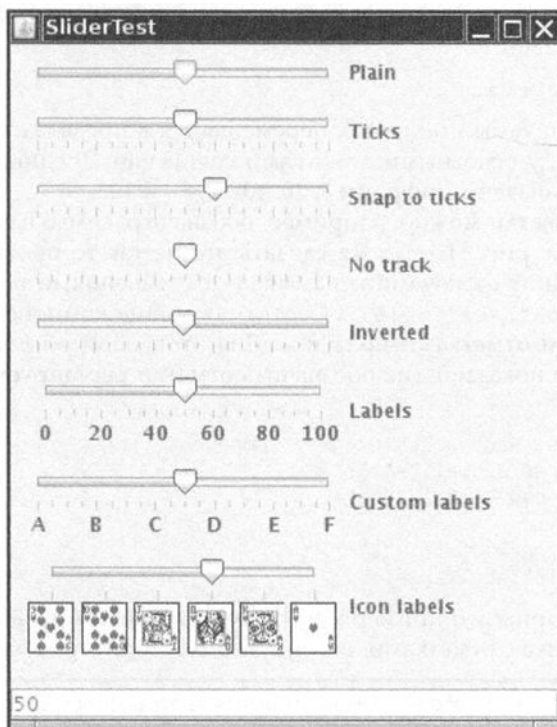


Рис. 11.16. Регулируемые ползунки

Регулируемый ползунок снабжается основными отметками, следующими через каждые 20 единиц измерения, а также вспомогательными, следующими через каждые 5 единиц измерения. Сами единицы измерения привязываются к значениям, на которых устанавливается ползунок, и не имеют никакого отношения к пикселям на экране. В приведенном выше фрагменте кода лишь устанавливаются отметки регулируемого ползунка. А для того чтобы вывести их на экран, нужно сделать следующий вызов:

```
slider.setPaintTicks(true);
```

Основные и вспомогательные отметки действуют независимо. Можно, например, установить основные отметки через каждые 20 единиц измерения, а вспомогательные — через каждые 7 единиц измерения, но в итоге шкала регулируемого ползунка получится беспорядочной.

Регулируемый ползунок можно принудительно *привязать к отметкам*. Всякий раз, когда пользователь завершает перемещение ползунка в режиме привязки к отметкам, ползунок сразу же устанавливается на ближайшей отметке. Такой режим задается с помощью следующего вызова:

```
slider.setSnapToTicks(true);
```



ВНИМАНИЕ! В режиме привязки к отметкам регулируемый ползунок ведет себя не совсем предсказуемым образом. До тех пор, пока ползунок не установится точно на отметке, приемник изменений получает значения, не соответствующие отметкам. Так, если щелкнуть кнопкой мыши рядом с ползунком, чтобы переместить его немного в нужную сторону, он все равно не установится на следующей отметке в режиме привязки к отметкам.

Сделав следующий вызов, можно обозначить основные отметки регулируемого ползунка:

```
slider.setPaintLabels(true);
```

Так, если регулируемый ползунок перемещается в пределах от 0 до 100 единиц, а промежуток между основными отметками составляет 20 единиц, отметки такого ползунка будут обозначены цифрами 0, 20, 40, 60, 80 и 100.

Кроме цифр, отметки можно, например, обозначить символьными строками или пиктограммами (см. рис. 11.16), хотя сделать это не так-то просто. Сначала нужно заполнить хеш-таблицу с ключами типа `Integer` и значениями типа `Component`, а затем вызвать метод `setLabelTable()`. Соответствующие компоненты располагаются под обозначаемыми отметками ползунка. Для этой цели обычно служат объекты типа `JLabel`. Ниже показано, как обозначить отметки регулируемого ползунка буквами A, B, C, D, E и F.

```
var labelTable = new Hashtable<Integer, Component>();
labelTable.put(0, new JLabel("A"));
labelTable.put(20, new JLabel("B"));
. . .
labelTable.put(100, new JLabel("F"));
slider.setLabelTable(labelTable);
```

В листинге 11.5 приведен пример программы, демонстрирующий построение регулируемого ползунка с отметками, обозначаемыми пиктограммами.



COBET. Если отметки и их обозначения не выводятся на экран, проверьте, вызываются ли методы `setPaintTicks(true)` и `setPaintLabels(true)`.

У четвертого регулируемого ползунка на рис. 11.16 отсутствует полоса перемещения. Подавить отображение полосы, по которой передвигается ползунок, можно, сделав следующий вызов:

```
slider.setPaintTrack(false);
```

Для пятого регулируемого ползунка на этом же рисунке направление движения изменено с помощью метода

```
slider.setInverted(true);
```

Регулируемые ползунки, создаваемые в рассматриваемом здесь примере программы, демонстрируют различные визуальные эффекты. Для каждого ползунка установлен приемник событий, отображающий значение, на котором в текущий момент установлен данный ползунок, в текстовом поле, расположенном в нижней части окна.

Листинг 11.5. Исходный код из файла `slider/SliderFrame.java`

```
1 package slider;
2
3 import java.awt.*;
4 import java.util.*;
5 import javax.swing.*;
6 import javax.swing.event.*;
7
8 /**
9  * Фрейм с несколькими ползунками и текстовым полем
```

```
10  * для показа значений, на которых по очереди
11  * устанавливаются ползунки
12  */
13  public class SliderFrame extends JFrame
14  {
15      private JPanel sliderPanel;
16      private JTextField textField;
17      private ChangeListener listener;
18
19      public SliderFrame()
20      {
21          sliderPanel = new JPanel();
22          sliderPanel.setLayout(new GridBagLayout());
23
24          // общий приемник событий для всех ползунков
25          listener = event -> {
26              // обновить текстовое поле, если
27              // выбранный ползунок установится
28              // на отметке с другим значением
29              JSlider source = (JSlider) event.getSource();
30              textField.setText("" + source.getValue());
31          };
32
33          // ввести простой ползунок
34
35          var slider = new JSlider();
36          addSlider(slider, "Plain");
37
38          // ввести ползунок с основными и
39          // неосновными отметками
40
41          slider = new JSlider();
42          slider.setPaintTicks(true);
43          slider.setMajorTickSpacing(20);
44          slider.setMinorTickSpacing(5);
45          addSlider(slider, "Ticks");
46
47          // ввести ползунок, привязываемый к отметкам
48
49          slider = new JSlider();
50          slider.setPaintTicks(true);
51          slider.setSnapToTicks(true);
52          slider.setMajorTickSpacing(20);
53          slider.setMinorTickSpacing(5);
54          addSlider(slider, "Snap to ticks");
55
56          // ввести ползунок без отметок
57
58          slider = new JSlider();
59          slider.setPaintTicks(true);
60          slider.setMajorTickSpacing(20);
61          slider.setMinorTickSpacing(5);
62          slider.setPaintTrack(false);
63          addSlider(slider, "No track");
64
65          // ввести обращенный ползунок
66
```



```
67     slider = new JSlider();
68     slider.setPaintTicks(true);
69     slider.setMajorTickSpacing(20);
70     slider.setMinorTickSpacing(5);
71     slider.setInverted(true);
72     addSlider(slider, "Inverted");
73
74     // ввести ползунок с числовыми обозначениями отметок
75
76     slider = new JSlider();
77     slider.setPaintTicks(true);
78     slider.setPaintLabels(true);
79     slider.setMajorTickSpacing(20);
80     slider.setMinorTickSpacing(5);
81     addSlider(slider, "Labels");
82
83     // ввести ползунок с буквенными
84     // обозначениями отметок
85
86     slider = new JSlider();
87     slider.setPaintLabels(true);
88     slider.setPaintTicks(true);
89     slider.setMajorTickSpacing(20);
90     slider.setMinorTickSpacing(5);
91
92     var labelTable =
93         new Hashtable<Integer, Component>();
94     labelTable.put(0, new JLabel("A"));
95     labelTable.put(20, new JLabel("B"));
96     labelTable.put(40, new JLabel("C"));
97     labelTable.put(60, new JLabel("D"));
98     labelTable.put(80, new JLabel("E"));
99     labelTable.put(100, new JLabel("F"));
100
101     slider.setLabelTable(labelTable);
102     addSlider(slider, "Custom labels");
103
104     // ввести ползунок с пиктограммными
105     // обозначениями отметок
106
107     slider = new JSlider();
108     slider.setPaintTicks(true);
109     slider.setPaintLabels(true);
110     slider.setSnapToTicks(true);
111     slider.setMajorTickSpacing(20);
112     slider.setMinorTickSpacing(20);
113
114     labelTable = new Hashtable<Integer, Component>();
115
116     // ввести изображения игральных карт
117
118     labelTable.put(0, new JLabel(
119         new ImageIcon("nine.gif")));
120     labelTable.put(20, new JLabel(
121         new ImageIcon("ten.gif")));
122     labelTable.put(40, new JLabel(
123         new ImageIcon("jack.gif"));
```

```

124     labelTable.put(60, new JLabel(
125         new ImageIcon("queen.gif")));
126     labelTable.put(80, new JLabel(
127         new ImageIcon("king.gif")));
128     labelTable.put(100, new JLabel(
129         new ImageIcon("ace.gif")));
130
131     slider.setLabelTable(labelTable);
132     addSlider(slider, "Icon labels");
133
134     // Вводит текстовое поле для показа значения,
135     // на котором установлен выбранный в настоящий
136     // момент ползунок
137
138     textField = new JTextField();
139     add(sliderPanel, BorderLayout.CENTER);
140     add(textField, BorderLayout.SOUTH);
141     pack();
142 }
143
144 /**
145  * Вводит ползунки на панели и привязывает
146  * к ним приемник событий
147  * @param s Ползунок
148  * @param description Описание ползунка
149  */
150 public void addSlider(JSlider s, String description)
151 {
152     s.addChangeListener(listener);
153     var panel = new JPanel();
154     panel.add(s);
155     panel.add(new JLabel(description));
156     panel.setAlignmentX(Component.LEFT_ALIGNMENT);
157     var gbc = new GridBagConstraints();
158     gbc.gridy = sliderPanel.getComponentCount();
159     gbc.anchor = GridBagConstraints.WEST;
160     sliderPanel.add(panel, gbc);
161 }
162 }

```

javax.swing.JSlider 1.2

- **JSlider()**
- **JSlider(int direction)c**
- **JSlider(int min, int max)**
- **JSlider(int min, int max, int initialValue)**
- **JSlider(int direction, int min, int max, int initialValue)**

Создают горизонтальный регулируемый ползунок с заданным направлением перемещения, минимальным и максимальным значениями. В качестве параметра **direction** указывается одна из констант **SwingConstants.HORIZONTAL** или **SwingConstants.VERTICAL**. По умолчанию устанавливаются значения 0, 50 и 100 параметров **min**, **max** и **initialValue** соответственно.

```
javax.swing.JSlider 1.2 (окончание)
```

- **void setPaintTicks(boolean b)**

Если параметр **b** принимает логическое значение **true**, то отображаются отметки, на которых устанавливается ползунок.

- **void setMajorTickSpacing(int units)**

- **void setMinorTickSpacing(int units)**

Устанавливают разные единицы измерения для основных и неосновных отметок.

- **void setPaintLabels(boolean b)**

Если параметр **b** принимает логическое значение **true**, то отображаются обозначения меток.

- **void setLabelTable(Dictionary table)**

Устанавливает компоненты для обозначения отметок. Каждая пара "ключ-значение" представлена в таблице в следующей форме:

- `new Integer(значение) / компонент.`

- **void setSnapToTicks(boolean b)**

Если параметр **b** принимает логическое значение **true**, то ползунок устанавливается на ближайшей отметке после каждого перемещения.

- **void setPaintTrack(boolean b)**

Если значение параметра **b** принимает логическое значение **true**, то отображается полоса, по которой перемещается ползунок.

11.5. Меню

В начале этой главы были рассмотрены наиболее употребительные компоненты пользовательского интерфейса, которые можно расположить в окне, в том числе разнообразные экранные кнопки, текстовые поля и комбинированные списки. В библиотеке Swing предусмотрены также ниспадающие меню, хорошо известные всем, кому когда-нибудь приходилось пользоваться прикладными программами с GUI.

Строка меню в верхней части окна содержит названия ниспадающих меню. Щелкая на таком имени мышью, пользователь открывает меню, состоящее из *пунктов* и *подменю*. Если пользователь щелкнет на пункте меню, все меню закроются и программе будет отправлено соответствующее уведомление. На рис. 11.17 показано типичное меню, состоящее из пунктов и подменю.

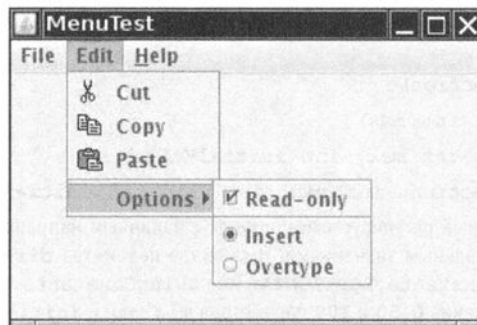


Рис. 11.17. Меню, состоящее из пунктов и подменю

11.5.1. Создание меню

Создать меню совсем не трудно. Для этого сначала создается строка меню следующим образом:

```
var = new JMenuBar();
```

Строка меню — это обычный компонент, который можно расположить где угодно. Как правило, она располагается в верхней части фрейма с помощью метода `setJMenuBar()`, как показано ниже.

```
frame.setJMenuBar(menuBar);
```

Для каждого меню создается свой объект следующим образом:

```
var editMenu = new JMenu("Edit");
```

Меню верхнего уровня размещаются в строке меню, как показано ниже.

```
menuBar.add(editMenu);
```

Затем в объект меню вводятся пункты, разделители и подменю:

```
var pasteItem = new JMenuItem("Paste");  
editMenu.add(pasteItem);  
editMenu.addSeparator();  
JMenu optionsMenu = . . .; // подменю  
editMenu.add(optionsMenu);
```

Разделители (рис. 11.17) отображаются под пунктами меню Paste (Вставка) и Read-only (Только для чтения). Когда пользователь выбирает пункт меню, инициируется событие действия. Следовательно, для каждого пункта меню следует определить обработчик:

```
ActionListener listener = . . .;  
pasteItem.addActionListener(listener);
```

Имеется удобный метод `JMenu.add(String s)`, позволяющий добавлять новый пункт в конце меню, например, так, как показано ниже.

```
editMenu.add("Paste");
```

Этот метод возвращает созданный пункт меню, для которого можно легко задать обработчик:

```
JMenuItem pasteItem = editMenu.add("Paste");  
pasteItem.addActionListener(listener);
```

Зачастую пункты меню связываются с командами, которые могут активизировать другие элементы пользовательского интерфейса, например, экранные кнопки. Как упоминалось в разделе 10.4.5, команды задаются с помощью объектов типа `Action`. Сначала следует определить класс, реализующий интерфейс `Action`. Обычно такой класс расширяет класс `AbstractAction`. Затем в конструкторе типа `AbstractAction` указывается метка пункта меню и переопределяется метод `actionPerformed()`, что позволяет реализовать обработку события, связанного с данным пунктом меню, как в приведенном ниже примере кода.

```
var exitAction = new AbstractAction("Exit")  
    // здесь указывается пункт меню  
{  
    public void actionPerformed(ActionEvent event)  
    {
```

а здесь следует код выполняемого действия

```
System.exit(0);  
}  
};
```

Затем объект типа `Action` вводится в меню следующим образом:

```
JMenuItem exitItem = fileMenu.add(exitAction);
```

В итоге новый пункт вводится в меню по имени действия. Объект этого действия становится обработчиком. Такой прием позволяет заменить следующие строки кода:

```
var exitItem = new JMenuItem(exitAction);  
fileMenu.add(exitItem);
```

`javax.swing.JMenu 1.2`

- **`JMenu(String label)`**
Создает меню с указанной меткой.
- **`JMenuItem add(JMenuItem item)`**
Добавляет пункт (или целое меню).
- **`JMenuItem add(String label)`**
Добавляет пункт в меню с указанной меткой и возвращает этот пункт меню.
- **`JMenuItem add(Action a)`**
Добавляет пункт и связанное с ним действие и возвращает этот пункт.
- **`void addSeparator()`**
Добавляет в меню разделитель.
- **`JMenuItem insert(JMenuItem menu, int index)`**
Добавляет новый пункт меню (или подменю) по указанному индексу.
- **`JMenuItem insert(Action a, int index)`**
Добавляет новый пункт меню и связанный с ним объект типа `Action` по указанному индексу.
- **`void insertSeparator(int index)`**
Добавляет в меню разделитель по указанному индексу.
- **`void remove(int index)`**
- **`void remove(JMenuItem item)`**
Удаляют указанный пункт меню.

`javax.swing.JMenuItem 1.2`

- **`JMenuItem(String label)`**
Создает пункт меню с указанной меткой.
- **`JMenuItem(Action a) 1.3`**
Создает пункт меню для указанного действия.

javax.swing.AbstractButton 1.2

- **void setAction(Action a) 1.3**

Устанавливает действие для данной экранной кнопки или пункта меню.

javax.swing.JFrame 1.2

- **void setJMenuBar(JMenuBar menubar)**

Устанавливает строку меню в данном фрейме.

11.5.2. Пиктограммы в пунктах меню

Пункты меню очень похожи на экранные кнопки. В действительности класс `JMenuItem` расширяет класс `AbstractButton`. Как и экранные кнопки, меню могут иметь текстовую метку, пиктограмму или и то и другое. Пиктограмму можно, с одной стороны, указать в конструкторе `JMenuItem(String, Icon)` или `JMenuItem(Icon)`, а с другой стороны, задать с помощью метода `setIcon()`, унаследованного классом `JMenuItem` от класса `AbstractButton`. Ниже приведен соответствующий пример.

```
var cutItem = new JMenuItem("Cut",
                           new ImageIcon("cut.gif"));
```

На рис. 11.17 показано меню с пиктограммами. По умолчанию названия пунктов меню располагаются справа от пиктограмм. Если же требуется, чтобы пиктограммы находились справа от названий пунктов меню, воспользуйтесь методом `setHorizontalTextPosition()`, унаследованным в классе `JMenuItem` от класса `AbstractButton`. Например, в приведенной ниже строке кода текст пункта меню размещается слева от пиктограммы.

```
cutItem.setHorizontalTextPosition(SwingConstants.LEFT);
```

Пиктограмму можно также связать с действием следующим образом:

```
aboutAction.putValue(Action.MNEMONIC_KEY, new Integer('A'));
```

Если пункт меню создается независимо от действия, представленного объектом типа `Action`, то значением поля `Action.NAME` становится название пункта меню, а значением поля `Action.SMALL_ICON` — его пиктограмма. Кроме того, пиктограмму можно задать в конструкторе класса `AbstractAction`:

```
cutAction = new AbstractAction("Cut",
                               new ImageIcon("cut.gif"))
{
    public void actionPerformed(ActionEvent event)
    {
        здесь следует код выполняемого действия
    }
};
```

javax.swing.JMenuItem 1.2

- **JMenuItem(String label, Icon icon)**
Создает пункт меню с указанными меткой и пиктограммой.

javax.swing.AbstractButton 1.2

- **void setHorizontalTextPosition(int pos)**
Задаёт взаимное расположение текста надписи и пиктограммы. В качестве параметра *pos* указывается константа **SwingConstants.RIGHT** (текст справа от пиктограммы) или же константа **SwingConstants.LEFT**.

javax.swing.AbstractAction 1.2

- **AbstractAction(String name, Icon smallIcon)**
Создает объект типа **AbstractAction** с указанным именем и пиктограммой.

11.5.3. Пункты меню с флажками и кнопками-переключателями

Пункты меню могут также содержать *флажки* или *кнопки-переключатели* (см. рис. 11.17). Когда пользователь щелкает мышью на пункте меню, флажок автоматически устанавливается или сбрасывается, а состояние кнопки-переключателя изменяется в соответствии с выбранным пунктом.

Помимо внешнего вида таких флажков и кнопок-переключателей, они мало чем отличаются от обычных пунктов меню. Ниже в качестве примера показано, каким образом создается пункт меню с флажком.

```
var readonlyItem = new JCheckBoxMenuItem("Read-only");  
optionsMenu.add(readonlyItem);
```

Пункты меню с кнопками-переключателями действуют точно так же, как и обычные кнопки-переключатели. Для этого в меню следует добавить группу кнопок-переключателей. Когда выбирается одна из таких кнопок, все остальные автоматически отключаются. Ниже приведен пример создания пунктов меню с кнопками-переключателями.

```
var group = new ButtonGroup();  
var insertItem = new JRadioButtonMenuItem("Insert");  
insertItem.setSelected(true);  
var overtypeItem = new JRadioButtonMenuItem("Overtypе");  
group.add(insertItem);  
group.add(overtypеItem);  
optionsMenu.add(insertItem);  
optionsMenu.add(overtypеItem);
```

В этих пунктах меню совсем не обязательно определять, когда именно пользователь сделал выбор. Вместо этого для проверки текущего состояния пункта меню достаточно вызвать метод `isSelected()`. (Разумеется, это означает, что в какой-то

переменной экземпляра придется хранить ссылку на данный пункт меню.) Кроме того, задать состояние пункта меню можно с помощью метода `setSelected()`.

`javax.swing.JCheckBoxMenuItem` 1.2

- `JCheckBoxMenuItem(String label)`
Создает пункт меню с флажком и заданной меткой.
- `JCheckBoxMenuItem(String label, boolean state)`
Создает пункт меню с флажком и заданными меткой и состоянием (если параметр `state` принимает логическое значение `true`, то пункт считается выбранным).

`javax.swing.JRadioButtonMenuItem` 1.2

- `JRadioButtonMenuItem(String label)`
Создает пункт меню с кнопкой-переключателем и заданной меткой.
- `JRadioButtonMenuItem(String label, boolean state)`
Создает пункт меню с кнопкой-переключателем и заданными меткой и состоянием (если параметр `state` принимает логическое значение `true`, то пункт считается выбранным).

`javax.swing.AbstractButton` 1.2

- `boolean isSelected()`
Возвращает состояние пункта меню.
- `void setSelected(boolean state)`
Устанавливает состояние пункта меню (если параметр `state` принимает логическое значение `true`, то пункт считается выбранным).

11.5.4. Всплывающие меню

Всплывающие, или контекстные, меню не связаны со строкой меню, а появляются в произвольно выбранном месте на экране (рис. 11.18).

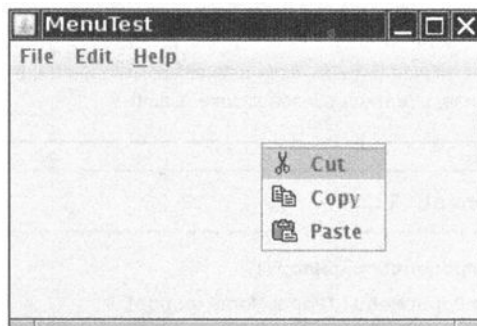


Рис. 11.18. Всплывающее меню

Всплывающее меню создается точно так же, как и обычное, за исключением того, что у него отсутствует заголовок. Ниже приведен типичный пример создания всплывающего меню в коде.

```
var popup = new JPopupMenu();
```

Пункты добавляются во всплывающее меню, как обычно:

```
var item = new JMenuItem("Cut");
item.addActionListener(listener);
popup.add(item);
```

В отличие от строки меню, которая всегда находится в верхней части фрейма, всплывающее меню следует явным образом выводить на экран с помощью метода `show()`. При вызове этого метода задается родительский компонент и расположение всплывающего меню в его системе координат:

```
popup.show(panel, x, y);
```

Обычно всплывающее меню отображается на экране, когда пользователь щелкает специально предназначенной для этого кнопкой — так называемым *триггером всплывающего меню*. В Windows и Linux это, как правило, правая кнопка мыши. Для всплывания меню после щелчка кнопкой мыши вызывается следующий метод:

```
component.setComponentPopupMenu(popup);
```

Нередко один компонент приходится размещать внутри другого компонента, с которым связано всплывающее меню. Чтобы производный компонент наследовал меню родительского компонента, достаточно сделать следующий вызов:

```
child.setInheritsPopupMenu(true);
```

javax.swing.JPopupMenu 1.2

- **void show(Component c, int x, int y)**
Отображает всплывающее меню над компонентом, определяемым параметром `c`, в левом верхнем углу с координатами `(x, y)` в пространстве данного компонента.
- **boolean isPopupTrigger(MouseEvent event) 1.3**
Возвращает логическое значение `true`, если событие инициировано триггером всплывающего меню (как правило, нажатием правой кнопки мыши).

java.awt.event.MouseEvent 1.1

- **boolean isPopupTrigger()**
Возвращает логическое значение `true`, если данное событие инициировано триггером всплывающего меню (как правило, нажатием правой кнопки мыши).

javax.swing.JComponent 1.2

- **JPopupMenu getComponentPopupMenu() 5**
- **void setComponentPopupMenu(JPopupMenu popup) 5**
Устанавливают или возвращают всплывающее меню для данного компонента.

`javax.swing.JComponent 1.2 (окончание)`

- `boolean getInheritsPopupMenu()` 5
- `void setInheritsPopupMenu(boolean b)` 5

Устанавливают или возвращают свойство `inheritsPopupMenu`. Если это свойство установлено, а вместо всплывающего меню данный компонент получает пустое значение `null`, то вызывается всплывающее меню родительского компонента.

11.5.5. Клавиши быстрого доступа и оперативные клавиши

Опытному пользователю удобно выбирать пункты меню с помощью *клавиши быстрого доступа*. Связать пункт меню с клавишей быстрого доступа можно, задав эту клавишу в конструкторе пункта меню следующим образом:

```
var aboutItem = new JMenuItem("About", 'A');
```

Буква в названии пункта меню, соответствующая клавише быстрого доступа, выделяется подчеркиванием (рис. 11.19). Так, если клавиша быстрого доступа задана с помощью приведенного выше выражения, то метка отобразится как `About`, т.е. с подчеркнутой буквой `A`. Теперь для выбора данного пункта меню пользователю достаточно нажать клавишу `<A>`. (Если буква, соответствующая назначенной клавише, не входит в название пункта меню, она не отображается на экране, но при ее нажатии этот пункт все равно будет выбран. Естественно, что польза от таких “невидимых” клавиш быстрого доступа сомнительна.)

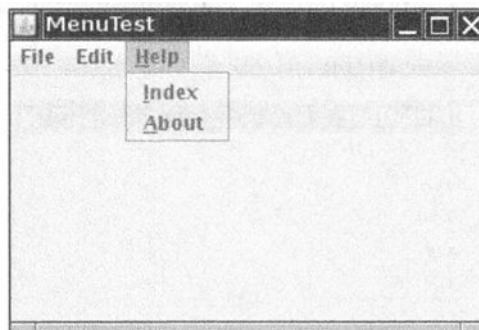


Рис. 11.19. Пункты меню, для которых назначены клавиши быстрого доступа

Задавая клавишу быстрого доступа, не всегда целесообразно выделять первое вхождение соответствующей буквы в названии пункта меню. Так, если для пункта `Save As` (Сохранить) назначена клавиша `<A>`, то гораздо уместнее выделить подчеркиванием букву `A` в слове `As` (`Save As`). Чтобы указать подчеркиваемый символ, следует вызвать метод `setDisplayMnemonicIndex()`. А имея в своем распоряжении объект типа `Action`, можно назначить клавишу быстрого доступа, указав нужное значение в поле `Action.MNEMONIC_KEY` следующим образом:

```
cutAction.putValue(Action.MNEMONIC_KEY, new Integer('A'));
```

Букву, соответствующую клавише быстрого доступа, следует задавать только в конструкторе пункта меню (но не в конструкторе всего меню). Чтобы связать какую-нибудь клавишу с меню в целом, следует вызвать метод `setMnemonic()`:

```
var helpMenu = new JMenu("Help");  
helpMenu.setMnemonic('H');
```

Теперь, чтобы сделать выбор из строки меню, достаточно нажать клавишу `<Alt>` вместе с клавишей назначенной буквы. Например, чтобы выбрать меню Help (Справка), следует нажать комбинацию клавиш `<Alt+H>`.

Клавиши быстрого доступа позволяют выбрать пункт в уже открытом меню. С другой стороны, *оперативные клавиши* позволяют выбрать пункт, не открывая меню. Например, во многих прикладных программах предусмотрены комбинации клавиш `<Ctrl+O>` и `<Ctrl+S>` для пунктов Open (Открыть) и Save (Сохранить) меню File (Файл). Для связывания оперативных клавиш с пунктом меню служит метод `setAccelerator()`. В качестве параметра он получает объект типа `KeyStroke`. Например, в приведенном ниже вызове комбинация оперативных клавиш `<Ctrl+O>` назначается для пункта меню `openItem`.

```
openItem.setAccelerator(KeyStroke.getKeyStroke("ctrl O"));
```

При нажатии оперативных клавиш автоматически выбирается соответствующий пункт меню и событие инициируется таким же образом, как и при выборе пункта меню обычным способом. Оперативные клавиши можно связывать только с пунктами меню, но не с меню в целом. Они не открывают меню, а только инициируют событие, связанное с указанным пунктом меню.

В принципе оперативные клавиши связываются с пунктами меню таким же образом, как и с остальными компонентами из библиотеки Swing. Но если оперативные клавиши назначены для пункта меню, то соответствующая комбинация клавиш автоматически отображается в меню (рис. 11.20).

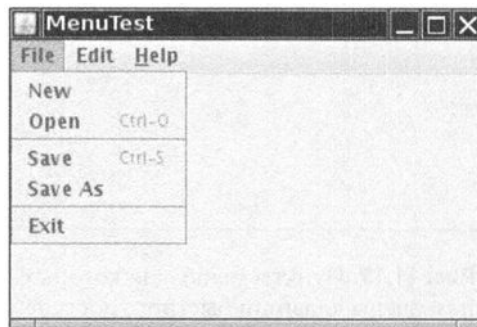


Рис. 11.20. Отображение оперативных клавиш в пунктах меню



НА ЗАМЕТКУ! При нажатии комбинации клавиш `<Alt+F4>` в Windows закрывается текущее окно. Но эти оперативные клавиши не могут быть переназначены средствами Java. Они определены в данной операционной системе и всегда инициируют событие `WindowClosing` для активного окна, независимо от того, имеется ли в меню пункт Close.

`javax.swing.JMenuItem 1.2`

- **`JMenuItem(String label, int mnemonic)`**
Создает пункт меню с указанной меткой и клавишей быстрого доступа.
- **`void setAccelerator(KeyStroke k)`**
Задаёт оперативную клавишу *k* для данного пункта меню. Соответствующая клавиша отображается в меню рядом с меткой данного пункта.

`javax.swing.AbstractButton 1.2`

- **`void setMnemonic(int mnemonic)`**
- Задаёт символ, мнемонически обозначающий клавишу быстрого доступа к экранной кнопке. В метке кнопки этот символ подчеркивается.
- **`void setDisplayedMnemonicIndex(int index) 1.4`**
- Задаёт расположение подчеркиваемого символа. Вызывается в том случае, если выделять первое вхождение символа, мнемонически обозначающего клавишу быстрого доступа, нецелесообразно.

11.5.6. Разрешение и запрет доступа к пунктам меню

Иногда некоторые пункты меню должны выбираться лишь в определенном контексте. Так, если документ открыт лишь для чтения, то пункт меню `Save` не имеет смысла. Разумеется, этот пункт можно удалить методом `JMenu.remove()`, но пользователя может удивить постоянно изменяющееся меню. Поэтому лучше всего запретить доступ к некоторым пунктам меню, временно лишив пользователя возможности выполнять соответствующие команды и операции. На рис. 11.21 запрещенный пункт меню выделен светло-серым цветом как недоступный.

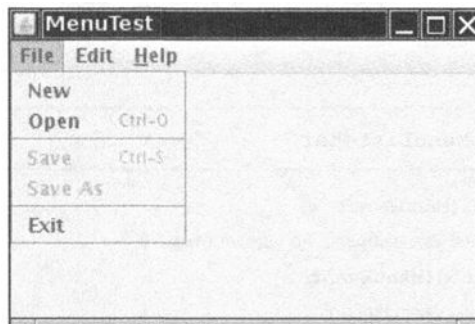


Рис. 11.21. Пункты меню, запрещенные для доступа

С целью разрешить или запретить доступ к пунктам меню вызывается метод `setEnabled()`:

```
saveItem.setEnabled(false);
```

Имеются две методики разрешения и запрета доступа к пунктам меню. При всяком изменении состояния программы можно вызывать метод `setEnabled()`

для соответствующего пункта меню. Например, открыв документ только для чтения, можно сделать недоступными пункты меню Save и Save As. С другой стороны, можно сделать недоступными пункты меню непосредственно перед их отображением. Для этого нужно зарегистрировать обработчик событий, связанный с выбором меню. В состав пакета `javax.swing.event` входит интерфейс `MenuListener`, в котором объявлены три метода:

```
void menuSelected(MenuEvent event)
void menuDeselected(MenuEvent event)
void menuCanceled(MenuEvent event)
```

Метод `menuSelected()` вызывается до отображения меню. Это самый подходящий момент для того, чтобы разрешить или запретить доступ к пунктам меню. В приведенном ниже фрагменте кода показано, как пункты меню Save и Save As делаются доступными и недоступными в зависимости от состояния флажка Read Only (Только для чтения).

```
public void menuSelected(MenuEvent event)
{
    saveAction.setEnabled(!readonlyItem.isSelected());
    saveAsAction.setEnabled(!readonlyItem.isSelected());
}
```



ВНИМАНИЕ! Запрещать доступ к пунктам меню непосредственно перед их отображением вполне разумно, но такая методика не подходит для пунктов меню, имеющих назначенные для них оперативные клавиши. При нажатии оперативной клавиши меню вообще не открывается, поэтому доступ к выбираемому пункту меню не запрещается, а следовательно, инициируется выполнение соответствующей команды.

`javax.swing.JMenuItem` 1.2

- `void setEnabled(boolean b)`
Разрешает и запрещает доступ к пункту меню.

`javax.swing.event.MenuListener` 1.2

- `void menuSelected(MenuEvent e)`
Вызывается, когда меню уже выбрано, но еще не открыто.
- `void menuDeselected(MenuEvent e)`
Вызывается, когда меню уже закрыто.
- `void menuCanceled(MenuEvent e)`
Вызывается, когда обращение к меню отменено; если, например, пользователь щелкнет кнопкой мыши за пределами меню.

В листинге 11.6 приведен исходный код программы, где формируется ряд меню. На примере данной программы демонстрируются все особенности меню, описанные в этом разделе: вложенные меню, недоступные пункты меню, флажки

и кнопки-переключатели в пунктах меню, а также клавиши быстрого доступа и оперативные клавиши выбора пунктов меню.

Листинг 11.6. Исходный код из файла `menu/MenuFrame.java`

```
1  package menu;
2
3  import java.awt.event.*;
4  import javax.swing.*;
5
6  /**
7   * Фрейм с образцом строки меню
8   */
9  public class MenuFrame extends JFrame
10 {
11     private static final int DEFAULT_WIDTH = 300;
12     private static final int DEFAULT_HEIGHT = 200;
13     private Action saveAction;
14     private Action saveAsAction;
15     private JCheckBoxMenuItem readonlyItem;
16     private JPopupMenu popup;
17
18     /**
19      * Обработчик действий, выводящий имя действия
20      * в стандартный поток System.out
21      */
22     class TestAction extends AbstractAction
23     {
24         public TestAction(String name)
25         {
26             super(name);
27         }
28
29         public void actionPerformed(ActionEvent event)
30         {
31             System.out.println(getValue(Action.NAME)
32                               + " selected.");
33         }
34     }
35
36     public MenuFrame()
37     {
38         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
39
40         var = new JMenu("File");
41         fileMenu.add(new TestAction("New"));
42
43         // продемонстрировать применение оперативных клавиш
44
45         var openItem = fileMenu.add(
46             new TestAction("Open"));
47         openItem.setAccelerator(
48             KeyStroke.getKeyStroke("ctrl O"));
49
50         fileMenu.addSeparator();
```

```
51
52     saveAction = new TestAction("Save");
53     JMenuItem saveItem = fileMenu.add(saveAction);
54     saveItem.setAccelerator(
55         KeyStroke.getKeyStroke("ctrl S"));
56
57     saveAsAction = new TestAction("Save As");
58     fileMenu.add(saveAsAction);
59     fileMenu.addSeparator();
60
61     fileMenu.add(new AbstractAction("Exit")
62     {
63         public void actionPerformed(ActionEvent event)
64         {
65             System.exit(0);
66         }
67     });
68
69     // продемонстрировать применение флажков
70     // и кнопок-переключателей
71
72     readonlyItem = new JCheckBoxMenuItem("Read-only");
73     readonlyItem.addActionListener(new ActionListener()
74     {
75         public void actionPerformed(ActionEvent event)
76         {
77             boolean saveOk = !readonlyItem.isSelected();
78             saveAction.setEnabled(saveOk);
79             saveAsAction.setEnabled(saveOk);
80         }
81     });
82
83     var group = new ButtonGroup();
84
85     var insertItem = new JRadioButtonMenuItem("Insert");
86     insertItem.setSelected(true);
87     var overtypeItem =
88         new JRadioButtonMenuItem("Overtype");
89
90     group.add(insertItem);
91     group.add(overtypeItem);
92
93     // продемонстрировать применение пиктограмм
94
95     var cutAction = new TestAction("Cut");
96     cutAction.putValue(Action.SMALL_ICON,
97         new ImageIcon("cut.gif"));
98     var copyAction = new TestAction("Copy");
99     copyAction.putValue(Action.SMALL_ICON,
100         new ImageIcon("copy.gif"));
101     Action pasteAction = new TestAction("Paste");
102     pasteAction.putValue(Action.SMALL_ICON,
103         new ImageIcon("paste.gif"));
104
105     var editMenu = new JMenu("Edit");
106     editMenu.add(cutAction);
```

```
107     editMenu.add(copyAction);
108     editMenu.add(pasteAction);
109
110     // продемонстрировать применение вложенных меню
111
112     var optionMenu = new JMenu("Options");
113
114     optionMenu.add(readonlyItem);
115     optionMenu.addSeparator();
116     optionMenu.add(insertItem);
117     optionMenu.add(overtypItem);
118
119     editMenu.addSeparator();
120     editMenu.add(optionMenu);
121
122     // продемонстрировать применение
123     // клавиш быстрого доступа
124
125     var helpMenu = new JMenu("Help");
126     helpMenu.setMnemonic('H');
127
128     JMenuItem indexItem = new JMenuItem("Index");
129     indexItem.setMnemonic('I');
130     helpMenu.add(indexItem);
131
132     // назначить клавишу быстрого доступа,
133     // используя объект действия
134     var aboutAction = new TestAction("About");
135     aboutAction.putValue(Action.MNEMONIC_KEY,
136                          new Integer('A'));
137     helpMenu.add(aboutAction);
138
139     // ввести все меню верхнего уровня в строку меню
140
141
142     var menuBar = new JMenuBar();
143     setJMenuBar(menuBar);
144
145     menuBar.add(fileMenu);
146     menuBar.add(editMenu);
147     menuBar.add(helpMenu);
148
149     // продемонстрировать применение всплывающих меню
150
151     popup = new JPopupMenu();
152     popup.add(cutAction);
153     popup.add(copyAction);
154     popup.add(pasteAction);
155
156     var panel = new JPanel();
157     panel.setComponentPopupMenu(popup);
158     add(panel);
159
160 }
161 }
```


11.5.7. Панели инструментов

Панель инструментов состоит из ряда экранных кнопок, обеспечивающих быстрый доступ к наиболее часто используемым командам (рис. 11.22).

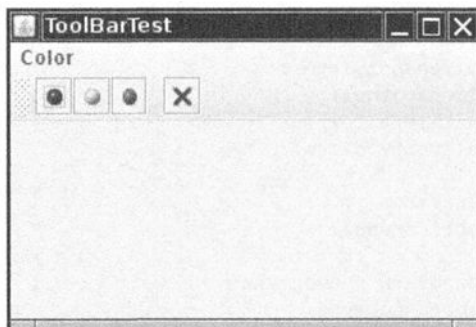


Рис. 11.22. Панель инструментов

Панель инструментов отличается от остальных элементов пользовательского интерфейса тем, что ее можно перетаскивать на любую из четырех сторон фрейма (рис. 11.23). При отпускании кнопки мыши панель инструментов фиксируется на новом месте (рис. 11.24).

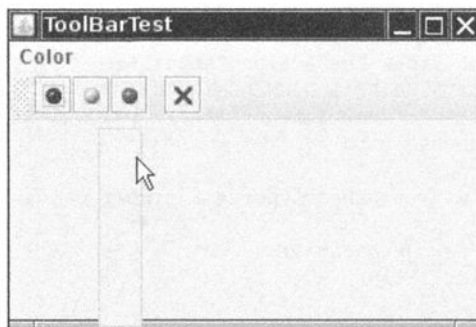


Рис. 11.23. Перетаскивание панели инструментов

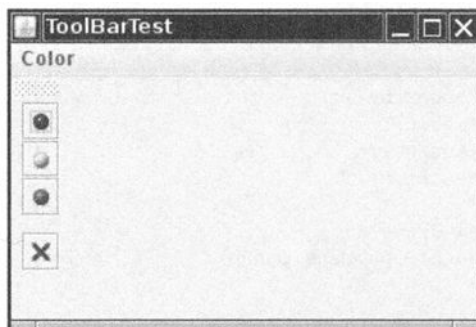


Рис. 11.24. Новое местоположение панели инструментов после перетаскивания



НА ЗАМЕТКУ! Перетаскивание панели инструментов допускается только в том случае, если она размещается в контейнере диспетчером граничной компоновки или любым другим диспетчером, поддерживающим расположение компонентов в северной, южной, восточной и западной областях фрейма.

Панель инструментов может быть даже обособленной от фрейма. Такая панель содержится в собственном фрейме (рис. 11.25). Если пользователь закрывает фрейм, содержащий обособленную панель инструментов, она перемещается в исходный фрейм.

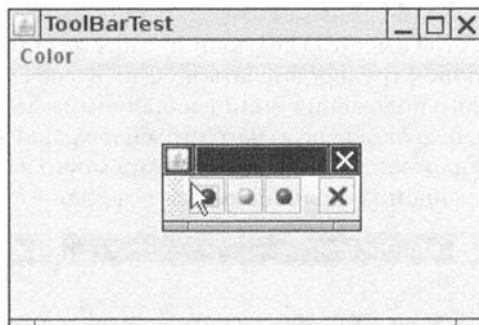


Рис. 11.25. Обособленная панель инструментов

Панель инструментов легко запрограммировать. Ниже приведен пример создания панели и добавления к ней компонента.

```
var bar = new JToolBar();  
toolbar.add(blueButton);
```

В классе `JToolBar` имеются также методы, предназначенные для ввода действия, представленного объектом типа `Action`. Для этого достаточно заполнить панель инструментов объектами типа `Action`, как показано ниже. Пиктограмма, соответствующая такому объекту, отображается на панели инструментов.

```
toolbar.add(blueAction);
```

Группы кнопок можно отделять друг от друга с помощью разделителей следующим образом:

```
toolbar.addSeparator();
```

Так, на панели инструментов, показанной на рис. 11.22, имеется разделитель третьей кнопки от четвертой. Обычно панель инструментов размещается в контейнере, как показано ниже.

```
add(toolbar, BorderLayout.NORTH);
```

Имеется также возможность указать заголовок панели, который появится, когда панель будет обособлена от фрейма. Ниже показано, каким образом заголовок панели указывается в коде.

```
toolbar = new JToolBar(titleString);
```

По умолчанию панель инструментов располагается горизонтально. А для того чтобы расположить ее вертикально, достаточно написать одну из следующих двух строк кода:

```
toolbar = new JToolBar(SwingConstants.VERTICAL)
```

или

```
toolbar = new JToolBar(titleString, SwingConstants.VERTICAL)
```

Чаще всего на панели инструментов располагаются экранные кнопки. Но никаких ограничений на вид компонентов, которые можно размещать на панели инструментов, не существует. Например, на панели инструментов можно расположить и комбинированный список.

11.5.8. Всплывающие подсказки

У панелей инструментов имеется следующий существенный недостаток: по внешнему виду экранных кнопок трудно догадаться об их назначении. В качестве выхода из этого затруднительного положения были изобретены *всплывающие подсказки*, которые появляются на экране, когда курсор наводится на экранную кнопку. Текст всплывающей подсказки отображается в закрашенном прямоугольнике (рис. 11.26). Когда курсор отводится от экранной кнопки, подсказка исчезает.

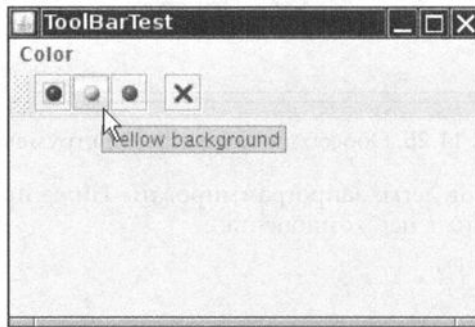


Рис. 11.26. Всплывающая подсказка

В библиотеке Swing допускается вводить всплывающие подсказки в любой объект типа `JComponent`, просто вызывая метод `setToolTipText()`:

```
exitButton.setToolTipText("Exit");
```

А если воспользоваться объектами типа `Action`, то всплывающая подсказка связывается с ключом `SHORT_DESCRIPTION` следующим образом:

```
exitAction.putValue(Action.SHORT_DESCRIPTION, "Exit");
```

javax.swing.JToolBar 1.2

- `JToolBar()`
- `JToolBar(String titleString)`
- `JToolBar(int orientation)`
- `JToolBar(String titleString, int orientation)`

Создают панель инструментов с заданной строкой заголовка и ориентацией. Параметр `orientation` может принимать значения констант `SwingConstants.HORIZONTAL` (по умолчанию) и `SwingConstants.VERTICAL`.

javax.swing.JToolBar 1.2 (окончание)

- **JButton add(Action a)**

Создает новую экранную кнопку на панели инструментов с именем, кратким описанием, пиктограммой и обратным вызовом действия. Экранная кнопка вводится в конце панели инструментов.

- **void addSeparator()**

Вводит разделитель в конце панели инструментов.

javax.swing.JComponent 1.2

- **void setToolTipText(String text)**

Задаёт текст для вывода во всплывающей подсказке, когда курсор мыши наводится на компонент.

11.6. Расширенные средства компоновки

В рассматривавшихся до сих пор примерах создания пользовательского интерфейса применялись только диспетчеры граничной, поточной и сеточной компоновки. Но для решения более сложных задач компоновки GUI этого явно недостаточно. В этом разделе будут подробно рассмотрены расширенные средства компоновки GUI.

Начиная с версии Java 1.0, в состав библиотеки AWT входят средства *сеточно-контейнерной компоновки* для расположения компонентов по рядам и столбцам. Размеры ряда и столбца допускают гибкую установку, а компоненты могут занимать несколько рядов и столбцов. Такой диспетчер компоновки действует очень гибко, но в то же время он довольно сложен, причем настолько, что само словосочетание “сеточно-контейнерная компоновка” способно вызвать невольный трепет у программистов на Java.

Безуспешные попытки разработать диспетчер компоновки, который избавил бы программистов от тирании сеточно-контейнерной компоновки, навели разработчиков библиотеки Swing на мысль о *блочной компоновке*. Как поясняется в документации на JDK, класс `BoxLayout` диспетчера блочной компоновки “осуществляет вложение многих панелей с горизонтальными и вертикальными размерами в разных сочетаниях, достигая такого же результата, как и класс `GridLayout` диспетчера сеточной компоновки, но без сложностей, присущих последней”. Но поскольку каждый компонуемый блок располагается независимо от других блоков, блочная компоновка не подходит для упорядочения соседних компонентов как по вертикали, так и по горизонтали.

В версии Java 1.4 была предпринята еще одна попытка найти замену сеточно-контейнерной компоновке так называемой *пружинной компоновкой*. В соответствии с этой разновидностью компоновки для соединения отдельных компонентов в контейнере используются воображаемые пружины. При изменении размеров контейнера эти пружины растягиваются и сжимаются, регулируя таким образом расположение компонентов. На практике такой подход к компоновке оказался слишком трудоемким и запутанным, поэтому пружинная компоновка быстро канула в небытие.

В IDE NetBeans внедрена технология Matisse, сочетающая в себе инструмент и диспетчер компоновки (теперь она называется Swing GUI Builder). Разработчики пользовательского интерфейса применяют инструментальное средство Swing GUI Builder, чтобы разместить компоненты в контейнере и указать те компоненты, по которым они должны быть выровнены. А инструмент переводит замысел разработчика в инструкции для диспетчера *групповой компоновки*. Это намного удобнее, чем написание кода управления компоновкой вручную.

В последующих разделах речь пойдет о диспетчере сеточно-контейнерной компоновки, потому что он применяется довольно широко и все еще является самым простым механизмом генерирования кода компоновки для прежних версий Java. Попутно будет представлена методика, благодаря которой применение этого диспетчера компоновки может стать сравнительно безболезненным в типичных случаях.

И в завершение темы диспетчеров компоновки будет показано, как вообще обойтись без них, располагая компоненты GUI вручную, и как написать свой собственный диспетчер компоновки.

11.6.1. Диспетчер сеточно-контейнерной компоновки

Диспетчер сеточно-контейнерной компоновки — предшественник всех остальных диспетчеров компоновки. Его можно рассматривать как диспетчер сеточной компоновки без ограничений, т.е. при сеточно-контейнерной компоновке ряды и столбцы могут иметь переменные размеры. Чтобы расположить крупный компонент, который не умещается в одной ячейке, несколько смежных ячеек можно соединить вместе. (Многие редакторы текста и HTML-документов предоставляют такие же возможности для построения таблиц: заполнение начинается с обычной сетки, а при необходимости некоторые ее ячейки соединяются вместе.) Компоненты совсем не обязательно должны заполнять всю ячейку, поэтому можно задать выравнивание в самих ячейках.

Рассмотрим в качестве примера диалоговое окно селектора шрифтов (рис. 11.27), состоящее из следующих компонентов.

- Два комбинированных списка для выбора начертания и размера шрифта.
- Метки для обоих комбинированных списков.
- Два флажка для выбора полужирного и наклонного начертания шрифта.
- Текстовая область для отображения образца текстовой строки.

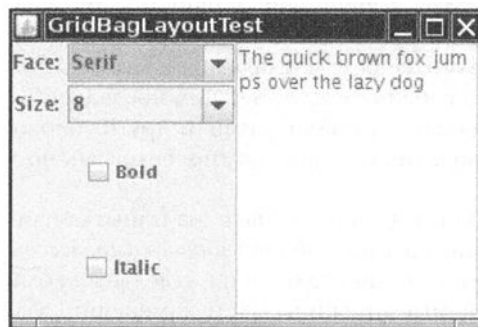


Рис. 11.27. Селектор шрифтов

А теперь разделим диалоговое окно как контейнер на ячейки в соответствии с рис. 11.28. (Ряды и столбцы совсем не обязательно должны иметь одинаковые размеры.) Как видите, каждый флажок занимает два столбца, а текстовая область — четыре ряда.



Рис. 11.28. Сетка разбиения диалогового окна, используемая для компоновки

Для описания компоновки, которое было бы понятно диспетчеру сеточно-контейнерной компоновки, выполните следующие действия.

1. Создайте объект типа `GridBagLayout`. Конструктору совсем не обязательно знать, из какого числа рядов и столбцов состоит сетка. Он попытается впоследствии сам уточнить эти параметры по сведениям, полученным от вас.
2. Установите объект типа `GridLayout` в качестве диспетчера компоновки для данного компонента.
3. Создайте для каждого компонента объект типа `GridBagConstraints`. Установите соответствующие значения в полях этого объекта, чтобы задать расположение компонентов в сеточном контейнере.
4. Введите каждый компонент с ограничениями, сделав следующий вызов:

```
add(Component, constraints);
```

Ниже приведен пример кода, в котором реализуются описанные выше действия. (Ограничения будут подробнее рассмотрены далее, а до тех пор не следует особенно беспокоиться об их назначении.)

```
var layout = new GridBagLayout();
panel.setLayout(layout);
var constraints = new GridBagConstraints();
constraints.weightx = 100;
constraints.weighty = 100;
constraints.gridx = 0;
constraints.gridy = 2;
constraints.gridwidth = 2;
constraints.gridheight = 1;
panel.add(component, constraints);
```

Самое главное — правильно установить состояние объекта типа `GridBagConstraints`. Поэтому в следующих далее подразделах поясняется, как пользоваться этим объектом.

11.6.1.1. Параметры `gridx`, `gridy`, `gridwidth` и `gridheight`

Эти параметры определяют место компонента в сетке компоновки. В частности, параметры `gridx` и `gridy` задают столбец и ряд для местоположения левого верхнего угла компонента. А параметры `gridwidth` и `gridheight` определяют число рядов и столбцов, занимаемых данным компонентом.

Координаты сетки отсчитываются от нуля. В частности, выражения `gridx=0` и `gridy=0` обозначают левый верхний угол. Например, местоположение текстовой области в рассматриваемом здесь примере определяется параметрами `gridx=2` и `gridy=0`, поскольку она начинается со столбца под номером 2 (т.е. с третьего столбца). А если текстовая область занимает один столбец и четыре ряда, то `gridwidth=1`, а `gridheight=4`.

11.6.1.2. Весовые поля

Для каждой области сеточно-контейнерной компоновки следует задать так называемые *весовые поля*, определяемые параметрами `weightx` и `weighty`. Если вес равен нулю, то область всегда сохраняет свои первоначальные размеры. В примере, приведенном на рис. 11.27, для текстовых меток установлено нулевое значение параметра `weightx`. Это позволяет сохранять постоянную ширину меток при изменении размеров окна. С другой стороны, если задать нулевой вес всех областей, контейнер расположит компоненты в центре выделенной для него области, не заполнив до конца все ее пространство.

Загруднение, возникающее при установке параметров весовых полей, состоит в том, что они являются свойствами рядов и столбцов, а не отдельных ячеек. Но их нужно задавать для ячеек, поскольку диспетчер сеточно-контейнерной компоновки не различает отдельные ряды и столбцы. В качестве веса ряда или столбца принимается максимальный вес среди всех содержащихся в них ячеек. Следовательно, если требуется сохранить фиксированными размеры рядов и столбцов, необходимо установить нулевым вес всех компонентов в этих рядах и столбцах.

Но на самом деле вес не позволяет определить относительные размеры столбцов. Он лишь указывает на ту часть свободного пространства, которая должна быть выделена для каждой области, если контейнер превышает рекомендуемые размеры. Чтобы научиться правильно подбирать вес, нужно иметь определенный опыт работы с рассматриваемым здесь диспетчером компоновки. Поэтому для начала рекомендуется поступать следующим образом. Установите вес равным 100, затем запустите свою программу на выполнение и посмотрите, как будет выглядеть пользовательский интерфейс. Чтобы выяснить, насколько правильно выравниваются ряды и столбцы, попробуйте изменить размеры окна. Если окажется, что какой-то ряд или столбец не должен увеличиваться, установите нулевой вес всех находящихся в нем компонентов. Можно, конечно, опробовать и другие весовые значения, но овчинка, как правило, выделки не стоит.

11.6.1.3. Параметры `fill` и `anchor`

Если требуется, чтобы компонент не растягивался и не заполнял все доступное пространство, на него следует наложить ограничение с помощью параметра `fill`. Этот параметр принимает значение одной из следующих констант: `GridBagConstraints.NONE`, `GridBagConstraints.HORIZONTAL`, `GridBagConstraints.VERTICAL` или `GridBagConstraints.BOTH`.

Если компонент не заполняет все доступное пространство, можно указать область, к которой его следует привязать, установив параметр `anchor`. Этот параметр может принимать значение одной из следующих констант: `GridBagConstraints.CENTER` (по умолчанию), `GridBagConstraints.NORTH`, `GridBagConstraints.NORTHEAST`, `GridBagConstraints.EAST` и т.д.

11.6.1.4. Заполнение пустого пространства

Установив соответствующее значение в поле `insets` объекта типа `GridBagConstraints`, можно ввести дополнительное пустое пространство вокруг компонента. А если требуется задать конкретные пределы пустого пространства вокруг компонента, то следует установить соответствующие значения в полях `left`, `top`, `right` и `bottom` объекта типа `Insets`. Этот процесс называется *внешним заполнением*.

Значения, устанавливаемые в полях `ipadx` и `ipady`, определяют *внутреннее заполнение*. Эти значения добавляются к минимальным ширине и высоте компонента, гарантируя тем самым, что компонент не уменьшится до своих минимальных размеров.

11.6.1.5. Альтернативные способы установки параметров `gridx`, `gridy`, `gridwidth` и `gridheight`

В документации на библиотеку AWT рекомендуется не задавать абсолютные значения параметров `gridx` и `gridy`, а использовать константу `GridBagConstraints.RELATIVE`. Затем компоненты нужно расположить, как обычно для сеточно-контейнерной компоновки: последовательными рядами слева направо.

Количество рядов и столбцов, занятых ячейкой, как правило, указывается в полях `gridheight` и `gridwidth`. Исключение из этого правила составляет компонент, занимающий последний ряд или столбец. Для него указывается не числовое значение, а специальная константа `GridBagConstraints.REMAINDER`. Этим диспетчер компоновки уведомляется, что данный компонент является последним в ряду.

Описанная выше схема компоновки выглядит вполне работоспособной. Но в то же время кажется довольно странным сначала скрывать от диспетчера компоновки сведения о фактическом расположении компонентов, а затем полагаться на то, что он сам угадает правильные параметры.

11.6.1.6. Рекомендации для сеточно-контейнерной компоновки

Следуя приведенным ниже рекомендациям, можно относительно просто выполнять сеточно-контейнерную компоновку компонентов GUI.

1. Набросайте схему расположения компонентов на бумаге.
2. Подберите такую сетку, чтобы мелкие компоненты умещались в отдельных ячейках, а крупные — в нескольких ячейках.
3. Пометьте ряды и столбцы сетки номерами 0, 1, 2, 3... После этого определите значения параметров `gridx`, `gridy`, `gridwidth` и `gridheight`.
4. Выясните для каждого компонента, должен ли он заполнять ячейку по горизонтали или по вертикали? Если не должен, то как его выровнять в ячейке? Для этой цели служат параметры `fill` и `anchor`.
5. Установите вес равным 100. Но если требуется, чтобы отдельный ряд или столбец всегда сохранял свои первоначальные размеры, задайте нулевое значение

параметров `weightx` или `weighty` всех компонентов, находящихся в данном ряду или столбце.

6. Напишите исходный код. Тщательно проверьте ограничения, накладываемые в классе `GridBagConstraints`. Одно неверное ограничение может нарушить всю компоновку.
7. Скомпилируйте исходный код, запустите его на выполнение и пользуйтесь им в свое удовольствие.

11.6.1.7. Вспомогательный класс для наложения ограничений на сеточно-контейнерную компоновку

Самая трудоемкая стадия сеточно-контейнерной компоновки относится к написанию кода, накладывающего ограничения. Многие программисты создают для этой цели вспомогательные функции или небольшие вспомогательные классы. Код одного из таких классов приведен после примера программы, где демонстрируется создание диалогового окна селектора шрифтов. Ниже перечислены характеристики, которыми должен обладать такой вспомогательный класс.

- Имеет имя `GBC` (прописные буквы из имени класса `GridBagConstraints`).
- Расширяет класс `GridBagConstraints`, поэтому константы можно указывать, используя более короткое имя, например `GBC.EAST`.
- Объект типа `GBC` используется при вводе компонента следующим образом:
`add(component, new GBC(1, 2));`
- Предусматривает два конструктора для установки наиболее употребительных параметров `gridx` и `gridy`, `gridx` и `gridy`, `gridwidth` и `gridheight`, как, например:
`add(component, new GBC(1, 2, 1, 4));`
- Предусматривает удобные методы для установки в полях пар значений координат `x` и `y`, как показано ниже.
`add(component, new GBC(1, 2).setWeight(100, 100));`
- Предусматривает методы для установки значений в полях и возврата ссылки `this`, чтобы объединять их в цепочки следующим образом:
`add(component, new GBC(1, 2).setAnchor(GBC.EAST).setWeight(100, 100));`
- Предусматривает метод `setInsets()` для построения объектов типа `Insets`. Так, если требуется создать пустое пространство размером в один пиксель, следует написать такую строку кода:
`add(component, new GBC(1, 2).setAnchor(GBC.EAST).setInsets(1));`

В листинге 11.7 представлен исходный код класса для рассматриваемого здесь примера диалогового окна селектора шрифтов, а в листинге 11.8 — исходный код вспомогательного класса `GBC`. Ниже приведен фрагмент кода для ввода компонентов в сеточный контейнер.

```
add(faceLabel, new GBC(0,0).setAnchor(GBC.EAST));
add(face, new GBC(1,0).setFill(GBC.HORIZONTAL)
    .setWeight(100,0).setInsets(1));
```

```

add(sizeLabel, new GBC(0, 1).setAnchor(GBC.EAST));
add(size, new GBC(1,1).setFill(GBC.HORIZONTAL)
    .setWeight(100,0).setInsets(1));
add(bold, new GBC(0, 2, 2, 1).setAnchor(GBC.CENTER)
    .setWeight(100, 100));
add(italic, new GBC(0, 3, 2, 1).setAnchor(GBC.CENTER)
    .setWeight(100, 100));
add(sample, new GBC(2, 0, 1, 4).setFill(GBC.BOTH)
    .setWeight(100, 100));

```

Уяснив принцип наложения ограничений при сеточно-контейнерной компоновке, разобраться в этом коде и отладить его будет несложно.

Листинг 11.7. Исходный код из файла `gridbag/FontFrame.java`

```

1 package gridbag;
2
3 import java.awt.Font;
4 import java.awt.GridBagLayout;
5 import java.awt.event.ActionListener;
6
7 import javax.swing.BorderFactory;
8 import javax.swing.JCheckBox;
9 import javax.swing.JComboBox;
10 import javax.swing.JFrame;
11 import javax.swing.JLabel;
12 import javax.swing.JTextArea;
13
14 /**
15  * Фрейм, в котором сеточно-контейнерная компоновка
16  * служит для расположения компонентов, предназначенных
17  * для выбора шрифтов
18  */
19 public class FontFrame extends JFrame
20 {
21     public static final int TEXT_ROWS = 10;
22     public static final int TEXT_COLUMNS = 20;
23
24     private JComboBox<String> face;
25     private JComboBox<Integer> size;
26     private JCheckBox bold;
27     private JCheckBox italic;
28     private JTextArea sample;
29
30     public FontFrame()
31     {
32         GridBagLayout layout = new GridBagLayout();
33         setLayout(layout);
34
35         ActionListener listener = event -> updateSample();
36
37         // сконструировать компоненты
38
39         var faceLabel = new JLabel("Face: ");
40
41         face = new JComboBox<>(new String[] {
42             "Serif", "SansSerif", "Monospaced",

```

```

43         "Dialog", "DialogInput" });
44
45     face.addActionListener(listener);
46
47     var sizeLabel = new JLabel("Size: ");
48
49     size = new JComboBox<>(new Integer[]
50         { 8, 10, 12, 15, 18, 24, 36, 48 });
49
50     size.addActionListener(listener);
51
52     bold = new JCheckBox("Bold");
53     bold.addActionListener(listener);
54
55     italic = new JCheckBox("Italic");
56     italic.addActionListener(listener);
57
58     sample = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
59     sample.setText("The quick brown fox jumps "
60         + "over the lazy dog");
61     sample.setEditable(false);
62     sample.setLineWrap(true);
63     sample.setBorder(
64         BorderFactory.createEtchedBorder());
65
66     // ввести компоненты в сетку,
67     // используя служебный класс GBC
68
69     add(faceLabel, new GBC(0, 0).setAnchor(GBC.EAST));
70     add(face, new GBC(1, 0).setFill(GBC.HORIZONTAL)
71         .setWeight(100, 0).setInsets(1));
72     add(sizeLabel, new GBC(0, 1).setAnchor(GBC.EAST));
73     add(size, new GBC(1, 1).setFill(GBC.HORIZONTAL)
74         .setWeight(100, 0).setInsets(1));
75     add(bold, new GBC(0, 2, 2, 1).setAnchor(GBC.CENTER)
76         .setWeight(100, 100));
77     add(italic, new GBC(0, 3, 2, 1).setAnchor(GBC.CENTER)
78         .setWeight(100, 100));
79     add(sample, new GBC(2, 0, 1, 4).setFill(GBC.BOTH)
80         .setWeight(100, 100));
81     pack();
82     updateSample();
83 }
84
85 public void updateSample()
86 {
87     var fontFace = (String) face.getSelectedItem();
88     int fontStyle = (bold.isSelected() ? Font.BOLD : 0)
89         + (italic.isSelected() ? Font.ITALIC : 0);
90     int fontSize = size.getItemAt(
91         size.getSelectedIndex());
92     var font = new Font(fontFace, fontStyle, fontSize);
93     sample.setFont(font);
94     sample.repaint();
95 }
96 }

```

Листинг 11.8. Исходный код из файла `gridbag/GBC.java`

```
1  package gridbag;
2
3  import java.awt.*;
4
5  /**
6   * Этот класс упрощает применение
7   * класса GridBagConstraints
8   * @version 1.01 2004-05-06
9   * @author Cay Horstmann
10  */
11  public class GBC extends GridBagConstraints
12  {
13      /**
14       * Строит объект типа GBC, накладывая ограничения
15       * с помощью параметров gridx и gridy, а все остальные
16       * ограничения накладываются на сеточно-контейнерную
17       * компоновку по умолчанию
18       * @param gridx Местоположение в сетке по горизонтали
19       * @param gridy Местоположение в сетке по вертикали
20       */
21      public GBC(int gridx, int gridy)
22      {
23          this.gridx = gridx;
24          this.gridy = gridy;
25      }
26
27      /**
28       * Строит объект типа GBC, накладывая ограничения
29       * с помощью параметров gridx, gridy, gridwidth,
30       * gridheight, а все остальные ограничения
31       * накладываются на сеточно-контейнерную
32       * компоновку по умолчанию
33       * @param gridx Местоположение в сетке по горизонтали
34       * @param gridy Местоположение в сетке по вертикали
35       * @param gridwidth Шаг сетки по горизонтали
36       * @param gridheight Шаг сетки по вертикали
37       */
38      public GBC(int gridx, int gridy, int gridwidth,
39                 int gridheight)
40      {
41          this.gridx = gridx;
42          this.gridy = gridy;
43          this.gridwidth = gridwidth;
44          this.gridheight = gridheight;
45      }
46
47      /**
48       * Устанавливает привязку к сетке
49       * @param anchor Степень привязки
50       * @return this Объект для последующего видоизменения
51       */
52      public GBC setAnchor(int anchor)
53      {
54          this.anchor = anchor;
55          return this;
56      }
57  }
```

```
56     }
57
58     /**
59      * Устанавливает направление для заполнения
60      * @param fill Направление заполнения
61      * @return this Объект для последующего видоизменения
62      */
63     public GBC setFill(int fill)
64     {
65         this.fill = fill;
66         return this;
67     }
68
69     /**
70      * Устанавливает веса ячеек
71      * @param weightx Вес ячейки по горизонтали
72      * @param weighty Вес ячейки по вертикали
73      * @return this Объект для последующего видоизменения
74      */
75     public GBC setWeight(double weightx, double weighty)
76     {
77         this.weightx = weightx;
78         this.weighty = weighty;
79         return this;
80     }
81
82     /**
83      * Вводит пробелы вокруг данной ячейки
84      * @param distance Пробел по всем направлениям
85      * @return this Объект для последующего видоизменения
86      */
87     public GBC setInsets(int distance)
88     {
89         this.insets = new Insets(distance, distance,
90                                 distance, distance);
91         return this;
92     }
93
94     /**
95      * Вводит пробелы вокруг данной ячейки
96      * @param top Пробел сверху
97      * @param left Пробел слева
98      * @param bottom Пробел снизу
99      * @param right Пробел справа
100     * @return this Объект для последующего видоизменения
101     */
102     public GBC setInsets(int top, int left,
103                          int bottom, int right)
104     {
105         this.insets = new Insets(top, left, bottom, right);
106         return this;
107     }
108     /**
109      * Устанавливает внутреннее заполнение
110      * @param ipadx Внутреннее заполнение по горизонтали
111      * @param ipady Внутреннее заполнение по вертикали
112      * @return this Объект для последующего видоизменения
```

```

113  */
114  public GBC setIpad(int ipadx, int ipady)
115  {
116      this.ipadx = ipadx;
117      this.ipady = ipady;
118      return this;
119  }
120 }

```

java.awt.GridBagConstraints 1.0

- **int gridx, gridy**
Задают начальный столбец и ряд для расположения ячейки. По умолчанию принимаются нулевые значения.
- **int gridwidth, gridheight**
Задают количество столбцов и рядов, занимаемых ячейкой. По умолчанию принимают значение 1.
- **double weightx, weighty**
Определяют способность ячейки увеличиваться в размерах. По умолчанию принимают нулевое значение.
- **int anchor**
Задаёт вид выравнивания компонента в ячейке. Допускает указывать константы, определяющие абсолютное расположение.

NORTHWEST NORTH NORTHEAST
WEST CENTER EAST
SOUTHWEST SOUTH SOUTHEAST

или константы, определяющие расположение независимо от ориентации:

FIRST_LINE_START LINE_START FIRST_LINE_END
PAGE_START CENTER PAGE_END
LAST_LINE_START LINE_END LAST_LINE_END

Вторая группа констант оказывается удобной в том случае, если приложение локализуется на языки, где символы следуют справа налево или сверху вниз.

- **int fill**
Задаёт способ заполнения ячейки компонентом. Допускаются значения **NONE**, **BOTH**, **HORIZONTAL** и **VERTICAL**. По умолчанию принимается значение **NONE**.
- **int ipadx, ipady**
Задаёт внутреннее заполнение вокруг компонента. По умолчанию принимают нулевое значение.
- **Insets insets**
Задаёт внешнее заполнение вокруг границ ячейки. По умолчанию заполнение отсутствует.
- **GridBagConstraints(int gridx, int gridy, int gridwidth, int gridheight, double weightx, double weighty, int anchor, int fill, Insets insets, int ipadx, int ipady) 1.2**

Создаёт объект типа **GridBagConstraints**, заполняя все его поля указанными значениями. Этот конструктор следует использовать только в программах автоматического построения GUI, поскольку получаемый в итоге код труден для восприятия.

11.6.2. Специальные диспетчеры компоновки

В принципе можно разработать собственный класс `LayoutManager`, управляющий расположением компонентов особым образом. В качестве любопытного примера покажем, как расположить все компоненты в контейнере по кругу (рис. 11.29).

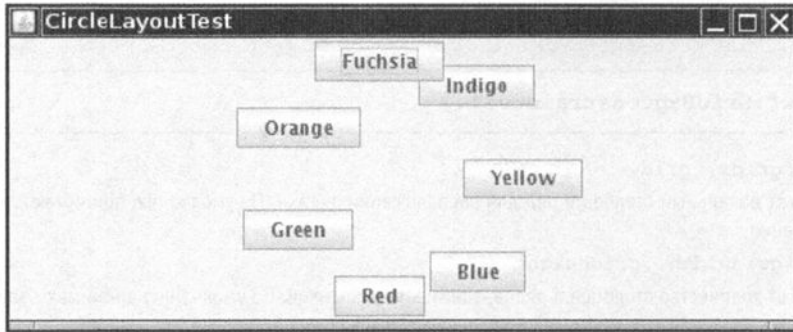


Рис. 11.29. Расположение компонентов по кругу

Класс специального диспетчера компоновки должен реализовывать интерфейс `LayoutManager`. Для этого придется переопределить следующие пять методов:

```
void addLayoutComponent(String s, Component c);  
void removeLayoutComponent(Component c);  
Dimension preferredLayoutSize(Container parent);  
Dimension minimumLayoutSize(Container parent);  
void layoutContainer(Container parent);
```

Первые два метода вызываются при добавлении или удалении компонента. Если дополнительные сведения о компоненте отсутствуют, тело этих методов можно оставить пустым. Следующие два метода вычисляют объем пространства, требующегося при минимальных и рекомендуемых размерах компонентов. Обычно эти величины равны. А пятый метод вызывает метод `setBounds()` для всех компонентов и выполняет основные операции по расположению компонента.



НА ЗАМЕТКУ! В библиотеке AWT имеется второй интерфейс под названием `LayoutManager2`. Он содержит десять, а не пять методов. Главная особенность этого интерфейса состоит в том, что с его помощью можно применять метод `add()` без всяких ограничений. Например, классы `BorderLayout` и `GridBagLayout` реализуют именно интерфейс `LayoutManager2`.

В листинге 11.9 приведен исходный код программы, реализующей специальный диспетчер компоновки типа `CircleLayout`, который располагает компоненты по кругу в контейнере. И хотя это довольно любопытная компоновка, она совершенно бесполезна. А в листинге 11.10 приведен исходный код класса фрейма, создаваемого в данном примере программы.

Листинг 11.9. Исходный код из файла `circleLayout/CircleLayout.java`

```
1 package circleLayout;  
2  
3 import java.awt.*;
```

```
4
5 /**
6  * Диспетчер компоновки, располагающий
7  * компоненты по кругу
8  */
9 public class CircleLayout implements LayoutManager
10 {
11     private int minWidth = 0;
12     private int minHeight = 0;
13     private int preferredWidth = 0;
14     private int preferredHeight = 0;
15     private boolean sizesSet = false;
16     private int maxComponentWidth = 0;
17     private int maxComponentHeight = 0;
18
19     public void addLayoutComponent(String name,
20                                     Component comp)
21     {
22     }
23     public void removeLayoutComponent(Component comp)
24     {
25     }
26
27     public void setSizes(Container parent)
28     {
29         if (sizesSet) return;
30         int n = parent.getComponentCount();
31
32         preferredWidth = 0;
33         preferredHeight = 0;
34         minWidth = 0;
35         minHeight = 0;
36         maxComponentWidth = 0;
37         maxComponentHeight = 0;
38
39         // вычислить максимальную ширину и высоту
40         // компонентов и установить предпочтительные
41         // размеры по сумме размеров компонентов
42
43         for (int i = 0; i < n; i++)
44         {
45             Component c = parent.getComponent(i);
46             if (c.isVisible())
47             {
48                 Dimension d = c.getPreferredSize();
49                 maxComponentWidth = Math.max(
50                     maxComponentWidth, d.width);
51                 maxComponentHeight = Math.max(
52                     maxComponentHeight, d.height);
53                 preferredWidth += d.width;
54                 preferredHeight += d.height;
55             }
56         }
57         minWidth = preferredWidth / 2;
58         minHeight = preferredHeight / 2;
59         sizesSet = true;
60     }
```



```
61
62 public Dimension preferredLayoutSize(Container parent)
63 {
64     setSizes(parent);
65     Insets insets = parent.getInsets();
66     int width = preferredWidth + insets.left
67         + insets.right;
68     int height = preferredHeight + insets.top
69         + insets.bottom;
70     return new Dimension(width, height);
71 }
72 public Dimension minimumLayoutSize(Container parent)
73 {
74     setSizes(parent);
75     Insets insets = parent.getInsets();
76     int width = minWidth + insets.left + insets.right;
77     int height = minHeight + insets.top + insets.bottom;
78     return new Dimension(width, height);
79 }
80
81 public void layoutContainer(Container parent)
82 {
83     setSizes(parent);
84
85     // вычислить центр круга
86
87     Insets insets = parent.getInsets();
88     int containerWidth = parent.getSize().width
89         - insets.left - insets.right;
90     int containerHeight = parent.getSize().height
91         - insets.top - insets.bottom;
92
93     int xcenter = insets.left + containerWidth / 2;
94     int ycenter = insets.top + containerHeight / 2;
95
96     // вычислить радиус окружности
97
98     int xradius = (containerWidth
99         - maxComponentWidth) / 2;
100     int yradius = (containerHeight
101         - maxComponentHeight) / 2;
102     int radius = Math.min(xradius, yradius);
103
104     // расположить компоненты по кругу
105
106     int n = parent.getComponentCount();
107     for (int i = 0; i < n; i++)
108     {
109         Component c = parent.getComponent(i);
110         if (c.isVisible())
111         {
112             double angle = 2 * Math.PI * i / n;
113
114             // центральная точка компонента
115             int x = xcenter + (int) (Math.cos(angle)
116                 * radius);
117             int y = ycenter + (int) (Math.sin(angle)
```

```

118             * radius);
119
120         // переместить компонент, расположив его
121         // в центральной точке с координатами (x,y)
122         // и предпочтительными размерами
123         Dimension d = c.getPreferredSize();
124         c.setBounds(x - d.width / 2, y - d.height / 2,
125                   d.width, d.height);
126     }
127 }
128 }
129 }

```

Листинг 11.10. Исходный код из файла circleLayout/CircleLayoutFrame.java

```

1 package circleLayout;
2
3 import javax.swing.*;
4
5 /**
6  * Фрейм, в котором демонстрируется расположение
7  * кнопок по кругу
8  */
9 public class CircleLayoutFrame extends JFrame
10 {
11     public CircleLayoutFrame()
12     {
13         setLayout(new CircleLayout());
14         add(new JButton("Yellow"));
15         add(new JButton("Blue"));
16         add(new JButton("Red"));
17         add(new JButton("Green"));
18         add(new JButton("Orange"));
19         add(new JButton("Fuchsia"));
20         add(new JButton("Indigo"));
21         pack();
22     }
23 }

```

java.awt.LayoutManager 1.0

- **void addLayoutComponent(String name, Component comp)**
Вводит компонент в текущую компоновку.
- **void removeLayoutComponent(Component comp)**
Удаляет компонент из текущей компоновки.
- **Dimension preferredLayoutSize(Container cont)**
Возвращает рекомендуемые размеры контейнера, в котором выполняется текущая компоновка.
- **void layoutContainer(Container cont)**
Располагает компоненты в контейнере.

11.7. Диалоговые окна

До сих пор рассматривались только компоненты пользовательского интерфейса, которые находились в окне фрейма, создаваемого в приложении, что характерно в основном для *апплетов*, выполняемых в окне браузера. Но при разработке приложений возникает потребность в отдельных всплывающих диалоговых окнах, которые должны появляться на экране и обеспечивать обмен данными с пользователем.

Как и в большинстве оконных систем, в AWT различаются *модальные* (т.е. режимные) и *немодальные* (т.е. безрежимные) диалоговые окна. Модальные диалоговые окна не дают пользователю возможности одновременно работать с другими окнами приложения. Такие окна требуются в том случае, если пользователь должен ввести данные, от которых зависит дальнейшая работа приложения. Например, при вводе файла пользователь должен сначала указать его имя. И только после того, как пользователь закроет модальное диалоговое окно, приложение начнет ввод файла.

Немодальное диалоговое окно дает пользователю возможность одновременно вводить данные как в этом окне, так и в других окнах приложения. Примером такого окна служит панель инструментов. Она постоянно находится на своем месте, и пользователь может одновременно взаимодействовать как с ней, так и с другими окнами.

В начале этого раздела будет представлено простейшее модальное диалоговое окно, содержащее единственную строку сообщения. В библиотеке Swing имеется удобный класс `JOptionPane`, позволяющий выводить на экран модальное диалоговое окно, не прибегая к написанию специального кода для его поддержки. Далее будет показано, как реализовать свое собственное диалоговое окно и каким образом данные передаются из приложения в диалоговое окно и обратно. И, наконец, обсуждение диалоговых окон в этом разделе завершается рассмотрением компонента `JFileChooser` из библиотеки Swing.

11.7.1. Диалоговые окна для выбора разных вариантов

В библиотеку Swing входит много готовых простых диалоговых окон, которые позволяют вводить данные отдельными фрагментами. Для этой цели в классе `JOptionPane` имеются перечисленные ниже статические методы.

ShowMessageDialog ()	Выводит на экран сообщение и ожидает до тех пор, пока пользователь не щелкнет на кнопке ОК
ShowConfirmDialog ()	Выводит на экран сообщение и ожидает от пользователя подтверждения (щелчок на кнопке ОК или Cancel)
ShowOptionDialog ()	Выводит на экран сообщение и предоставляет пользователю возможность выбора среди нескольких вариантов
showInputDialog ()	Выводит на экран сообщение и поле, в котором пользователь должен ввести данные

На рис. 11.30 показано типичное диалоговое окно для выбора разных вариантов. Как видите, в нем содержатся следующие компоненты:

- пиктограмма;
- сообщение;
- одна или несколько экранных кнопок.

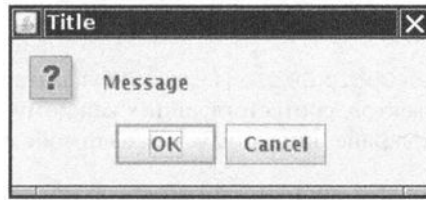


Рис. 11.30. Типичное диалоговое окно для выбора разных вариантов

Диалоговое окно может содержать дополнительный компонент для ввода данных. Этим компонентом может быть текстовое поле, в котором пользователь вводит произвольную символьную строку, или комбинированный список, один из элементов которого пользователь должен выбрать. Компоновка подобных диалоговых окон и выбор пиктограмм для стандартных сообщений зависит от визуального стиля оформления GUI.

Пиктограмма в левой части диалогового окна выбирается в зависимости от типа сообщения. Существуют пять типов сообщений:

```
ERROR_MESSAGE
INFORMATION_MESSAGE
WARNING_MESSAGE
QUESTION_MESSAGE
PLAIN_MESSAGE
```

Для сообщения типа `PLAIN_MESSAGE` пиктограмма не предусмотрена. А для каждого типа диалоговых окон существует метод, позволяющий указывать свою собственную пиктограмму.

С каждым типом диалоговых окон можно связать определенное сообщение, которое может быть представлено символьной строкой, пиктограммой, компонентом пользовательского интерфейса или любым другим объектом. Ниже поясняется, каким образом отображается объект сообщений.

String	Выводит символьную строку
Icon	Отображает пиктограмму
Component	Отображает компонент
Object[]	Выводит все объекты из массива, отображая их один над другим
Любой другой объект	Вызывает метод <code>toString()</code> и выводит получаемую в итоге символьную строку

Безусловно, на экран чаще всего выводится символьная строка сообщения. В то же время возможность отображать в диалоговом окне объекты типа `Component` дает немало удобств, поскольку, вызвав метод `paintComponent()`, можно нарисовать все, что угодно.

Внешний вид кнопок, расположенных в нижней части диалогового окна, зависит от его типа, а также от *типа вариантов*. При вызове метода `showMessageDialog()` или `showInputDialog()` выбор ограничивается только стандартным набором экранных кнопок (ОК или ОК и Cancel). А вызывая метод `showConfirmDialog()`, можно выбрать один из четырех типов вариантов:

```
DEFAULT_OPTION
YES_NO_OPTION
```

YES_NO_CANCEL_OPTION
OK_CANCEL_OPTION

С помощью метода `showOptionDialog()` можно указать произвольный набор вариантов, задав массив объектов, соответствующих каждому из них. Элементы этого массива отображаются на экране описанным ниже способом.

String	Создает кнопку, меткой которой служит указанная символьная строка
Icon	Создает кнопку, меткой которой служит указанная пиктограмма
Component	Отображает компонент
Любой другой объект	Вызывает метод <code>toString()</code> и создает кнопку, меткой которой служит получаемая в итоге символьная строка

Статические методы, предназначенные для создания диалоговых окон, возвращают перечисленные ниже значения.

ShowMessageDialog()	Возвращаемое значение отсутствует
showConfirmDialog()	Целое значение, соответствующее выбранному варианту
showOptionDialog()	Целое значение, соответствующее выбранному варианту
showInputDialog()	Символьная строка, введенная или выбранная пользователем

Методы `showConfirmDialog()` и `showOptionDialog()` возвращают целое значение, обозначающее кнопку, на которой щелкнул пользователь. В диалоговом окне для выбора разных вариантов это числовое значение является порядковым номером. Если вместо выбора варианта пользователь закрыл диалоговое окно, возвращается константа `CLOSED_OPTION`. Ниже приведены константы, используемые в качестве возвращаемых значений.

OK_OPTION
CANCEL_OPTION
YES_OPTION
NO_OPTION
CLOSED_OPTION

Несмотря на обилие упомянутых выше мнемонических обозначений разных вариантов выбора, создать диалоговое окно данного типа совсем не трудно. Для этого выполните следующие действия.

1. Выберите тип диалогового окна (для вывода сообщения, получения подтверждения, выбора разных вариантов или ввода данных).
2. Выберите пиктограмму (с ошибкой, важной информацией, предупреждением, вопросом, свою собственную) или вообще откажитесь от нее.
3. Выберите сообщение (в виде символьной строки, пиктограммы, пользовательского компонента или массива компонентов).
4. Если вы выбрали диалоговое окно для подтверждения выбора, задайте тип вариантов (по умолчанию Yes/No, No/Cancel или OK/Cancel).
5. Если вы создаете диалоговое окно для выбора разных вариантов, задайте варианты выбора (в виде символьных строк, пиктограмм или собственных компонентов), а также вариант, выбираемый по умолчанию.

6. Если вы создаете диалоговое окно для ввода данных, выберите текстовое поле или комбинированный список.

7. Найдите подходящий метод в классе `JOptionPane`.

Допустим, на экране требуется отобразить диалоговое окно, показанное на рис. 11.30. В этом окне выводится сообщение, и пользователю предлагается подтвердить или отклонить его. Следовательно, это диалоговое окно для подтверждения выбора. Пиктограмма, отображаемая в этом окне, относится к разряду вопросов, а сообщение выводится символьной строкой. Тип вариантов выбора обозначается константой `OK_CANCEL_OPTION`. Для создания такого диалогового окна служит следующий фрагмент кода:

```
int selection = JOptionPane.showConfirmDialog(parent,
    "Message", "Title",
    JOptionPane.OK_CANCEL_OPTION,
    JOptionPane.QUESTION_MESSAGE);
if (selection == JOptionPane.OK_OPTION) . . .
```



СОБЕТ. Символьная строка сообщения может содержать символ перевода строки (`'\n'`). В этом случае сообщение выводится в нескольких строках.

`javax.swing.JOptionPane 1.2`

- `static void showMessageDialog(Component parent, Object message, String title, int messageType, Icon icon)`
- `static void showMessageDialog(Component parent, Object message, String title, int messageType)`
- `static void showMessageDialog(Component parent, Object message)`
- `static void showInternalMessageDialog(Component parent, Object message, String title, int messageType, Icon icon)`
- `static void showInternalMessageDialog(Component parent, Object message, String title, int messageType)`
- `static void showInternalMessageDialog(Component parent, Object message)`

Выводят на экран обычное диалоговое окно или внутреннее диалоговое окно для сообщения. (Внутреннее диалоговое окно воспроизводится только в пределах фрейма.) Родительский компонент может быть пустым (`null`). В качестве сообщения в диалоговом окне может быть выведена символьная строка, пиктограмма, компонент или массив всех этих элементов. А в качестве параметра `messageType` может быть указана одна из следующих констант: `ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `WARNING_MESSAGE`, `QUESTION_MESSAGE`, `PLAIN_MESSAGE`.
- `static int showConfirmDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon)`
- `static int showConfirmDialog(Component parent, Object message, String title, int optionType, int messageType)`
- `static int showConfirmDialog(Component parent, Object message, String title, int optionType)`

javax.swing.JOptionPane 1.2 (продолжение)

- `static int showConfirmDialog(Component parent, Object message)`
- `static int showInternalConfirmDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon)`
- `static int showInternalConfirmDialog(Component parent, Object message, String title, int optionType, int messageType)`
- `static int showInternalConfirmDialog(Component parent, Object message, String title, int optionType)`
- `static int showInternalConfirmDialog(Component parent, Object message)`

Отображают обычное или внутреннее диалоговое окно для подтверждения. (Внутреннее диалоговое окно воспроизводится только в пределах фрейма.) Возвращают вариант, выбранный пользователем (одну из следующих констант: `OK_OPTION`, `YES_OPTION` или `NO_OPTION`), или же константу `CLOSED_OPTION`, если пользователь закрыл диалоговое окно. Родительский компонент может быть пустым (`null`). В качестве сообщения в диалоговом окне может быть выведена символьная строка, пиктограмма, компонент или массив всех этих элементов. А в качестве параметра `messageType` может быть указана одна из следующих констант: `ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `WARNING_MESSAGE`, `QUESTION_MESSAGE`, `PLAIN_MESSAGE`. И в качестве параметра `optionType` может быть указана одна из таких констант: `DEFAULT_OPTION`, `YES_NO_OPTION`, `YES_NO_CANCEL_OPTION`, `OK_CANCEL_OPTION`.

- `static int showOptionDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon, Object[] options, Object default)`
- `static int showInternalOptionDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon, Object[] options, Object default)`
- Отображают обычное или внутреннее диалоговое окно для выбора разных вариантов. (Внутреннее диалоговое окно воспроизводится только в пределах фрейма.) Возвращают индекс варианта, выбранного пользователем, или же константу `CLOSED_OPTION`, если пользователь закрыл диалоговое окно. Родительский компонент может быть пустым (`null`). В качестве сообщения в диалоговом окне может быть выведена символьная строка, пиктограмма, компонент или массив всех этих элементов. В качестве параметра `messageType` может быть указана одна из следующих констант: `ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `WARNING_MESSAGE`, `QUESTION_MESSAGE`, `PLAIN_MESSAGE`. А в качестве параметра `optionType` может быть указана одна из таких констант: `DEFAULT_OPTION`, `YES_NO_OPTION`, `YES_NO_CANCEL_OPTION`, `OK_CANCEL_OPTION`. И в качестве параметра `options` указывается массив символьных строк, пиктограмм или компонентов.
- `static Object showInputDialog(Component parent, Object message, String title, int messageType, Icon icon, Object[] values, Object default)`
- `static String showInputDialog(Component parent, Object message, String title, int messageType)`
- `static String showInputDialog(Component parent, Object message)`

`javax.swing.JOptionPane` 1.2 (окончание)

- `static String showInputDialog(Object message)`
 - `static String showInputDialog(Component parent, Object message, Object default)` 1.4
 - `static String showInputDialog(Object message, Object default)` 1.4
 - `static Object showInternalInputDialog(Component parent, Object message, String title, int messageType, Icon icon, Object[] values, Object default)`
 - `static String showInternalInputDialog(Component parent, Object message, String title, int messageType)`
 - `static String showInternalInputDialog(Component parent, Object message)`
- Отображают обычное или внутреннее диалоговое окно для ввода. (Внутреннее диалоговое окно воспроизводится только в пределах фрейма.) Возвращают символьную строку, введенную пользователем, или же пустое значение `null`, если пользователь закрыл диалоговое окно. Родительский компонент может быть пустым (`null`). В качестве сообщения в диалоговом окне может быть выведена символьная строка, пиктограмма, компонент или массив всех этих элементов. А в качестве параметра `messageType` может быть указана одна из следующих констант: `ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `WARNING_MESSAGE`, `QUESTION_MESSAGE`, `PLAIN_MESSAGE`.

11.7.2. Создание диалоговых окон

Как упоминалось ранее, для применения предопределенных диалоговых окон служит класс `JOptionPane`. В этом разделе будет показано, как создать диалоговое окно самостоятельно. На рис. 11.31 показано типичное модальное диалоговое окно, содержащее сообщение. Подобное окно обычно появляется на экране после того, как пользователь выберет пункт меню `About` (О программе).

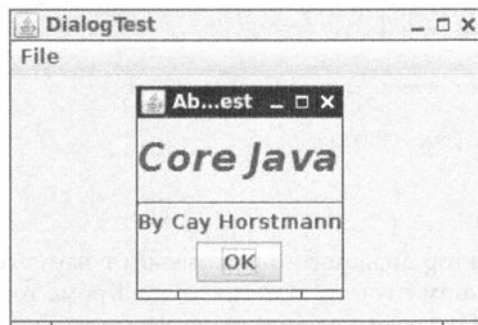


Рис. 11.31. Диалоговое окно `About`

Чтобы реализовать такое окно, необходимо создать подкласс, производный от класса `JDialog`. По существу, этот процесс ничем не отличается от создания главного окна приложения расширением класса `JFrame`. А точнее говоря, для этого нужно сделать следующее.

1. Вызовите конструктор суперкласса `JDialog` в конструкторе вашего диалогового окна.
2. Введите в свое диалоговое окно компоненты пользовательского интерфейса.
3. Введите обработчики событий, наступающих в данном окне.
4. Установите размеры своего диалогового окна.

При вызове конструктора суперкласса следует указать *фрейм-владелец*, заголовок окна и признак *модальности*. Фрейм-владелец управляет местом отображения диалогового окна. Вместо владельца можно указать пустое значение `null`, и тогда диалоговое окно будет принадлежать скрытому фрейму.

Признак модальности означает, должны ли блокироваться другие окна приложения до тех пор, пока отображается данное диалоговое окно. Немодальные (т.е. безрежимные) окна не блокируют другие окна. А модальное (т.е. режимное) диалоговое окно блокирует все остальные окна приложения (за исключением производных от этого диалогового окна). Как правило, немодальные диалоговые окна служат для создания панелей инструментов, к которым постоянно открыт доступ. С другой стороны, модальные диалоговые окна обычно служат для того, чтобы принудить пользователя ввести необходимую информацию, прежде чем продолжить работу с приложением.

Ниже приведен фрагмент кода, в котором создается диалоговое окно.

```
public AboutDialog extends JDialog
{
    public AboutDialog(JFrame owner)
    {
        super(owner, "About DialogTest", true);
        add(new JLabel(
            "<html><h1><i>Core Java</i>
            </h1><hr>By Cay Horstmann</html>"),
            BorderLayout.CENTER);

        var panel = new JPanel();
        var ok = new JButton("OK");

        ok.addActionListener(event -> setVisible(false));
        panel.add(ok);
        add(panel, BorderLayout.SOUTH);
        setSize(250, 150);
    }
}
```

Как видите, конструктор диалогового окна вводит в него элементы пользовательского интерфейса, в данном случае метки и кнопку. Кроме того, он вводит обработчик событий от кнопки и задает размеры окна. Чтобы отобразить диалоговое окно, необходимо создать новый объект типа `JDialog` и вызвать метод `setVisible()` следующим образом:

```
var dialog = new AboutDialog(this);
dialog.setVisible(true);
```

На самом деле в программе, рассматриваемой здесь в качестве примера, диалоговое окно создается только один раз, а затем оно используется повторно всякий раз, когда пользователь щелкает на кнопке `About`:

```
if (dialog == null) // в первый раз
    dialog = new AboutDialog(this);
dialog.setVisible(true);
```

Когда пользователь щелкает на кнопке ОК, диалоговое окно должно закрываться. Такая реакция на действия пользователя определяется в обработчике событий от кнопки ОК, как следует из приведенной ниже строки кода.

```
ok.addActionListener(event -> setVisible(false));
```

Когда же пользователь закрывает диалоговое окно, щелкая на кнопке Close, оно исчезает из виду. Как и в классе `JFrame`, такое поведение можно изменить с помощью метода `setDefaultCloseOperation()`.

В листинге 11.11 приведен исходный код класса фрейма для примера программы, где демонстрируется применение модального диалогового окна, создаваемого самостоятельно. А в листинге 11.12 представлен исходный код класса для создания этого диалогового окна.

Листинг 11.11. Исходный код из файла `dialog/DialogFrame.java`

```
1 package dialog;
2
3 import javax.swing.JFrame;
4 import javax.swing.JMenu;
5 import javax.swing.JMenuBar;
6 import javax.swing.JMenuItem;
7
8 /**
9  * Фрейм со строкой меню, при выборе команды File⇒About
10  * из которого появляется диалоговое окно About
11  */
12 public class DialogFrame extends JFrame
13 {
14     private static final int DEFAULT_WIDTH = 300;
15     private static final int DEFAULT_HEIGHT = 200;
16     private AboutDialog dialog;
17
18     public DialogFrame()
19     {
20         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
21
22         // сконструировать меню File
23
24         var = new JMenuBar();
25         setJMenuBar(menuBar);
26         var fileMenu = new JMenu("File");
27         menuBar.add(fileMenu);
28
29         // ввести в меню пункты About и Exit
30
31         // При выборе пункта меню About открывается
32         // одноименное диалоговое окно
33
34         var aboutItem = new JMenuItem("About");
35         aboutItem.addActionListener(event -> {
```

```
36     if (dialog == null) // в первый раз
37         dialog = new AboutDialog(DialogFrame.this);
38     dialog.setVisible(true);
39         // показать диалоговое окно
40     });
41     fileMenu.add(aboutItem);
42
43     // При выборе пункта меню Exit происходит
44     // выход из программы
45
46     var exitItem = new JMenuItem("Exit");
47     exitItem.addActionListener(event -> System.exit(0));
48     fileMenu.add(exitItem);
49 }
50 }
```

Листинг 11.12. Исходный код из файла `dialog/AboutDialog.java`

```
1  package dialog;
2
3  import java.awt.BorderLayout;
4
5  import javax.swing.JButton;
6  import javax.swing.JDialog;
7  import javax.swing.JFrame;
8  import javax.swing.JLabel;
9  import javax.swing.JPanel;
10
11 /**
12  * Образец модального диалогового окна, в котором
13  * выводится сообщение и ожидается до тех пор, пока
14  * пользователь не щелкнет на кнопке OK
15  */
16 public class AboutDialog extends JDialog
17 {
18     public AboutDialog(JFrame owner)
19     {
20         super(owner, "About DialogTest", true);
21
22         // ввести HTML-метку по центру окна
23
24         add(
25             new JLabel("<html><h1><i>Core Java</i></h1>
26                        <hr>By Cay Horstmann</html>"),
27             BorderLayout.CENTER);
28
29         // При выборе кнопки OK диалоговое окно закрывается
30
31         var ok = new JButton("OK");
32         ok.addActionListener(event -> setVisible(false));
33
34         // ввести кнопку OK в нижней части окна
35         // у южной его границы
36
37         var panel = new JPanel();
```

```
38     panel.add(ok);
39     add(panel, BorderLayout.SOUTH);
40
41     pack();
42 }
43 }
```

javax.swing.JDialog 1.2

- `public JDialog(Frame parent, String title, boolean modal)`

Создает диалоговое окно, которое оказывается невидимым до тех пор, пока оно не будет показано явным образом.

11.7.3. Обмен данными

Чаще всего диалоговые окна создаются для того, чтобы получить информацию от пользователя. Выше было показано, насколько просто создаются объекты типа `JDialog`: достаточно указать исходные данные и вызвать метод `setVisible(true)`, чтобы вывести окно на экран. А теперь покажем, как вводить данные в диалоговом окне. Обратите внимание на диалоговое окно, показанное на рис. 11.32. Его можно использовать для получения имени пользователя и пароля при подключении к оперативно доступной службе.

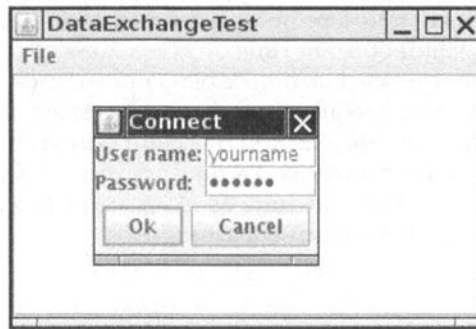


Рис. 11.32. Диалоговое окно для ввода имени пользователя и пароля

Для нормальной работы такого диалогового окна должны быть предусмотрены методы, формирующие данные по умолчанию. Например, в классе `PasswordChooser` имеется метод `setUser()` для ввода исходных значений, задаваемых по умолчанию в следующих полях:

```
public void setUser(User u)
{
    username.setText(u.getName());
}
```

После установки значений по умолчанию, если требуется, окно выводится на экран с помощью метода `setVisible(true)`. Далее пользователь должен ввести данные в указанных полях и щелкнуть на кнопке `OK` или `Cancel`. В обработчиках событий

от обеих кнопок вызывается метод `setVisible(false)`. При таком вызове выполняются действия, обратные тем, что происходят при вызове метода `setVisible(true)`. С другой стороны, пользователь может просто закрыть диалоговое окно. Если приемник событий в диалоговом окне не установлен, то выполняются обычные операции по закрытию окон. В итоге диалоговое окно становится невидимым, а вызов `setVisible(true)` завершается.

Следует особо подчеркнуть, что при вызове метода `setVisible(true)` выполнение программы приостанавливается до тех пор, пока пользователь не выполнит действие, приводящее к закрытию окна. Такой подход существенно упрощает реализацию модальных диалоговых окон.

В большинстве случаев требуется знать, подтвердил ли пользователь введенные им данные или отказался от их ввода. В программе, рассматриваемой здесь в качестве примера, применяется следующий подход. Перед обращением к диалоговому окну в переменной `ok` устанавливается логическое значение `false`. А логическое значение `true` присваивается этой переменной только в обработчике событий от кнопки ОК. В этом случае введенные пользователем данные могут быть использованы в программе.



НА ЗАМЕТКУ! Передать данные из немодального диалогового окна не так-то просто. При отображении такого окна вызов метода `setVisible(true)` не приводит к приостановке выполнения программы. Если пользователь выполнит манипуляции над элементами в немодальном диалоговом окне и затем щелкнет на кнопке ОК, это окно должно уведомить соответствующий приемник событий в самой программе.

В рассматриваемом здесь примере программы имеются и другие заметные усовершенствования. При создании объекта типа `JDialog` должен быть указан фрейм-владелец. Но зачастую одно и то же диалоговое окно приходится отображать с разными фреймами. Поэтому намного удобнее, если фрейм-владелец задается, когда диалоговое окно *готово* к выводу на экран, а не при создании объекта типа `PasswordChooser`.

Этого можно добиться, если класс `PasswordChooser` будет расширять класс `JPanel`, а не класс `JDialog`. А объект типа `JDialog` можно создать по ходу выполнения метода `showDialog()` следующим образом:

```
public boolean showDialog(Frame owner, String title)
{
    ok = false;
    if (dialog == null || dialog.getOwner() != owner)
    {
        dialog = new JDialog(owner, true);
        dialog.add(this);
        dialog.pack();
    }

    dialog.setTitle(title);
    dialog.setVisible(true);
    return ok;
}
```

Следует заметить, что для большей надежности значение параметра `owner` должно быть равно `null`. Можно пойти еще дальше. Ведь иногда фрейм-владелец оказывается недоступным. Но его можно вычислить, как и любой родительский компонент из параметра `parent`, следующим образом:

```

Frame owner;
if (parent instanceof Frame)
    owner = (Frame) parent;
else
    owner = (Frame) SwingUtilities.getAncestorOfClass(
        Frame.class, parent);

```

Именно такой подход применяется в приведенном ниже примере программы. Этот механизм используется и в классе `JOptionPane`.

Во многих диалоговых окнах имеется кнопка по умолчанию, которая выбирается автоматически, если пользователь нажимает клавишу ввода (в большинстве визуальных стилей оформления GUI для этого служит клавиша `<Enter>`). Кнопка по умолчанию выделяется среди остальных компонентов GUI, чаще всего контуром. Для установки кнопки по умолчанию на *корневой панели* диалогового окна делается следующий вызов:

```
dialog.getRootPane().setDefaultButton(okButton);
```

Если вы собираетесь разместить элементы диалогового окна на панели, будьте внимательны: устанавливайте кнопку по умолчанию только после оформления панели в виде диалогового окна. Такая панель сама по себе не имеет *корневой панели*.

В листинге 11.13 приведен исходный код класса фрейма для примера программы, где демонстрируется обмен данными с диалоговым окном. А в листинге 11.14 представлен исходный код класса для этого диалогового окна.

Листинг 11.13. Исходный код из файла `dataExchange/DataExchangeFrame.java`

```

1 package dataExchange;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 /**
8  * Фрейм со строкой меню, при выборе команды
9  * File⇌Connect из которого появляется
10 * диалоговое окно для ввода пароля
11 */
12 public class DataExchangeFrame extends JFrame
13 {
14     public static final int TEXT_ROWS = 20;
15     public static final int TEXT_COLUMNS = 40;
16     private PasswordChooser dialog = null;
17     private JTextArea textArea;
18
19     public DataExchangeFrame()
20     {
21         // сконструировать меню File
22
23         var = new JMenuBar();
24         setJMenuBar(mbar);
25         var fileMenu = new JMenu("File");
26         mbar.add(fileMenu);
27
28         // ввести в меню пункты Connect и Exit
29

```

```
30    var connectItem = new JMenuItem("Connect");
31    connectItem.addActionListener(new ConnectAction());
32    fileMenu.add(connectItem);
33
34    // При выборе пункта меню Exit происходит
35    // выход из программы
36
37    var exitItem = new JMenuItem("Exit");
38    exitItem.addActionListener(event -> System.exit(0));
39    fileMenu.add(exitItem);
40
41    var = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
42    add(new JScrollPane(textArea), BorderLayout.CENTER);
43    pack();
44 }
45
46 /**
47  * При выполнении команды Connect появляется
48  * диалоговое окно для ввода пароля
49  */
50 private class ConnectAction implements ActionListener
51 {
52     public void actionPerformed(ActionEvent event)
53     {
54         // При первом обращении конструируется
55         // диалоговое окно
56
57         if (dialog == null)
58             dialog = new PasswordChooser();
59
60         // установить значения по умолчанию
61         dialog.setUser(new User("yourname", null));
62
63         // показать диалоговое окно
64         if (dialog.showDialog(DataExchangeFrame.this,
65                               "Connect"))
66         {
67             // Если пользователь подтвердил введенные
68             // данные, извлечь их для последующей обработки
69             User u = dialog.getUser();
70             textArea.append("user name = " + u.getName()
71                            + ", password = "
72                            + (new String(u.getPassword()))
73                            + "\n");
74         }
75     }
76 }
77 }
```

Листинг 11.14. Исходный код из файла `dataExchange/PasswordChooser.java`

```
1  package dataExchange;
2
3  import java.awt.BorderLayout;
4  import java.awt.Component;
```

```
5  import java.awt.Frame;
6  import java.awt.GridLayout;
7
8  import javax.swing.JButton;
9  import javax.swing.JDialog;
10 import javax.swing.JLabel;
11 import javax.swing.JPanel;
12 import javax.swing.JPasswordField;
13 import javax.swing.JTextField;
14 import javax.swing.SwingUtilities;
15
16 /**
17  * Окно для ввода пароля, отображаемое в диалоговом окне
18  */
19 public class PasswordChooser extends JPanel
20 {
21     private JTextField username;
22     private JPasswordField password;
23     private JButton okButton;
24     private boolean ok;
25     private JDialog dialog;
26
27     public PasswordChooser()
28     {
29         setLayout(new BorderLayout());
30
31         // сконструировать панель с полями для
32         // ввода имени пользователя и пароля
33
34         var panel = new JPanel();
35         panel.setLayout(new GridLayout(2, 2));
36         panel.add(new JLabel("User name:"));
37         panel.add(username = new JTextField(""));
38         panel.add(new JLabel("Password:"));
39         panel.add(password = new JPasswordField(""));
40         add(panel, BorderLayout.CENTER);
41
42         // создать кнопки OK и Cancel для закрытия
43         // диалогового окна
44
45         okButton = new JButton("Ok");
46         okButton.addActionListener(event -> {
47             ok = true;
48             dialog.setVisible(false);
49         });
50
51         var cancelButton = new JButton("Cancel");
52         cancelButton.addActionListener(event ->
53             dialog.setVisible(false));
54
55         // ввести кнопки в нижней части окна
56         // у южной его границы
57
58         var buttonPanel = new JPanel();
59         buttonPanel.add(okButton);
60         buttonPanel.add(cancelButton);
61         add(buttonPanel, BorderLayout.SOUTH);
```



```
62     }
63
64     /**
65      * Устанавливает диалоговое окно в исходное состояние
66      * @param u Данные о пользователе по умолчанию
67      */
68     public void setUser(User u)
69     {
70         username.setText(u.getName());
71     }
72
73     /**
74      * Получает данные, введенные в диалоговом окне
75      * @return Объект типа User, состояние которого
76      *         отражает введенные пользователем данные
77      */
78     public User getUser()
79     {
80         return new User(username.getText(),
81                         password.getPassword());
82     }
83
84     /**
85      * Отображает панель для ввода пароля
86      * в диалоговом окне
87      * @param parent Компонент из фрейма-владельца
88      *              или пустое значение null
89      * @param title Заголовок диалогового окна
90      */
91     public boolean showDialog(Component parent,
92                             String title)
93     {
94         ok = false;
95
96         // обнаружить фрейм-владелец
97
98         Frame owner = null;
99         if (parent instanceof Frame)
100             owner = (Frame) parent;
101         else
102             owner = (Frame) SwingUtilities
103                 .getAncestorOfClass(Frame.class, parent);
104
105         // создать новое диалоговое окно при первом
106         // обращении или изменении фрейма-владельца
107
108         if (dialog == null || dialog.getOwner() != owner)
109         {
110             dialog = new JDialog(owner, true);
111             dialog.add(this);
112             dialog.getRootPane().setDefaultButton(okButton);
113             dialog.pack();
114         }
115
116         // установить заголовок и отобразить
117         // диалоговое окно
```

```
119     dialog.setTitle(title);
120     dialog.setVisible(true);
121     return ok;
122 }
123 }
```

javax.swing.SwingUtilities 1.2

- **Container getAncestorOfClass(Class c, Component comp)**

Возвращает наиболее глубоко вложенный родительский контейнер указанного компонента, принадлежащего заданному классу или одному из его подклассов.

javax.swing.JComponent 1.2

- **JRootPane getRootPane()**

Определяет корневую панель, содержащую данный компонент. Если у компонента отсутствует предшественник с корневой панелью, то возвращает пустое значение **null**.

javax.swing.JRootPane 1.2

- **void setDefaultButton(JButton button)**

Устанавливает кнопку по умолчанию на данной корневой панели. Чтобы запретить доступ к кнопке по умолчанию, этот метод вызывается с пустым значением **null** параметра **button**.

javax.swing.JButton 1.2

- **boolean isDefaultButton()**

Возвращает логическое значение **true**, если это кнопка, выбираемая по умолчанию на своей корневой панели.

11.7.4. Диалоговые окна для выбора файлов

Во многих приложениях требуется открывать и сохранять файлы. Написать код для создания диалогового окна, позволяющего свободно перемещаться по каталогам файловой системы, не так-то просто. Впрочем, этого и не требуется, чтобы не изобретать заново колесо! В библиотеке Swing имеется класс **JFileChooser**, позволяющий отображать диалоговое окно для манипулирования файлами, удовлетворяющее потребностям большинства приложений. Это диалоговое окно всегда является модальным. Следует, однако, иметь в виду, что класс **JFileChooser** не расширяет класс **JDialog**. Вместо метода **setVisible(true)** для отображения диалогового окна при открытии файлов вызывается метод **showOpenDialog()**, а при сохранении файлов — метод **showSaveDialog()**. Экранная кнопка, подтверждающая выбор файла, автоматически помечается как **Open** или **Save**. С помощью метода **showDialog()** можно

задать свою собственную метку данной кнопки. На рис. 11.33 приведен пример диалогового окна для выбора файлов.

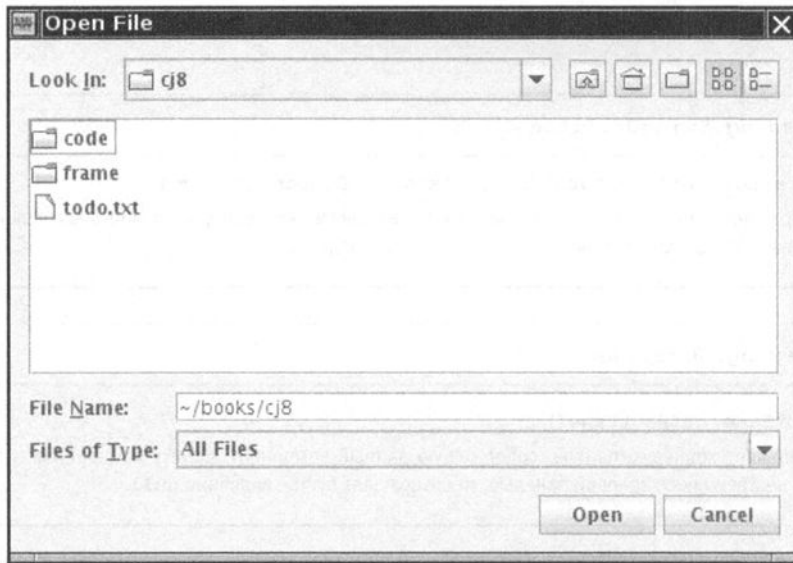


Рис. 11.33. Диалоговое окно для выбора файлов

Чтобы отобразить на экране диалоговое окно для выбора файлов и восстановить выбор, сделанный пользователем, выполните следующие действия.

1. Создайте объект класса `JFileChooser`, как показано ниже. В отличие от конструктора класса `JDialog`, в данном случае указывать родительский компонент не нужно. Такой подход позволяет повторно использовать диалоговое окно для выбора файлов во многих фреймах.

```
var chooser = new JFileChooser();
```



СОВЕТ. Повторно использовать диалоговое окно для выбора файлов целесообразно потому, что конструктор класса `JFileChooser` иногда действует очень медленно. Особенно это заметно в Windows, когда пользователю доступно много подключенных сетевых дисков.

2. Укажите нужный каталог, вызвав метод `setCurrentDirectory()`. Например, чтобы задать текущий каталог, сделайте следующий вызов:

```
chooser.setCurrentDirectory(new File("."));
```

В этом вызове необходимо указать объект типа `File`. Подробнее класс `File` описывается в главе 2 второго тома настоящего издания, а до тех пор достаточно сказать, что у этого класса имеется конструктор `File(String filename)`, преобразующий символьную строку `filename` с именем файла или каталога в объект типа `File`.

3. Если требуется предоставить пользователю имя файла, выбираемого по умолчанию, укажите его при вызове метода `setSelectedFile()` следующим образом:

```
chooser.setSelectedFile(new File(filename));
```

4. Вызовите метод `setMultiSelectionEnabled()`, как показано ниже, чтобы дать пользователю возможность выбрать одновременно несколько файлов. Разумеется, делать это совсем не обязательно, ведь подобная возможность требуется далеко не всегда.

```
chooser.setMultiSelectionEnabled(true);
```

5. Установите *фильтр файлов*, чтобы ограничить выбор пользователя файлами с определенным расширением (например, `.gif`). Подробнее о фильтрах файлов речь пойдет далее в этом разделе.
6. По умолчанию в данном диалоговом окне пользователь может выбирать только файлы. Если же требуется предоставить пользователю возможность выбирать целые каталоги, вызовите метод `setFileSelectionMode()`, указав в качестве его параметра один из следующих режимов: `JFileChooser.FILES_ONLY` (по умолчанию), `JFileChooser.DIRECTORIES_ONLY` или `JFileChooser.FILES_AND_DIRECTORIES`.

7. Отобразите данное диалоговое окно с помощью метода `showOpenDialog()` или `showSaveDialog()`. При вызове этих методов следует указать родительский компонент, как показано ниже.

```
int result = chooser.showOpenDialog(parent);
```

или

```
int result = chooser.showSaveDialog(parent)
```

Эти вызовы отличаются лишь меткой для кнопки подтверждения выбора, т.е. той экранной кнопки, на которой пользователь щелкает мышью, выбирая файл. Можно также вызвать метод `showDialog()` и явно указать текст надписи на кнопке следующим образом:

```
int result = chooser.showDialog(parent, "Select");
```

При подобных вызовах управление возвращается только после того, как пользователь подтвердит свой выбор файла или откажется сделать выбор, закрыв диалоговое окно. В качестве возвращаемого значения может служить одна из следующих констант: `JFileChooser.APPROVE_OPTION`, `JFileChooser.CANCEL_OPTION` или `JFileChooser.ERROR_OPTION`.

8. Получите выбранный файл или несколько файлов с помощью метода `getSelectedFile()` или `getSelectedFiles()`. Эти методы возвращают один объект типа `File` или массив таких объектов. Если пользователю нужно знать лишь имя файла, вызовите метод `getPath()`, как в приведенной ниже строке кода.

```
String filename = chooser.getSelectedFile().getPath();
```

Большая часть описанных выше действий довольно проста. Основные трудности возникают при использовании диалогового окна, в котором ограничивается выбор файлов пользователем. Допустим, пользователь должен выбирать только графические файлы формата GIF. Следовательно, в диалоговом окне должны отображаться только файлы, имеющие расширение `.gif`. Кроме того, пользователю необходимо подсказать, что это за файлы. Но дело может усложниться, когда выбираются файлы изображений формата JPEG, поскольку они могут иметь расширение `.jpg` или `.jpeg`. Для преодоления подобных трудностей разработчики предложили следующий изящный механизм: чтобы ограничить круг отображаемых файлов, достаточно

предоставить объект подкласса, производного от абстрактного класса `javax.swing.filechooser.FileFilter`. В этом случае диалоговое окно для выбора файлов передает фильтру каждый файл и отображает только отфильтрованные файлы.

На момент написания данной книги были известны только два таких подкласса: фильтр, задаваемый по умолчанию и пропускающий все файлы, а также фильтр, пропускающий все файлы с указанным расширением. Но можно создать и свой собственный фильтр. Для этого достаточно реализовать следующие два метода, которые объявлены в классе `FileFilter` как абстрактные:

```
public boolean accept(File f);  
public String getDescription();
```

Первый из приведенных выше методов проверяет, удовлетворяет ли файл накладываемым ограничениям, а второй возвращает описание типа файлов, которые могут отображаться в диалоговом окне.



НА ЗАМЕТКУ! В состав пакета `java.io` входит также интерфейс `FileFilter` (совершенно не связанный с описанным выше одноименным абстрактным классом). В этом интерфейсе объявлен только один метод — `boolean accept(File f)`. Он используется в методе `listFiles()` из класса `File` для вывода списка файлов, находящихся в каталоге. Совершенно непонятно, почему разработчики библиотеки Swing не стали расширять этот интерфейс. Возможно, библиотека классов Java настолько сложная, что даже ее разработчики не знают обо всех стандартных классах и интерфейсах!

Если в программе одновременно импортируются пакеты `java.io` и `javax.swing.filechooser`, то придется каким-то образом разрешать конфликт имен. Поэтому вместо пакета `javax.swing.filechooser.*` проще импортировать класс `javax.swing.filechooser.FileFilter`.

Имея в своем распоряжении объект фильтра файлов, можно воспользоваться методом `setFileFilter()` из класса `JFileChooser`, чтобы установить этот фильтр в объекте диалогового окна для выбора файлов следующим образом:

```
chooser.setFileFilter(new FileNameExtensionFilter(  
    "Image files", "gif", "jpg");
```

Аналогичным образом можно установить несколько фильтров в следующей последовательности вызовов:

```
chooser.addChoosableFileFilter(filter1);  
chooser.addChoosableFileFilter(filter2);  
...
```

Пользователь выбирает фильтр из комбинированного списка в нижней части диалогового окна для выбора файлов. По умолчанию в нем всегда присутствует фильтр `All files` (Все файлы). Это удобно, особенно в том случае, если пользователю прикладной программы нужно выбрать файл с нестандартным расширением. Но если требуется подавить фильтр `All files`, то достаточно сделать следующий вызов:

```
chooser.setAcceptAllFileFilterUsed(false)
```



ВНИМАНИЕ! Если одно и то же диалоговое окно используется для загрузки и сохранения разнотипных файлов, следует вызвать метод `chooser.resetChoosableFilters()`, чтобы очистить старые фильтры перед установкой новых.

И, наконец, в диалоговом окне для выбора файлов каждый файл можно сопроводить специальной пиктограммой и кратким описанием. Для этого следует предоставить экземпляр класса, расширяющего класс `FileView` из пакета `javax.swing.filechooser`, хотя такая дополнительная возможность используется нечасто. Обычно

внешний вид файла не интересует разработчика прикладной программы, поскольку эту задачу автоматически решают подключаемые визуальные стили оформления GUI. Но если файлы определенного типа требуется сопроводить специальными пиктограммами, то можно задать свой собственный стиль отображения файлов. Для этого придется расширить класс `FileView` и реализовать пять приведенных ниже методов, а затем вызвать метод `setFileView()`, чтобы связать нужное представление файлов с диалоговым окном для их выбора.

```
Icon getIcon(File f);
String getName(File f);
String getDescription(File f);
String getTypeDescription(File f);
Boolean isTraversable(File f);
```

Эти методы вызываются для каждого файла или каталога, отображаемого в диалоговом окне для выбора файлов. Если метод возвращает пустую ссылку типа `null` на пиктограмму, имя или описание файла, то в данном диалоговом окне используется визуальный стиль, устанавливаемый по умолчанию. И это хорошо, поскольку позволяет применять особый стиль только к отдельным типам файлов.

Чтобы выяснить, следует ли открывать каталог, выбранный пользователем, в диалоговом окне для выбора файлов, необходимо вызвать метод `isTraversable()`. Однако этот метод возвращает объект класса `Boolean`, а не значение типа `boolean`! И хотя это не совсем обычно, тем не менее, очень удобно. Так, если нет особых требований к визуальному стилю, достаточно вернуть пустое значение `null`. В таком случае в диалоговом окне для выбора файлов используется стиль отображения файлов, устанавливаемый по умолчанию. Иными словами, данный метод возвращает объект типа `Boolean`, чтобы дать возможность выбрать один из трех вариантов: открывать каталог (`Boolean.TRUE`), не открывать каталог (`Boolean.FALSE`) и неважно (`null`).

Рассмотрим в качестве примера простой класс представления файлов. Этот класс отображает определенную пиктограмму всякий раз, когда файл совпадает с заданным фильтром файлов. В данном случае этот класс служит для отображения панели с пиктограммами для всех графических файлов:

```
class FileIconView extends FileView
{
    private FileFilter filter;
    private Icon icon;

    public FileIconView(FileFilter aFilter, Icon anIcon)
    {
        filter = aFilter;
        icon = anIcon;
    }

    public Icon getIcon(File f)
    {
        if (!f.isDirectory() && filter.accept(f))
            return icon;
        else return null;
    }
}
```

Чтобы установить это представление файлов в диалоговом окне для выбора файлов, следует вызвать метод `setFileView()`, как показано ниже.

```
chooser.setFileView(new FileIconView(filter,
    new ImageIcon("palette.gif")));
```

Панель с пиктограммами отображается в диалоговом окне для выбора файлов рядом со всеми отфильтрованными файлами, а для отображения всех остальных файлов используется представление, устанавливаемое по умолчанию. Естественно, что для отбора файлов служит один тот же фильтр, установленный в диалоговом окне для выбора файлов.

И, наконец, диалоговое окно для выбора файлов можно снабдить *вспомогательным компонентом*. В качестве примера на рис. 11.34 показан такой компонент, позволяющий отображать в миниатюрном виде содержимое выбранного в данный момент файла, помимо списка файлов.

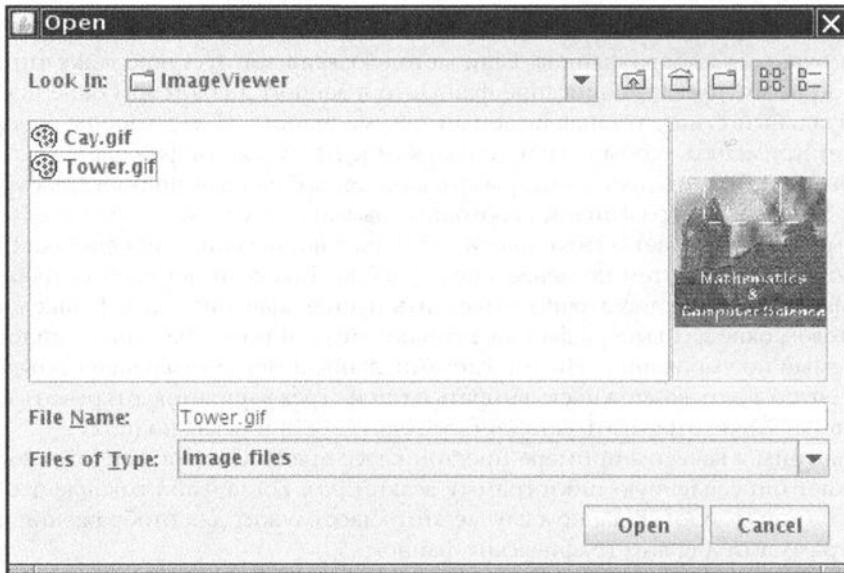


Рис. 11.34. Диалоговое окно для выбора файлов со вспомогательным компонентом для предварительного просмотра выбранных файлов

В качестве вспомогательного может служить любой компонент из библиотеки Swing. В данном примере расширяется класс `JLabel`, а в качестве пиктограммы используется уменьшенная копия изображения, хранящегося в выбранном файле:

```
class ImagePreviewer extends JLabel
{
    public ImagePreviewer(JFileChooser chooser)
    {
        setPreferredSize(new Dimension(100, 100));
        setBorder(BorderFactory.createEtchedBorder());
    }

    public void loadImage(File f)
    {
        var icon = new ImageIcon(f.getPath());
        if(icon.getIconWidth() > getWidth())
            icon = new ImageIcon(icon.getImage().getScaledInstance(
```

```

        getWidth(), -1, Image.SCALE_DEFAULT));
setIcon(icon);
repaint();
    }
}

```

Остается лишь преодолеть еще одно затруднение. Предварительно просматриваемое изображение требуется обновлять всякий раз, когда пользователь выбирает новый файл. С этой целью в диалоговом окне для выбора файлов применяется механизм компонентов JavaBeans, уведомляющий заинтересованные приемники событий об изменениях свойств данного окна. Выбранный файл — это свойство, которое можно отслеживать с помощью установленного приемника событий типа `PropertyChangeListener`. В приведенном ниже фрагменте кода показано, каким образом организуется перехват уведомлений, направляемых приемнику событий.

```

chooser.addPropertyChangeListener(event -> {
    if (event.getPropertyName() ==
        JFileChooser.SELECTED_FILE_CHANGED_PROPERTY)
    {
        var newFile = (File) event.getNewValue();
        // обновить вспомогательный компонент
        . . .
    }
});

```

javax.swing.JFileChooser 1.2

- **JFileChooser()**
Создает диалоговое окно для выбора файлов, которое можно использовать во многих фреймах.
- **void setCurrentDirectory(File dir)**
Задает исходный каталог, содержимое которого отображается в диалоговом окне для выбора файлов.
- **void setSelectedFile(File file)**
- **void setSelectedFiles(File[] file)**
Задают файл, выбираемый в диалоговом окне по умолчанию.
- **void setMultiSelectionEnabled(boolean b)**
Устанавливает или отменяет режим выбора нескольких файлов.
- **void setFileSelectionMode(int mode)**
Позволяет выбирать только файлы (по умолчанию), только каталоги или же каталоги вместе с файлами. Параметр `mode` может принимать следующие значения: `JFileChooser.FILES_ONLY`, `JFileChooser.DIRECTORIES_ONLY` и `JFileChooser.FILES_AND_DIRECTORIES`.
- **int showOpenDialog(Component parent)**
- **int showSaveDialog(Component parent)**
- **int showDialog(Component parent, String approveButtonText)**
Отображают диалоговое окно с кнопкой подтверждения выбора, обозначенной меткой `Open`, `Save` или произвольной меткой, указанной в символьной строке `approveButtonText`. Возвращают следующие значения: `APPROVE_OPTION`, `CANCEL_OPTION` (если пользователь щелкнул на кнопке `Cancel` или закрыл диалоговое окно) или `ERROR_OPTION` (если возникла ошибка).

javax.swing.JFileChooser 1.2 (окончание)

- **File getSelectedFile()**
- **File[] getSelectedFiles()**
Возвращают файл или несколько файлов, выбранных пользователем, а если он ничего не выбрал — пустое значение `null`.
- **void setFileFilter(FileFilter filter)**
Устанавливает маску файлов в диалоговом окне для выбора файлов. В этом окне отображаются только те файлы, для которых метод `filter.accept()` возвращает логическое значение `true`. Кроме того, вводит фильтр в список выбираемых фильтров.
- **void addChoosableFileFilter(FileFilter filter)**
Вводит фильтр в список выбираемых фильтров.
- **void setAcceptAllFileFilterUsed(boolean b)**
Вводит все файлы в комбинированный список выбираемых фильтров или удаляет их из этого списка.
- **void resetChoosableFileFilters()**
Очищает список фильтров, где остается фильтр всех файлов, если только он не удален из списка специально.
- **void setFileView(FileView view)**
Устанавливает представление файлов для предоставления сведений о файлах, отображаемых в диалоговом окне для выбора файлов.
- **void setAccessory(JComponent component)**
Устанавливает вспомогательный компонент.

javax.swing.filechooser.FileFilter 1.2

- **boolean accept(File f)**
Возвращает логическое значение `true`, если указанный файл должен отображаться в диалоговом окне.
- **String getDescription()**
Возвращает описание указанного фильтра, например `"Image files (*.gif, *.jpeg)"` (Файлы изображений с расширением `*.gif` и `*.jpeg`).

javax.swing.filechooser.FileNameExtensionFilter 6

- **FileNameExtensionFilter(String description, String ... extensions)**
Конструирует фильтр файлов с заданным описанием, принимающий все каталоги и все файлы, имена которых оканчиваются точкой и последующей символьной строкой одного из указанных расширений.

javax.swing.filechooser.FileView 1.2

- **String getName(File f)**

Возвращает имя указанного файла *f* или пустое значение **null**. Обычно возвращается результат вызова метода *f.getName()*.

- **String getDescription(File f)**

Возвращает удобочитаемое описание указанного файла *f* или пустое значение **null**. Так, если указанный файл *f* представляет собой HTML-документ, этот метод может вернуть его заголовок.

- **String getTypeDescription(File f)**

Возвращает удобочитаемое описание типа указанного файла *f* или пустое значение **null**. Так, если указанный файл *f* представляет собой HTML-документ, этот метод может вернуть символическую строку **"Hypertext document"**.

- **Icon getIcon(File f)**

Возвращает пиктограмму, назначенную для указанного файла *f*, или пустое значение **null**. Так, если указанный файл *f* относится к формату **JPEG**, этот метод может вернуть пиктограмму с миниатюрным видом его содержимого.

- **Boolean isTraversable(File f)**

Если пользователь может открыть указанный каталог, возвращается значение **Boolean.TRUE**. Если же каталог представляет собой составной документ, может быть возвращено значение **Boolean.FALSE**. Подобно методам из класса **FileView**, этот метод может возвращать пустое значение **null**, отмечая тот факт, что в диалоговом окне для выбора файлов должно быть использовано представление файлов, устанавливаемое по умолчанию.

На этом обсуждение особенностей программирования GUI на Java завершается. А более развитые компоненты Swing и усовершенствованные методики работы с графикой обсуждаются во втором томе настоящего издания.

Параллелизм

В этой главе...

- ▶ Назначение потоков исполнения
- ▶ Состояния потоков исполнения
- ▶ Свойства потоков исполнения
- ▶ Синхронизация
- ▶ Потокобезопасные коллекции
- ▶ Задачи и пулы потоков исполнения
- ▶ Асинхронные вычисления
- ▶ Процессы

Вам, вероятно, хорошо известна *многозадачность* используемой вами операционной системы — возможность одновременно выполнять несколько программ. Например, вы можете печатать во время редактирования документа или приема электронной почты. Ваш современный компьютер, вероятнее всего, оснащен не одним центральным процессором, но число одновременно выполняющихся процессов не ограничивается количеством процессоров. Операционная система выделяет время процессора квантами для каждого процесса, создавая впечатление параллельного выполнения программ.

Многопоточные программы расширяют принцип многозадачности, перенося его на один уровень ниже, чтобы отдельные приложения могли выполнять многие задачи одновременно. Каждая задача обычно называется *поток*ом исполнения, или потоком управления. Программы, способные одновременно выполнять больше одного потока исполнения, называются *многопоточными*.

Так в чем же отличие во многих *процессах* и *потоках* исполнения? Оно состоит в следующем: если у каждого процесса имеется собственный набор переменных, то потоки исполнения могут разделять одни и те же общие данные. Это кажется несколько рискованным, и на самом деле так оно и есть, как станет ясно далее в этой главе. Но разделяемые переменные обеспечивают более высокую эффективность

взаимодействия потоков исполнения и облегчают связь между ними. Кроме того, в некоторых операционных системах потоки исполнения являются более “легковесными”, чем процессы, — они требуют меньших издержек на свое создание и уничтожение по сравнению с процессами.

Многопоточная обработка имеет исключительную практическую ценность. Например, браузер должен уметь одновременно загружать многие изображения. Веб-серверу приходится обслуживать параллельные запросы. Программы с графическим интерфейсом имеют отдельные потоки исполнения для сбора событий в пользовательском интерфейсе из среды операционной системы. В этой главе речь пойдет о том, как внедрять многопоточные возможности в прикладные программы на Java.

Однако многопоточная обработка может оказаться очень сложным делом. В этой главе будут рассмотрены все инструментальные средства, которые понадобятся прикладным программистам. Но для более сложного программирования на системном уровне рекомендуется обратиться к таким основательным первоисточникам, как, например, книга Брайана Гоеца (Brian Goetz) *Java Concurrency in Practice* (издательство Addison-Wesley Professional, 2006 г.).

12.1. Назначение потоков исполнения

Начнем с рассмотрения примера программы, в которой применяются два потока исполнения. Эта программа служит для перевода денежных средств между банковскими счетами. В ней используется класс `Bank` для хранения остатков на банковских счетах и метод `transfer()` — для перевода денежных сумм с одного счета на другой. Конкретная реализация данной программы приведена в листинге 12.2. В первом потоке исполнения денежные средства переводятся со счета 0 на счет 1, а во втором потоке — со счета 2 на счет 3.

Ниже приведена простая процедура исполнения конкретной задачи в отдельном потоке.

1. Введите код выполняемой задачи в тело метода `run()` из класса, реализующего интерфейс `Runnable`. Этот интерфейс очень прост и содержит следующий единственный метод:

```
public interface Runnable
{
    void run();
}
```

2. Интерфейс `Runnable` является функциональным, поэтому его экземпляр можно создать с помощью лямбда-выражения следующим образом:

```
Runnable r = () -> { код задачи };
```

3. Сконструируйте объект типа `Thread` из объекта `r` типа `Runnable`, как показано ниже.

```
var = new Thread(r);
```

4. Запустите поток на исполнение следующим образом:

```
t.start();
```

Чтобы перевести денежные средства в отдельном потоке исполнения, достаточно ввести соответствующий код в тело метода `run()` и запустить данный поток на исполнение:

```
Runnable r = () -> {
    try
    {
        for (int i = 0; i < STEPS; i++)
        {
            int toAccount = (int) (bank.size() * Math.random());
            double amount = MAX_AMOUNT * Math.random();
            bank.transfer(fromAccount, toAccount, amount);
            Thread.sleep((int) (DELAY * Math.random()));
        }
    }
    catch (InterruptedException e)
    {
    }
};
var t = new Thread(r);
t.start();
```

В течение заданного количества шагов цикла в данном потоке исполнения переводится произвольная денежная сумма, после чего он переходит в состояние ожидания на время, определяемое произвольной задержкой. А поскольку в методе `sleep()` может быть сгенерировано исключение типа `InterruptedException`, то оно должно быть перехвачено. Более подробно о перехвате исключений в потоках исполнения речь пойдет далее в разделе 12.3.1. Как правило, прерывание потока исполнения служит для отправки запроса на завершение данного потока. И когда возникает исключение типа `InterruptedException`, метод `run()` завершается.

Данная программа запускает второй поток на исполнение, а также переводит денежные средства со счета 2 на счет 3. Выполнение данной программы приводит к результату, аналогичному следующему:

```
Thread[Thread-1,5,main] 606.77 from 2 to 3
    Total Balance: 400000.00
Thread[Thread-0,5,main] 98.99 from 0 to 1
    Total Balance: 400000.00
Thread[Thread-1,5,main] 476.78 from 2 to 3
    Total Balance: 400000.00
Thread[Thread-0,5,main] 653.64 from 0 to 1
    Total Balance: 400000.00
Thread[Thread-1,5,main] 807.14 from 2 to 3
    Total Balance: 400000.00
Thread[Thread-0,5,main] 481.49 from 0 to 1
    Total Balance: 400000.00
Thread[Thread-0,5,main] 203.73 from 0 to 1
    Total Balance: 400000.00
Thread[Thread-1,5,main] 111.76 from 2 to 3
    Total Balance: 400000.00
Thread[Thread-1,5,main] 794.88 from 2 to 3
    Total Balance: 400000.00
. . .
```

Как видите, результаты исполнения двух потоков перемежаются. Это ясно указывает на то, что они исполняются параллельно. В действительности эти результаты становятся несколько запутанными, когда две выводимые строки перемежаются.

Вот, собственно, и все! Данный пример ясно показывает, каким образом организуется параллельное выполнение задач. В остальной части этой главы поясняется, как управлять взаимодействием потоков исполнения. Исходный код рассмотренной здесь программы полностью приведен в листингах 12.1 и 12.2.



НА ЗАМЕТКУ! Потоки исполнения можно также определить, создавая подклассы, производные от класса **Thread**, как показано ниже.

```
class MyThread extends Thread
{
    public void run()
    {
        код задачи
    }
}
```

Далее конструируется объект этого подкласса и вызывается его метод **start()**. Но такой подход не рекомендуется. Задачу, которая должна выполняться параллельно, следует отделять от механизма ее выполнения. Если имеется много задач, то было бы слишком неэффективно создавать отдельный поток исполнения для каждой из них. Вместо этого можно организовать пул потоков исполнения, как поясняется в разделе 12.6.2.



ВНИМАНИЕ! Не вызывайте метод **run()** из класса **Thread** или объекта типа **Runnable**. При прямом вызове этого метода конкретная задача будет выполнена в том же потоке, а новый поток исполнения не будет запущен. Вместо этого вызывайте метод **Thread.start()**, который создаст новый поток, где будет выполнен метод **run()**.

Листинг 12.1. Исходный код из файла **threads/ThreadTest.java**

```
1 package threads;
2
3 /**
4  * @version 1.30 2004-08-01
5  * @author Cay Horstmann
6  */
7 public class ThreadTest
8 {
9     public static final int DELAY = 10;
10    public static final int STEPS = 100;
11    public static final double MAX_AMOUNT = 1000;
12
13    public static void main(String[] args)
14    {
15        var bank = new Bank(4, 100000);
16        Runnable task1 = () ->
17        {
18            try
19            {
20                for (int i = 0; i < STEPS; i++)
21                {
```

```
22         double amount = MAX_AMOUNT * Math.random();
23         bank.transfer(0, 1, amount);
24         Thread.sleep((int) (DELAY * Math.random()));
25     }
26 }
27 catch (InterruptedException e)
28 {
29 }
30 };
31
32 Runnable task2 = () ->
33 {
34     try
35     {
36         for (int i = 0; i < STEPS; i++)
37         {
38             double amount = MAX_AMOUNT * Math.random();
39             bank.transfer(2, 3, amount);
40             Thread.sleep((int) (DELAY * Math.random()));
41         }
42     }
43     catch (InterruptedException e)
44     {
45     }
46 };
47
48 new Thread(task1).start();
49 new Thread(task2).start();
50 }
51 }
```

Листинг 12.2. Исходный код из файла `threads/Bank.java`

```
1 package threads;
2
3 import java.util.*;
4
5 public class Bank
6 {
7     private final double[] accounts;
8
9     /**
10      * Конструирует объект банка
11      * @param n Количество счетов
12      * @param initialBalance Первоначальный остаток
13      *                      на каждом счете
14      */
15     public Bank(int n, double initialBalance)
16     {
17         accounts = new double[n];
18         Arrays.fill(accounts, initialBalance);
19     }
20
21     /**
22      * Переводит деньги с одного счета на другой
```



```
23  * @param from Счет, с которого переводятся деньги
24  * @param to Счет, на который переводятся деньги
25  * @param amount Сумма перевода
26  */
27  public void transfer(int from, int to, double amount)
28  {
29      if (accounts[from] < amount) return;
30      System.out.print(Thread.currentThread());
31      accounts[from] -= amount;
32      System.out.printf(" %10.2f from %d to %d",
33                      amount, from, to);
34      accounts[to] += amount;
35      System.out.printf(" Total Balance: %10.2f%n",
36                      getTotalBalance());
37  }
38
39  /**
40   * Получает сумму остатков на всех счетах
41   * @return Возвращает общий баланс
42   */
43  public double getTotalBalance()
44  {
45      double sum = 0;
46
47      for (double a : accounts)
48          sum += a;
49      return sum;
50  }
51
52  /**
53   * Получает количество счетов в банке
54   * @return Возвращает количество счетов
55   */
56  public int size()
57  {
58      return accounts.length;
59  }
60 }
```

java.lang.Thread 1.0

- **Thread(Runnable target)**
Конструирует новый поток исполнения, вызывающий метод **run()** для указанного целевого объекта.
- **void start()**
Запускает поток исполнения, иницируя вызов метода **run()**. Этот метод немедленно возвращает управление. Новый поток исполняется параллельно.
- **void run()**
Вызывает метод **run()** для связанного с ним объекта типа **Runnable**.
- **static void sleep(long millis)**
Переходит в состояние ожидания на заданное число миллисекунд.

`java.lang Runnable 1.0`

- `void run()`

Этот метод следует переопределить и ввести в него инструкции для исполнения требуемой задачи в потоке.

12.2. Состояния потоков исполнения

Потоки могут находиться в одном из шести состояний:

- новый;
- исполняемый;
- блокированный;
- ожидающий;
- временно ожидающий;
- завершенный.

Каждое из этих состояний поясняется в последующих разделах. Чтобы определить текущее состояние потока исполнения, достаточно вызвать метод `getState()`.

12.2.1. Новые потоки исполнения

Если поток исполнения создан в результате операции `new`, например `new Thread(r)`, то он еще не запущен на выполнение. Это означает, что он находится в новом состоянии и программа еще не запустила на исполнение код в данном потоке. Прежде чем поток исполнения будет запущен, необходимо выполнить определенные подготовительные операции.

12.2.2. Исполняемые потоки

Как только вызывается метод `start()`, поток оказывается в исполняемом состоянии. Исполняемый поток может выполняться или не выполняться в данный момент, поскольку от операционной системы зависит, будет ли выделено потоку время на исполнение. (Но в спецификации Java это отдельное состояние не указывается. Поток по-прежнему находится в исполняемом состоянии.)

Если поток запущен, он не обязательно продолжает исполняться. На самом деле даже желательно, чтобы исполняемый поток периодически приостанавливался, давая возможность выполниться другим потокам. Особенности планирования потоков исполнения зависят от конкретных служб, предоставляемых операционной системой. Системы приоритетного планирования выделяют каждому исполняемому потоку по кванту времени для выполнения его задачи. Когда этот квант времени истекает, операционная система выгружает поток исполнения и дает возможность выполнить-ся другому потоку (рис. 12.2). Выбирая следующий поток исполнения, операционная система принимает во внимание *приоритеты* потоков исполнения, как поясняется далее, в разделе 12.3.5.

Во всех современных настольных и серверных операционных системах применяется приоритетное (вытесняющее) планирование. Но на переносных устройствах вроде

мобильных телефонов может применяться кооперативное планирование. В таких устройствах поток исполнения теряет управление только в том случае, если он вызывает метод `yield()`, заблокирован или находится в состоянии ожидания.

На машинах с несколькими процессорами каждый процессор может исполнять поток, что позволяет иметь несколько потоков, работающих одновременно. Очевидно, что если потоков больше, чем процессоров, то планировщик вынужден заниматься разделением времени для их исполнения. Не следует, однако, забывать, что в любой момент времени исполняемый поток может выполняться или не выполняться. Именно поэтому рассматриваемое здесь состояние потока называется исполняемым, а не исполняющимся.

`java.lang.Thread 1.0`

- `static void yield()`

Вынуждает текущий исполняемый поток уступить управление другому потоку. Следует, однако, иметь в виду, что этот метод — статический.

12.2.3. Блокированные и ожидающие потоки исполнения

Когда поток исполнения находится в состоянии блокировки или ожидания, он временно не активен. Он не выполняет никакого кода и потребляет минимум ресурсов. На планировщике потоков лежит обязанность повторно активизировать его. Подробности зависят от того, как было достигнуто неактивное состояние.

- Когда поток исполнения пытается получить встроенную блокировку объектов (но не объект типа `Lock` из библиотеки `java.util.concurrent`), которая в настоящий момент захвачена другим потоком исполнения, он становится *блокированным*. (Блокировки из библиотеки `java.util.concurrent` будут обсуждаться в разделе 12.4.3, а встроенные блокировки объектов — в разделе 12.4.5.) Поток исполнения разблокируется, когда все остальные потоки освобождают блокировку и планировщик потоков позволяет данному потоку захватить ее.
- Когда поток исполнения ожидает от другого потока уведомления планировщика о наступлении некоторого условия, он входит в состояние ожидания. Эти условия рассматриваются в разделе 12.4.4. Переход в состояние ожидания происходит при вызове метода `Object.wait()` или `Thread.join()` или в ожидании объекта типа `Lock` или `Condition` из библиотеки `java.util.concurrent`. Но на практике отличия состояний блокировки и ожидания несущественны.
- Несколько методов принимают в качестве параметра время ожидания. Их вызов вводит поток исполнения в состояние *временного ожидания*, которое сохраняется до тех пор, пока не истечет заданное время ожидания или не будет получено соответствующее уведомление. К числу методов со временем ожидания относятся `Object.wait()`, `Thread.join()`, `Lock.tryLock()` и `Condition.await()`.

На рис. 12.1 показаны состояния, в которых может находиться поток исполнения, а также возможные переходы между ними. Когда поток исполнения находится в состоянии блокировки или ожидания (и, конечно, когда он завершается), к запуску планируется другой поток. А когда поток исполнения активизируется повторно (например, по истечении времени ожидания или в том случае, если ему удастся захватить

блокировку), планировщик потоков сравнивает его приоритет с приоритетом выполняющихся в данный момент потоков. Если приоритет данного потока исполнения ниже, он приостанавливается и запускается новый поток.

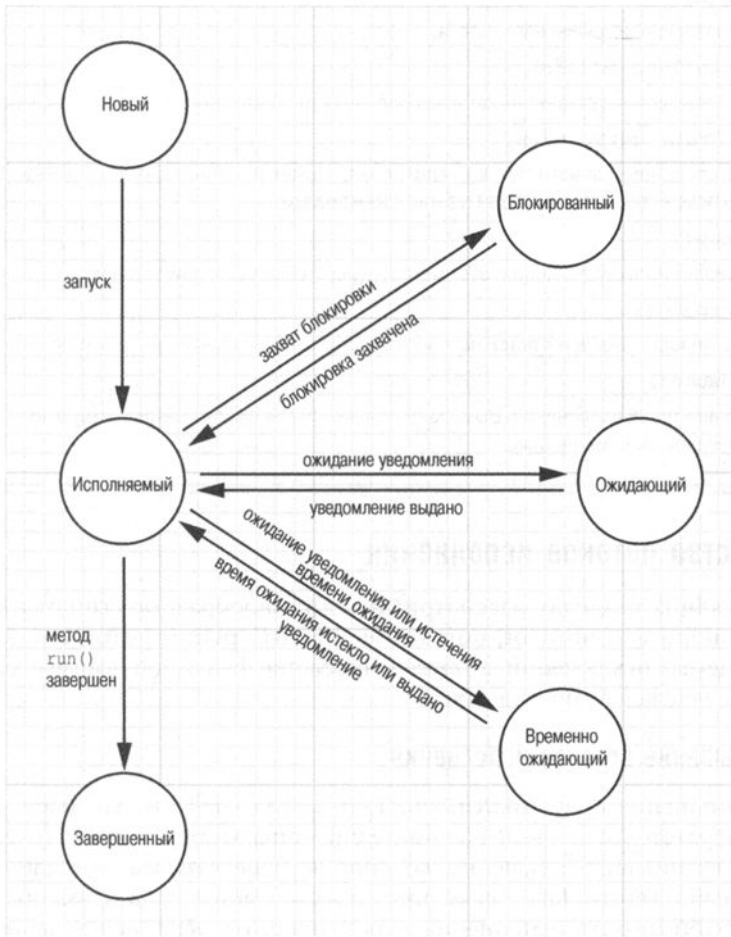


Рис. 12.1. Состояния потока исполнения

12.2.4. Завершенные потоки исполнения

Поток исполнения завершается по одной из следующих причин.

- Прекращает свое существование естественным образом при нормальном завершении метода `run()`.
- Прекращает свое существование внезапно, поскольку неперехваченное исключение прерывает выполнение метода `run()`.

В частности, поток исполнения можно уничтожить, вызвав его метод `stop()`, генерирующий объект ошибки типа `ThreadDeath`, который уничтожает поток исполнения. Но метод `stop()` не рекомендован к применению, поэтому следует избегать его применения в прикладном коде.

java.lang.Thread 1.0

- **void join()**
Ожидает завершения указанного потока.
- **void join(long millis)**
Ожидает завершения указанного потока исполнения или истечения заданного периода времени.
- **Thread.State getState()** 5
Получает состояние данного потока исполнения. Может принимать значения **NEW**, **RUNNABLE**, **BLOCKED**, **WAITING**, **TIMED_WAITING** или **TERMINATED**.
- **void stop()**
Останавливает поток исполнения. Этот метод не рекомендован к применению.
- **void suspend()**
Временно приостанавливает поток исполнения. Этот метод не рекомендован к применению.
- **void resume()**
Возобновляет поток исполнения. Вызывается только после вызова метода **suspend()**. Этот метод не рекомендован к применению.

12.3. Свойства потоков исполнения

В последующих разделах рассматриваются разнообразные свойства потоков исполнения: приоритеты потоков, потоковые демоны, группы потоков и обработчики необрабатываемых исключений, а также устаревшие функциональные средства, которыми вообще не следует пользоваться.

12.3.1. Прерывание потоков исполнения

Поток исполнения прерывается, когда его метод **run()** возвращает управление, выполнив оператор **return** вслед за последним оператором в своем теле, или в том случае, если возникает исключение, которое не перехватывается в данном методе. В первоначальной версии Java также присутствовал метод **stop()**, который мог быть вызван из другого потока исполнения, чтобы прервать исполнение данного потока. Но теперь этот метод не рекомендуется к применению, как поясняется далее, в разделе 12.4.13.

Поток можно прервать *принудительно* только с помощью метода **stop()**, не рекомендованного к применению. Но для *запроса* на прерывание потока исполнения может быть вызван метод **interrupt()**. Когда метод **interrupt()** вызывается для потока исполнения, устанавливается *состояние прерывания* данного потока. Это состояние устанавливается с помощью признака типа **boolean**, имеющегося у каждого потока исполнения. В каждом потоке исполнения следует периодически проверять значение данного признака, чтобы знать, когда поток должен быть прерван. Чтобы выяснить, было ли установлено состояние прерывания потока исполнения, следует вызвать статический метод **Thread.currentThread()**, получающий текущий поток исполнения и вызывающий далее метод **isInterrupted()**, как показано ниже.

```
while (!Thread.currentThread().isInterrupted())  
    && дополнительные действия)
```

```
{  
    выполнить дополнительные действия  
}
```

Но если поток исполнения заблокирован, то проверить состояние его прерывания нельзя. И здесь на помощь приходит исключение типа `InterruptedException`. Когда метод `interrupt()` вызывается для потока исполнения, который заблокирован, например, в результате вызова метода `sleep()` или `wait()`, блокирующий вызов прерывается исключением типа `InterruptedException`. (Существуют блокирующие вызовы ввода-вывода, которые не могут быть прерваны. В таких случаях следует рассмотреть альтернативные способы прерывания потока исполнения. Подробнее об этом речь пойдет в главах 2 и 4 второго тома настоящего издания.)

В языке Java не существует такого требования, чтобы прерванный поток прекратил свое исполнение. Прерывание лишь привлекает внимание потока. А прерванный поток может сам решить, как реагировать на прерывание его исполнения. Некоторые потоки настолько важны, что должны обрабатывать исключение и продолжать свое исполнение. Но зачастую поток должен просто интерпретировать прерывание как запрос на прекращение своего исполнения. Метод `run()` такого потока имеет следующий вид:

```
Runnable r = () -> {  
    try  
    {  
        . . .  
        while (!Thread.currentThread().isInterrupted()  
            && дополнительные действия)  
        {  
            выполнить дополнительные действия  
        }  
    }  
    catch(InterruptedException e)  
    {  
        // поток прерван во время ожидания или приостановки  
    }  
    finally  
    {  
        выполнить очистку, если требуется  
    }  
    // выходом из метода run() завершается исполнение потока  
};
```

Вызывать метод `isInterrupted()` для того, чтобы проверить состояние прерывания потока исполнения, совсем не обязательно, да и неудобно, если вместо этого можно вызвать метод `sleep()` (или другой прерываемый метод) после каждого рабочего шага цикла. Если же метод `sleep()` вызывается, когда установлено состояние прерывания, поток исполнения не переходит в состояние ожидания. Вместо этого он очищает свое состояние (!) и генерирует исключение типа `InterruptedException`. Так, если метод `sleep()` вызывается в цикле, то проверять состояние прерывания не следует. Вместо этого лучше организовать перехват исключения типа `InterruptedException`, как показано ниже.

```
Runnable r = () -> {  
    try  
    {
```

```

    . . .
    while (дополнительные действия)
    {
        выполнить дополнительные действия
        Thread.sleep(delay);
    }
}
catch (InterruptedException e)
{
    // поток прерван во время ожидания
}
finally
{
    выполнить очистку, если требуется
}
// выходом из метода run() завершается исполнение потока
};

```



НА ЗАМЕТКУ! Для проверки состояния прерывания потока исполнения имеются два очень похожих метода: `interrupted()` и `isInterrupted()`. Статический метод `interrupted()` проверяет, был ли прерван текущий поток исполнения. Более того, вызов этого метода приводит к очистке состояния прерывания потока исполнения. С другой стороны, метод экземпляра `isInterrupted()` можно использовать для проверки, был ли прерван любой поток исполнения. Его вызов не приводит к изменению состояния прерывания.

Можно найти немало опубликованных примеров кода, где прерывание типа `InterruptedException` подавляется на низком уровне, как показано ниже.

```

void mySubTask()
{
    . . .
    try { sleep(delay); }
    catch (InterruptedException e) {} // НЕ ИГНОРИРОВАТЬ!
    . . .
}

```

Не поступайте так! Если вы не можете придумать ничего полезного из того, что можно было бы сделать в блоке оператора `catch`, вам остаются на выбор два обоснованных варианта.

- Сделать в блоке оператора `catch` вызов `Thread.currentThread().interrupt()`, чтобы установить состояние прерывания потока исполнения, как выделено ниже полужирным. И тогда это состояние может быть проверено в вызывающей части программы.

```

void mySubTask()
{
    . . .
    try { sleep(delay); }
    catch (InterruptedException e)
    { Thread.currentThread().interrupt(); }
    . . .
}

```

- А еще лучше указать выражение `throws InterruptedException` в сигнатуре метода, как выделено ниже полужирным, а также удалить блок оператора `try` из его тела. И тогда это прерывание может быть перехвачено в вызывающей части программы, а в крайнем случае — в методе `run()`.

```
void mySubTask() throws InterruptedException
{
    . . .
    sleep(delay);
    . . .
}
```

java.lang.Thread 1.0

- **void interrupt()**
Посылает потоку исполнения запрос на прерывание. Признак состояния прерывания потока исполнения устанавливается равным логическому значению **true**. Если поток в данный момент блокирован вызовом метода `sleep()`, генерируется исключение типа `InterruptedException`.
- **static boolean interrupted()**
Проверяет, был ли прерван текущий поток исполнения. Следует, однако, иметь в виду, что это статический метод. Его вызов имеет побочный эффект: признак состояния прерывания текущего потока исполнения устанавливается равным логическому значению **false**.
- **boolean isInterrupted()**
Проверяет, был ли прерван поток исполнения. В отличие от статического метода `interrupted()`, вызов этого метода не изменяет состояние прерывания потока исполнения.
- **static Thread currentThread()**
Возвращает объект типа `Thread`, представляющий текущий поток исполнения.

12.3.2. Потокосные демоны

Превратить поток исполнения в *потокосный демон* можно, сделав следующий вызов:

```
t.setDaemon(true);
```

Правда, в таком потоке исполнения нет ничего демонического. Демон — это лишь поток, у которого нет других целей, кроме служения другим. В качестве примера можно привести потоки исполнения таймера, посылающие регулярные “такты” другим потокам, или же потоки исполнения, очищающие устаревшие записи в кеше. Когда остаются только потокосные демоны, виртуальная машина завершает работу. Нет смысла продолжать выполнение программы, когда все оставшиеся потоки исполнения являются демонами.

java.lang.Thread 1.0

- **void setDaemon(boolean isDaemon)**
Помечает данный поток исполнения как демон или пользовательский поток. Этот метод должен вызываться перед запуском потока исполнения.

12.3.3. Именованние потоков исполнения

По умолчанию потоку исполнения присваиваются привлекающие внимание имена вроде Thread-2. Тем не менее потоку исполнения можно присвоить любое имя, вызвав метод `setName()`, как показано ниже. Такой возможностью удобно пользоваться при выводе потоков исполнения из оперативной памяти на экран или печать.

```
var t = new Thread(runnable);  
t.setName("Web crawler");
```

12.3.4. Обработчики необрабатываемых исключений

Метод `run()` потока исполнения не может генерировать никаких проверяемых исключений, но может быть прерван непроверяемым исключением. В этом случае поток исполнения уничтожается. Но такой конструкции `catch`, куда может распространиться исключение, не существует. Вместо этого, перед тем, как поток исполнения прекратит свое существование, исключение передается обработчику необрабатываемых исключений. Такой обработчик должен относиться к классу, реализующему интерфейс `Thread.UncaughtExceptionHandler`. У этого интерфейса имеется единственный метод:

```
void uncaughtException(Thread t, Throwable e)
```

Этот обработчик можно установить в любом потоке исполнения с помощью метода `setUncaughtExceptionHandler()`. Кроме того, можно установить обработчик по умолчанию для всех потоков с помощью статического метода `setDefaultUncaughtExceptionHandler()` из класса `Thread`. В заменяющем обработчике может использоваться прикладной интерфейс API протоколирования для отправки отчетов о необрабатываемых исключениях в файл протокола.

Если не установить обработчик по умолчанию, то такой обработчик оказывается пустым (`null`). Но если не установить обработчик для отдельного потока исполнения, то им становится объект потока типа `ThreadGroup`.



НА ЗАМЕТКУ! Группа потоков — это коллекция потоков исполнения, которой можно управлять совместно. По умолчанию все создаваемые потоки исполнения относятся к одной и той же группе потоков, но можно устанавливать и другие группы. Теперь в Java имеются более совершенные средства для выполнения операций над коллекциями потоков исполнения, поэтому пользоваться группами потоков в собственных программах не рекомендуется.

Класс `ThreadGroup` реализует интерфейс `Thread.UncaughtExceptionHandler`. Его метод `uncaughtException()` выполняет следующие действия.

1. Если у группы потоков имеется родительская группа, то из нее вызывается метод `uncaughtException()`.
2. Иначе, если метод `Thread.getDefaultExceptionHandler()` возвращает непустой обработчик (т.е. не `null`), то вызывается именно этот обработчик.
3. Иначе, если объект типа `Throwable` является экземпляром класса `ThreadDeath`, то ничего не происходит.
4. Иначе имя потока исполнения и трассировка стека объекта типа `Throwable` выводятся в стандартный поток сообщений об ошибках `System.err`.

В итоге производится трассировка стека, которую, без сомнения, приходилось не раз наблюдать в своих программах.

`java.lang.Thread 1.0`

- `static void setDefaultUncaughtExceptionHandler (Thread.UncaughtExceptionHandler handler) 5`
- `static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler () 5`
- Устанавливают или получают обработчик по умолчанию для необрабатываемых исключений.
- `void setUncaughtExceptionHandler (Thread.UncaughtExceptionHandler handler) 5`
- `Thread.UncaughtExceptionHandler getUncaughtExceptionHandler () 5`
Устанавливают или получают обработчик для необрабатываемых исключений. Если обработчик вообще не установлен, таким обработчиком становится объект группы потоков исполнения.

`java.lang.Thread.UncaughtExceptionHandler 5`

- `void uncaughtException(Thread t, Throwable e)`
Определяется для протоколирования специального отчета по завершении потока исполнения с необрабатываемым исключением.

`java.lang.ThreadGroup 1.0`

- `void uncaughtException(Thread t, Throwable e)`
Этот метод вызывается из родительской группы потоков, если таковая имеется, или же вызывается обработчик по умолчанию из класса `Thread`, если таковой имеется, а иначе выводится трассировка стека в стандартный поток сообщений об ошибках. (Но если `e` — объект типа `ThreadDeath`, то трассировка стека подавляется. Объекты типа `ThreadDeath` формируются устаревшим и не рекомендованным к применению методом `stop()`.)

12.3.5. Приоритеты потоков исполнения

В языке программирования Java у каждого потока исполнения имеется свой *приоритет*. По умолчанию поток исполнения наследует приоритет того потока, который его создал. Повысить или понизить приоритет любого потока исполнения можно, вызвав метод `setPriority()`. А установить приоритет потока исполнения можно, указав любое значение в пределах от `MIN_PRIORITY` (определено в классе `Thread` равным 1) до `MAX_PRIORITY` (равно 10). Обычному приоритету соответствует значение `NORM_PRIORITY`, равное 5.

Всякий раз, когда планировщик потоков выбирает новый поток для исполнения, он предпочитает потоки с более высоким приоритетом. Но приоритеты потоков исполнения *сильно зависят* от системы. Когда виртуальная машина полагается на реализацию потоков средствами главной платформы, на которой она выполняется,

приоритеты потоков Java привязываются к уровням приоритетов этой платформы, где может быть больше или меньше уровней приоритетов.

Например, в Windows предусмотрено семь уровней приоритетов. Некоторые приоритеты Java привязываются к тому же самому уровню приоритета операционной системы. В виртуальной машине Oracle для Linux приоритеты потоков исполнения вообще игнорируются. Все потоки исполнения имеют одинаковый приоритет.

Приоритеты потоков исполнения могли приносить пользу в прежних версиях Java, где использовались потоки исполнения операционной системы, а теперь пользоваться ими не рекомендуется.

`java.lang.Thread 1.0`

- **`void setPriority(int newPriority)`**

Устанавливает приоритет потока исполнения. Приоритет должен находиться в пределах от `Thread.MIN_PRIORITY` до `Thread.MAX_PRIORITY`. Для нормального приоритета указывается значение `Thread.NORM_PRIORITY`.

- **`static int MIN_PRIORITY`**

Минимальный приоритет, который может иметь объект типа `Thread`. Значение минимального приоритета равно 1.

- **`static int NORM_PRIORITY`**

Приоритет объекта типа `Thread` по умолчанию. Значение нормального приоритета по умолчанию равно 5.

- **`static int MAX_PRIORITY`**

Максимальный приоритет, который может иметь объект типа `Thread`. Значение максимального приоритета равно 10.

12.4. Синхронизация

В большинстве практических многопоточных приложений двум или более потокам исполнения приходится разделять общий доступ к одним и тем же данным. Что произойдет, если два потока исполнения имеют доступ к одному объекту и каждый из них вызывает метод, изменяющий состояние этого объекта? Нетрудно догадаться, что потоки исполнения станут наступать друг другу на пятки. В зависимости от порядка обращения к данным можно в конечном итоге получить поврежденный объект. Такая ситуация обычно называется *состоянием гонок*.

12.4.1. Пример состояния гонок

Чтобы избежать повреждения данных, совместно используемых многими потоками, нужно научиться *синхронизировать доступ* к ним. В этом разделе будет показано, что произойдет, если не применять синхронизацию. А в следующем разделе поясняется, как синхронизировать обращение к данным.

В следующем примере тестовой программы продолжается имитация работы банка. Но, в отличие от примера, рассматривавшегося ранее в разделе 12.1, в данном случае произвольно выбирается источник и место назначения обмена данными. Но поскольку это может вызвать осложнения, проанализируем более тщательно исходный код метода `transfer()` из класса `Bank`.

```

public void transfer(int from, int to, double amount)
    // ВНИМАНИЕ: вызывать этот метод из
    // нескольких потоков небезопасно!
{
    System.out.print(Thread.currentThread());
    accounts[from] -= amount;
    System.out.printf(" %10.2f from %d to %d",
                      amount, from, to);
    accounts[to] += amount;
    System.out.printf(" Total Balance: %10.2f%n",
                      getTotalBalance());
}

```

Ниже приведен исходный код для экземпляров типа `Runnable`. Метод `run()` переводит деньги с фиксированного банковского счета. В каждой транзакции метод `run()` выбирает случайный целевой счет и произвольную сумму, вызывает метод `transfer()` для объекта `Bank`, а затем переводит поток исполнения в состояние ожидания.

```

Runnable r = () -> {
    try
    {
        while (true)
        {
            int toAccount = (int) (bank.size() * Math.random());
            double amount = MAX_AMOUNT * Math.random();
            bank.transfer(fromAccount, toAccount, amount);
            Thread.sleep((int) (DELAY * Math.random()));
        }
    }
    catch (InterruptedException e)
    {
    }
};

```

Когда выполняется данная имитация банка, неизвестно, какая именно сумма находится на любом банковском счете в произвольный момент времени. Но в то же время известно, что общая сумма денег по всем счетам должна оставаться неизменной, поскольку деньги только переводятся с одного счета на другой, а не снимаются окончательно.

В конце каждой транзакции метод `transfer()` заново вычисляет итоговую сумму на счетах и выводит ее. Данная программа вообще не прекращает выполняться. Чтобы удалить ее, следует нажать комбинацию клавиш `<Ctrl+C>`. Ниже приведен типичный результат, выводимый данной программой.

```

. . .
Thread[Thread-11,5,main] 588.48 from 11 to 44
    Total Balance: 100000.00
Thread[Thread-12,5,main] 976.11 from 12 to 22
    Total Balance: 100000.00
Thread[Thread-14,5,main] 521.51 from 14 to 22
    Total Balance: 100000.00
Thread[Thread-13,5,main] 359.89 from 13 to 81
    Total Balance: 100000.00
. . .
Thread[Thread-36,5,main] 401.71 from 36 to 73

```

```

Total Balance: 99291.06
Thread[Thread-35,5,main] 691.46 from 35 to 77
Total Balance: 99291.06
Thread[Thread-37,5,main] 78.64 from 37 to 3
Total Balance: 99291.06
Thread[Thread-34,5,main] 197.11 from 34 to 69
Total Balance: 99291.06
Thread[Thread-36,5,main] 85.96 from 36 to 4
Total Balance: 99291.06
. . .
Thread[Thread-4,5,main]Thread[Thread-33,5,main]
7.31 from 31 to 32
Total Balance: 99979.24
627.50 from 4 to 5 Total Balance: 99979.24
. . .

```

Как видите, что-то в этой программе пошло не так. В течение нескольких транзакций общий баланс в имитируемом банке оставался равным сумме \$100000, что совершенно верно, поскольку первоначально было 100 счетов по \$1000 на каждом. Но через некоторое время общий баланс немного изменился. Запустив эту программу на выполнение, вы можете обнаружить, что ошибка возникнет очень быстро или же общий баланс будет нарушен нескоро. Такая ситуация не внушает доверия, и вы вряд ли захотите положить свои заработанные тяжким трудом денежки в такой банк!

Попробуйте сами найти ошибку в исходном коде из листинга 12.3 и в классе Bank из листинга 12.2. А причины ее появления будут раскрыты в следующем разделе.

Листинг 12.3. Исходный код из файла `unsynch/UnsynchBankTest.java`

```

1 package unsynch;
2
3 /**
4  * В этой программе демонстрируется нарушение
5  * данных при произвольном доступе к структуре
6  * данных из многих потоков
7  * @version 1.32 2018-04-10
8  * @author Cay Horstmann
9  */
10 public class UnsynchBankTest
11 {
12     public static final int NACCOUNTS = 100;
13     public static final double INITIAL_BALANCE = 1000;
14     public static final double MAX_AMOUNT = 1000;
15     public static final int DELAY = 10;
16
17     public static void main(String[] args)
18     {
19         var bank = new Bank(NACCOUNTS, INITIAL_BALANCE);
20         for (int i = 0; i < NACCOUNTS; i++)
21         {
22             int fromAccount = i;
23             Runnable r = () -> {
24                 try

```

```
25     {
26         while (true)
27         {
28             int toAccount = (int) (bank.size()
29                                     * Math.random());
30             double amount = MAX_AMOUNT * Math.random();
31             bank.transfer(fromAccount, toAccount,
32                           amount);
33             Thread.sleep((int) (DELAY * Math.random()));
34         }
35     }
36     catch (InterruptedException e)
37     {
38     }
39 };
40 var t = new Thread(r);
41 t.start();
42 }
43 }
44 }
```

12.4.2. Объяснение причин, приводящих к состоянию гонок

В предыдущем разделе рассмотрен пример программы, где остатки на банковских счетах обновлялись в нескольких потоках исполнения. По истечении некоторого времени в ней накапливаются ошибки, а в итоге некоторая сумма теряется или появляется неизвестно откуда. Подобная ошибка возникает в том случае, если в двух потоках исполнения предпринимается одновременная попытка обновить один и тот же счет. Допустим, что в двух потоках исполнения одновременно выполняется следующая операция:

```
accounts[to] += amount;
```

Дело в том, что такие операции не являются *атомарными*. Приведенная выше операция может быть выполнена поэтапно следующим образом.

1. Загрузить значение из элемента массива `accounts[to]` в регистр.
2. Добавить значение `amount`.
3. Переместить результат обратно в элемент массива `accounts[to]`.

А теперь представим, что в первом потоке выполняются операции из пп. 1 и 2, после чего его исполнение приостанавливается. Допустим, что второй поток исполнения выходит из состояния ожидания и в нем обновляется тот же самый элемент массива `accounts`. Затем из состояния ожидания выходит первый поток, и в нем выполняется операция из п. 3. Такое действие уничтожает изменения, внесенные во втором потоке исполнения. В итоге общий баланс подсчитан неверно (рис. 12.2). Такое нарушение данных обнаруживается в рассматриваемой здесь тестовой программе. (Разумеется, существует вероятность получить сигнал ложной тревоги, если поток исполнения будет прерван во время тестирования!)

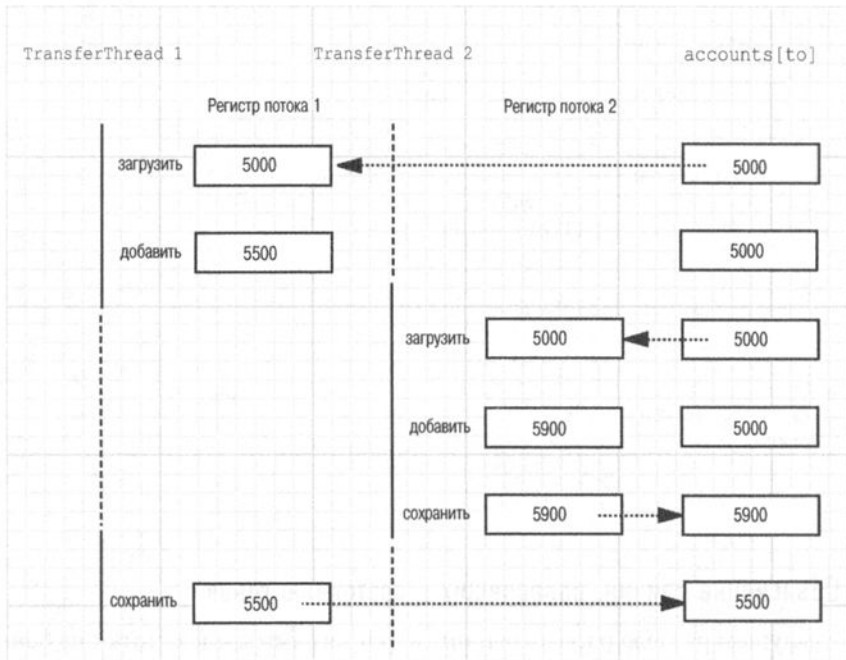


Рис. 12.2. Одновременный доступ к данным из двух потоков исполнения



НА ЗАМЕТКУ! Байт-коды, выполняемые виртуальной машиной в классе `Bank`, можно просмотреть. Для этого достаточно ввести следующую команду декомпиляции файла `Bank.class`:

```
javap -c -v Bank
```

Например, строка кода

```
accounts[to] += amount;
```

транслируется в следующий байт-код:

```
aload_0
getfield #2; // поле accounts:[D
iload_2
dup2
daload
dload_3
dadd
dastore
```

Неважно, что именно означают эти байт-коды. Важнее другое: операция инкрементирования состоит из нескольких команд, и исполняющий их поток может быть прерван на любой из них.

Какова вероятность повредить данные? В современных многоядерных процессорах риск нарушения режима параллельной обработки довольно велик. В рассматриваемом здесь примере вероятность проявления этого недостатка была увеличена за счет того, что операторы вывода перемежались операторами обновления общего баланса.

Если пропустить операторы вывода, то риск повреждения немного снизится, поскольку в каждом потоке будет выполняться настолько мало операций, прежде чем он перейдет в состояние ожидания, что прерывание посреди вычислений окажется

маловероятным. Но и в этом случае риск повреждения данных не исключается полностью. Если запустить достаточно много потоков исполнения на сильно загруженной машине, то программа даст сбой даже при отсутствии в ней операторов вывода. Сбой произойдет через минуты, часы или даже дни. Вообще говоря, для программиста нет ничего хуже, чем ошибка в программе, которая проявляется только раз в несколько дней.

Суть рассматриваемой здесь программной ошибки состоит в том, что выполнение метода `transfer()` может быть прервано на полпути к его завершению. Если удастся гарантировать нормальное завершение этого метода до того, как его поток утратит управление, то состояние объекта банковского счета вообще не будет нарушено.

12.4.3. Объекты блокировки

Имеются два механизма для защиты блока кода от параллельного доступа. В языке Java для этой цели предусмотрено ключевое слово `synchronized`, а в версии Java 5 появился еще и класс `ReentrantLock`. Ключевое слово `synchronized` автоматически обеспечивает блокировку, как и связанное с ней “условие”, которое удобно указывать в большинстве случаев, когда требуется явная блокировка. Но понять ключевое слово `synchronized` проще, если рассмотреть блокировки и условия по отдельности. В библиотеке `java.util.concurrent` предоставляются отдельные классы для реализации этих основополагающих механизмов, принцип действия которых поясняется в разделе 12.4.4. Разъяснив, каким образом устроены эти основные составляющие многопоточной обработки, мы перейдем к разделу 12.4.5.

Защита блока кода средствами класса `ReentrantLock` в общих чертах выглядит следующим образом:

```
myLock.lock(); // объект типа ReentrantLock
try
{
    критический раздел кода
}
finally
{
    myLock.unlock(); // непременно снять блокировку,
                    // даже если генерируется исключение
}
```

Такая конструкция гарантирует, что только один поток исполнения в единицу времени сможет войти в критический раздел кода. Как только один поток исполнения заблокирует объект блокировки, никакой другой поток не сможет обойти вызов метода `lock()`. И если другие потоки исполнения попытаются вызвать метод `lock()`, то они будут деактивизированы до тех пор, пока первый поток не снимет блокировку с объекта блокировки.



ВНИМАНИЕ! Крайне важно расположить вызов метода `unlock()` в блоке `finally`. Если код в критическом разделе сгенерирует исключение, блокировка должна быть снята. В противном случае другие потоки исполнения будут заблокированы навсегда.



НА ЗАМЕТКУ! Пользоваться блокировками вместе с оператором `try` с ресурсами нельзя. Ведь метод разблокировки не называется `close()`. Но даже если бы он так назывался, то его все равно нельзя было бы применять вместе с оператором `try` с ресурсами, поскольку в заголовке этого оператора предполагается объявление новой переменной. Но производя блокировку, следует использовать одну и ту же переменную, общую для всех потоков исполнения.

Воспользуемся блокировкой для защиты метода `transfer()` из класса `Bank`, как показано ниже.

```
public class Bank
{
    private Lock bankLock = new ReentrantLock();
        // объект класса ReentrantLock,
        // реализующего интерфейс Lock
    . . .
    public void transfer(int from, int to, int amount)
    {
        bankLock.lock();
        try
        {
            System.out.print(Thread.currentThread());
            accounts[from] -= amount;
            System.out.printf(" %10.2f from %d to %d",
                               amount, from, to);
            accounts[to] += amount;
            System.out.printf(" Total Balance: %10.2f%n",
                               getTotalBalance());
        }
        finally
        {
            bankLock.unlock();
        }
    }
}
```

Допустим, в одном потоке исполнения вызывается метод `transfer()` и до его завершения этот поток приостанавливается, и во втором потоке исполнения также вызывается метод `transfer()`. Второй поток не сможет захватить блокировку и останется заблокированным при вызове метода `lock()`. Напротив, он будет деактивизирован и вынужден ждать до тех пор, пока выполнение метода `transfer()` не завершится в первом потоке. И только тогда, когда первый поток снимет блокировку, второй поток сможет продолжить свое исполнение (рис. 12.3).

Опробуйте описанный выше механизм синхронизации потоков исполнения, введя блокирующий код в метод `transfer()` и снова запустив рассматриваемую здесь программу. Можете прогонять ее бесконечно, но общий баланс банка на этот раз не нарушится.

Однако у каждого объекта типа `Bank` имеется свой собственный объект типа `ReentrantLock`. Если два потока исполнения попытаются обратиться к одному и тому же объекту типа `Bank`, блокировка послужит для сериализации доступа. Но если два потока исполнения обращаются к разным объектам типа `Bank`, то каждый из них захватывает свою блокировку и ни один из потоков не блокируется. Так и должно быть, потому что потоки исполнения никак не мешают друг другу, оперируя разными экземплярами класса `Bank`.

Блокировка называется *реентерабельной*, потому что поток исполнения может повторно захватывать блокировку, которой он уже владеет. Для блокировки предусмотрен *счетчик захватов*, отслеживающий вложенные вызовы метода `lock()`. И для каждого вызова `lock()` в потоке должен быть вызван метод `unlock()`, чтобы, в конце концов, снять блокировку. Благодаря этому средству код, защищенный блокировкой, может вызывать другой метод, использующий ту же самую блокировку.

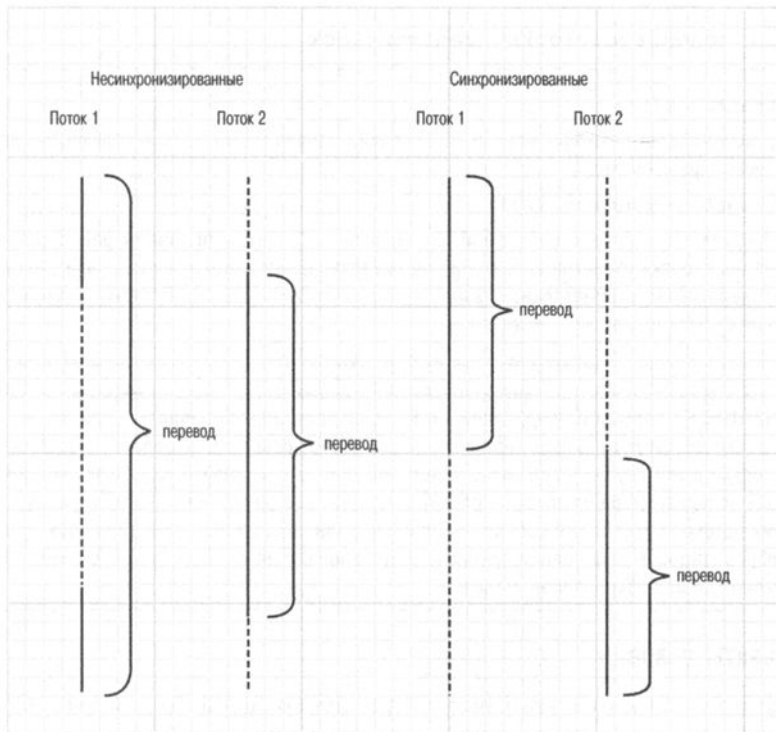


Рис. 12.3. Сравнение синхронизированных и не синхронизированных потоков исполнения

Например, метод `transfer()` вызывает метод `getTotalBalance()`, который также блокирует объект `bankLock`, у которого теперь значение счетчика захватов равно 2. Когда метод `getTotalBalance()` завершается, значение счетчика захватов возвращается к 1. При выходе из метода `transfer()` счетчик захватов имеет значение 0, и поток исполнения снимает блокировку.

Как правило, требуется защищать блоки кода, обновляющие и проверяющие объект, разделяемый потоками. Тогда можно не сомневаться, что эти операции завершатся, прежде чем тот же самый объект сможет быть использован в другом потоке исполнения.



ВНИМАНИЕ! Будьте внимательны, чтобы код в критическом разделе программы не был пропущен из-за генерирования исключения. Если исключение сгенерировано до конца критического раздела кода, блокировка будет снята в блоке **finally**, но объект может оказаться в поврежденном состоянии.

```
java.util.concurrent.locks.Lock 5
```

- **void lock()**

Захватывает блокировку. Если же в данный момент она захвачена другим потоком, текущий поток блокируется.

- **void unlock()**

Снимает блокировку.

java.util.concurrent.locks.ReentrantLock 5

- **ReentrantLock ()**

Конструирует объект реентерабельной блокировки, которая может быть использована для защиты критического раздела кода.

- **ReentrantLock (boolean fair)**

Конструирует объект реентерабельной блокировки с заданным правилом равноправия. Равноправная блокировка отдает предпочтение потоку исполнения, ожидающему дольше всех. Но такое равноправие может отрицательно сказаться на производительности. Поэтому по умолчанию равноправия от блокировок не требуется.



ВНИМАНИЕ! На первый взгляд, лучше, чтобы блокировка была равноправной, но равноправные блокировки действуют *намного* медленнее обычных. Разрешить равноправную блокировку вы можете только в том случае, если точно знаете, что делаете, и имеете на то особые причины. Даже если вы используете равноправную блокировку, у вас все равно нет никаких гарантий, что планировщик потоков будет также соблюдать правило равноправия. Если планировщик потоков решит пренебречь потоком исполнения, который длительное время ожидает снятия блокировки, то никакое равноправие блокировок не поможет.

12.4.4. Объекты условий

Нередко поток входит в критический раздел кода только для того, чтобы обнаружить, что он не может продолжить свое исполнение до тех пор, пока не будет соблюдено определенное условие. В подобных случаях для управления теми потоками, которые захватили блокировку, но не могут выполнить полезные действия, служит *объект условия*. В этом разделе будет представлена реализация объектов условий в библиотеке Java (по ряду исторических причин объекты условий нередко называются *условными переменными*).

Попробуем усовершенствовать рассматриваемую здесь программу имитации банка. Не будем перемещать деньги со счета, если он не содержит достаточной суммы, чтобы покрыть расходы на перевод. Но для этого не годится код, подобный следующему:

```
if (bank.getBalance(from) >= amount)
    bank.transfer(from, to, amount)
```

Ведь вполне возможно, что текущий поток будет деактивизирован в промежутке между успешным выполнением проверки и вызовом метода `transfer()`:

```
if (bank.getBalance(from) >= amount)
    // поток исполнения может быть
    // деактивизирован в этом месте кода
bank.transfer(from, to, amount);
```

В тот момент, когда возобновляется исполнение потока, остаток на счете может измениться, т.е. уменьшиться ниже допустимого предела. Поэтому нужно каким-то образом гарантировать, что никакой другой поток не сможет изменить остаток на счете между его проверкой и переводом денег. Для этого придется защитить как проверку остатка на счете, так и сам перевод денег с помощью следующей блокировки:

```
public void transfer(int from, int to, int amount)
{
```

```
bankLock.lock();
try
{
    while (accounts[from] < amount)
    {
        // ожидать
        . . .
    }
    // перевести денежные средства
    . . .
}
finally
{
    bankLock.unlock();
}
```

Что делать дальше, если на счете нет достаточной суммы? Ожидать до тех пор, пока счет не будет пополнен в каком-то другом потоке исполнения. Но ведь данный поток только что получил монопольный доступ к объекту `bankLock`, так что ни в одном другом потоке нет возможности пополнить счет. И здесь на помощь приходит объект условия.

С объектом блокировки может быть связан один или несколько объектов условий, которые получены с помощью метода `newCondition()`. Каждому объекту условия можно присвоить имя, напоминающее об условии, которое он представляет. Например, объект, представляющий условие “достаточных денежных средств”, устанавливается следующим образом:

```
class Bank
{
    private Condition sufficientFunds;
    . . .
    public Bank()
    {
        . . .
        sufficientFunds = bankLock.newCondition();
    }
}
```

Если в методе `transfer()` будет обнаружено, что средств на счете недостаточно, он сделает следующий вызов:

```
sufficientFunds.await();
```

Текущий поток исполнения теперь деактивизирован и снимает блокировку. Это дает возможность пополнить счет в другом потоке.

Имеется существенное отличие между потоком, ожидающим возможности захватить блокировку, и потоком, который вызвал метод `await()`. Как только в потоке исполнения вызывается метод `await()`, он входит в *набор ожидания*, установленный для данного условия. Поток не становится исполняемым, когда доступна блокировка. Вместо этого он остается деактивизированным до тех пор, пока другой поток не вызовет метод `signalAll()` по тому же условию.

Когда перевод денег будет произведен в другом потоке исполнения, в нем должен быть сделан следующий вызов:

```
sufficientFunds.signalAll();
```

В результате этого вызова активизируются все потоки исполнения, ожидающие данного условия. Когда потоки удаляются из набора ожидания, они опять становятся исполняемыми, и в конечном итоге планировщик потоков активизирует их снова. В этот момент они попытаются повторно захватить объект блокировки. И как только он окажется доступным, один из этих потоков захватит блокировку и *продолжит* свое исполнение с *того места*, где он *остановился*, получив управление после вызова метода `await()`.

В этот момент условие должно быть снова проверено в потоке исполнения. Но нет никаких гарантий, что условие теперь выполнится. Ведь метод `signalAll()` просто сигнализирует ожидающим потокам о том, что условие теперь *может* быть удовлетворено и что его стоит проверить заново.



НА ЗАМЕТКУ! Вообще говоря, вызов метода `await()` должен быть введен в цикл следующей формы:

```
while (!(можно продолжить))
    условие.await();
```

Крайне важно, чтобы в конечном итоге метод `signalAll()` был вызван в каком-нибудь другом потоке исполнения. Когда метод `await()` вызывается в потоке исполнения, последний не имеет возможности повторно активизировать самого себя. И здесь он полностью полагается на другие потоки. Если ни один из них не позаботится о повторной активизации ожидающего потока, его выполнение никогда не возобновится. Это может привести к неприятной ситуации *взаимной блокировки*. Если все прочие потоки исполнения будут заблокированы, а метод `await()` будет вызван в последнем активном потоке без разблокировки какого-нибудь другого потока, этот поток также окажется заблокированным. И тогда не останется ни одного потока исполнения, где можно было бы разблокировать другие потоки, а следовательно, программа зависнет.

Когда же следует вызывать метод `signalAll()`? Существует эмпирическое правило: вызывать этот метод при таком изменении состояния объекта, которое может быть выгодно ожидающим потокам исполнения. Например, всякий раз, когда изменяются остатки на счетах, ожидающим потокам исполнения следует давать очередную возможность для проверки остатков на счетах. В данном примере метод `signalAll()` вызывается по завершении перевода денег, как показано ниже.

```
public void transfer(int from, int to, int amount)
{
    bankLock.lock();
    try
    {
        while (accounts[from] < amount)
            sufficientFunds.await();
        // перевести денежные средства
        . . .
        sufficientFunds.signalAll();
    }
    finally
    {
        bankLock.unlock();
    }
}
```

Однако вызов метода `signalAll()` не влечет за собой немедленной активизации ожидающего потока исполнения. Он лишь разблокирует ожидающие потоки, чтобы они могли соперничать за объект блокировки после того, как текущий поток снимет блокировку.

Другой метод, `signal()`, разблокирует только один поток из набора ожидания, выбирая его случайным образом. Это более эффективно, чем разблокировать все потоки, хотя здесь существует определенная опасность. Если случайно выбранный поток обнаружит, что еще не может продолжить свое исполнение, он вновь заблокируется. И если никакой другой поток не вызовет снова метод `signal()`, то система перейдет в состояние взаимной блокировки.



ВНИМАНИЕ! По условию в потоке исполнения может быть вызван только метод `await()`, `signalAll()` или `signal()`, когда этот поток владеет блокировкой по данному условию.

Запустив на выполнение видоизмененный вариант имитирующей банк программы из листинга 12.4, вы обнаружите, что теперь она работает правильно. Общий баланс в \$100000 сохраняется неизменным, и ни на одном из счетов нет отрицательного остатка. (Но для того чтобы прервать выполнение этой программы, вам снова придется нажать комбинацию клавиш `<Ctrl+C>`.) Вы можете также заметить, что программа работает немного медленнее — это та цена, которую приходится платить за дополнительные служебные операции, связанные с механизмом синхронизации.

На практике правильно употреблять условия не так-то просто. Поэтому, прежде чем пытаться реализовать собственные объекты условий, стоит рассмотреть применение конструкций, описанных далее, в разделе 12.5.

Листинг 12.4. Исходный код из файла `synch/Bank.java`

```
1 package synch;
2
3 import java.util.*;
4 import java.util.concurrent.locks.*;
5
6 /**
7  * Программа, имитирующая банк со счетами и использующая
8  * блокировки для организации последовательного доступа
9  * к остаткам на счетах
10 */
11 public class Bank
12 {
13     private final double[] accounts;
14     private Lock bankLock;
15     private Condition sufficientFunds;
16     /**
17      * Конструирует объект банка
18      * @param n Количество счетов
19      * @param initialBalance Первоначальный остаток
20      *                      на каждом счете
21      */
22     public Bank(int n, double initialBalance)
23     {
24         accounts = new double[n];
25         Arrays.fill(accounts, initialBalance);
```

```
26     bankLock = new ReentrantLock();
27     sufficientFunds = bankLock.newCondition();
28 }
29 /**
30  * Переводит деньги с одного счета на другой
31  * @param from Счет, с которого переводятся деньги
32  * @param to Счет, на который переводятся деньги
33  * @param amount Сумма перевода
34  */
35 public void transfer(int from, int to, double amount)
36     throws InterruptedException
37 {
38     bankLock.lock();
39     try
40     {
41         while (accounts[from] < amount)
42             sufficientFunds.await();
43         System.out.print(Thread.currentThread());
44         accounts[from] -= amount;
45         System.out.printf(" %10.2f from %d to %d",
46             amount, from, to);
47         accounts[to] += amount;
48         System.out.printf(" Total Balance: %10.2f%n",
49             getTotalBalance());
50         sufficientFunds.signalAll();
51     }
52     finally
53     {
54         bankLock.unlock();
55     }
56 }
57
58 /**
59  * Получает сумму остатков на всех счетах
60  * @return Возвращает общий баланс
61  */
62 public double getTotalBalance()
63 {
64     bankLock.lock();
65     try
66     {
67         double sum = 0;
68
69         for (double a : accounts)
70             sum += a;
71
72         return sum;
73     }
74     finally
75     {
76         bankLock.unlock();
77     }
78 }
79 /**
80  * Получает количество счетов в банке
81  * @return Возвращает количество счетов
```

```
82    */
83    public int size()
84    {
85        return accounts.length;
86    }
87 }
```

java.util.concurrent.locks.Lock 5

- **Condition newCondition()**
Возвращает объект условия, связанный с данной блокировкой.

java.util.concurrent.locks.Condition 5

- **void await()**
Вводит поток исполнения в набор ожидания по данному условию.
- **void signalAll()**
Разблокирует все потоки исполнения в наборе ожидания по данному условию.
- **void signal()**
Разблокирует один произвольно выбранный поток исполнения в наборе ожидания по данному условию.

12.4.5. Ключевое слово **synchronized**

В предыдущих разделах было показано, как пользоваться объектами блокировки типа `Lock` и условиями типа `Condition`. Прежде чем двигаться дальше, подведем краткие итоги, перечислив главные особенности блокировок и условий.

- Блокировка защищает критические разделы кода, позволяя выполнять этот код только в одном потоке в единицу времени.
- Блокировка управляет потоками исполнения, которые пытаются войти в защищенный раздел кода.
- Каждый объект условия управляет потоками, которые вошли в защищенный раздел кода, но пока еще не в состоянии продолжить свое исполнение.

Интерфейсы `Lock` и `Condition` предоставляют программистам высокую степень контроля над блокировками. Но зачастую такой контроль не требуется, и оказывается достаточно механизма, встроенного в язык Java. Еще со времен версии 1.0 *каждый объект* в Java обладает встроенной блокировкой. Если метод объявлен с ключевым словом `synchronized`, то блокировка объекта защищает весь этот метод. Следовательно, поток исполнения должен захватить встроенную блокировку объекта, чтобы вызвать такой метод.

Иными словами, следующий фрагмент кода:

```
public synchronized void метод()
{
    тело метода
}
```


равнозначен такому фрагменту:

```
public void метод()  
{  
    this.встроеннаяБлокировка.lock();  
    try  
    {  
        тело метода  
    }  
    finally { this.встроеннаяБлокировка.unlock(); }  
}
```

Например, вместо явной блокировки можно просто объявить метод `transfer()` из класса `Bank` как `synchronized`. Встроенная блокировка объектов имеет единственное связанное с ней условие. Метод `wait()` вводит поток исполнения в набор ожидания, а методы `notifyAll()/notify()` разблокируют ожидающие потоки. Иными словами, вызов метода `wait()` или `notifyAll()` равнозначен следующему коду:

```
встроенноеУсловие.await();  
встроенноеУсловие.signalAll();
```



НА ЗАМЕТКУ! Методы `wait()`, `notifyAll()` и `notify()` являются конечными (`final`) методами из класса `Object`. А методы из интерфейса `Condition` должны именоваться `await`, `signalAll` и `signal`, чтобы не вступать в конфликт с этими методами.

Например, класс `Bank` можно реализовать только языковыми средствами Java следующим образом:

```
class Bank  
{  
    private double[] accounts;  
  
    public synchronized void transfer(int from, int to,  
        int amount) throws InterruptedException  
    {  
        while (accounts[from] < amount)  
            wait(); // ожидать по единственному условию  
                // встроенной блокировки объектов  
        accounts[from] -= amount;  
        accounts[to] += amount;  
        notifyAll(); // уведомить все потоки,  
                    // ожидающие по данному условию  
    }  
    public synchronized double getTotalBalance() { . . . }  
}
```

Как видите, применение ключевого слова `synchronized` порождает намного более краткий код. Разумеется, чтобы понять такой код, нужно знать, что каждый объект обладает встроенной блокировкой и что эта блокировка имеет встроенное условие. Блокировка управляет потоками исполнения, которые пытаются войти в метод `synchronized`. А условие управляет потоками исполнения, вызвавшими метод `wait()`.



СОВЕТ. Синхронизированные методы относительно просты. Но начинающие программировать на Java нередко испытывают затруднения в обращении с условиями. Поэтому, прежде чем применять методы `wait()/notifyAll()`, рекомендуется ознакомиться с одной из конструкций, описанных в разделе 12.5.

Статические методы также допускается объявлять синхронизированными. Когда вызывается такой метод, он захватывает встроенную блокировку объекта соответствующего класса. Так, если в классе `Bank` имеется статический синхронизированный метод, при его вызове захватывается блокировка объекта типа `Bank.class`. В результате к этому объекту не может обратиться никакой другой поток исполнения и никакой другой синхронизированный статический метод того же класса.

Встроенным блокировкам и условиям присущи некоторые ограничения, в том числе приведенные ниже.

- Нельзя прервать поток исполнения, который пытается захватить блокировку.
- Нельзя указать время ожидания, пытаясь захватить блокировку.
- Наличие единственного условия на блокировку может оказаться неэффективным.

Что же лучше использовать в прикладном коде: объекты типа `Lock` и `Condition` или синхронизированные методы? Ниже приведены некоторые рекомендации, которые дают ответ на этот вопрос.

- Лучше не пользоваться ни объектами типа `Lock/Condition`, ни ключевым словом `synchronized`. Зачастую вместо этого можно выбрать подходящий механизм из пакета `java.util.concurrent`, который организует блокировку автоматически. Так, в разделе 12.5.1 будет показано, как пользоваться блокирующими очередями для синхронизации потоков, выполняющих общую задачу.
- Если ключевое слово `synchronized` подходит в конкретной ситуации, непременно воспользуйтесь им. В этом случае вам придется написать меньше кода, а следовательно, допустить меньше ошибок. В листинге 12.5 приведен пример очередного варианта программы, имитирующей банк и реализованной на основе синхронизированных методов.
- Пользуйтесь объектами типа `Lock/Condition`, если действительно нуждаетесь в дополнительных возможностях подобных конструкций.

Листинг 12.5. Исходный код из файла `synch2/Bank.java`

```
1 package synch2;
2
3 import java.util.*;
4
5 /**
6  * Программа, имитирующая банк со счетами, используя
7  * примитивные языковые конструкции для синхронизации
8  * потоков исполнения
9  */
10 public class Bank
11 {
12     private final double[] accounts;
13
14     /**
15      * Конструирует объект банка
16      * @param n Количество счетов
17      * @param initialBalance Первоначальный остаток
18      *                      на каждом счете
```

```
19  */
20  public Bank(int n, double initialBalance)
21  {
22      accounts = new double[n];
23      Arrays.fill(accounts, initialBalance);
24  }
25
26  /**
27   * Переводит деньги с одного счета на другой
28   * @param from Счет, с которого переводятся деньги
29   * @param to Счет, на который переводятся деньги
30   * @param amount Сумма перевода
31   */
32  public synchronized void transfer(int from, int to,
33      double amount) throws InterruptedException
34  {
35      while (accounts[from] < amount)
36          wait();
37      System.out.print(Thread.currentThread());
38      accounts[from] -= amount;
39      System.out.printf(" %10.2f from %d to %d",
40          amount, from, to);
41      accounts[to] += amount;
42      System.out.printf(" Total Balance: %10.2f%n",
43          getTotalBalance());
44      notifyAll();
45  }
46
47  /**
48   * Получает сумму остатков на всех счетах
49   * @return Возвращает общий баланс
50   */
51  public synchronized double getTotalBalance()
52  {
53      double sum = 0;
54
55      for (double a : accounts)
56          sum += a;
57
58      return sum;
59  }
60
61  /**
62   * Получает сумму остатков на всех счетах
63   * @return Возвращает общий баланс
64   */
65  public int size()
66  {
67      return accounts.length;
68  }
69 }
```

java.lang.Object 1.0

- **void notifyAll()**

Разблокирует потоки исполнения, вызвавшие метод **wait()** для данного объекта. Может быть вызван только из тела синхронизированного метода или блока кода. Генерирует исключение типа **IllegalMonitorStateException**, если поток исполнения не владеет блокировкой данного объекта.

- **void notify()**

Разблокирует один произвольно выбранный поток исполнения среди потоков, вызвавших метод **wait()** для данного объекта. Может быть вызван только из тела синхронизированного метода или блока кода. Генерирует исключение типа **IllegalMonitorStateException**, если поток исполнения не владеет блокировкой данного объекта.

- **void wait()**

Вынуждает поток исполнения ожидать уведомления в течение указанного периода времени. Вызывается только из синхронизированного метода или блока кода. Генерирует исключение типа **IllegalMonitorStateException**, если поток исполнения не владеет блокировкой данного объекта.

- **void wait(long millis)**

- **void wait(long millis, int nanos)**

Вынуждают поток исполнения ожидать уведомления в течение указанного периода времени. Вызываются только из синхронизированного метода или блока кода. Генерируют исключение типа **IllegalMonitorStateException**, если поток исполнения не владеет блокировкой данного объекта. Задаваемое количество миллисекунд не может превышать 1000000.

12.4.6. Синхронизированные блоки

Как упоминалось выше, у каждого объекта в Java имеется собственная встроенная блокировка. Поток исполнения может захватить эту блокировку, вызвав синхронизированный метод. Но есть и другой механизм захвата блокировки — вхождение в *синхронизированный блок*. Когда поток исполнения входит в блок кода, объявляемый в приведенной ниже форме, он захватывает блокировку объекта `obj`.

```
synchronized (obj) // это синтаксис
                  // синхронизированного блока
{
    критический раздел кода
}
```

Иногда в прикладном коде встречаются специальные блокировки вроде следующей:

```
public class Bank
{
    private double[] accounts;
    private var lock = new Object();
    . . .
    public void transfer(int from, int to, int amount)
    {
        synchronized (lock) // специальная блокировка
        {
            accounts[from] -= amount;
            accounts[to] += amount;
        }
    }
}
```

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

Здесь объект `lock` создается только для использования встроенной блокировки, которая имеется у каждого объекта в Java. Встроенной блокировкой объекта иногда пользуются для реализации дополнительных атомарных операций. Такая практика получила название *клиентской блокировки*. Рассмотрим для примера класс `Vector`, где реализуется список с синхронизированными методами. А теперь допустим, что остатки на банковских счетах сохранены в объекте типа `Vector<Double>`. Ниже приведена наивная реализация метода `transfer()`.

```
public void transfer(Vector<Double> accounts, int from,  
                    int to, int amount) // ОШИБКА!  
{  
    accounts.set(from, accounts.get(from) - amount);  
    accounts.set(to, accounts.get(to) + amount);  
    System.out.println(. . .);  
}
```

Методы `get()` и `set()` из класса `Vector` синхронизированы, но это вряд ли поможет. Вполне возможно, что поток исполнения будет приостановлен в методе `transfer()` по завершении первого вызова метода `get()`. В другом потоке исполнения может быть затем установлено иное значение на той же позиции. Блокировку можно захватить следующим образом:

```
public void transfer(Vector<Double> accounts, int from,  
                    int to, int amount)  
{  
    synchronized (accounts)  
    {  
        accounts.set(from, accounts.get(from) - amount);  
        accounts.set(to, accounts.get(to) + amount);  
    }  
    System.out.println(. . .);  
}
```

Такой подход вполне работоспособен, но он полностью зависит от того факта, что встроенная блокировка используется в классе `Vector` для всех его модифицирующих методов. Но так ли это на самом деле? В документации на класс `Vector` этого не обещается. Поэтому следует очень тщательно проанализировать исходный код этого класса, надеясь, что в последующие его версии не будут внедрены несинхронизированные модифицирующие методы. Как видите, клиентская блокировка — весьма ненадежный прием, и поэтому он обычно не рекомендуется для применения.



НА ЗАМЕТКУ! В виртуальную машину Java встроена поддержка синхронизированных методов. Тем не менее синхронизированные блоки компилируются в длинные последовательности байт-кодов для управления встроенной блокировкой.

12.4.7. Принцип монитора

Блокировки и условия — эффективные инструментальные средства синхронизации потоков исполнения, но они не слишком объектно-ориентированы. В течение многих лет исследователи искали способы обеспечения безопасности многопоточной обработки,

чтобы избавить программистов от необходимости думать о явных блокировках. Одно из наиболее успешных решений — принцип монитора, который был впервые предложен Пером Бринчем Хансеном (Per Brinch Hansen) и Тони Хоаром (Tony Hoare) в 1970-х годах. В терминологии Java монитор обладает следующими свойствами.

- Монитор — это класс, имеющий только закрытые поля.
- У каждого объекта такого класса имеется связанная с ним блокировка.
- Все методы блокируются этой блокировкой. Иными словами, если клиент вызывает метод `obj.method()`, блокировка объекта `obj` автоматически захватывается в начале этого метода и снимается по его завершении. А поскольку все поля класса монитора закрытые, то такой подход гарантирует, что к ним нельзя будет обратиться ни в одном из потоков исполнения до тех пор, пока ими манипулирует какой-то другой поток.
- У блокировки может быть любое количество связанных с ней условий.

В первоначальных версиях мониторов имелось единственное условие с довольно изящным синтаксисом. Так, можно было просто сделать вызов `await accounts[from] >= balance`, не указывая явную условную переменную. Но исследования показали, что неразборчивая повторная проверка условий может оказаться неэффективной. Проблема была решена благодаря применению явных условных переменных, каждая из которых управляет отдельным рядом потоков исполнения.

Создатели Java вполне адаптировали принцип монитора. *Каждый* объект в Java обладает встроенной блокировкой и встроенным условием. Если метод объявлен с ключевым словом `synchronized`, он действует как метод монитора. А переменная условия доступна через вызовы методов `wait()`, `notifyAll()`, `notify()`.

Но объекты в Java отличаются от мониторов в следующих трех важных отношениях, нарушающих безопасность потоков исполнения.

- Поля не обязательно должны быть закрытыми (`private`).
- Методы не обязаны быть синхронизированными (`synchronized`).
- Встроенная блокировка доступна клиентам.

Это — явное пренебрежение требованиями безопасности, изложенными Пером Бринчем Хансеном. В уничижительном обозрении, посвященном примитивам многозадачной обработки в Java, он пишет: “Для меня является непостижимым тот факт, что небезопасный параллелизм столь серьезно принят сообществом программистов, и это спустя четверть века после изобретения мониторов и языка *Concurrent Pascal*. Этому нет оправданий”. [*Java’s Insecure Parallelism, ACM SIGPLAN Notices* 34:38–45, April 1999].

12.4.8. Поля и переменные типа `volatile`

Плата за синхронизацию кажется порой непомерной, когда нужно просто прочитать или записать данные в одно или два поля экземпляра. В конце концов, что такого страшного может при этом произойти? К сожалению, современные процессоры и компиляторы оставляют немало места для появления ошибок.

- Компьютеры с несколькими процессорами могут временно удерживать значения из памяти в регистрах или локальных кешах. Вследствие этого в потоках, исполняемых на разных процессорах, могут быть доступны разные значения в одной и той же области памяти!

- Компиляторы могут менять порядок выполнения команд для достижения максимальной производительности. Они не меняют этот порядок таким образом, чтобы изменился смысл кода, а лишь делают предположения, что значения в памяти изменяются только явными командами в коде. Но значение в памяти может быть изменено из другого потока исполнения!

Если вы пользуетесь блокировками для защиты кода, который может выполняться в нескольких потоках, то вряд ли столкнетесь с подобными затруднениями. Компиляторы обязаны соблюдать блокировки, очищая при необходимости локальные кеши и не изменяя порядок следования команд. Подробнее об этом можно узнать из документа *Java Memory Model and Thread Specification* (Спецификация модели памяти и потоков исполнения в Java), разработанного экспертной группой JSR 133 (<https://www.jcp.org/en/jsr/detail?id=133>). Большая часть этого документа довольно сложна и полна технических подробностей, но в нем приведен также целый ряд наглядных примеров. Более доступный обзор данной темы, автором которого является Брайан Гоецц, доступен по адресу <https://www.ibm.com/developerworks/library/j-jtp02244/>.



НА ЗАМЕТКУ! Брайан Гоецц предложил следующий “девиз синхронизации”: если вы записываете в переменную данные, которые могут быть затем прочитаны в другом потоке исполнения, или же читаете из переменной данные, которые были записаны в другом потоке исполнения, то обязаны использовать синхронизацию.

Ключевое слово `volatile` обозначает неблокирующий механизм синхронизированного доступа к полю экземпляра. Если поле объявляется как `volatile`, то компилятор и виртуальная машина принимают во внимание тот факт, что поле может быть параллельно обновлено в другом потоке исполнения.

Допустим, у объекта имеется поле признака `done` типа `boolean`, который устанавливается в одном потоке исполнения и опрашивается в другом. Как пояснялось ранее, для этой цели можно организовать встроенную блокировку следующим образом:

```
private boolean done;  
public synchronized boolean isDone() { return done; }  
public synchronized void setDone() { done = true; }
```

Применять встроенную блокировку объекта — вероятно, не самая лучшая идея. Ведь методы `isDone()` и `setDone()` могут блокироваться, если другой поток исполнения заблокировал объект. В таком случае можно воспользоваться отдельной блокировкой только для данной переменной. Но это повлечет за собой немало хлопот. Поэтому в данном случае имеет смысл объявить поле как `volatile` следующим образом:

```
private volatile boolean done;  
public boolean isDone() { return done; }  
public void setDone() { done = true; }
```

В таком случае компилятор вставит подходящий код, чтобы изменения, вносимые в переменную `done` в одном потоке исполнения, были доступны в любом другом потоке исполнения, где читается ее содержимое.



ВНИМАНИЕ! Изменчивые переменные типа `volatile` не гарантируют никакой атомарности операций. Например, приведенный ниже метод не гарантирует смены значения поля на противоположное.

```
public void flipDone() { done = !done; } // не атомарная операция!
```

12.4.9. Поля и переменные типа `final`

Как было показано в предыдущем разделе, благополучно прочитать содержимое поля из нескольких потоков исполнения не удастся, если не применить блокировки или модификатор доступа `volatile`. Но имеется еще одна возможность получить надежный доступ к общему полю, если оно объявлено как `final`. Рассмотрим следующую строку кода:

```
final var accounts = new HashMap<>();
```

Переменная `accounts` станет доступной из других потоков исполнения по завершении конструктора. Если не объявить ее как `final`, то нет никакой гарантии, что обновленное значение переменной `accounts` окажется доступным из других потоков исполнения. Ведь если конструктор класса `HashMap` не завершится нормально, значение этой переменной может оказаться пустым (`null`). Разумеется, операции над отображением не являются потокобезопасными. Если содержимое отображения видоизменяется или читается в нескольких потоках исполнения, то по-прежнему требуется их синхронизация.

12.4.10. Атомарность операций

Общие переменные могут быть объявлены как `volatile`, при условии, что над ними не выполняется никаких операций, кроме присваивания. В пакете `java.util.concurrent.atomic` имеется целый ряд классов, в которых эффективно используются команды машинного уровня, гарантирующие атомарность других операций без применения блокировок. Например, в классе `AtomicInteger` имеются методы `incrementAndGet()` и `decrementAndGet()`, атомарно инкрементирующие или декрементирующие целое значение. Так, безопасно сформировать последовательность чисел можно следующим образом:

```
public static AtomicLong nextNumber = new AtomicLong();  
// В некотором потоке исполнения...  
long id = nextNumber.incrementAndGet();
```

Метод `incrementAndGet()` автоматически инкрементирует переменную типа `AtomicLong` и возвращает ее значение после инкрементирования. Это означает, что операции получения значения, прибавления 1, установки и получения нового значения переменной не могут быть прерваны. Этим гарантируется правильное вычисление и возврат значения даже при одновременном доступе к одному и тому же экземпляру из нескольких потоков исполнения.

Имеются также методы для автоматической установки, сложения и вычитания значений, но если требуется выполнить более сложное их обновление, то придется вызывать метод `compareAndSet()`. Допустим, в нескольких потоках исполнения требуется отслеживать наибольшее значение. Приведенный ниже код для этой цели не годится.

```
public static AtomicLong largest = new AtomicLong();  
// В некотором потоке исполнения...  
largest.set(Math.max(largest.get(), observed));  
// ОШИБКА из-за условия гонок!
```

Такое обновление не является атомарным. Вместо этого следует предоставить лямбда-выражение для автоматического обновления переменной. Так, в рассматриваемом здесь примере можно сделать один из следующих вызовов:

```
largest.updateAndGet(x -> Math.max(x, observed));
```


или

```
largest.accumulateAndGet(observed, Math::max);
```

Метод `accumulateAndGet()` принимает двоичную операцию, применяемую для объединения атомарного значения и предоставляемого аргумента. Имеются также методы `getAndUpdate()` и `getAndAccumulate()`, возвращающие прежнее значение.



НА ЗАМЕТКУ! Упомянутые выше методы предоставляются также для классов `AtomicInteger`, `AtomicIntegerArray`, `AtomicIntegerFieldUpdater`, `AtomicLongArray`, `AtomicLongFieldUpdater`, `AtomicReference`, `AtomicReferenceArray` и `AtomicReferenceFieldUpdater`.

При наличии очень большого количества потоков исполнения, где осуществляется доступ к одним и тем же атомарным значениям, резко снижается производительность, поскольку для оптимистичных обновлений требуется слишком много попыток. В качестве выхода из этого затруднительного положения в версии Java 8 предоставляются классы `LongAdder` и `LongAccumulator`. В частности, класс `LongAdder` состоит из нескольких полей, общая сумма значений в которых составляет текущее значение. Разные слагаемые этой суммы могут обновляться во многих потоках исполнения, а новые слагаемые автоматически предоставляются по мере увеличения количества потоков. В общем случае такой подход к параллельным вычислениям оказывается довольно эффективным, поскольку суммарное значение не требуется до тех пор, пока не будет завершена вся операция. Благодаря этому значительно повышается производительность.

Если предвидится высокая степень состязательности потоков исполнения за доступ к общим данным, то вместо класса `AtomicLong` следует воспользоваться классом `LongAdder`. Методы в этом классе называются несколько иначе. Так, для инкрементирования счетчика вызывается метод `increment()`, для прибавления величины — метод `add()`, а для извлечения итоговой суммы — метод `sum()`, как показано ниже.

```
var adder = new LongAdder();
for (. . .)
    pool.submit(() -> {
        while (. . .) {
            . . .
            if (. . .) adder.increment();
        }
    });
. . .
long total = adder.sum();
```



НА ЗАМЕТКУ! Безусловно, метод `increment()` не возвращает прежнее значение. Ведь это свело бы на нет весь выигрыш в эффективности от разделения суммы на многие слагаемые.

Подобный принцип обобщается в классе `LongAccumulator` до произвольной операции накопления. Конструктору этого класса предоставляется нужная операция, а также нейтральный элемент. Для внедрения новых значений вызывается метод `accumulate()`, а для получения текущего значения — метод `get()`. Так, следующий фрагмент кода дает такой же результат, как и приведенный выше, где применялся класс `LongAdder`:

```
var adder = new LongAccumulator(Long::sum, 0);
// В некотором потоке исполнения...
adder.accumulate(value);
```

В накапливающем сумматоре имеются переменные a_1, a_2, \dots, a_n . Каждая переменная инициализируется нейтральным элементом (в данном случае — 0). Когда метод `accumulate()` вызывается со значением v , одна из этих переменных автоматически обновляется следующим образом: $a_i \text{ op } v$, где `op` — операция накопления в infix-фиксной форме записи. В данном примере в результате вызова метода `accumulate()` вычисляется сумма $a_i = a_i + v$ для некоторой величины i . А вызов метода `get()` приводит к такому результату: $a_1 \text{ op } a_2 \text{ op } \dots \text{ op } a_n$. В данном примере это сумма всех накапливающих сумматоров $a_1 + a_2 + \dots + a_n$.

Если выбрать другую операцию, то можно вычислить максимум или минимум. В общем, операция должна быть ассоциативной или коммутативной. Это означает, что конечный результат не должен зависеть от порядка, в котором объединяются промежуточные значения.

Имеются также классы `DoubleAdder` и `DoubleAccumulator`, которые действуют аналогичным образом, только оперируют значениями типа `double`.

12.4.11. Взаимные блокировки

Блокировки и условия не могут решить всех проблем, которые возникают при многопоточной обработке. Рассмотрим следующую ситуацию.

1. Счет 1: сумма 200 долл.
2. Счет 2: сумма 300 долл.
3. Поток 1: переводит сумму 300 долл. со счета 1 на счет 2.
4. Поток 2: переводит сумму 400 долл. со счета 2 на счет 1.

Как следует из рис. 12.4, потоки 1 и 2 явно блокируются. Ни один из них не выполняется, поскольку остатков на счетах 1 и 2 недостаточно для выполнения транзакции. Может случиться так, что все потоки исполнения будут заблокированы, поскольку каждый из них будет ожидать пополнения счета. Такая ситуация называется *взаимной блокировкой*.

В рассматриваемой здесь программе взаимная блокировка не может произойти по следующей простой причине: сумма каждого перевода не превышает 1000 долл. А поскольку имеется всего 100 счетов с общим балансом 100000 долл., то как минимум на одном счете в любой момент времени должна быть сумма больше 1000 долл. Поэтому тот поток, где производится перевод денежных средств с данного счета, может продолжить свое исполнение.

Но если внести изменение в метод `run()`, исключив лимит 1000 долл. на транзакцию, то взаимная блокировка произойдет очень скоро. Можете убедиться в этом сами. Для этого установите значение константы `NACCOUNTS` равным 10, сконструируйте объект типа `Runnable` со значением поля `max`, равным `2*INITIAL_BALANCE`, и запустите программу на выполнение. Она будет работать довольно долго, но в конечном итоге зависнет.



СОВЕТ. Когда программа зависнет, нажмите комбинацию клавиш `<Ctrl+\>`. В итоге будет выведено содержимое памяти с перечислением всех потоков исполнения. Для каждого потока исполнения производится трассировка стека, из которой видно, где именно произошла блокировка. В качестве альтернативы запустите утилиту `jconsole`, как описано в главе 7, и перейдите к панели `Threads` (Потоки), приведенной на рис. 12.5.

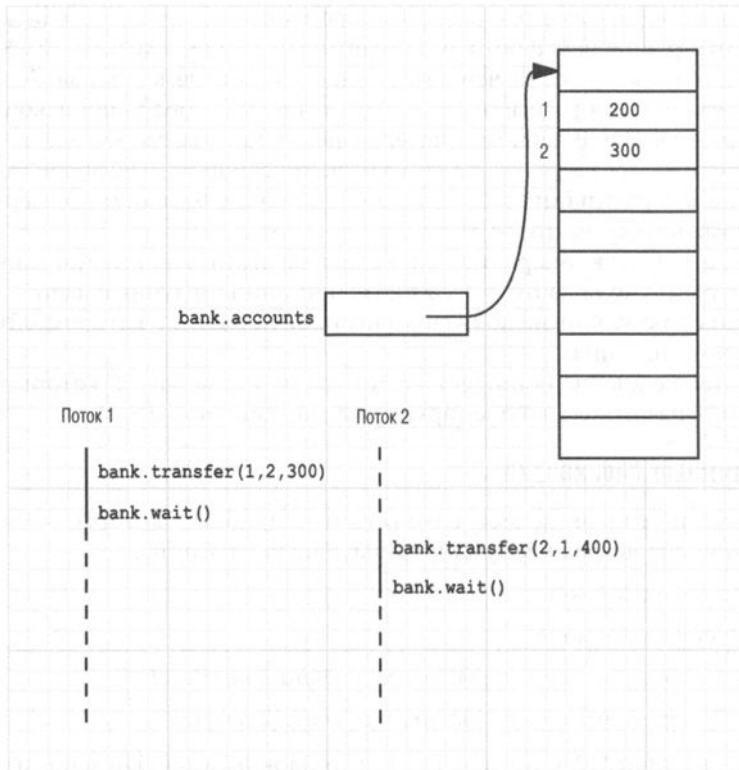


Рис. 12.4. Ситуация взаимной блокировки

Другой способ создать взаимную блокировку — сделать i -й поток исполнения ответственным за размещение денежных средств на i -м счете вместо их снятия с i -го счета. В этом случае имеется вероятность, что все потоки исполнения набросятся на один и тот же счет, и в каждом из них будет предпринята попытка снять деньги с этого счета. Попробуйте смоделировать подобную ситуацию. Для этого обратитесь в программе `SynchBankTest` к методу `run()` из класса `TransferRunnable`, а в вызове метода `transfer()` поменяйте местами параметры `fromAccount` и `toAccount`. После запуска программы вы почти сразу же обнаружите взаимную блокировку.

Взаимная блокировка может легко возникнуть и в том случае, если заменить метод `signalAll()` на `signal()` в программе `SynchBankTest`. Сделав это, вы обнаружите, что программа в конечном итоге зависнет. (И в этом случае лучше установить значение 10 константы `NACCOUNTS`, чтобы как можно скорее добиться желаемого результата.) В отличие от метода `signalAll()`, который извещает все потоки, ожидающие пополнения счета, метод `signal()` разблокирует только один поток исполнения. Если этот поток не может продолжить свое исполнение, то все потоки могут оказаться заблокированными. Рассмотрим следующий простой сценарий создания взаимной блокировки.

1. Счет 1: 1990 долл.
2. Все прочие счета: сумма 990 долл. на каждом.
3. Поток 1: переводит сумму 995 долл. со счета 1 на счет 2.
4. Все прочие потоки: переводят сумму 995 долл. со своего счета на другой счет.

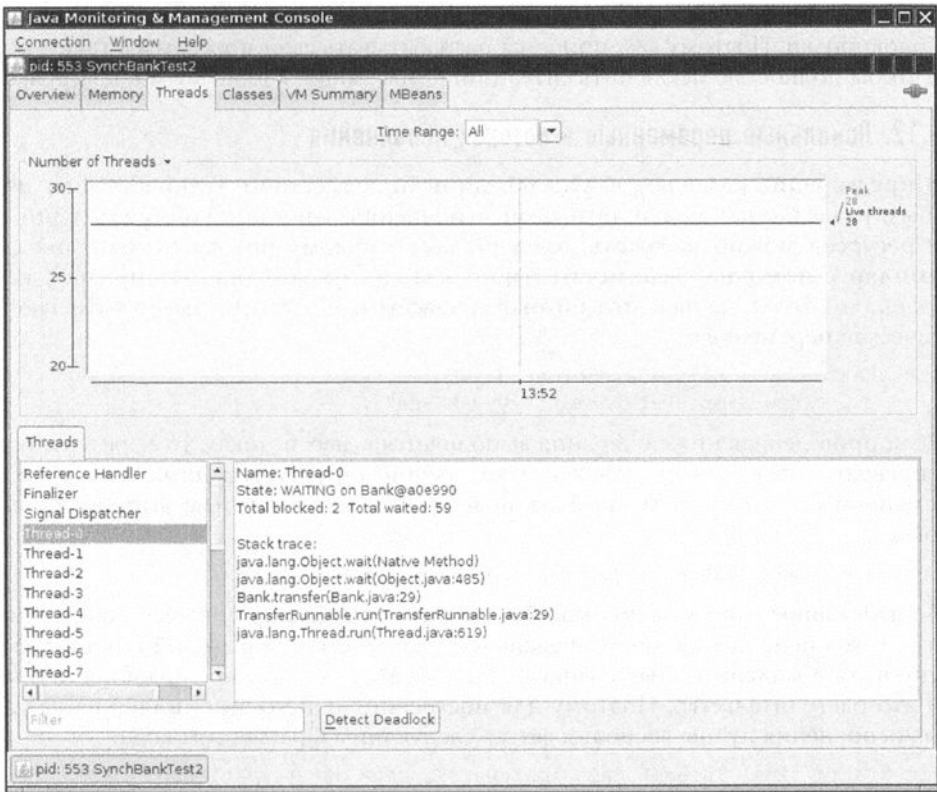


Рис. 12.5. Панель Threads в окне утилиты jconsole

Ясно, что все потоки исполнения, кроме потока 1, заблокированы, поскольку на их счетах недостаточно денег. Поток 1 выполняет перевод денег, после чего возникает следующая ситуация.

1. Счет 1: сумма 995 долл.
2. Счет 2: сумма 1985 долл.
3. Все прочие счета: сумма 990 долл. на каждом счете.

Затем в потоке 1 вызывается метод `signal()`, который произвольным образом выбирает поток исполнения для разблокировки. Допустим, он выберет поток 3. Этот поток исполнения активизируется, затем обнаруживает, что на его счете недостаточно денег, и снова вызывается метод `await()`. Но поток 1 все еще исполняется. В итоге формируется новая произвольная транзакция вроде следующей.

1. Поток 1: переводит сумму 997 долл. со счета 1 на счет 2.

Теперь метод `await()` вызывается и в потоке 1, а в итоге *все* потоки исполнения блокируются. Система входит во взаимную блокировку. И виновником всему — вызов метода `signal()`. Он разблокирует только один поток исполнения, а таким потоком может оказаться совсем не тот, который позволит программе выполняться дальше. (В данном случае перевод денег со счета 2 должен быть произведен в потоке 2.)

К сожалению, в Java отсутствуют средства, исключающие или снимающие взаимные блокировки. Поэтому вам придется разрабатывать свои программы таким образом, чтобы полностью исключить ситуации, приводящие к взаимным блокировкам.

12.4.12. Локальные переменные в потоках исполнения

В предыдущих разделах обсуждались риски совместного использования переменных, разделяемых между потоками исполнения. Иногда такого разделения общих ресурсов можно избежать, предоставляя каждому потоку исполнения свой экземпляр с помощью вспомогательного класса `ThreadLocal`. Например, класс `SimpleDateFormat` не является потокобезопасным. Допустим, имеется следующая статическая переменная:

```
public static final SimpleDateFormat dateFormat =  
    new SimpleDateFormat("yyyy-MM-dd");
```

Если приведенная ниже операция выполняется в двух потоках, то ее результат может превратиться в “мусор”, поскольку внутренние структуры данных, используемые переменной `dateFormat`, могут быть повреждены в параллельно выполняющемся потоке.

```
String dateStamp = dateFormat.format(new Date());
```

Во избежание этого можно было бы, с одной стороны, организовать синхронизацию потоков исполнения, но это недешевое удовольствие, а с другой стороны, сконструировать локальный объект типа `SimpleDateFormat` по мере надобности в нем, но и это расточительство. Поэтому для построения одного экземпляра на каждый поток исполнения лучше воспользоваться следующим фрагментом кода:

```
public static final ThreadLocal<SimpleDateFormat> dateFormat =  
    ThreadLocal.withInitial(() ->  
        new SimpleDateFormat("yyyy-MM-dd"));
```

Для доступа к конкретному средству форматирования делается следующий вызов:

```
String dateStamp = dateFormat.get().format(new Date());
```

При первом вызове метода `get()` в данном потоке исполнения вызывается также метод `initialValue()`. А по его завершении метод `get()` возвращает экземпляр, принадлежащий текущему потоку исполнения.

Аналогичные трудности вызывает генерирование случайных чисел в нескольких потоках исполнения. Для этой цели служит класс `java.util.Random`, который является потокобезопасным. Но и он недостаточно эффективный, если нескольким потокам исполнения приходится ожидать доступа к единственному разделяемому между ними генератору случайных чисел.

Для предоставления каждому потоку отдельного генератора случайных чисел можно было бы воспользоваться вспомогательным классом `ThreadLocal`, но, начиная с версии Java 7, для этой цели предоставляется удобный класс `ThreadLocalRandom`. Достаточно сделать приведенный ниже вызов `ThreadLocalRandom.current()`, и в результате возвратится экземпляр класса `Random`, однозначный для текущего потока исполнения.

```
int random = ThreadLocalRandom.current()  
    .nextInt(upperBound);
```

java.lang.ThreadLocal<T> 1.2

- **T get()**

Получает текущее значение в данном потоке исполнения. Если этот метод вызывается в первый раз, то значение получается в результате вызова метода `initialize()`.

- **void remove()**

Удаляет значение из потока исполнения.

- **void set(T t)**

Устанавливает новое значение для данного потока исполнения.

- **static <S> ThreadLocal<S> withInitial(Supplier<? extends S> supplier) 8**

Создает в потоке исполнения локальную переменную, исходное значение которой получается в результате вызова заданного поставщика информации.

java.util.concurrent.ThreadLocalRandom 7

- **static ThreadLocalRandom current()**

Возвращает экземпляр класса `Random`, однозначный для текущего потока исполнения.

12.4.13. Причины, по которым методы `stop()` и `suspend()` не рекомендованы к применению

В первоначальной версии Java был определен метод `stop()`, который просто останавливал поток исполнения, а также метод `suspend()`, который блокировал поток исполнения до тех пор, пока другой поток не вызывал метод `resume()`. У методов `stop()` и `suspend()` имеется нечто общее: оба пытаются контролировать поведение данного потока исполнения без учета взаимодействия потоков.

Методы `stop()`, `suspend()` и `resume()` не рекомендованы больше к применению. Метод `stop()`, по существу, небезопасен, а что касается метода `suspend()`, то, как показывает опыт, он часто приводит к взаимным блокировкам. В этом разделе поясняется, почему применение этих методов проблематично и что нужно делать, чтобы избежать проблем, которые они вызывают.

Обратимся сначала к методу `stop()`. Он прекращает выполнение любых незавершенных методов, включая и `run()`. Когда поток исполнения останавливается, данный метод немедленно снимает блокировки со всех объектов, которые он блокировал. Это может привести к тому, что объекты останутся в несогласованном состоянии. Допустим, метод `TransferThread()` остановлен посередине перевода денежных средств с одного счета на другой: после снятия денежных средств с одного счета, но перед их переносом на другой счет. В этом случае объект имитируемого банка оказывается поврежденным. А поскольку блокировка снята, этот поврежденный объект будет доступен и другим потокам исполнения, которые не были остановлены.

Когда одному потоку исполнения требуется остановить другой поток, он никоим образом не может знать, когда вызов метода `stop()` безопасен, а когда он может привести к повреждению объектов. Поэтому данный метод был объявлен не

рекомендованным к применению. Когда требуется остановить поток исполнения, его нужно прервать. Прерванный поток может затем остановиться сам, когда это можно будет сделать безопасно.



НА ЗАМЕТКУ! Некоторые авторы требовали объявления метода `stop()` нежелательным, поскольку он мог приводить к появлению объектов, заблокированных остановленным потоком исполнения навсегда. Но подобное требование некорректно. Остановленный поток исполнения выходит из всех синхронизированных методов, которые он вызывал, генерируя исключение типа `ThreadDeath`. Как следствие, поток исполнения освобождает все встроенные блокировки объектов, которые он удерживает.

А теперь рассмотрим, что же не так делает метод `suspend()`. В отличие от метода `stop()`, метод `suspend()` не повреждает объекты. Но если приостановить поток исполнения, владеющий блокировкой, то эта блокировка останется недоступной до тех пор, пока поток не будет возобновлен. Если поток исполнения, вызвавший метод `suspend()`, попытается захватить ту же самую блокировку, программа перейдет в состояние взаимной блокировки: приостановленный поток ожидает, когда его возобновят, а поток, который приостановил его, ожидает снятия блокировки.

Подобная ситуация часто возникает в GUI. Допустим, имеется графический имитатор банка. Экранная кнопка `Pause` приостанавливает потоки денежных переводов, а кнопка `Resume` возобновляет их, как показано ниже.

```
pauseButton.addActionListener(event -> {
    for (int i = 0; i < threads.length; i++)
        threads[i].suspend(); // Не делайте этого!
});
resumeButton.addActionListener(event -> {
    for (int i = 0; i < threads.length; i++)
        threads[i].resume();
});
```

Допустим также, что метод `paintComponent()` рисует график каждого счета, вызывая метод `getBalances()` для получения массива остатков на счетах. Как поясняется в разделе 12.7.3, действия обеих экранных кнопок и перерисовка происходят в одном и том же потоке исполнения, называемом *потоком диспетчеризации событий*. Рассмотрим следующий сценарий.

1. Один из потоков исполнения, производящих денежные переводы, захватывает блокировку объекта `bank`.
2. Пользователь щелкает на кнопке `Pause`.
3. Все потоки исполнения денежных переводов приостанавливаются, но один из них продолжает удерживать блокировку для объекта `bank`.
4. По той же причине график счета должен быть перерисован.
5. Метод `paintComponent()` вызывает метод `getBalance()`.
6. Этот метод пытается захватить блокировку объекта `bank`.

В итоге программа зависает. Поток диспетчеризации событий не может продолжить свое выполнение, поскольку блокировкой владеет один из приостановленных потоков исполнения. Поэтому пользователь не может активизировать кнопку `Resume`, чтобы возобновить исполнение потоков.

Если требуется безопасно приостановить поток исполнения, следует ввести переменную `suspendRequested` и проверить ее в безопасном месте метода `run()`, т.е. там, где данный поток не блокирует объекты, необходимые другим потокам. Когда в данном потоке исполнения обнаружится, что переменная `suspendRequested` установлена, ему придется подождать до тех пор, пока она станет вновь доступной.

12.5. Потокобезопасные коллекции

Если во многих потоках исполнения одновременно вносятся изменения в структуру данных вроде хеш-таблицы, такую структуру данных очень легко повредить. (Подробнее о хеш-таблицах см. в главе 9.) Например, в одном потоке исполнения может быть начат ввод нового элемента. Допустим, что этот поток приостанавливается в процессе переназначения ссылок между группами хеш-таблицы. Если в другом потоке начнется проход по тому же самому списку, он может последовать по неправильным ссылкам, внося полный беспорядок, а возможно, сгенерировав попутно исключение или войдя в бесконечный цикл.

Общую структуру данных, разделяемую среди потоков исполнения, можно защитить, установив блокировку, но обычно проще выбрать потокобезопасную реализацию такой структуры данных. В последующих разделах будут рассмотрены другие потокобезопасные коллекции, предоставляемые в библиотеке `Java`.

12.5.1. Блокирующие очереди

Многие затруднения, связанные с потоками исполнения, можно изящно и безопасно сформулировать, применив одну или несколько очередей. В частности, поставляющий поток исполнения вводит элементы в очередь, а потребляющие потоки извлекают их. Таким образом, очередь позволяет безопасно передавать данные из одного потока в другой. Обратимся снова к примеру программы банковских переводов. Вместо того чтобы обращаться к объекту банка напрямую, потоки исполнения, производящие денежные переводы, вводят в очередь объекты команд на денежный перевод. А другой поток исполнения удаляет эти объекты из очереди и сам выполняет денежные переводы. И только этот поток имеет доступ к внутреннему содержимому объекта банка. В итоге никакой синхронизации не требуется. (Конечно, разработчики потокобезопасных классов очередей должны позаботиться о блокировках и условиях, но это их забота, а не ваша как пользователя таких классов.)

Блокирующая очередь вынуждает поток исполнения блокироваться при попытке ввести элемент в переполненную очередь или удалить элемент из пустой очереди. Блокирующие очереди — удобное инструментальное средство для координации работы многих потоков исполнения. Одни рабочие потоки могут периодически размещать промежуточные результаты в блокирующей очереди, а другие — удалять промежуточные результаты и видоизменять их далее. Методы для организации блокирующих очередей перечислены в табл. 12.1.

Методы блокирующих очередей разделяются на три категории, в зависимости от выполняемых действий, когда очередь заполнена или пуста. Для применения блокирующей очереди в качестве инструментального средства управления потоками понадобятся методы `put()` и `take()`. Методы `add()`, `remove()` и `element()` генерируют исключение при попытке ввести элемент в заполненную очередь или получить элемент из головы пустой очереди. Разумеется, в многопоточной программе очередь может заполниться или опустеть в любой момент, поэтому вместо этих методов, возможно,

потребуется методы `offer()`, `poll()` и `peek()`. Эти методы просто возвращают признак сбоя вместо исключения, если они не могут выполнить свои функции.



НА ЗАМЕТКУ! Методы `poll()` и `peek()` возвращают пустое значение `null` для обозначения неудачного исхода. Поэтому пустые значения `null` недопустимо вводить в блокирующие очереди.

Таблица 12.1. Методы блокирующих очередей

Метод	Обычное действие	Действие в особых случаях
<code>add()</code>	Вводит элемент в очередь	Генерирует исключение типа <code>IllegalStateException</code> , если очередь заполнена
<code>element()</code>	Возвращает элемент, находящийся в голове очереди	Генерирует исключение типа <code>NoSuchElementException</code> , если очередь пуста
<code>offer()</code>	Вводит элемент в очередь и возвращает логическое значение <code>true</code>	Возвращает логическое значение <code>false</code> , если очередь заполнена
<code>peek()</code>	Возвращает элемент, находящийся в голове очереди	Возвращает пустое значение <code>null</code> , если очередь пуста
<code>poll()</code>	Возвращает элемент, находящийся в голове очереди, удаляя его из очереди	Возвращает пустое значение <code>null</code> , если очередь пуста
<code>put()</code>	Вводит элемент в очередь	Блокирует, если очередь пуста
<code>remove()</code>	Возвращает элемент, находящийся в голове очереди, удаляя его из очереди	Генерирует исключение типа <code>NoSuchElementException</code> , если очередь пуста
<code>take()</code>	Возвращает элемент, находящийся в голове очереди, удаляя его из очереди	Блокирует, если очередь пуста

Существуют варианты методов `offer()` и `poll()` с указанием времени ожидания. Например, при приведенном ниже вызове в течение 100 миллисекунд предпринимается попытка ввести элемент в хвост очереди. Если это удастся сделать, то возвращается логическое значение `true`, в противном случае — логическое значение `false` по истечении времени ожидания.

```
boolean success = q.offer(x, 100, TimeUnit.MILLISECONDS);
```

Аналогично при приведенном ниже вызове в течение 100 миллисекунд предпринимается попытка удалить элемент из головы очереди. Если это удастся сделать, то возвращается элемент из головы очереди, а иначе — пустое значение `null` по истечении времени ожидания.

```
Object head = q.poll(100, TimeUnit.MILLISECONDS)
```

Метод `put()` блокирует, когда очередь заполнена, а метод `take()` блокирует, когда очередь пуста. Существуют также эквиваленты методов `offer()` и `put()` без указания времени ожидания.

В пакете `java.util.concurrent` предоставляется несколько вариантов блокирующих очередей. По умолчанию у очереди типа `LinkedBlockingQueue` отсутствует верхняя граница емкости, но такая граница емкости может быть указана. `LinkedBlockingDeque` — это вариант блокирующей двухсторонней очереди. А блокирующая очередь типа `ArrayBlockingQueue` конструируется с заданной емкостью

и признаком равноправия блокировки в качестве необязательного параметра. Если этот параметр указан, то предпочтение отдается очередям, дольше всего находящимся в состоянии ожидания. Как всегда, от этого существенно страдает производительность. Потому правило равноправия блокировки следует применять только в том случае, если оно действительно разрешает затруднения, возникающие при многопоточной обработке.

`PriorityBlockingQueue` — это блокирующая очередь по приоритету, а не просто действующая по принципу “первым пришел — первым обслужен”. Элементы удаляются из такой очереди по их приоритетам. Эта очередь имеет неограниченную емкость, но при попытке извлечь элемент из пустой очереди происходит блокирование. (Подробнее об очередях по приоритету см. в главе 9.)

И, наконец, блокирующая очередь типа `DelayQueue` содержит объекты, реализующие интерфейс `Delayed`, объявляемый следующим образом:

```
interface Delayed extends Comparable<Delayed>
{
    long getDelay(TimeUnit unit);
}
```

Метод `getDelay()` возвращает оставшееся время задержки объекта. Отрицательное значение указывает, что время задержки истекло. Элементы могут быть удалены из очереди типа `DelayQueue` только в том случае, если истечет время их задержки. Кроме того, придется реализовать метод `compareTo()`, который используется в очереди типа `DelayQueue` для сортировки ее элементов.

В версии Java 7 внедрен интерфейс `TransferQueue`, дающий поставляющему потоку исполнения возможность ожидать до тех пор, пока потребляющий поток не будет готов принять элемент из очереди. Когда в поставляющем потоке делается приведенный ниже вызов, блокировка устанавливается до тех пор, пока не будет снята в другом потоке. Данный интерфейс реализуется в классе `LinkedTransferQueue`.

```
q.transfer(item);
```

В примере программы, приведенном в листинге 12.6, демонстрируется применение блокирующей очереди для управления многими потоками исполнения. Эта программа осуществляет поиск среди всех файлов в каталоге и его подкаталогах, выводя строки кода, содержащие заданное ключевое слово.

В поставляющем потоке исполнения перечисляются все файлы во всех подкаталогах, а затем они размещаются в блокирующей очереди. Эта операция выполняется быстро, поэтому очередь быстро заполняется всеми файлами из файловой системы, если не установлена верхняя граница ее емкости.

Кроме того, запускается огромное количество поисковых потоков исполнения. В каждом таком потоке файл извлекается из очереди, открывается, а затем из него выводятся все строки, содержащие ключевое слово, после чего из очереди извлекается следующий файл. Чтобы прекратить выполнение данной программы, когда никакой другой обработки файлов больше не требуется, применяется специальный прием: из перечисляющего потока в очередь вводится фиктивный объект, чтобы уведомить о завершении потока. (Это похоже на муляж чемодана с надписью “последний чемодан” на ленточном транспортере в зале выдачи багажа.) Когда поисковый поток получает такой объект, он возвращает его обратно и завершается.

Следует также заметить, что в данном примере программы не требуется никакой явной синхронизации потоков исполнения. А в качестве синхронизирующего механизма используется сама структура очереди.

Листинг 12.6. Исходный код из файла `blockingQueue/BlockingQueueTest.java`

```
1  package blockingQueue;
2
3  import java.io.*;
4  import java.nio.charset.*;
5  import java.nio.file.*;
6  import java.util.*;
7  import java.util.concurrent.*;
8  import java.util.stream.*;
9
10 /**
11  * @version 1.03 2018-03-17
12  * @author Cay Horstmann
13  */
14 public class BlockingQueueTest
15 {
16     private static final int FILE_QUEUE_SIZE = 10;
17     private static final int SEARCH_THREADS = 100;
18     private static final Path DUMMY = Path.of("");
19     private static BlockingQueue<Path> queue =
20         new ArrayBlockingQueue<>(FILE_QUEUE_SIZE);
21
22     public static void main(String[] args)
23     {
24         try (var in = new Scanner(System.in))
25         {
26             System.out.print("Enter base directory "
27                 + "(e.g. /opt/jdk-9-src): ");
28             String directory = in.nextLine();
29             System.out.print("Enter keyword "
30                 + "(e.g. volatile): ");
31             String keyword = in.nextLine();
32
33             Runnable enumerator = () -> {
34                 try
35                 {
36                     enumerate(Path.of(directory));
37                     queue.put(DUMMY);
38                 }
39                 catch (IOException e)
40                 {
41                     e.printStackTrace();
42                 }
43                 catch (InterruptedException e)
44                 {
45                 }
46             };
47
48             new Thread(enumerator).start();
49             for (int i = 1; i <= SEARCH_THREADS; i++) {
50                 Runnable searcher = () -> {
51                     try
52                     {
53                         var done = false;
54                         while (!done)
55                         {
```

[illegible]

```

113     int lineNumber = 0;
114     while (in.hasNextLine())
115     {
116         lineNumber++;
117         String line = in.nextLine();
118         if (line.contains(keyword))
119             System.out.printf("%s:%d:%s\n", file,
120                               lineNumber, line);
121     }
122 }
123 }
124 }
125 }

```

java.util.concurrent.ArrayBlockingQueue<E> 5.0

- **ArrayBlockingQueue(int capacity)**
- **ArrayBlockingQueue(int capacity, boolean fair)**

Конструируют блокирующую очередь заданной емкости с установленным правилом равноправия блокировки, реализованную в виде циклического массива.

java.util.concurrent.LinkedBlockingQueue<E> 5

java.util.concurrent.LinkedBlockingDeque<E> 6

- **LinkedBlockingQueue()**
- **LinkedBlockingDeque()**

Конструируют неограниченную блокирующую одностороннюю или двустороннюю очередь, реализованную в виде связанного списка.

- **LinkedBlockingQueue(int capacity)**
- **LinkedBlockingDeque(int capacity)**

Конструируют ограниченную блокирующую одностороннюю или двустороннюю очередь, реализованную в виде связанного списка.

java.util.concurrent.DelayQueue<E extends Delayed> 5

- **DelayQueue()**

Конструирует неограниченную блокирующую очередь элементов типа **Delayed**. Из очереди могут быть удалены только элементы, время задержки которых истекло.

java.util.concurrent.Delayed 5

- **long getDelay(TimeUnit unit)**

Получает задержку для данного объекта, измеряемую в заданных единицах времени.

java.util.concurrent.PriorityBlockingQueue<E> 5

- **PriorityBlockingQueue()**
- **PriorityBlockingQueue(int initialCapacity)**
- **PriorityBlockingQueue(int initialCapacity, Comparator<? super E> comparator)**

Конструируют неограниченную блокирующую очередь по приоритетам, реализованную в виде "кучи". По умолчанию параметр **initialCapacity** принимает значение 11. Если же компаратор не указан, элементы очереди должны реализовывать интерфейс **Comparable**.

java.util.concurrent.BlockingQueue<E> 5

- **void put(E element)**
Вводит элемент в очередь, устанавливая, если требуется, блокировку.
- **E take()**
Удаляет элемент из головы очереди и возвращает его, устанавливая, если требуется, блокировку.
- **boolean offer(E element, long time, TimeUnit unit)**
Вводит заданный элемент в очередь и возвращает логическое значение **true** при удачном исходе, устанавливая, если требуется, блокировку на время ввода элемента в очередь или до истечения времени ожидания.
- **E poll(long time, TimeUnit unit)**
Удаляет элемент из головы очереди и возвращает его, устанавливая, если требуется, блокировку до тех пор, пока элемент доступен, или же до тех пор, пока не истечет время ожидания. При неудачном исходе возвращает пустое значение **null**.

java.util.concurrent.BlockingDeque<E> 6

- **void putFirst(E element)**
- **void putLast(E element)**
Вводят элемент в очередь, устанавливая, если требуется, блокировку.
- **E takeFirst()**
- **E takeLast()**
Удаляют элемент из головы очереди и возвращают его, устанавливая, если требуется, блокировку.
- **boolean offerFirst(E element, long time, TimeUnit unit)**
- **boolean offerLast(E element, long time, TimeUnit unit)**
Вводят заданный элемент в очередь и возвращают логическое значение **true** при удачном исходе, устанавливая, если требуется, блокировку на время ввода элемента или до истечения времени ожидания.
- **E pollFirst(long time, TimeUnit unit)**
- **E pollLast(long time, TimeUnit unit)**
Удаляют и возвращают элемент из головы очереди, устанавливая, если требуется, блокировку до тех пор, пока элемент доступен, или же до тех пор, пока не истечет время ожидания. При неудачном исходе возвращают пустое значение **null**.

```
java.util.concurrent.TransferQueue<E> 7
```

- `void transfer(E element)`
- `boolean tryTransfer(E element, long time, TimeUnit unit)`

Передают значение или пытаются передать его в течение заданного времени ожидания, устанавливая, если требуется, блокировку до тех пор, пока элемент не будет удален из очереди в другом потоке исполнения. Второй метод возвращает логическое значение `true` при удачном исходе.

12.5.2. Эффективные отображения, множества и очереди

В пакете `java.util.concurrent` предоставляются следующие эффективные реализации отображений, отсортированных множеств и очередей: `ConcurrentHashMap`, `ConcurrentSkipListMap`, `ConcurrentSkipListSet` и `ConcurrentLinkedQueue`. В этих коллекциях используются изоцированные алгоритмы, сводящие к минимуму вероятность состязаний, обеспечивая параллельный доступ к разным частям структуры данных.

В отличие от большинства коллекций, метод `size()` совсем не обязательно выполняется в течение постоянного промежутка времени. Для определения размера одной из перечисленных выше коллекций обычно требуется обратиться ко всем ее элементам.



НА ЗАМЕТКУ! В некоторых приложениях применяются настолько крупные параллельные хеш-отображения, что обращаться к ним с помощью метода `size()` неэффективно, поскольку он возвращает размер коллекции в виде значения типа `int`. Как же тогда обращаться к отображению, в котором хранится больше двух миллионов записей? Для этого имеется метод `mappingCount()`, возвращающий размер коллекции в виде значения типа `long`.

Коллекции возвращают *слабо совместные* итераторы. Это означает, что итератор может отражать все изменения, внесенные в коллекцию после его создания, а может и не отражать их. Но такие итераторы никогда не возвращают одно и то же значение дважды и не генерируют исключение типа `ConcurrentModificationException`.



НА ЗАМЕТКУ! Напротив, итератор коллекции из пакета `java.util` генерирует исключение типа `ConcurrentModificationException`, если в коллекцию внесены изменения после создания этого итератора.

Хеш-отображение параллельного действия способно эффективно поддерживать большое количество читающих потоков и фиксированное количество записывающих потоков. По умолчанию допускается до 16 *параллельно* действующих записывающих потоков. Таких потоков исполнения может быть намного больше, но если запись одновременно выполняется в более чем 16 потоках, то остальные временно блокируются. В конструкторе такой коллекции можно, конечно, указать большее количество потоков исполнения, но вряд ли это вообще понадобится.



НА ЗАМЕТКУ! Все записи с одним и тем же хеш-кодом хранятся в одной и той же "группе" хеш-отображения. В некоторых приложениях применяются неудачные хеш-функции, и в результате все записи оказываются в небольшом количестве групп, что значительно снижает производительность. И даже применение таких, в общем, подходящих хеш-функций может оказаться проблематичным.

Например, взломщик может замедлить выполнение программы, состряпав огромное количество символьных строк с одинаковым хеш-значением. Но в последних версиях Java группы в параллельном хеш-отображении организуются в виде деревьев, а не списков, когда тип ключа реализует интерфейс `Comparable`, гарантирующий производительность порядка $O(\log n)$.

`java.util.concurrent.ConcurrentLinkedQueue<E> 5`

- `ConcurrentLinkedQueue<E>()`

Конструирует неограниченную и неблокирующую очередь, безопасно доступную из многих потоков исполнения.

`java.util.concurrent.ConcurrentSkipListSet<E> 6`

- `ConcurrentSkipListSet<E>()`
- `ConcurrentSkipListSet<E>(Comparator<? super E> comp)`

Конструируют отсортированное множество, безопасно доступное из многих потоков исполнения. Первый конструктор требует, чтобы элементы множества относились к классу, реализующему интерфейс `Comparable`.

`java.util.concurrent.ConcurrentHashMap<K, V> 5.0`

`java.util.concurrent.ConcurrentSkipListMap<K, V> 6`

- `ConcurrentHashMap<K, V>()`
- `ConcurrentHashMap<K, V>(int initialCapacity)`
- `ConcurrentHashMap<K, V>(int initialCapacity, float loadFactor, int concurrencyLevel)`

Конструируют отсортированное хеш-отображение, безопасно доступное из многих потоков. По умолчанию параметр `initialCapacity` принимает значение 16. Если же средняя нагрузка на группу превышает коэффициент загрузки, определяемый параметром `loadFactor`, то размер данной коллекции изменяется. По умолчанию параметр `loadFactor` принимает значение 0.75. А параметр `concurrencyLevel` определяет предполагаемое количество параллельных записывающих потоков.

- `ConcurrentSkipListMap<K, V>()`
- `ConcurrentSkipListSet<K, V>(Comparator<? super K> comp)`

Конструируют отсортированное хеш-отображение, безопасно доступное из многих потоков исполнения. Первый конструктор требует, чтобы элементы множества относились к классу, реализующему интерфейс `Comparable`.

12.5.3. Атомарное обновление записей в отображениях

В первоначальной версии класса `ConcurrentHashMap` имелось лишь несколько методов для атомарного обновления хеш-отображений, что затрудняло в какой-то

степени программирование. Допустим, требуется подсчитать, насколько часто наблюдаются определенные свойства. В качестве простого примера допустим, что в нескольких потоках исполнения встречаются слова, частоту появления которых требуется подсчитать.

Можно ли в таком случае воспользоваться хеш-отображением типа `ConcurrentHashMap<String, Long>`? Рассмотрим пример инкрементирования счета. Очевидно, что приведенный ниже фрагмент кода не является потокобезопасным, поскольку в следующем потоке исполнения может быть одновременно обновлен тот же самый счет.

```
Long oldValue = map.get(word);
Long newValue = oldValue == null ? 1 : oldValue + 1;
map.put(word, newValue); // ОШИБКА - заменить значение
                          // переменной oldValue, возможно, не удастся
```



НА ЗАМЕТКУ! Некоторых программистов удивляет, что в потокобезопасной, предположительно, структуре данных разрешаются операции, не являющиеся потокобезопасными. Но этому имеются два совершенно противоположных объяснения. Если простое хеш-отображение типа `HashMap` модифицируется в нескольких потоках исполнения, они могут нарушить его внутреннюю структуру (т.е. массив связанных списков). В итоге некоторые связи могут быть пропущены и даже зациклены, приведя структуру данных в полную негодность. Ничего подобного не может произойти с хеш-отображением типа `ConcurrentHashMap`. В приведенном выше примере кода вызовы методов `get()` и `put()` вообще не нарушают структуру данных. Но поскольку последовательность выполняемых операций не является атомарной, то ее результат непредсказуем.

В прежних версиях Java приходилось вызывать метод `replace()`, где прежнее значение атомарно заменяется новым значением, при условии, что прежнее значение не было раньше заменено на нечто иное ни в одном из других потоков исполнения. Эту операцию приходилось продолжать до тех пор, пока метод `replace()` не завершится успешно, как показано ниже.

```
do
{
    oldValue = map.get(word);
    newValue = oldValue == null ? 1 : oldValue + 1;
} while (!map.replace(word, oldValue, newValue));
```

В качестве альтернативы можно было воспользоваться хеш-отображением типа `ConcurrentHashMap<String, AtomicLong>` или же хеш-отображением типа `ConcurrentHashMap<String, LongAdder>`, как показано ниже. Но, к сожалению, новый объект типа `AtomicLong` создается для каждой операции инкремента независимо от того, требуется это или нет.

```
map.putIfAbsent(word, new LongAdder());
map.get(word).increment();
```

В прикладном интерфейсе Java API предоставляются методы, делающие атомарные обновления более удобными. В частности, метод `compute()` вызывается с ключом и функцией для вычисления нового значения. Эта функция получает ключ и связанное с ним значение, а если таковое отсутствует, то пустое значение `null`, а затем она вычисляет новое значение. В качестве примера ниже показано, как обновить отображение целочисленных счетчиков.

```
map.compute(word, (k, v) -> v == null ? 1 : v + 1);
```



НА ЗАМЕТКУ! В хеш-отображении типа `ConcurrentHashMap` пустые значения `null` не допускаются. Имеется немало методов, в которых пустое значение `null` служит для указания на то, что заданный ключ отсутствует в отображении.

Имеются также варианты методов `computeIfPresent()` и `computeIfAbsent()`, в которых новое значение вычисляется только в том случае, если прежнее значение уже имеется или еще отсутствует соответственно. Так, отображение счетчиков типа `LongAdder` может быть обновлено следующим образом:

```
map.computeIfAbsent(word, k -> new LongAdder()).increment();
```

Это почти равнозначно рассмотренному ранее вызову метода `putIfAbsent()`. Но конструктор класса `LongAdder` может быть вызван только в том случае, если действительно требуется новый счетчик.

Когда ключ вводится в отображение в первый раз, нередко требуется сделать нечто особенное, и для этой цели очень удобен метод `merge()`. В качестве одного параметра ему передается исходное значение, которое используется, когда ключ еще отсутствует. В противном случае вызывается функция, предоставляемая этому методу в качестве другого параметра. Эта функция объединяет существующее значение с исходным. (Но в отличие от метода `compute()`, эта функция не обрабатывает ключ.) Метод `merge()` можно вызвать следующим образом:

```
map.merge(word, 1L, (existingValue, newValue) ->
    existingValue + newValue);
```

или же так:

```
map.merge(word, 1L, Long::sum);
```

Как говорится, проще и короче не бывает.



НА ЗАМЕТКУ! Если функция, которая передается методу `merge()` для вычисления или объединения значений, возвращает пустое значение `null`, существующая запись удаляется из отображения.



ВНИМАНИЕ! Применяя метод `compute()` или `merge()`, следует иметь в виду, что функция, которой он снабжается, не должна выполнять много операций. Ведь в процессе выполнения этой функции может быть заблокирован ряд других обновлений отображения. Безусловно, эта функция также не должна обновлять другие части отображения.

В примере программы из листинга 12.7 параллельное хеш-отображение применяется для подсчета всех слов в исходных файлах Java, находящихся в дереве каталогов.

Листинг 12.7. Исходный код из файла `concurrentHashMap/CHMDemo.java`

```
1 package concurrentHashMap;
2
3 import java.io.*;
4 import java.nio.file.*;
5 import java.util.*;
6 import java.util.concurrent.*;
7 import java.util.stream.*;
8
```

```
9  /**
10  * В этой программе демонстрируется применение
11  * параллельных хеш-отображений
12  * @version 1.0 2018-01-04
13  * @author Cay Horstmann
14  */
15  public class CHMDemo
16  {
17      public static ConcurrentHashMap<String, Long> map =
18          new ConcurrentHashMap<>();
19
20      /**
21       * Вводит все слова из заданного файла в
22       * параллельное хеш-отображение
23       * @param file Заданный файл
24       */
25      public static void process(Path file)
26      {
27          try (var in = new Scanner(file))
28          {
29              while (in.hasNext())
30              {
31                  String word = in.next();
32                  map.merge(word, 1L, Long::sum);
33              }
34          }
35          catch (IOException e)
36          {
37              e.printStackTrace();
38          }
39      }
40
41      /**
42       * Возвращает все каталоги, порожденные заданным
43       * каталогом – см. главы 1 и 2 второго тома
44       * настоящего издания
45       * @param rootDir Корневой каталог
46       * @return Возвращает все каталоги, порожденные
47       *         корневым каталогом
48       */
49      public static Set<Path> descendants(Path rootDir)
50          throws IOException
51      {
52          try (Stream<Path> entries = Files.walk(rootDir))
53          {
54              return entries.collect(Collectors.toSet());
55          }
56      }
57
58      public static void main(String[] args)
59          throws InterruptedException,
60              ExecutionException, IOException
61      {
62          int processors =
63              Runtime.getRuntime().availableProcessors();
64          ExecutorService executor =
```

```
65         Executors.newFixedThreadPool(processors);
66     Path pathToRoot = Path.of(".");
67     for (Path p : descendants(pathToRoot))
68     {
69         if (p.getFileName().toString().endsWith(".java"))
70             executor.execute(() -> process(p));
71     }
72     executor.shutdown();
73     executor.awaitTermination(10, TimeUnit.MINUTES);
74     map.forEach((k, v) ->
75     {
76         if (v >= 10)
77             System.out.println(k + " occurs " + v
78                               + " times");
79     });
80 }
81 }
```

12.5.4. Групповые операции над параллельными хеш-отображениями

В прикладном интерфейсе Java API предоставляются групповые операции, которые можно безопасно выполнять над параллельными хеш-отображениями даже в том случае, если в других потоках исполнения производятся определенные действия с этим отображением. Групповые операции осуществляют обход всего отображения, оперируя по ходу дела обнаруживаемыми в нем элементами. И для этого не требуется специально фиксировать моментальный снимок отображения во времени. Результат групповой операции следует интерпретировать как некоторую аппроксимацию, приближенно отражающую состояние отображения, кроме тех случаев, когда становится известно, что отображение не модифицируется при выполнении групповой операции.

Имеются следующие разновидности групповых операций.

- **Операция `search`.** Применяет функцию к каждой паре “ключ–значение” до тех пор, пока не будет получен непустой результат. После этого поиск прекращается и возвращается результат выполнения функции.
- **Операция `reduce`.** Объединяет все пары “ключ–значение” с помощью предоставляемой функции накопления.
- **Операция `forEach`.** Применяет функцию ко всем ключам и/или значениям.

У каждой групповой операции имеются следующие варианты.

- **Операция `Keys`.** Оперирует ключами.
- **Операция `Values`.** Оперирует значениями.
- **Операция.** Оперирует ключами и значениями.
- **Операция `Entries`.** Оперирует объектами типа `Map.Entry`.

В каждой из этих операций необходимо указать *порог параллелизма*. Если отображение содержит больше элементов, чем заданный порог, групповая операция распараллеливается. Если же требуется выполнить групповую операцию в одном потоке, то следует указать порог `Long.MAX_VALUE`. А если для групповой операции требуется максимальное количество потоков исполнения, то следует указать порог 1.

Рассмотрим сначала методы, выполняющие групповую операцию `search`. Ниже приведены варианты этих методов.

```
U searchKeys(long threshold, BiFunction<? super K,
           ? extends U> f)
U searchValues(long threshold, BiFunction<? super V,
           ? extends U> f)
U search(long threshold, BiFunction<? super K, ? super V,
           ? extends U> f)
U searchEntries(long threshold, BiFunction<Map.Entry<K, V>,
           ? extends U> f)
```

Допустим, требуется найти первое слово, встречающееся больше 1000 раз. Для этого необходимо обнаружить сначала ключи и значения:

```
String result = map.search(threshold, (k, v) -> v > 1000
           ? k : null);
```

Таким образом, в переменной `result` устанавливается результат первого совпадения или пустое значение `null`, если функция поиска возвращает пустое значение `null` для всех входных данных.

У методов, выполняющих групповую операцию `forEach`, имеются два варианта. В первом варианте функция *потребителя* просто применяется к каждой записи отображения, например, следующим образом:

```
map.forEach(threshold, (k, v) ->
           System.out.println(k + " -> " + v));
```

Во втором варианте принимается дополнительная функция *преобразователя*, которая применяется в первую очередь, а ее результат передается потребителю, как показано ниже.

```
map.forEach(threshold, (k, v) ->
           k + " -> " + v, // Преобразователь
           System.out::println); // Потребитель
```

Преобразователь может быть использован в качестве фильтра. Всякий раз, когда преобразователь возвращает пустое значение `null`, это значение негласно пропускается. Так, в следующем фрагменте кода из отображения выводятся только те записи, в которых хранятся крупные значения:

```
map.forEach(threshold, (k, v) ->
           v > 1000 ? k + " -> " + v : null,
           // Фильтр и преобразователь
           System.out::println); // Пустые значения
           // не передаются потребителю
```

В операциях `reduce` их входные данные объединяются с помощью функции накопления. Например, в следующей строке кода может быть вычислена сумма всех значений:

```
Long sum = map.reduceValues(threshold, Long::sum);
```

Как и операцию `forEach`, в данном случае операцию `reduce` можно также снабдить функцией преобразователя. Так, в приведенном ниже фрагменте кода вычисляется длина наибольшего ключа.

```
Integer maxLength = map.reduceKeys(threshold,
           String::length, // Преобразователь
           Integer::max); // Накопитель
```

Преобразователь может действовать как фильтр, возвращая пустое значение `null` для исключения нежелательных входных данных. В следующем фрагменте кода подсчитывается количество записей, имеющих значение больше 100:

```
Long count = map.reduceValues(threshold,
    v -> v > 1000 ? 1L : null,
    Long::sum);
```



НА ЗАМЕТКУ! Если отображение оказывается пустым или же все его записи отсеяны, операция **reduce** возвратит пустое значение `null`. Если же в отображении имеется только один элемент, то возвращается результат его преобразования, а накопитель не применяется.

Для вывода результатов типа `int`, `long` и `double` имеются специальные варианты групповых операций, обозначаемые суффиксами **ToInt**, **ToLong** и **ToDouble** соответственно. Для их выполнения необходимо преобразовать входные данные в значение соответствующего примитивного типа, а также указать значение по умолчанию и функцию накопителя, как показано ниже. Значение по умолчанию возвращается в том случае, если отображение оказывается пустым.

```
long sum = map.reduceValuesToLong(threshold,
    Long::longValue, // Преобразователь в примитивный тип
    0, // Значение по умолчанию для пустого отображения
    Long::sum); // Накопитель значений примитивного типа
```



ВНИМАНИЕ! Упомянутые выше специальные варианты групповых операций над значениями примитивных типов действуют иначе, чем их аналоги для объектов, где во внимание принимается только один элемент. Вместо возврата преобразованного элемента в последнем случае накапливается элемент по умолчанию. Следовательно, элемент по умолчанию должен быть нейтральным элементом накопителя.

12.5.5. Параллельные представления множеств

Допустим, что вместо отображения требуется потокбезопасное множество. Для этой цели отсутствует соответствующий класс `ConcurrentHashSet`, поэтому можно попытаться создать собственный класс. В качестве выхода из этого положения можно было бы, конечно, воспользоваться классом `ConcurrentHashMap` с поддельными значениями, но тогда в конечном итоге получилось бы отображение, а не множество, к которому нельзя применять операции, определяемые в интерфейсе `Set`.

Статический метод `newKeySet()` выдает множество типа `Set<K>`, которое фактически служит оболочкой, в которую заключается параллельное хеш-отображение типа `ConcurrentHashMap<K, Boolean>`. (Все значения в таком отображении равны `Boolean.TRUE`, что, в общем, неважно, поскольку это отображение используется лишь как множество.)

```
Set<String> words = ConcurrentHashMap.<String>newKeySet();
```

Безусловно, если уже имеется отображение, то метод `keySet()` выдает изменяемое множество ключей. Если удалить элементы из такого множества, ключи (и их значения) удаляются из множества. Но вряд ли имеет смысл вводить элементы во множество ключей, поскольку для них отсутствуют соответствующие значения. В классе `ConcurrentHashMap` имеется еще один метод `keySet()` со значением по умолчанию, которое используется при вводе элементов во множество, как показано ниже. Если

символьная строка "Java" раньше отсутствовала во множестве words, то теперь она имеет единичное значение.

```
Set<String> words = map.keySet(1L);
words.add("Java");
```

12.5.6. Массивы, копируемые при записи

Классы CopyOnWriteArrayList и CopyOnWriteArraySet представляют потоко-безопасные коллекции, при любых изменениях в которых создается копия базового массива. Это удобно, если количество потоков, выполняющих обход коллекции, значительно превышает количество потоков, изменяющих ее. Когда конструируется итератор, он содержит ссылку на текущий массив. Если в дальнейшем массив изменяется, итератор по-прежнему ссылается на старую копию массива, а текущий массив коллекции заменяется новым. Как следствие, старому итератору доступно согласованное (хотя и потенциально устаревшее) представление коллекции, не требующее дополнительных издержек на синхронизацию.

12.5.7. Алгоритмы обработки параллельных массивов

В классе Arrays реализуется целый ряд распараллеливаемых операций. В частности, статический метод Arrays.parallelSort() может отсортировать массив примитивных значений или объектов, как показано в следующем примере кода:

```
var contents = new String(Files.readAllBytes(
    Paths.get("alice.txt"), StandardCharsets.UTF_8);
    // прочитать данные из файла в символьную строку
String[] words = contents.split("[\\P{L}]+");
    // разделить небуквенные символы
Arrays.parallelSort(words);
```

Для сортировки объектов можно предоставить компаратор типа Comparator следующим образом:

```
Arrays.parallelSort(words, Comparator.comparing(String::length));
```

При вызове всех методов из класса Arrays можно предоставить границы диапазона, как показано ниже.

```
values.parallelSort(values.length / 2, values.length);
    // отсортировать верхнюю половину диапазона
```



НА ЗАМЕТКУ На первый взгляд кажется несколько странным, что упомянутые выше методы имеют слово **parallel** в своих именах. Ведь пользователя вообще не интересует, каким образом выполняются операции установки и сортировки значений в массиве. Но разработчики прикладного интерфейса Java API стремились к тому, чтобы из наименования подобных операций пользователям было ясно, что они распараллеливаются. Подобным образом пользователи предупреждаются о недопустимости передачи функций с побочными эффектами.

Метод parallelSetAll() заполняет массив значениями, вычисляемыми соответствующей функцией, как показано ниже. Эта функция принимает в качестве параметра индекс элемента и вычисляет значение в данном месте массива.

```
Arrays.parallelSetAll(values, i -> i % 10);
    // заполнить массив values значениями
    // 0 1 2 3 4 5 6 7 8 9 0 1 2 . . .
```

Очевидно, что такая операция только выиграет от распараллеливания. Имеются разные ее версии как для массивов примитивных типов, так и для массивов объектов.

И, наконец, имеется метод `parallelPrefix()`, заменяющий каждый элемент массива накоплением префикса заданной ассоциативной операции. Чтобы стало понятнее назначение этого метода, обратимся к конкретному примеру, рассмотрев массив `[1, 2, 3, 4, . . .]` и операцию умножения. В результате вызова `Arrays.parallelPrefix(values, (x, y) -> x * y)` этот массив будет содержать следующее:

```
[1, 1
  × 2, 1
  × 2
  × 3, 1
  × 2
  × 3
  × 4, ...]
```

Как ни странно, подобное вычисление можно распараллелить. Сначала необходимо соединить смежные элементы:

```
[1, 1
  × 2, 3, 3
  × 4, 5, 5
  × 6, 7, 7
  × 8]
```

Значения, выделенные обычным шрифтом, не затрагиваются данной операцией. Очевидно, что ее можно выполнить параллельно на отдельных участках массива, а затем обновить выделенные выше полужирным элементы, перемножив их с элементами, находящимися на одну или две позиции раньше:

```
[1, 1 × 2
  , 1 × 2 × 3, 1 × 2 × 3 × 4
  , 5, 5 × 6, 5 × 6 × 7, 5 × 6 × 7 × 8]
```

И эту операцию можно выполнить параллельно. После $\log n$ шагов процесс будет завершен. Это более выгодный способ, чем простое линейное вычисление при наличии достаточного количества процессоров. Такой алгоритм нередко применяется на специальном оборудовании, а его пользователи проявляют немалую изобретательность, приспособлявая его к решению самых разных задач.

12.5.8. Устаревшие потокобезопасные коллекции

С самых первых версий Java классы `Vector` и `Hashtable` предоставляли потокобезопасные реализации динамического массива и хеш-таблицы. Теперь эти классы считаются устаревшими и заменены классами `ArrayList` и `HashMap`. Но эти последние классы не являются потокобезопасными, хотя в библиотеке коллекций предусмотрен другой механизм для обеспечения безопасности потоков исполнения. Любой класс может быть сделан потокобезопасным благодаря *синхронизирующей оболочке* следующим образом:

```
List<E> synchArrayList =
    Collections.synchronizedList(new ArrayList<E>());
Map<K, V> synchHashMap =
    Collections.synchronizedMap(new HashMap<K, V>());
```


Методы получаемой в итоге коллекции защищены блокировкой, обеспечивая потокобезопасный доступ. Но при этом необходимо гарантировать, что ни один поток исполнения не обращается к данным через исходные, не синхронизированные методы. Это проще всего сделать, не сохраняя никаких ссылок на исходный объект. Достаточно сконструировать коллекцию и сразу же передать ее оболочке, как показано в приведенных ранее примерах. Но если требуется сделать обход коллекции в то время, как в другом потоке имеется возможность изменить ее, то придется установить клиентскую блокировку, как показано ниже.

```
synchronized (synchHashMap)
{
    Iterator<K> iter = synchHashMap.keySet().iterator();
    while (iter.hasNext()) . . . ;
}
```

Такой же код следует использовать и при организации цикла в стиле `for each`, поскольку в этом цикле применяется итератор. Но при попытке изменить коллекцию из другого потока исполнения, когда осуществляется ее обход, итератор генерирует исключение типа `ConcurrentModificationException`, свидетельствующее о неудачном исходе обхода коллекции. Кроме того, требуется синхронизация, чтобы просто и надежно обнаружить попытки внести изменения в данные из параллельно действующих потоков исполнения.

Но, как правило, вместо синхронизирующих оболочек лучше пользоваться коллекциями, определенными в пакете `java.util.concurrent`. В частности, хеш-отображение типа `ConcurrentHashMap` тщательно реализовано с таким расчетом, чтобы к нему можно было обращаться из многих потоков исполнения, не блокирующих друг друга, если они работают с разными группами данных в хеш-таблице. Исключением из этого правила является часто обновляемый списочный массив. В этом случае вместо синхронизированного списочного массива типа `ArrayList` лучше выбрать списочный массив типа `CopyOnWriteArrayList`.

java.util.Collections 1.2

- `static <E> Collection<E> synchronizedCollection(Collection<E> c)`
- `static <E> List synchronizedList(List<E> c)`
- `static <E> Set synchronizedSet(Set<E> c)`
- `static <E> SortedSet synchronizedSortedSet(SortedSet<E> c)`
- `static <K, V> Map<K, V> synchronizedMap(Map<K, V> c)`
- `static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> c)`

Конструируют представление коллекции с синхронизированными методами.

12.6. Задачи и пулы потоков исполнения

Создание нового потока исполнения — довольно дорогостоящая операция с точки зрения потребляемых ресурсов, поскольку она включает в себя взаимодействие с операционной системой. Если в программе создается большое количество

кратковременных потоков исполнения, то имеет смысл использовать *пул потоков*. В пуле потоков содержится целый ряд простаивающих потоков, готовых к запуску. Так, объект типа `Runnable` размещается в пуле, а один из потоков вызывает его метод `run()`. Когда метод `run()` завершается, его поток не уничтожается, но остается в пуле готовым обслужить новый запрос. В последующих разделах будут представлены инструментальные средства, предоставляющие в каркасе параллелизма Java для согласованного выполнения задач.

12.6.1. Интерфейсы `Callable` и `Future`

Интерфейс `Runnable` инкапсулирует задачу, выполняющуюся асинхронно. Его можно рассматривать как асинхронный метод без параметров и возвращаемого значения. А интерфейс `Callable` подобен интерфейсу `Runnable`, но в нем предусмотрен возврат значения. Интерфейс `Callable` относится к параметризованному типу и имеет единственный метод `call()`:

```
public interface Callable<V>
{
    V call() throws Exception;
}
```

Параметр типа обозначает тип возвращаемого значения. Например, интерфейс `Callable<Integer>` представляет асинхронно выполняемую задачу, в результате чего возвращается объект типа `Integer`. А сохранение результатов асинхронного вычисления обеспечивает интерфейс `Future`. В частности, вычисление можно начать, предоставив кому-нибудь другому объект типа `Future`, а затем просто забыть о нем. Владелец объекта типа `Future` может получить результат, когда он будет готов. В интерфейсе `Future` объявляются следующие методы:

```
V get()
V get(long timeout, TimeUnit unit)
void cancel(boolean mayInterrupt)
boolean isCancelled()
boolean isDone()
```

При первом вызове метода `get()` блокировка устанавливается до тех пор, пока не завершится вычисление. А при втором вызове метода `get()` генерируется исключение типа `TimeoutException`, если время ожидания истекает до завершения вычислений. Если же прерывается поток, в котором выполняется вычисление, в обоих случаях генерируется исключение типа `InterruptedException`. А если вычисление уже завершено, то метод `get()` сразу же возвращает управление.

Метод `isDone()` возвращает логическое значение `false`, если вычисление продолжается, и логическое значение `true`, если оно завершено. Вычисление можно прервать, вызвав метод `cancel()`. Если вычисление еще не начато, оно отменяется и вообще не начнется. Если же вычисление уже выполняется, оно прерывается, когда параметр метода `mayInterrupt()` принимает логическое значение `true`.



ВНИМАНИЕ! Отмена задачи выполняется в два этапа. В частности, базовый поток исполнения должен быть сначала обнаружен, а затем прерван. Реализация конкретной задачи (в методе `call()`) должна отреагировать на прерывание и прекратить свою работу. Если объекту типа `Future` неизвестно, в каком именно потоке выполняется задача, а в самой задаче не осуществляется текущий контроль состояния прерванного потока исполнения, то отмена задачи не вызовет никакого действия.

Чтобы выполнить асинхронное вычисление типа `Callable`, можно, например, воспользоваться классом `FutureTask`, реализующим оба интерфейса `Future` и `Runnable`. Это дает возможность построить поток для последующего исполнения, как показано ниже.

```
Callable<Integer> task = . . . ;
var futuretask = new FutureTask<Integer>(task);
var t = new Thread(futuretask); // это объект типа Runnable
t.start();
. . .
Integer result = task.get(); // а это объект типа Future
```

Чаще всего объект типа `Callable` передается исполнителю. Более подробно об этом речь пойдет в следующем разделе.

```
java.util.concurrent.Callable<V> 5
```

- **V call()**
Запускает на выполнение задачи, выдающие результат.

```
java.util.concurrent.Future<V> 5
```

- **V get()**
- **V get(long time, TimeUnit unit)**
Получают результат, устанавливая блокировку до момента его готовности или истечения заданного промежутка времени. Второй метод генерирует исключение типа `TimeoutException` при неудачном исходе.
- **boolean cancel(boolean mayInterrupt)**
Пытается отменить выполнение задачи. Если задача уже запущена на выполнение и параметр `mayInterrupt` принимает логическое значение `true`, она прерывается. Возвращает логическое значение `true` при удачном исходе операции отмены.
- **boolean isCancelled()**
Возвращает логическое значение `true`, если задача была отменена до ее завершения.
- **boolean isDone()**
Возвращает логическое значение `true`, если задача завершена нормально, отменена или прервана вследствие исключения.

```
java.util.concurrent.FutureTask<V> 5.0
```

- **FutureTask(Callable<V> task)**
- **FutureTask(Runnable task, V result)**
Конструируют объект, реализующий одновременно интерфейсы `Future<V>` и `Runnable`.

12.6.2. Исполнители

В состав класса `Executors` входит целый ряд статических фабричных методов для построения пулов потоков. Их перечень и краткое описание приводятся в табл. 12.2.

Таблица 12.2. Фабричные методы из класса `Executors`

Метод	Описание
<code>newCachedThreadPool()</code>	Новые потоки исполнения создаются по мере необходимости, а простаивающие потоки сохраняются в течение 60 секунд
<code>newWorkStealingPool()</code>	Пул, пригодный для выполнения задач вилочного соединения (см. раздел 12.6.4), где сложные задачи разбиваются на более простые, которые "похищаются" простаивающими потоками исполнения
<code>newFixedThreadPool()</code>	Пул содержит фиксированный ряд потоков исполнения, а простаивающие потоки сохраняются неопределенно долго
<code>newSingleThreadExecutor()</code>	Пул с единственным потоком, исполняющим переданные ему задачи поочередно
<code>newScheduledThreadPool()</code>	Пул фиксированных потоков для планового запуска на исполнение
<code>newSingleThreadScheduledExecutor()</code>	Пул с единственным потоком для планового запуска на исполнение

Метод `newCachedThreadPool()` строит пул потоков, выполняющий каждую задачу немедленно, используя существующий простаивающий поток, если он доступен, а иначе — создавая новый поток. Метод `newFixedThreadPool()` строит пул потоков фиксированного размера. Если количество задач превышает количество простаивающих потоков, то лишние задачи ставятся в очередь на обслуживание, чтобы быть запущенными, когда завершатся текущие исполняемые задачи. Метод `newSingleThreadExecutor()` создает вырожденный пул размером в один поток. Задачи, передаваемые в единственный поток, исполняются по очереди. Все три упомянутых здесь метода возвращают объект класса `ThreadPoolExecutor`, реализующего интерфейс `ExecutorService`.

Передать задачу типа `Runnable` или `Callable` объекту типа `ExecutorService` можно одним из следующих вариантов метода `submit()`:

```
Future<T> submit(Callable<T> task)
Future<?> submit(Runnable task)
Future<T> submit(Runnable task, T result)
```

Пул запускает переданную ему задачу при первом удобном случае. В результате вызова первого варианта метода `submit()` возвращается объект типа `Future`, который можно использовать для получения результата или отмены задачи. Второй вариант метода `submit()` возвращает объект необычного типа `Future<?>`. Он служит для вызова метода `isDone()`, `cancel()` или `isCancelled()`. И третий вариант метода `submit()` возвращает объект типа `Future`, метод которого `get()`, в свою очередь, возвращает по завершении объект результата.

По завершении работы с пулом потоков исполнения следует вызвать метод `shutdown()`. Этот метод инициирует последовательность закрытия пула, после чего новые задачи не принимаются на выполнение. По завершении всех задач потоки в пуле уничтожаются. В качестве альтернативы можно вызвать метод `shutdownNow()`. В этом случае пул отменяет все задачи, которые еще не запущены, пытаясь прервать исполняемые потоки.

Ниже перечислены действия для организации пула потоков исполнения.

1. Вызовите статический метод `newCachedThreadPool()` или `newFixedThreadPool()` из класса `Executors`.
2. Вызовите метод `submit()` для передачи объектов `Runnable` или `Callable` в пул потоков исполнения.
3. Полагайтесь на возвращаемые объекты типа `Future`, чтобы иметь возможность получить результат и прервать задачу.
4. Вызовите метод `shutdown()`, если больше не собираетесь запускать новые задачи.

В состав интерфейса `ScheduledExecutorService` входят методы для планирования или многократного выполнения задач. Это — обобщение класса `java.util.Timer`, позволяющее организовать пул потоков исполнения. Методы `newScheduledThreadPool()` и `newSingleThreadScheduledExecutor()` из класса `Executors` возвращают объекты, реализующие интерфейс `ScheduledExecutorService`. Имеется возможность запланировать однократный запуск задач типа `Runnable` или `Callable` по истечении некоторого времени, а также периодический запуск задач типа `Runnable`. Более подробно средства планового выполнения потоков представлены ниже, в описании соответствующего прикладного интерфейса API.

`java.util.concurrent.Executors` 5

- **`ExecutorService newCachedThreadPool()`**
Возвращает кешированный пул потоков, создающий потоки исполнения по мере необходимости и закрывающий те потоки, которые простаивают больше 60 секунд.
- **`ExecutorService newFixedThreadPool(int threads)`**
Возвращает пул потоков, использующий заданное количество потоков исполнения для запуска задач.
- **`ExecutorService newSingleThreadExecutor()`**
Возвращает исполнитель, поочередно запускающий задачи на исполнение в одном потоке.
- **`ScheduledExecutorService newScheduledThreadPool(int threads)`**
Возвращает пул потоков исполнения, использующий заданное их количество для планирования запуска задачи.
- **`ScheduledExecutorService newSingleThreadScheduledExecutor()`**
Возвращает исполнитель, планирующий поочередный запуск задач в одном потоке исполнения.

`java.util.concurrent.ExecutorService` 5

- **`Future<T> submit(Callable<T> task)`**
- **`Future<T> submit(Runnable task, T result)`**
- **`Future<?> submit(Runnable task)`**
Передают указанную задачу на выполнение.
- **`void shutdown()`**
Останавливает службу, завершая все запущенные задачи и не принимая новые.

java.util.concurrent.ThreadPoolExecutor 5

- `int getLargestPoolSize()`

Возвращает максимальный размер пула потоков, который был достигнут в течение срока его существования.

java.util.concurrent.ScheduledExecutorService 5

- `ScheduledFuture<V> schedule(Callable<V> task, long time, TimeUnit unit)`
- `ScheduledFuture<?> schedule(Runnable task, long time, TimeUnit unit)`
- `ScheduledFuture<?> scheduleAtFixedRate(Runnable task, long initialDelay, long period, TimeUnit unit)`

Планируют запуск указанной задачи по истечении заданного промежутка времени.

Планирует периодический запуск указанной задачи через каждые `period` промежутков времени в заданных единицах `unit` по истечении первоначального времени задержки, определяемого параметром `initialDelay`.

- `ScheduledFuture<?> scheduleWithFixedDelay(Runnable task, long initialDelay, long delay, TimeUnit unit)`

Планирует периодический запуск указанной задачи с указанным временем задержки `delay` в заданных единицах `unit` между запусками по истечении первоначального времени задержки, определяемого параметром `initialDelay`.

12.6.3. Управление группами задач

Ранее было показано, как пользоваться службой исполнителей в виде пула потоков для повышения эффективности исполнения задач. Но иногда исполнитель применяется скорее по причинам тактического характера только для управления группами взаимосвязанных задач. Например, все задачи можно прервать в исполнителе, вызвав его метод `shutdownNow()`.

Метод `invokeAny()` передает исполнителю все объекты типа `Callable` из коллекции и возвращает результат выполненной задачи. Но заранее неизвестно, к какой именно задаче относится возвращаемый результат. Можно лишь предположить, что это будет результат выполнения задачи, которая завершилась быстрее всех. Поэтому данный метод вызывается для поиска любого приемлемого решения поставленной задачи. Допустим, требуется разложить на множители большие целые числа — вычисление, необходимое для взлома шифра RSA. С этой целью можно запустить целый ряд задач, в каждой из которых будет предпринята попытка найти множители в отдельном диапазоне чисел. Как только в одной из задач будет найдено решение, все остальные вычисления можно прекратить.

Метод `invokeAll()` запускает все объекты типа `Callable` из коллекции и возвращает список объектов типа `Future`, представляющих результаты выполнения всех задач. Обработку этих результатов можно организовать по мере их готовности следующим образом:

```
List<Callable<T>> tasks = . . . ;
List<Future<T>> results = executor.invokeAll(tasks);
```

```
for (Future<T> result : results)
    processFurther(result.get());
```

В цикле `for` первый вызов `result.get()` блокируется до тех пор, пока не станет доступным первый результат. Это не вызовет особых трудностей, если все задачи завершатся приблизительно в одно и то же время. Но результаты стоило бы получить в том порядке, в каком они окажутся доступными. И это можно сделать с помощью класса `ExecutorCompletionService`.

С этой целью запускается исполнитель, полученный обычным способом. Затем получается объект типа `ExecutorCompletionService`. Далее полученному экземпляру службы исполнителей передаются исполняемые задачи. Эта служба управляет блокирующей очередью объектов типа `Future`, содержащих результаты переданных на выполнение задач, причем очередь заполняется по мере готовности результатов. Таким образом, вычисления можно организовать более эффективно, как показано ниже.

```
var service = new ExecutorCompletionService<T>(executor);
for (Callable<T> task : tasks) service.submit(task);
for (int i = 0; i < tasks.size(); i++)
    processFurther(service.take().get());
```

В примере программы из листинга 12.8 демонстрируется применение асинхронно выполняемых задач и исполнителей. На первой стадии вычисления подсчитывается количество файлов в дереве каталога, где содержится заданное слово. Для обработки каждого файла создается отдельная задача, как показано ниже.

```
Set<Path> files = descendants(Path.of(start));
var tasks = new ArrayList<Callable<Long>>();
for (Path file : files)
{
    Callable<Long> task = () -> occurrences(word, file);
    tasks.add(task);
}
```

Затем задачи передаются службе исполнителя:

```
ExecutorService executor = Executors.newCachedThreadPool();
List<Future<Long>> results = executor.invokeAll(tasks);
```

Чтобы получить сводный подсчет, все результаты необходимо просуммировать. Но эта операция блокируется до тех пор, пока они не станут доступными.

```
long total = 0;
for (Future<Long> result : results)
    total += result.get();
```

В рассматриваемой здесь программе отображается также время, затраченное на поиск. Распакуйте в избранном каталоге исходный код комплекта JDK и произведите в нем поиск. Затем замените службу исполнителя однопоточным исполнителем и сделайте еще одну попытку, чтобы выяснить, насколько быстрее было выполнено параллельное вычисление.

На второй стадии вычисления в данной программе осуществляется поиск первого файла, содержащего заданное слово. Для распараллеливания поиска служит метод `invokeAny()`. И здесь необходимо более тщательно формулировать исполняемые задачи. Метод `invokeAny()` завершится, как только произойдет *возврат* из любой задачи. Поэтому задачи поиска не должны возвращать логическое значение, обозначающее удачное или неудачное завершение. Ведь поиск не должен прерываться из-за неудачного завершения отдельной задачи. Вместо этого неудачно завершившаяся

задача должна сгенерировать исключение типа `NoSuchElementException`. А если одна задача завершится удачно, остальные задачи отменяются. И для этого требуется текущий контроль состояния прерванного потока исполнения. Если базовый поток исполнения прерывается, задача поиска выводит сообщение, прежде чем завершиться, чтобы было ясно видно, что завершение произошло эффективно.

```
public static Callable<Path> searchForTask(
    String word, Path path)
{
    return () -> {
        try (var in = new Scanner(path))
        {
            while (in.hasNext())
            {
                if (in.next().equals(word)) return path;
                if (Thread.currentThread().isInterrupted())
                {
                    System.out.println("Search in " + path
                        + " canceled.");
                    return null;
                }
            }
            throw new NoSuchElementException();
        }
    };
}
```

Ради большей информативности в данной программе выводится наибольший размер пула потоков, достигаемый во время исполнения. Но эти сведения недоступны через интерфейс `ExecutorService`, поэтому объект пула потоков исполнения пришлось привести к типу `ThreadPoolExecutor`.



СОВЕТ. Анализируя исходный код данной программы, можете оценить пользу, которую приносит служба исполнителя. Для управления потоками исполнения в своих программах пользуйтесь службами исполнителей вместо того, чтобы запускать потоки на исполнение по отдельности.

Листинг 12.8. Исходный код из файла `executors/ExecutorDemo.java`

```
1  package executors;
2
3  import java.io.*;
4  import java.nio.file.*;
5  import java.time.*;
6  import java.util.*;
7  import java.util.concurrent.*;
8  import java.util.stream.*;
9
10 /**
11  * В этой программе демонстрируется применение
12  * интерфейса Callable и исполнителей
13  * @version 1.0 2018-01-04
14  * @author Cay Horstmann
15  */
16 public class ExecutorDemo
```



```
17 {
18     /**
19      * Подсчитывает количество вхождений заданного
20      * слова в указанном файле
21      * @return Возвращает количество вхождений
22      *         заданного слова в указанном файле
23      */
24     public static long occurrences(String word, Path path)
25     {
26         try (var in = new Scanner(path))
27         {
28             int count = 0;
29             while (in.hasNext())
30                 if (in.next().equals(word)) count++;
31             return count;
32         }
33         catch (IOException ex)
34         {
35             return 0;
36         }
37     }
38
39     /**
40      * Возвращает все каталоги, порожденные заданным
41      * каталогом — см. главы 1 и 2 второго тома
42      * @param rootDir Корневой каталог
43      * @return Возвращает все каталоги, порожденные
44      *         корневым каталогом
45      */
46     public static Set<Path> descendants(Path rootDir)
47         throws IOException
48     {
49         try (Stream<Path> entries = Files.walk(rootDir))
50         {
51             return entries.filter(Files::isRegularFile)
52                 .collect(Collectors.toSet());
53         }
54     }
55
56     /**
57      * Выдает задачу поиска слова в файле
58      * @param word the word to search
59      * @param path Путь к файлу для поиска слова
60      * @return Возвращает задачу поиска, выдающую
61      *         путь при удачном завершении
62      */
63     public static Callable<Path> searchForTask(
64         String word, Path path)
65     {
66         return () -> {
67             try (var in = new Scanner(path))
68             {
69                 while (in.hasNext())
70                 {
71                     if (in.next().equals(word)) return path;
72                     if (Thread.currentThread().isInterrupted())
```

```
73         {
74             System.out.println("Search in " + path
75                                 + " canceled.");
76             return null;
77         }
78     }
79     throw new NoSuchElementException();
80 }
81 };
82 }
83
84 public static void main(String[] args)
85     throws InterruptedException,
86         ExecutionException, IOException
87 {
88     try (var in = new Scanner(System.in))
89     {
90         System.out.print("Enter base directory "
91                         + "(e.g. /opt/jdk-9-src): ");
92         String start = in.nextLine();
93         System.out.print("Enter keyword "
94                         + "(e.g. volatile): ");
95         String word = in.nextLine();
96
97         Set<Path> files = descendants(Path.of(start));
98         var tasks = new ArrayList<Callable<Long>>();
99         for (Path file : files)
100         {
101             Callable<Long> task = () ->
102                 occurrences(word, file);
103             tasks.add(task);
104         }
105         ExecutorService executor =
106             Executors.newCachedThreadPool();
107         // использовать один исполнитель потоков вместо
108         // того, чтобы выяснять, можно ли ускорить поиск
109         // с помощью нескольких потоков исполнения
110         // ExecutorService executor =
111             Executors.newSingleThreadExecutor();
112
113         Instant startTime = Instant.now();
114         List<Future<Long>> results =
115             executor.invokeAll(tasks);
116         long total = 0;
117         for (Future<Long> result : results)
118             total += result.get();
119         Instant endTime = Instant.now();
120         System.out.println("Occurrences of " + word
121                             + ": " + total);
122         System.out.println("Time elapsed: "
123                             + Duration.between(startTime, endTime)
124                                 .toMillis() + " ms");
125         var searchTasks = new ArrayList<Callable<Path>>();
126         for (Path file : files)
127             searchTasks.add(searchForTask(word, file));
128         Path found = executor.invokeAny(searchTasks);
```

```

129     System.out.println(word + " occurs in: " + found);
130
131     if (executor instanceof ThreadPoolExecutor)
132         // не единственный исполнитель потоков
133         System.out.println("Largest pool size: "
134             + ((ThreadPoolExecutor) executor)
135             .getLargestPoolSize());
136     executor.shutdown();
137 }
138 }
139 }

```

java.util.concurrent.ExecutorService 5

- **T invokeAny(Collection<Callable<T>> tasks)**
- **T invokeAny(Collection<Callable<T>> tasks, long timeout, TimeUnit unit)**

Выполняют указанные задачи и возвращают результат выполнения одной из них. Второй метод генерирует исключение типа **TimeoutException** по истечении времени ожидания.

- **List<Future<T>> invokeAll(Collection<Callable<T>> tasks)**
- **List<Future<T>> invokeAll(Collection<Callable<T>> tasks, long timeout, TimeUnit unit)**

Выполняют указанные задачи и возвращают результаты выполнения всех задач. Второй метод генерирует исключение типа **TimeoutException** по истечении времени ожидания.

java.util.concurrent.ExecutorCompletionService 5

- **ExecutorCompletionService(Executor e)**

Конструирует службу завершения исполнителя, которая собирает результаты работы заданного исполнителя.

- **Future<T> submit(Callable<T> task)**
- **Future<T> submit(Runnable task, T result)**

Передают на выполнение задачу базовому исполнителю.

- **Future<T> take()**

Удаляет следующий готовый результат, устанавливая блокировку, если готовых результатов нет.

- **Future<T> poll()**

- **Future<T> poll(long time, TimeUnit unit)**

Удаляют следующий готовый результат или пустое значение **null**, если готовых результатов нет. Второй метод ожидает в течение указанного промежутка времени.

12.6.4. Архитектура вилочного соединения

В части приложений используется большое количество потоков исполнения, которые в основном простаивают. Примером тому может служить веб-сервер, использующий по одному потоку исполнения на каждое соединение. А в других приложениях

на каждое ядро процессора приходится по одному потоку для исполнения таких задач, требующих интенсивных вычислений, как обработка изображений и видео-записей. Для поддержки именно таких приложений в версии Java 7 появилась архитектура вилочного соединения. Допустим, имеется задача обработки, естественно разделяемая на подзадачи следующим образом:

```
if (problemSize < threshold)
    решить задачу непосредственно
else
{
    разделить задачу на подзадачи, решить каждую подзадачу
    рекурсивно и объединить полученные результаты
}
```

Примером такой задачи служит обработка изображений. Для увеличения изображения можно преобразовать по отдельности верхнюю и нижнюю его половины. Если для подобных операций имеется достаточно свободных процессоров, то их выполнение можно распараллелить. (Безусловно, для разделения обрабатываемого изображения на две половины потребуются дополнительные действия, но это уже технические подробности, не имеющие отношения к делу.)

Обратимся к более простому примеру. Допустим, требуется подсчитать количество элементов в массиве с определенным свойством. Для этого массив разделяется на две половины, в каждой из них подсчитываются соответствующие элементы, а затем результаты складываются.

Чтобы облечь рекурсивные вычисления в форму, пригодную для архитектуры вилочного соединения, сначала предоставляется класс, расширяющий класс `RecursiveTask<T>` (если результат вычисления относится к типу `T`) или же класс `RecursiveAction` (если получение результата не предполагается). Затем переопределяется метод `compute()` для формирования и вызова подзадач, а также объединения результатов их выполнения. Ниже приведен соответствующий пример кода.

```
class Counter extends RecursiveTask<Integer>
{
    . . .
    protected Integer compute()
    {
        if (to - from < THRESHOLD)
        {
            решить задачу непосредственно
        }
        else
        {
            int mid = (from + to) / 2;
            var first = new Counter(values, from, mid, filter);
            var second = new Counter(values, mid, to, filter);
            invokeAll(first, second);
            return first.join() + second.join();
        }
    }
}
```

В данном примере метод `invokeAll()` получает ряд задач и блоков до тех пор, пока их выполнение не будет завершено, а метод `join()` объединяет полученные результаты. В частности, метод `join()` вызывается для каждой подзадачи, а в итоге возвращается сумма результатов их выполнения.



НА ЗАМЕТКУ! Имеется также метод `get()` для получения текущего результата, но он менее привлекателен, поскольку может сгенерировать проверяемые исключения, которые не допускаются генерировать в методе `compute()`.

Весь исходный код рассматриваемого здесь примера приведен в листинге 12.9. В самой архитектуре вилочного соединения применяется эффективный эвристический алгоритм для уравнивания рабочей нагрузки на имеющиеся потоки исполнения, называемый *перехватом работы*. Для каждого рабочего потока исполнения организуется двухсторонняя очередь выполняемых задач. Рабочий поток исполнения размещает подзадачи в голове своей двухсторонней очереди, причем только один поток исполнения имеет доступ к голове этой очереди, благодаря чему исключается потребность в блокировке потоков. Когда рабочий процесс простаивает, он пытается перехватить задачу из хвоста другой двухсторонней очереди, но поскольку в хвосте очереди обычно располагаются крупные подзадачи, то потребность в перехвате задач возникает редко.



ВНИМАНИЕ! Пулы вилочного соединения оптимизированы для неблокирующих нагрузок. Если ввести много блокирующих задач в пул вилочного соединения, его можно истощить. Но это препятствие можно преодолеть, если реализовать в задачах интерфейс `ForkJoinPool.ManagedBlocker`. Впрочем, этот непростой прием здесь не рассматривается.

Листинг 12.9. Исходный код из файла `forkJoin/forkJoinTest.java`

```
1 package forkJoin;
2
3 import java.util.concurrent.*;
4 import java.util.function.*;
5
6 /**
7  * В этой программе демонстрируется
8  * архитектура вилочного соединения
9  * @version 1.01 2015-06-21
10 * @author Cay Horstmann
11 */
12 public class ForkJoinTest
13 {
14     public static void main(String[] args)
15     {
16         final int SIZE = 10000000;
17         var numbers = new double[SIZE];
18         for (int i = 0; i < SIZE; i++)
19             numbers[i] = Math.random();
20         var = new Counter(numbers, 0, numbers.length,
21                             x -> x > 0.5);
22         var pool = new ForkJoinPool();
23         pool.invoke(counter);
24         System.out.println(counter.join());
25     }
26 }
27
28 class Counter extends RecursiveTask<Integer>
```

```
29 {
30     public static final int THRESHOLD = 1000;
31     private double[] values;
32     private int from;
33     private int to;
34     private DoublePredicate filter;
35
36     public Counter(double[] values, int from, int to,
37                   DoublePredicate filter)
38     {
39         this.values = values;
40         this.from = from;
41         this.to = to;
42         this.filter = filter;
43     }
44
45     protected Integer compute()
46     {
47         if (to - from < THRESHOLD)
48         {
49             int count = 0;
50             for (int i = from; i < to; i++)
51             {
52                 if (filter.test(values[i])) count++;
53             }
54             return count;
55         }
56         else
57         {
58             int mid = (from + to) / 2;
59             Counter first =
60                 new Counter(values, from, mid, filter);
61             Counter second =
62                 new Counter(values, mid, to, filter);
63             invokeAll(first, second);
64             return first.join() + second.join();
65         }
66     }
67 }
```

12.7. Асинхронные вычисления

До сих пор рассматривался способ организации параллельных вычислений, который состоял в том, чтобы разбить задачу на мелкие части и ожидать их завершения. Но ожидать не всегда уместно. Поэтому в последующих разделах будет показано, как организовать *асинхронные* вычисления без ожидания.

12.7.1. Завершаемые будущие действия

Если имеется объект типа `Future`, придется вызвать метод `get()`, чтобы получить значение, установив блокировку до тех пор, пока данное значение не станет доступным. В классе `CompletableFuture` реализуется интерфейс `Future` и предоставляется второй механизм для получения результата. Остается лишь зарегистрировать

обратный вызов, который будет сделан (в некотором потоке исполнения), чтобы получить результат, как только он будет доступен. Подобным образом можно обработать результат без блокировки после того, как он будет доступен.

```
CompletableFuture<String> f = . . .;
f.thenAccept(s -> обработать строку результата s);
```

В прикладном интерфейсе API имеется ряд методов, возвращающих объекты типа `CompletableFuture`. Например, веб-страницу можно извлечь асинхронно с помощью экспериментального класса `HttpClient`, упоминаемого в главе 4 второго тома настоящего издания, как показано ниже.

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder()
    .uri(create(urlString)).GET().build();
CompletableFuture<HttpResponse<String>> f =
    client.sendAsync(request, BodyHandler.asString());
```

Было бы неплохо иметь в своем распоряжении готовый метод, выдающий готовый объект типа `CompletableFuture`, но зачастую для этого приходится создавать собственный метод. Чтобы выполнить задачу асинхронно и получить в итоге объект типа `CompletableFuture`, совсем не обязательно передавать его непосредственно службе исполнителя. Вместо этого можно вызвать статический метод `CompletableFuture.supplyAsync()`. Ниже показано, как прочитать веб-страницу, не пользуясь преимуществами класса `HttpClient`.

```
public CompletableFuture<String> readPage(URL url)
{
    return CompletableFuture.supplyAsync(() ->
    {
        try
        {
            return new String(url.openStream().readAllBytes(),
                "UTF-8");
        }
        catch (IOException e)
        {
            throw new UncheckedIOException(e);
        }
    }, executor);
}
```

Если опустить исполнитель, задача будет выполнена используемым по умолчанию исполнителем, возвращаемым методом `ForkJoinPool.commonPool()`. Но, как правило, это нежелательно.



ВНИМАНИЕ! Следует, однако, иметь в виду, что в качестве первого аргумента при вызове метода `supplyAsync()` указывается объект типа `Supplier<T>`, а не типа `Callable<T>`. Оба функциональных интерфейса описывают функции без аргументов и возвращают значение обобщенного типа `T`, но функция из интерфейса `Supplier` неспособна сгенерировать исключение. Как следует из приведенного выше примера кода, такой выбор был сделан явно не по вдохновению.

Будущее действие типа `CompletableFuture` может завершиться двояко: полученным результатом или необрабатываемым исключением. Чтобы принять во внимание оба случая, следует воспользоваться методом `whenComplete()`. Предоставляемая

функция вызывается с полученным результатом, а в его отсутствие — с пустым значением `null`, или же с исключением, а в его отсутствие — с пустым значением `null`.

```
f.whenComplete((s, t) -> {
    if (t == null) { обработать результат s; }
    else { обработать исключение Throwable t; }
});
```

Будущее действие типа `CompletableFuture` называется завершаемым потому, что значение завершения может быть установлено вручную. (В других библиотеках параллелизма такой объект называется *обещанием* или *обязательством*.) Безусловно, при создании объекта типа `CompletableFuture` с помощью метода `supplyAsync()` значение завершения устанавливается неявно по завершении задачи. Но установка результата явным образом доставляет дополнительные удобства. Например, две задачи могут одновременно работать над вычислением ожидаемого ответа, как показано ниже.

```
var f = new CompletableFuture<Integer>();
executor.execute(() ->
{
    int n = workHard(arg);
    f.complete(n);
});
executor.execute(() ->
{
    int n = workSmart(arg);
    f.complete(n);
});
```

Вместо завершения будущего действия исключением достаточно сделать следующий вызов:

```
Throwable t = . . .;
f.completeExceptionally(t);
```



НА ЗАМЕТКУ! Метод `complete()` или `completeExceptionally()` надежнее вызвать для одного и того же будущего действия в нескольких потоках исполнения. Если же будущее действие уже завершено, подобные вызовы не возымеют никакого действия.

Метод `isDone()` сообщает, было ли будущее действие (объект типа `Future`) завершено (нормально или исключением). В приведенном выше примере эти сведения могут быть использованы в методах `workHard()` и `workSmart()`, чтобы прервать выполнение одного из этих методов, когда результат определен в другом из них.



ВНИМАНИЕ! В отличие от простого будущего действия типа `Future`, вычисление завершаемого будущего действия типа `CompletableFuture` не прерывается при вызове его метода `cancel()`. Отмена просто приводит к завершению будущего действия типа `Future` исключением типа `CancellationException`. И в этом, как правило, есть свой смысл, поскольку для завершаемого будущего действия типа `CompletableFuture` может не оказаться единственного потока исполнения, отвечающего за его завершение. Но такое ограничение накладывается и на экземпляры типа `CompletableFuture`, возвращаемые методами вроде `supplyAsync()`, которые в принципе могут прерываться.

12.7.2. Составление завершаемых будущих действий

Неблокирующие вызовы реализуются через обратные вызовы. Программист регистрирует обратный вызов для действия, которое должно произойти по завершении задачи. Разумеется, если следующее действие также является асинхронным, то и следующее после него действие должно происходить в другом обработчике событий. Несмотря на то что программист мыслит следующими категориями: сначала сделать шаг 1, затем шаг 2 и далее шаг 3, логика программы становится распределенной среди разных обработчиков. А добавление обработки ошибок только усложняет дело. Допустим, что на шаге 2 пользователь входит в систему. Этот шаг, возможно, придется повторить, поскольку пользователь может ввести свои учетные данные с опечатками. Реализовать такой поток управления в ряде обработчиков событий непросто, а еще труднее другим понять, как он был реализован.

Совсем иной подход принят в классе `CompletableFuture`. В отличие от обработчиков событий, завершаемые будущие действия могут быть *составлены*. Допустим, что с веб-страницы требуется извлечь все ссылки, чтобы построить поисковый робот. Допустим также, что для этой цели имеется следующий метод, получающий текст из веб-страницы, как только он становится доступным:

```
public void CompletableFuture<String> readPage(URL url)
```

Если следующий метод:

```
public List<URL> getImageURLs(String page)
```

получает URL на HTML-странице, то его вызов можно запланировать на момент, когда страница будет доступна:

```
CompletableFuture<String> contents = readPage(url);  
CompletableFuture<List<URL>> imageURLs =  
    contents.thenApply(this::getLinks);
```

Метод `thenApply()` вообще не блокируется. Он возвращает другое будущее действие. По завершении первого будущего действия его результат передается методу `getLinks()`, а значение, возвращаемое этим методом, становится окончательным результатом.

Применяя завершаемые будущие действия, достаточно указать, что и в каком порядке требуется сделать. Безусловно, все это происходит не сразу, но самое главное, что весь код оказывается в одном месте.

Принципиально класс `CompletableFuture` является простым прикладным интерфейсом API, но для составления завершаемых будущих действий имеется немало вариантов его методов. Рассмотрим сначала те из них, которые обращаются с единственным будущим действием. Для каждого метода, перечисленного в табл. 12.3, имеются еще два варианта типа `Async`, которые в этой таблице не представлены. В одном из этих вариантов используется общий объект типа `ForkJoinPool`, а в другом имеется параметр типа `Executor`. Кроме того, в табл. 12.3 употребляется следующее сокращенное обозначение громоздких функциональных интерфейсов: `T -> U` вместо `Function<? super T, U>`. Обозначения `T` и `U`, разумеется, не имеют никакого отношения к конкретным типам данных в Java.

Метод `thenApply()` уже был представлен выше. Допустим, что `f` — это функция, получающая значения типа `T` и возвращающая значения типа `U`. Так, в результате следующих вызовов:

```
CompletableFuture<U> future.thenApply(f);
CompletableFuture<U> future.thenApplyAsync(f);
```

возвращается будущее действие, применяющее функцию *f* к результату будущего действия *future*, как только он станет доступным. А в результате второго вызова функция *f* выполняется еще в одном потоке.

Вместо функции, преобразующей тип *T* в тип *U*, метод `thenCompose()` принимает функцию, преобразующую тип *T* в тип `CompletableFuture<U>`. На первый взгляд это кажется довольно абстрактным, тем не менее, может быть вполне естественным. Рассмотрим действие чтения веб-страницы по заданному URL. Вместо того чтобы вызывать следующий метод:

```
public String blockingReadPage(URL url)
```

изящнее вернуть из метода будущее действие следующим образом:

```
public CompletableFuture<String> readPage(URL url)
```

А теперь допустим, что имеется еще один, приведенный ниже метод, получающий URL из вводимых пользователем данных, возможно, в диалоговом окне, где ответ не появляется до тех пор, пока пользователь не щелкнет на экранной кнопке ОК. И это считается событием в будущем действии.

```
public CompletableFuture<URL> getURLInput(String prompt)
```

В данном случае имеются две функции: *T* → `CompletableFuture<U>` и *U* → `CompletableFuture<V>`. Очевидно, что они составляют функцию *T* → `CompletableFuture<V>`, если вызывается вторая функция, когда завершается первая. Именно это и делается в методе `thenCompose()`.

В предыдущем разделе был представлен метод `whenComplete()`, предназначенный для обработки исключений. Имеется также метод `handle()`, требующий функцию для обработки результата или исключения и вычисления нового результата. Но зачастую вместо этого оказывается проще вызвать метод `exceptionally()`. Этот метод вычисляет фиктивное значение, когда возникает исключение, как показано ниже.

```
CompletableFuture<List<URL>> imageURLs = readPage(url)
    .exceptionally(ex -> "<html></html>")
    .thenApply(this::getImageURLs)
```

Аналогичным образом можно обработать выдержку времени:

```
CompletableFuture<List<URL>> imageURLs = readPage(url)
    .completeOnTimeout("<html></html>", 30,
        TimeUnit.SECONDS)
    .thenApply(this::getImageURLs)
```

С другой стороны, можно сгенерировать исключение по истечении выдержки времени:

```
CompletableFuture<String> = readPage(url).orTimeout(30, TimeUnit.SECONDS)
```

Те методы из табл. 12.3, которые возвращают результат типа `void`, обычно вызываются в конце конвейера обработки.

Таблица 12.3. Методы ввода будущего действия в объект типа `CompletableFuture<T>`

Метод	Параметр	Описание
<code>thenApply()</code>	<code>T -> U</code>	Применить функцию к результату
<code>thenAccept()</code>	<code>T -> void</code>	Аналогично методу <code>thenApply()</code> , но с результатом типа <code>void</code>
<code>thenCompose()</code>	<code>T -> CompletableFuture<U></code>	Вызвать функцию для результата и выполнить возвращаемое будущее действие
<code>handle()</code>	<code>(T, Throwable) -> U</code>	Обработать результат или ошибку
<code>whenComplete()</code>	<code>(T, Throwable) -> void</code>	Аналогично методу <code>handle()</code> , но с результатом типа <code>void</code>
<code>exceptionally()</code>	<code>Throwable -> T</code>	Вычислить результат из ошибки
<code>completeOnTimeout()</code>	<code>T, long, TimeUnit</code>	Выдать заданное значение как результат по истечении выдержки времени
<code>orTimeout()</code>	<code>long, TimeUnit</code>	Выдать исключение типа <code>TimeoutException</code> по истечении выдержки времени
<code>thenRun()</code>	<code>Runnable</code>	Выполнить задачу типа <code>Runnable</code> с результатом типа <code>void</code>

А теперь рассмотрим методы, объединяющие несколько будущих действий (табл. 12.4). Три первых метода выполняют действия типа `CompletableFuture<T>` и `CompletableFuture<U>` параллельно и объединяют полученные результаты.

Следующие три метода выполняют два действия типа `CompletableFuture<T>` параллельно. Как только одно из них завершается, передается его результат, а результат другого действия игнорируется.

И, наконец, статические методы `allOf()` и `anyOf()` принимают переменное количество завершаемых будущих действий и получают завершаемое действие типа `CompletableFuture<Void>`, которое завершается, когда завершаются все они или одно из них. В таком случае результаты не распространяются дальше.

Таблица 12.4. Методы объединения нескольких объектов составления будущих действий

Метод	Параметры	Описание
<code>thenCombine()</code>	<code>CompletableFuture<U>, (T, U) -> V</code>	Выполнить оба действия и объединить полученные результаты с помощью заданной функции
<code>thenAcceptBoth()</code>	<code>CompletableFuture<U>, (T, U) -> void</code>	Аналогично методу <code>thenCombine()</code> , но с результатом типа <code>void</code>
<code>runAfterBoth()</code>	<code>CompletableFuture<?>, Runnable</code>	Выполнить задачу типа <code>Runnable</code> по завершении обоих действий
<code>applyToEither()</code>	<code>CompletableFuture<T>, T -> V</code>	Если доступен результат выполнения одного или другого действия, передать его заданной функции
<code>acceptEither()</code>	<code>CompletableFuture<T>, T -> void</code>	Аналогично методу <code>applyToEither()</code> , но с результатом типа <code>void</code>

Окончание табл. 12.4

Метод	Параметры	Описание
<code>runAfterEither()</code>	<code>CompletableFuture<?>, Runnable</code>	Выполнить задачу типа Runnable по завершении одного или другого действия
<code>static allOf()</code>	<code>CompletableFuture<?>...</code>	Завершить с результатом типа void по окончании всех заданных будущих действий
<code>static anyOf()</code>	<code>CompletableFuture<?>...</code>	Завершить с результатом типа void по окончании любого из заданных будущих действий



НА ЗАМЕТКУ Формально методы, рассматриваемые в этом разделе, принимают параметры типа **CompletionStage**, а не типа **CompletableFuture**. Интерфейс **CompletionStage** состоит почти из сорока абстрактных методов, реализуемых в классе **CompletableFuture**. Этот интерфейс предоставляется для того, чтобы его можно было реализовать в сторонних каркасах.

В листинге 12.10 демонстрируется исходный код примера готовой программы, читающей веб-страницу, просматривающей на ней изображения, загружающей и сохраняющей их локально. Обратите внимание, каким образом все методы, потребляющие немало времени, возвращают объект типа **CompletableFuture**. Чтобы начать асинхронное вычисление, в данном случае применяется небольшая хитрость. Так, вместо непосредственного вызова метода `readPage()` сначала выполняется завершаемое будущее действие с URL в качестве аргумента, а затем составляется будущее действие по ссылке на метод `this::readPage()`. Благодаря этому конвейер приобретает вполне единообразный вид:

```
CompletableFuture.completedFuture(url)
    .thenComposeAsync(this::readPage, executor)
    .thenApply(this::getImageURLs)
    .thenCompose(this::getImages)
    .thenAccept(this::saveImages);
```

Листинг 12.10. Исходный код из файла **completableFutures/CompletableFutureDemo.java**

```
1  package completableFutures;
2
3  import java.awt.image.*;
4  import java.io.*;
5  import java.net.*;
6  import java.nio.charset.*;
7  import java.util.*;
8  import java.util.concurrent.*;
9  import java.util.regex.*;
10
11 import javax.imageio.*;
12
13 public class CompletableFutureDemo
14 {
15     private static final Pattern IMG_PATTERN =
16         Pattern.compile("<[<|\\s*[iI][mM][gG]"
17             + "\\s*[>]*[sS][rR][cC]"
```

```
18             + "\\s*=[]\\s*['\"]"
19             + "([^\"]*)['\"](?:>)*[>]");
20 private ExecutorService executor =
21     Executors.newCachedThreadPool();
22 private URL urlToProcess;
23
24 public CompletableFuture<String> readPage(URL url)
25 {
26     return CompletableFuture.supplyAsync(() ->
27     {
28         try
29         {
30             var contents = new String(url.openStream()
31                                     .readAllBytes(),
32                                     StandardCharsets.UTF_8);
33             System.out.println("Read page from " + url);
34             return contents;
35         }
36         catch (IOException e)
37         {
38             throw new UncheckedIOException(e);
39         }
40     }, executor);
41 }
42
43 public List<URL> getImageURLs(String webpage)
44     // без потребления времени
45 {
46     try
47     {
48         var result = new ArrayList<URL>();
49         Matcher matcher = IMG_PATTERN.matcher(webpage);
50         while (matcher.find())
51         {
52             var url = new URL(urlToProcess,
53                             matcher.group(1));
54             result.add(url);
55         }
56         System.out.println("Found URLs: " + result);
57         return result;
58     }
59     catch (IOException e)
60     {
61         throw new UncheckedIOException(e);
62     }
63 }
64
65 public CompletableFuture<List<BufferedImage>>
66     getImages(List<URL> urls)
67 {
68     return CompletableFuture.supplyAsync(() ->
69     {
70         try
71         {
72             var result = new ArrayList<BufferedImage>();
73             for (URL url : urls)
74             {
```

```
75         result.add(ImageIO.read(url));
76         System.out.println("Loaded " + url);
77     }
78     return result;
79 }
80 catch (IOException e)
81 {
82     throw new UncheckedIOException(e);
83 }
84 }, executor);
85 }
86
87 public void saveImages(List<BufferedImage> images)
88 {
89     System.out.println("Saving " + images.size()
90         + " images");
91     try
92     {
93         for (int i = 0; i < images.size(); i++)
94         {
95             String filename = "/tmp/image" + (i + 1)
96                 + ".png";
97             ImageIO.write(images.get(i), "PNG",
98                 new File(filename));
99         }
100     }
101     catch (IOException e)
102     {
103         throw new UncheckedIOException(e);
104     }
105     executor.shutdown();
106 }
107
108 public void run(URL url)
109     throws IOException, InterruptedException
110 {
111     urlToProcess = url;
112     CompletableFuture.completedFuture(url)
113         .thenComposeAsync(this::readPage, executor)
114         .thenApply(this::getImageURLs)
115         .thenCompose(this::getImages)
116         .thenAccept(this::saveImages);
117
118     /* или воспользоваться экспериментальным
119     // HTTP-клиентом:
120
121     HttpClient client = HttpClient.newBuilder()
122         .executor(executor).build();
123     HttpRequest request = HttpRequest.newBuilder(
124         urlToProcess.toURI()).GET().build();
125     client.sendAsync(request, BodyProcessor.asString())
126         .thenApply(HttpResponse::body)
127         .thenApply(this::getImageURLs)
128         .thenCompose(this::getImages)
129         .thenAccept(this::saveImages);
130
131     */
132 }
```

```
133
134 public static void main(String[] args)
135     throws IOException, InterruptedException
136 {
137     new CompletableFutureDemo().run(new URL(
138         "http://horstmann.com/index.html"));
139 }
140 }
```

12.7.3. Длительные задачи в обратных вызовах пользовательского интерфейса

Применение потоков исполнения объясняется, в частности, тем, что прикладные программы оперативно реагируют на действия пользователя. И это особенно важно в прикладных программах с графическим пользовательским интерфейсом. Когда в прикладной программе требуется выполнить какую-нибудь операцию, потребляющую немало времени, этого нельзя сделать в потоке исполнения пользовательского интерфейса, иначе его работа застопорится. Вместо этого следует запустить еще один рабочий поток исполнения.

Так, если требуется прочитать содержимое файла, как только пользователь щелкнет на экранной кнопке, то для достижения этой цели не следует поступать так, как показано ниже.

```
var open = new JButton("Open");
open.addActionListener(event ->
{ // НЕУДАЧНО - длительное действие выполняется
  // в потоке исполнения пользовательского интерфейса
  var in = new Scanner(file);
  while (in.hasNextLine())
  {
    String line = in.nextLine();
    . . .
  }
});
```

Вместо этого длительное действие лучше выполнить в отдельном потоке следующим образом:

```
open.addActionListener(event ->
{ // УДАЧНО - длительное действие выполняется
  // в отдельном потоке исполнения
  Runnable task = () ->
  {
    var in = new Scanner(file);
    while (in.hasNextLine())
    {
      String line = in.nextLine();
      . . .
    }
  };
  executor.execute(task);
});
```

Но пользовательский интерфейс нельзя обновить непосредственно из рабочего потока исполнения, в котором выполняется длительная задача. Пользовательские интерфейсы, построенные на основе библиотеки Swing, JavaFX или платформы

Android, не являются потокобезопасными. Манипулировать элементами пользовательского интерфейса из нескольких потоков исполнения нельзя без риска нарушить их нормальное функционирование. На самом деле в JavaFX и Android подобные попытки получить доступ к пользовательскому интерфейсу из другого потока исполнения проверяются, а в итоге генерируется соответствующее исключение.

Таким образом, любые обновления, которые должны происходить в пользовательском интерфейсе, необходимо планировать. В каждой библиотеке для построения пользовательского интерфейса предоставляется свой механизм, позволяющий планировать выполнение задачи, представленной объектом типа `Runnable`, в потоке исполнения пользовательского интерфейса. Например, в библиотеке `Swing` с этой целью делается следующий вызов:

```
EventQueue.invokeLater() ->  
    label.setText(percentage + "% complete");
```

Реализовать реакцию пользователя в рабочем потоке исполнения непросто, и поэтому в каждой библиотеке для построения пользовательского интерфейса предоставляется особый вспомогательный класс, берущий на себя все хлопоты, в том числе класс `SwingWorker` в `Swing`, класс `Task` в `JavaFX` или класс `AsyncTask` в `Android`. А программисту достаточно указать действия для выполнения длительной задачи (в отдельном потоке) и порядок обновлений по ходу выполнения задачи и окончательное решение (в потоке исполнения пользовательского интерфейса).

В примере программы из листинга 12.11 предоставляются команды для загрузки текстового файла и отмены процесса загрузки. Попробуйте запустить ее вместе с крупным текстовым файлом, например, файлом, содержащим полный англоязычный текст романа “Граф Монте-Кристо” и находящимся в каталоге `gutenberg` загружаемого исходного кода примеров к этой книге. Такой файл загружается в отдельном потоке исполнения, и в этот момент пункт меню `Open` (Открыть) остается недоступным, а пункт меню `Cancel` (Отмена), наоборот, доступным (рис. 12.6). После ввода из файла каждой текстовой строки обновляется счетчик в строке состояния. По завершении процесса загрузки пункт меню `Open` становится доступным, тогда как пункт меню `Cancel` — недоступным, а в строке состояния появляется сообщение “Done” (Готово).

Данный пример демонстрирует следующие типичные действия в пользовательском интерфейсе для выполнения фоновой задачи.

- После каждой единицы работы пользовательский интерфейс обновляется, чтобы показать ход ее выполнения.
- По завершении всей работы в пользовательский интерфейс вносятся окончательные изменения.

Класс `SwingWorker` облегчает решение данной задачи. В нем переопределяется метод `doInBackground()`, чтобы выполнять продолжительную задачу и периодически вызывать метод `publish()` для отображения хода ее выполнения. Этот метод выполняется в рабочем потоке. А метод `publish()`, в свою очередь, вызывает метод `process()`, чтобы вынудить поток диспетчеризации событий обработать данные о ходе выполнения задачи. По завершении задачи в потоке диспетчеризации событий вызывается метод `done()`, завершающий обновление пользовательского интерфейса.

Всякий раз, когда требуется выполнить какую-нибудь задачу в рабочем потоке, следует сконструировать новый экземпляр объекта типа `SwingWorker`. (Каждый такой рабочий объект предназначен для однократного применения.) Затем вызывается метод `execute()`. Обычно этот метод вызывается в потоке диспетчеризации событий, но делать этого не рекомендуется.

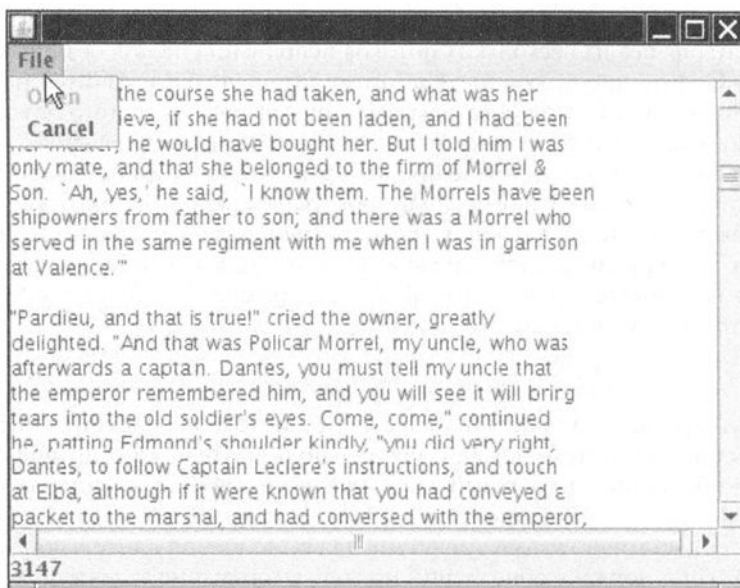


Рис. 12.6. Загрузка файла в отдельном потоке исполнения

Предполагается, что рабочий объект производит некоторый результат, и поэтому класс `SwingWorker<T, V>` реализует интерфейс `Future<T>`. Этот результат может быть получен с помощью метода `get()` из интерфейса `Future`. Метод `get()` устанавливает блокировку до тех пор, пока не будет доступен результат, поэтому вряд ли стоит вызывать его сразу же после метода `execute()`. Это лучше сделать, когда уже известно, что задача завершена. Обычно метод `get()` вызывается из метода `done()`. (Вызывать метод `get()` необязательно. Иногда достаточно и обработки данных о ходе выполнения задачи.)

Как промежуточные данные о ходе выполнения задачи, так и конечный результат могут иметь произвольные типы. Класс `SwingWorker` получает эти типы через параметры типа. Так, класс `SwingWorker<T, V>` выдает результат типа `T` и данные типа `V` о ходе выполнения задачи. Чтобы прервать выполнение задачи, достаточно вызвать метод `cancel()` из интерфейса `Future`. Как только задача будет отменена, метод `get()` сгенерирует исключение типа `CancellationException`.

Как упоминалось выше, вызов метода `publish()` из рабочего потока исполнения повлечет за собой вызов метода `process()` в потоке диспетчеризации событий. Ради повышения эффективности результаты нескольких вызовов метода `publish()` могут стать причиной только одного вызова метода `process()`. Метод `process()` принимает в качестве параметра список типа `List<V>`, содержащий все промежуточные результаты.

Обратимся к практическому применению описанного выше механизма на примере чтения из текстового файла. Компонент `JTextArea` оказывается довольно медленным. Так, для ввода строк из длинного текстового файла (вроде романа "Граф Монте-Кристо") потребуется немало времени. Чтобы показать пользователю ход выполнения данного процесса, требуется отображать в строке состояния количество прочитанных текстовых строк. Таким образом, данные о ходе чтения из текстового файла будут состоять из текущего номера строки и самой текстовой строки. Эти данные можно упаковать в тривиальный внутренний класс следующим образом:

```
private class ProgressData
{
    public int number;
    public String line;
}
```

В конечном итоге получается текст, прочитанный и сохраненный в объекте типа `StringBuilder`. А для его обработки в рабочем потоке понадобится класс `SwingWorker<StringBuilder, ProgressData>`.

В методе `doInBackground()` осуществляется построчное чтение из текстового файла. После каждой прочитанной текстовой строки вызывается метод `publish()` для передачи номера прочитанной строки и ее содержимого, как показано ниже.

```
@Override public StringBuilder doInBackground()
    throws IOException, InterruptedException
{
    int lineNumber = 0;
    Scanner in = new Scanner(new FileInputStream(file));
    while (in.hasNextLine())
    {
        String line = in.nextLine();
        lineNumber++;
        text.append(line);
        text.append("\n");
        var data = new ProgressData();
        data.number = lineNumber;
        data.line = line;
        publish(data);
        Thread.sleep(1); // только для проверки отмены;
                        // а в конкретных программах не требуется
    }
    return text;
}
```

После каждой прочитанной строки чтение из текстового файла приостанавливается на одну миллисекунду, чтобы благополучно проверить возможность отмены, но выполнение конкретных прикладных программ вряд ли потребует замедлять задержками подобного рода. Если закомментировать строку кода `Thread.sleep(1);`, то обнаружится, что текст романа “Граф Монте-Кристо” загружается достаточно быстро — всего в течение несколько групповых обновлений пользовательского интерфейса.

В методе `process()` игнорируются номера всех строк, за исключением последней, а все прочитанные текстовые строки сцепляются для единого обновления текстовой области:

```
@Override public void process(List<ProgressData> data)
{
    if (isCancelled()) return;
    var b = new StringBuilder();
    statusLine.setText("" + data.get(data.size() - 1).number);
    for (ProgressData d : data)
    { b.append(d.line); b.append("\n"); }
    textArea.append(b.toString());
}
```

В методе `done()` текстовая область обновляется полным текстом, а пункт меню `Cancel` становится недоступным. Следует также иметь в виду, что рабочий поток исполнения запускается в приемнике событий при выборе пункта меню `Open`. Описанная выше простая методика позволяет выполнять продолжительные задачи, сохраняя для GUI возможность реагировать на действия пользователя.

Листинг 12.11. Исходный код из файла `swingWorker/SwingWorkerTest.java`

```
5  import java.nio.charset.*;
6  import java.util.*;
7  import java.util.List;
8  import java.util.concurrent.*;
9
10 import javax.swing.*;
11
12 /**
13  * В этой программе демонстрируется рабочий
14  * поток, в котором выполняется потенциально
15  * продолжительная задача
16  * @version 1.12 2018-03-17
17  * @author Cay Horstmann
18  */
19 public class SwingWorkerTest
20 {
21     public static void main(String[] args)
22         throws Exception
23     {
24         EventQueue.invokeLater(() -> {
25             var frame = new SwingWorkerFrame();
26             frame.setDefaultCloseOperation(
27                 JFrame.EXIT_ON_CLOSE);
28             frame.setVisible(true);
29         });
30     }
31 }
32
33 /**
34  * Этот фрейм содержит текстовую область для
35  * отображения содержимого текстового файла,
36  * меню для открытия файла и отмены его открытия,
37  * а также строку состояния для отображения
38  * процесса загрузки файла
39  */
40 class SwingWorkerFrame extends JFrame
41 {
42     private JFileChooser chooser;
43     private JTextArea textArea;
44     private JLabel statusLine;
45     private JMenuItem openItem;
46     private JMenuItem cancelItem;
47     private SwingWorker<StringBuilder,
48         ProgressData> textReader;
49     public static final int TEXT_ROWS = 20;
50     public static final int TEXT_COLUMNS = 60;
```

```
51
52 public SwingWorkerFrame()
53 {
54     chooser = new JFileChooser();
55     chooser.setCurrentDirectory(new File("."));
56
57     textArea = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
58     add(new JScrollPane(textArea));
59
60     statusLine = new JLabel(" ");
61     add(statusLine, BorderLayout.SOUTH);
62
63     var menuBar = new JMenuBar();
64     setJMenuBar(menuBar);
65
66     var menu = new JMenu("File");
67     menuBar.add(menu);
68
69     openItem = new JMenuItem("Open");
70     menu.add(openItem);
71     openItem.addActionListener(event -> {
72         // показать диалоговое окно для выбора файлов
73         int result = chooser.showOpenDialog(null);
74
75         // если файл выбран, задать его в качестве
76         // пиктограммы для метки
77         if (result == JFileChooser.APPROVE_OPTION)
78         {
79             textArea.setText("");
80             openItem.setEnabled(false);
81             textReader = new TextReader(
82                 chooser.getSelectedFile());
83             textReader.execute();
84             cancelItem.setEnabled(true);
85         }
86     });
87
88     cancelItem = new JMenuItem("Cancel");
89     menu.add(cancelItem);
90     cancelItem.setEnabled(false);
91     cancelItem.addActionListener(event ->
92         textReader.cancel(true));
93     pack();
94 }
95
96 private class ProgressData
97 {
98     public int number;
99     public String line;
100 }
101
102 private class TextReader extends
103     SwingWorker<StringBuilder, ProgressData>
104 {
105     private File file;
106     private StringBuilder text = new StringBuilder();
```

```
107
108 public TextReader(File file)
109 {
110     this.file = file;
111 }
112
113 // Следующий метод выполняется в рабочем потоке,
114 // не затрагивая компоненты Swing
115
116 public StringBuilder doInBackground()
117     throws IOException, InterruptedException
118 {
119     int lineNumber = 0;
120     try (var in = new Scanner(
121         new FileInputStream(file),
122         StandardCharsets.UTF_8))
123     {
124         while (in.hasNextLine())
125         {
126             String line = in.nextLine();
127             lineNumber++;
128             text.append(line).append("\n");
129             var data = new ProgressData();
130             data.number = lineNumber;
131             data.line = line;
132             publish(data);
133             Thread.sleep(1); // только для проверки
134                 // отмены; а в конкретных программах
135                 // не требуется
136         }
137     }
138     return text;
139 }
140
141 // Следующие методы выполняются в потоке
142 // диспетчеризации событий
143
144 public void process(List<ProgressData> data)
145 {
146     if (isCancelled()) return;
147     var builder = new StringBuilder();
148     statusLine.setText("" +
149         data.get(data.size() - 1).number);
150     for (ProgressData d : data)
151         builder.append(d.line).append("\n");
152     textArea.append(builder.toString());
153 }
154
155 public void done()
156 {
157     try
158     {
159         StringBuilder result = get();
160         textArea.setText(result.toString());
161         statusLine.setText("Done");
162     }
```

```
163     catch (InterruptedException ex)
164     {
165     }
166     catch (CancellationException ex)
167     {
168         textArea.setText("");
169         statusLine.setText("Cancelled");
170     }
171     catch (ExecutionException ex)
172     {
173         statusLine.setText("" + ex.getCause());
174     }
175
176     cancelItem.setEnabled(false);
177     openItem.setEnabled(true);
178 }
179 };
180 }
```

javax.swing.SwingWorker<T, V> 6

- **abstract T doInBackground()**

Этот метод переопределяется, чтобы выполнять фоновую задачу и возвращать результат ее выполнения.

- **void process(List<V> data)**

Этот метод переопределяется, чтобы обрабатывать промежуточные данные о ходе выполнения задачи в потоке диспетчеризации событий.

- **void publish(V... data)**

Направляет промежуточные данные о ходе выполнения задачи в поток диспетчеризации событий. Вызывается из метода **doInBackground()**.

- **void execute()**

Планирует запуск данного рабочего задания в рабочем потоке исполнения.

- **SwingWorker.StateValue getState()**

Получает состояние рабочего потока исполнения, которое может принимать одно из значений: **PENDING**, **STARTED** или **DONE**.

12.8. Процессы

До сих пор было показано, как выполнять код Java в отдельных потоках одной и той же программы, но иногда в них требуется выполнять отдельные программы. И для этой цели служат классы **ProcessBuilder** и **Process**. В частности, класс **Process** выполняет команду в отдельном процессе операционной системы, давая возможность взаимодействовать со своими стандартными потоками ввода-вывода данных и ошибок. А класс **ProcessBuilder** позволяет сконфигурировать процесс в виде объекта типа **Process**.



НА ЗАМЕТКУ! Класс **ProcessBuilder** служит более удобной заменой вызовов **Runtime.exec()**.

12.8.1. Построение процесса

Построение процесса следует начинать с той команды, которую требуется выполнить. Для этого достаточно предоставить список типа `List<String>` или просто те символьные строки, которые составляют исполняемую команду:

```
var builder = new ProcessBuilder("gcc", "myapp.c");
```



ВНИМАНИЕ! В первой символьной строке должна быть указана исполняемая команда, а не встроенная командная оболочка. Например, чтобы выполнить команду `dir` в Windows, необходимо построить процесс с символьными строками `"cmd.exe"`, `"/C"` и `"dir"`.

У каждого процесса имеется свой *рабочий каталог*, который служит для разрешения имен относительных путей к каталогам. По умолчанию у процесса имеется такой же рабочий каталог, как и у виртуальной машины. Как правило, это каталог, из которого запускается утилита `java`. Его можно изменить с помощью метода `directory()` следующим образом:

```
builder = builder.directory(pathToFile());
```



НА ЗАМЕТКУ! Каждый метод для конфигурирования объекта типа `ProcessBuilder` возвращает ссылку на самого себя, что дает возможность составлять команды в цепочку. В конечном счете необходимо сделать следующий вызов:

```
Process p = new ProcessBuilder(команда).directory(файл)....start();
```

Далее требуется указать, что именно должно произойти со стандартными потоками ввода-вывода данных и ошибок в процессе. По умолчанию каждый из них является каналом, доступным с помощью следующих методов:

```
OutputStream processIn = p.getOutputStream();  
InputStream processOut = p.getInputStream();  
InputStream processErr = p.getErrorStream();
```

Следует, однако, иметь в виду, что поток ввода в процесс является потоком вывода из виртуальной машины Java! Все, что направляется в этот поток, появляется на входе в процесс. И напротив, все, что выводится из процесса, направляется в потоки вывода данных и ошибок. А для программиста они являются потоками ввода.

Потоки ввода-вывода данных и ошибок для нового процесса можно указать такими же, как и для виртуальной машины Java. Если пользователь выполняет виртуальную машину Java на консоли, то любые вводимые им данные направляются процессу, а выводимые процессом данные — на консоль. Чтобы распространить эту настройку на все три стандартных потока ввода-вывода, достаточно сделать следующий вызов:

```
builder.redirectIO()
```

Если же требуется лишь наследовать некоторые потоки ввода-вывода, для этого достаточно передать значение `ProcessBuilder.Redirect.INHERIT` методу `redirectInput()`, `redirectOutput()` или `redirectError()`, как демонстрируется в следующей строке кода:

```
builder.redirectOutput(ProcessBuilder.Redirect.INHERIT);
```

Потоки ввода-вывода можно переадресовать из процесса в файлы, предоставив объекты типа `File`, как показано ниже.

```
builder.redirectInput(inputFile)
    .redirectOutput(outputFile)
    .redirectError(errorFile)
```

Файлы для вывода данных и ошибок создаются или усекаются, когда начинается процесс. Чтобы присоединить существующие файлы, необходимо сделать следующий вызов:

```
builder.redirectOutput(ProcessBuilder.Redirect
    .appendTo(outputFile));
```

Зачастую оказывается полезно объединить потоки вывода данных и ошибок, чтобы наблюдать выводимые результаты и сообщения об ошибках именно в той последовательности, в какой они формируются в процессе. Чтобы активизировать объединение потоков вывода, достаточно сделать приведенный ниже вызов. Но если сделать это, то вызвать метод `redirectError()` для объекта типа `ProcessBuilder` или метод `getErrorStream()` для объекта типа `Process` больше не удастся.

```
builder.redirectErrorStream(true)
```

Возможно, потребуется также видоизменить переменные окружения процесса. Но для этого цепочный синтаксис построения процесса не годится. Ведь сначала необходимо получить окружение для строителя процесса, которое инициализируется с помощью переменных окружения процесса, выполняемого виртуальной машиной Java, а затем ввести или удалить отдельные элементы, как показано ниже.

```
Map<String, String> env = builder.environment();
env.put("LANG", "fr_FR");
env.remove("JAVA_HOME");
Process p = builder.start();
```

Если данные требуется направить по каналу с выхода одного процесса на вход другого процесса (как это делается, например, в командной оболочке с помощью операции `|`), то для этой цели в версии Java 9 предоставляется метод `startPipeline()`. В следующем примере демонстрируется перечисление однозначных расширений в дереве каталогов:

```
List<Process> processes = ProcessBuilder.startPipeline(
    List.of(
        new ProcessBuilder("find", "/opt/jdk-9"),
        new ProcessBuilder("grep", "-o", "\\.[^./*]*$"),
        new ProcessBuilder("sort"),
        new ProcessBuilder("uniq")
    ));
Process last = processes.get(processes.size() - 1);
var result = new String(last.getInputStream()
    .readAllBytes());
```

Безусловно, данную конкретную задачу можно решить более эффективно языковыми средствами Java, обойдя дерево каталогов вместо того, чтобы выполнять четыре отдельных процесса. О том, как это делается, речь пойдет в главе 2 второго тома настоящего издания.

12.8.2. Выполнение процесса

После того как строитель процесса будет сконфигурирован, следует вызвать его метод `start()`, чтобы начать процесс. Если потоки ввода-вывода данных и ошибок

сконфигурированы как каналы, то данные можно направлять в поток ввода и извлекать их из потоков вывода результатов и ошибок, как демонстрируется в следующем примере кода:

```
Process process = new ProcessBuilder("/bin/ls", "-l")
    .directory(Path.of("/tmp").toFile())
    .start();
try (var in = new Scanner(process.getInputStream())) {
    while (in.hasNextLine())
        System.out.println(in.nextLine());
}
```



ВНИМАНИЕ! Для потоков ввода-вывода в процессе выделяется ограниченное буферное пространство. Поэтому объем входных данных должен быть умеренным, а выходные данные следует извлекать оперативно. Если же имеется немало входных и выходных данных, то поставлять и потреблять их, возможно, придется в отдельных потоках исполнения.

Чтобы организовать ожидание завершения процесса, достаточно сделать следующий вызов:

```
int result = process.waitFor();
```

А если ожидать до бесконечности нежелательно, то можно воспользоваться приведенным ниже фрагментом кода.

```
long delay = . . . ;
if (process.waitFor(delay, TimeUnit.SECONDS)) {
    int result = process.exitValue();
    . . .
} else {
    process.destroyForcibly();
}
```

При первом вызове метода `waitFor()` возвращается значение выхода из процесса (нулевым значением выхода принято обозначать удачное завершение процесса, а ненулевым — неудачное завершение с кодом ошибки). При втором вызове метода `waitFor()` возвращается логическое значение `true`, если процесс не был блокирован по времени. В таком случае придется извлечь значение выхода из процесса, вызвав метод `exitValue()`.

Вместо того чтобы ожидать завершения процесса, достаточно предоставить ему выполняться дальше, периодически вызывая метод `isAlive()` для проверки его активности. Чтобы уничтожить процесс, следует вызвать метод `destroy()` или `destroyForcibly()`. Эти методы отличаются в зависимости от конкретной платформы. Так, в UNIX метод `destroy()` прерывает процесс сигналом `SIGTERM`, а метод `destroyForcibly()` — сигналом `SIGKILL`. (Если методу `destroy()` удастся завершить процесс нормально, то метод `supportsNormalTermination()` возвратит логическое значение `true`.)

И, наконец, можно получить асинхронное уведомление, когда процесс завершен. Так, в результате вызова `process.onExit()` выдается объект типа `CompletableFuture<Process>`, которым можно воспользоваться для планирования любого действия.

```
process.onExit().thenAccept(p ->
    System.out.println("Exit value: " + p.exitValue()));
```

12.8.3. Дескрипторы процессов

Чтобы получить сведения о процессе, в котором была запущена прикладная программа, или о любом другом процессе, выполняющемся в настоящий момент на компьютере, следует воспользоваться интерфейсом `ProcessHandle`, описывающим дескриптор процесса. Получить дескриптор процесса можно одним из следующих способов.

1. Если имеется объект `p` типа `Process`, то в результате вызова `p.toHandle()` получается объект типа `ProcessHandle`, представляющий дескриптор данного процесса.
2. Если имеется процесс типа `long` на уровне операционной системы, то в результате вызова `ProcessHandle.of(id)` получается дескриптор данного процесса.
3. В результате вызова `Process.current()` получается дескриптор того процесса, в котором выполняется данная виртуальная машина Java.
4. В результате вызова `ProcessHandle.allProcesses()` получают все процессы операционной системы, доступные текущему процессу.

Имея в своем распоряжении дескриптор процесса, можно получить его идентификатор, родительский, дочерний и прочие порожденные процессы, как показано ниже.

```
long pid = handle.pid();
Optional<ProcessHandle> parent = handle.parent();
Stream<ProcessHandle> children = handle.children();
Stream<ProcessHandle> descendants = handle.descendants();
```



НА ЗАМЕТКУ! Экземпляры типа `Stream<ProcessHandle>`, возвращаемые методами `allProcesses()`, `children()` и `descendants()`, служат лишь моментальными снимками в течение времени. Одни процессы, выполняемые в потоке данных, могут быть прерваны к тому моменту, когда к ним происходит обращение, а другие могут быть запущены вне этого потока.

Метод `info()` выдает объект типа `ProcessHandle.Info` с перечисленными ниже методами для получения сведений о процессе.

```
Optional<String[]> arguments()
Optional<String> command()
Optional<String> commandLine()
Optional<String> startInstant()
Optional<String> totalCpuDuration()
Optional<String> user()
```

Все эти методы возвращают значения типа `Optional`, поскольку вполне вероятно, что конкретная операционная система может и не сообщить сведения о процессе. Для текущего контроля или принудительного прерывания процесса в интерфейсе `ProcessHandle` имеются такие же методы `isAlive()`, `supportsNormalTermination()`, `destroy()`, `destroyForcibly()` и `onExit()`, как и в классе `Process`. Хотя в нем отсутствует эквивалент метода `waitFor()`.

java.lang.ProcessBuilder 5

- **ProcessBuilder(String... command)**
- **ProcessBuilder(List<String> command)**
Конструируют постройитель процесса по заданной команде и аргументам.
- **ProcessBuilder directory(File directory)**
Устанавливает рабочий каталог для процесса.
- **ProcessBuilder inheritIO() 9**
Использует стандартные для виртуальной машины Java потоки ввода-вывода данных и ошибок.
- **ProcessBuilder redirectErrorStream(boolean redirectErrorStream)**
Если параметр **redirectErrorStream** принимает логическое значение **true**, стандартный поток вывода ошибок из процесса объединяется со стандартным потоком вывода данных.
- **ProcessBuilder redirectInput(File file) 7**
- **ProcessBuilder redirectOutput(File file) 7**
- **ProcessBuilder redirectError(File file) 7**
Переадресовывают стандартные потоки ввода-вывода данных и ошибок из процесса в указанный файл.
- **ProcessBuilder redirectInput(ProcessBuilder.Redirect source) 7**
- **ProcessBuilder redirectOutput(ProcessBuilder.Redirect destination) 7**
- **ProcessBuilder redirectError(ProcessBuilder.Redirect destination) 7**
Переадресовывают стандартные потоки ввода-вывода данных и ошибок из процесса в одно из следующих мест назначения, определяемых параметром **destination**:
 - **Redirect.PIPE** — стандартное место назначения, доступное через объект типа **Process**
 - **Redirect.INHERIT** — поток вывода из виртуальной машины Java
 - **Redirect.DISCARD**
 - **Redirect.from(file)**
 - **Redirect.to(file)**
 - **Redirect.appendTo(file)**
- **Map<String,String> environment()**
Выдает изменяемое отображение для установки переменных окружения данного процесса.
- **Process start()**
Запускает процесс и выдает его объект типа **Process**.
- **static List<Process> startPipeline(List<ProcessBuilder> builders) 9**
Запускает конвейер процессов, соединяя стандартный выход из предыдущего процесса со стандартным входом следующего процесса.

java.lang.Process 1.0

- **abstract OutputStream getOutputStream()**
Получает поток для записи данных в поток ввода данных в текущем процессе.

java.lang.Process 1.0 (окончание)

- **abstract InputStream getInputStream()**
Получают поток для чтения данных из потока вывода данных или ошибок в текущем процессе.
- **abstract InputStream getErrorStream()**
Получают поток для чтения данных из потока вывода ошибок в текущем процессе.
- **abstract int waitFor()**
Ожидает завершения процесса и выдает значение выхода из процесса.
- **boolean waitFor(long timeout, TimeUnit unit) 8**
Ожидает завершения процесса, но не дольше, чем заданная выдержка времени. Возвращает логическое значение **true**, если процесс завершился.
- **abstract int exitValue()**
Возвращает значение выхода из процесса. Ненулевым значением принято обозначать выход из процесса.
- **boolean isAlive() 8**
Проверяет, активен ли все еще текущий процесс.
- **abstract void destroy()**
- **Process destroyForcibly() 8**
Завершают текущий процесс нормально или принудительно.
- **boolean supportsNormalTermination() 9**
Проверяет, может ли текущий процесс завершиться нормально или же его придется уничтожить принудительно.
- **ProcessHandle toHandle() 9**
Выдает объект типа **ProcessHandle**, описывающий текущий процесс.
- **CompletableFuture<Process> onExit() 9**
Выдает объект типа **CompletableFuture**, выполняемый после выхода из текущего процесса.

java.lang.ProcessHandle 9

- **static Optional<ProcessHandle> of(long pid)**
- **static Stream<ProcessHandle> allProcesses()**
- **static ProcessHandle current()**
Выдают дескрипторы с заданными идентификаторами всех процессов или же процесса виртуальной машины Java.
- **Stream<ProcessHandle> children()**
- **Stream<ProcessHandle> descendants()**
Выдают дескрипторы процессов, являющихся дочерними или порожденными данным процессом.
- **long pid()**
Выдает идентификатор текущего процесса.
- **ProcessHandle.Info info()**
Выдает подробные сведения о текущем процессе.

`java.lang.ProcessHandle.Info` 9

- `Optional<String[]> arguments()`
- `Optional<String> command()`
- `Optional<String> commandLine()`
- `Optional<Instant> startInstant()`
- `Optional<Instant> totalCpuDuration()`
- `Optional<String> user()`

Выдают требующиеся подробные сведения, если они доступны.

На этом описание языковых средств Java для организации параллельного, синхронного, асинхронного и длительного выполнения задач и процессов завершается. В заключительной главе речь пойдет о построении GUI прикладных программ на Java современными средствами библиотеки JavaFX.

Библиотека JavaFX

В этой главе...

- ▶ Отображение данных на сцене
- ▶ Обработка событий
- ▶ Компоновка
- ▶ Элементы управления пользовательского интерфейса
- ▶ Свойства и привязки
- ▶ Длительные задачи в обратных вызовах пользовательского интерфейса

В библиотеке JavaFX предоставляются инструментальные средства для построения графического пользовательского интерфейса (GUI) клиентских приложений, обладающего широкими функциональными возможностями. Эта библиотека входит в состав версий Java 7–10 и доступна через проект OpenJFX (<https://wiki.openjdk.java.net/display/OpenJFX/Main>) для последующих версий Java. История развития инструментальных средств для разработки GUI на Java была представлена в разделе 10.1, а в этой главе описаны основы разработки GUI средствами библиотеки JavaFX.

13.1. Отображение данных на сцене

В последующих разделах описывается основная архитектура JavaFX-приложений. По ходу изложения материала будет показано, как разрабатывать простые JavaFX-приложения, отображающие текст и геометрические формы.

13.1.1. Первое JavaFX-приложение

Начнем с простой программы, отображающей сообщение (рис. 13.1). Для этой цели используется *текстовый узел* и задаются координаты x и y расположения отображаемого сообщения приблизительно по центру. В частности, базовая точка первого символа в строке будет начинаться с позиции, расположенной на 75 пикселей

вправо и 100 пикселей вниз. (Далее в этой главе будет показан порядок точного расположения текста на экране.)

```
Text message = new Text(75, 100,  
    "Not a Hello World program");
```

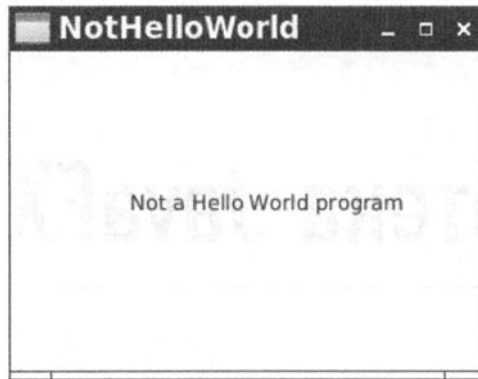


Рис. 13.1. Окно, в котором отображается информация

Все, что отображается на экране, считается в библиотеке JavaFX узлом типа `Node`, к которому относятся как геометрические формы, так и элементы пользовательского интерфейса. Узлы накапливаются в корневом узле типа `Parent` (т.е. узле типа `Node`, в который можно организовать другие узлы). Если не требуется располагать узлы автоматически, в качестве корневого узла можно воспользоваться объектом типа `Pane`.

Целесообразно также установить предпочтительные размеры панели, как показано ниже. В противном случае размеры панели автоматически задаются без полей, чтобы ровно вмещать геометрические формы.

```
Pane root = new Pane(message);  
root.setPrefSize(PREFERRED_WIDTH, PREFERRED_HEIGHT);
```

Затем из панели строится сцена следующим образом:

```
Scene scene = new Scene(root);
```

Далее сцена должна располагаться на *подмостках* — в окне, находящемся на рабочем столе (рис. 13.2). Подмостки передаются в виде параметра методу `start()`, который переопределяется в подклассе, производном от класса `Application`. Дополнительно можно задать заголовок окна. И, наконец, для отображения окна вызывается метод `show()`, как показано ниже.

```
public class NotHelloWorld extends Application  
{  
    public void start(Stage stage)  
    {  
        . . .  
        stage.setScene(scene);  
        stage.setTitle("NotHelloWorld");  
        stage.show();  
    }  
}
```

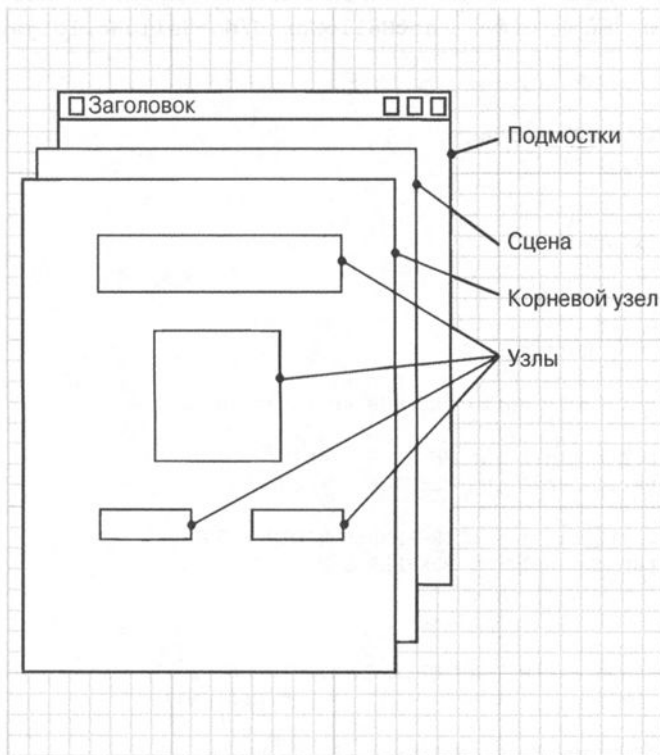


Рис. 13.2. Внутренняя структура подмоетков

Исходный код рассматриваемой здесь программы полностью приведен в листинге 13.1. А на рис. 13.3 представлена диаграмма, составленная на языке UML и отображающая иерархические взаимоотношения классов, применяемых в данной программе из библиотеки JavaFX.

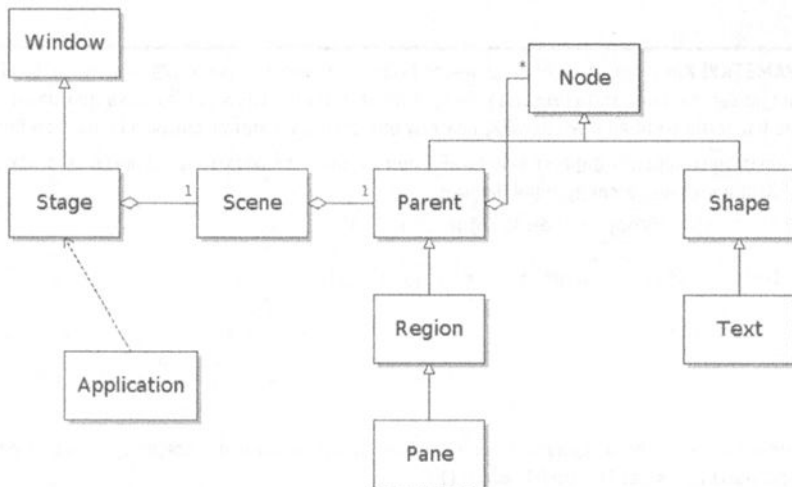


Рис. 13.3. Иерархические взаимоотношения базовых классов из библиотеки JavaFX

Листинг 13.1. Исходный код из файла `notHelloWorld/NotHelloWorld.java`

```
1 package notHelloWorld;
2
3 import javafx.application.*;
4 import javafx.scene.*;
5 import javafx.scene.layout.*;
6 import javafx.scene.text.*;
7 import javafx.stage.*;
8
9 /**
10  * @version 1.4 2017-12-23
11  * @author Cay Horstmann
12  */
13 public class NotHelloWorld extends Application
14 {
15     private static final int MESSAGE_X = 75;
16     private static final int MESSAGE_Y = 100;
17
18     private static final int PREFERRED_WIDTH = 300;
19     private static final int PREFERRED_HEIGHT = 200;
20
21     public void start(Stage stage)
22     {
23         Text message = new Text(MESSAGE_X, MESSAGE_Y,
24                                 "Not a Hello World program");
25
26         Pane root = new Pane(message);
27         root.setPrefSize(PREFERRED_WIDTH, PREFERRED_HEIGHT);
28
29         Scene scene = new Scene(root);
30         stage.setScene(scene);
31         stage.setTitle("NotHelloWorld");
32         stage.show();
33     }
34 }
```



НА ЗАМЕТКУ! Как следует из приведенного выше примера программы, для запуска JavaFX-приложения на выполнение метод `main()` не требуется. Утилите `java` для запуска программ на выполнение известно о библиотеке JavaFX, поэтому она вызывает метод `launch()` из этой библиотеки.

В предыдущих версиях библиотеки JavaFX приходилось включать метод `main()` в исходный код JavaFX-приложения в следующей форме:

```
public class MyApp extends Application
{
    public static void main(String[] args)
    {
        launch(args);
    }
    . . .
}
```

Тем не менее это можно сделать, если в цепочке инструментальных средств недостает объявления метода `public static void main()`.

javafx.stage.Stage

- **void setScene(Scene value)**
Устанавливает сцену для отображения на данных подмостках.
- **void setTitle(String value)**
Устанавливает заголовок, отображаемый в строке заголовка окна.
- **void show()**
Отображает окно.

javafx.scene.layout.Pane

- **Pane(Node... children)**
Конструирует панель, содержащую заданные порожденные узлы.

javafx.scene.layout.Region

- **void setPrefSize(double prefWidth, double prefHeight)**
Устанавливает предпочтительные размеры данной области по заданной ширине и высоте.

javafx.scene.text.Text

- **Text(double x, double y, String text)**
Конструирует текстовый узел типа **Text** по заданному положению и содержанию.

13.2.2. Рисование геометрических форм

В библиотеке JavaFX геометрические формы представлены подклассами, производными от класса **Shape**, который, в свою очередь, является производным от класса **Node**. Чтобы воспроизвести изображение, состоящее из прямоугольников, линий, окружностей и других геометрических форм, достаточно сконструировать сначала геометрические формы, а затем корневой узел, содержащий эти формы:

```
Rectangle rect = new Rectangle(leftX, topY, width, height);  
Line line = new Line(centerX, centerY, centerX + radius,  
                    centerY);  
Pane root = new Pane(rect, line);
```

Если узел требуется ввести впоследствии, для этого следует вызвать метод **getChildren()** из корневой панели, выдающий изменяемый список типа

`List<Node>`, как показано ниже. Вводя или удаляя узлы, можно обновлять узлы, порожденные панелью.

```
Circle circle = new Circle(centerX, centerY, radius);
root.getChildren().add(circle);
```



НА ЗАМЕТКУ! Пуристы объектно-ориентированного проектирования жалуются, что такие методы, как `getChildren()`, нарушают так называемый “закон Деметры”, поскольку они выдают изменяемую внутреннюю структуру объекта. Но в библиотеке JavaFX это весьма распространенная практика.



НА ЗАМЕТКУ! Окружности и эллипсы конструируются в библиотеке JavaFX, исходя из заданных центральных точек и радиусов. Это совсем иной и более удобный способ, чем в библиотеках AWT и Swing, где требуется указывать прямоугольник, ограничивающий конструируемую форму.



НА ЗАМЕТКУ! Чтобы нарисовать геометрические формы в библиотеке Swing или на платформе Android, операции рисования необходимо разместить в обратном вызове метода `paintComponent()` или `onDraw()`. В прикладном интерфейсе JavaFX API это делается намного проще. Для этого достаточно ввести на сцене те узлы, которые требуется воспроизвести. Если же переместить узлы, вся сцена будет перерисована автоматически.

Узлы типа `Line` (линия), `Path` (контур) и `Polygon` (многоугольник) по умолчанию рисуются черным цветом. Если же их требуется нарисовать другим цветом, достаточно вызвать метод `setStroke()` следующим образом:

```
radius.setStroke(Color.RED);
```

Другие геометрические формы, кроме линий, контуров и многоугольников, исходно заполняются черным цветом. Цвет заливки этих форм можно изменить, сделав следующий вызов:

```
rect.setFill(Color.YELLOW);
```

А если заполнять геометрическую форму цветом не требуется, следует выбрать сначала прозрачное заполнение, а затем установить цвет обводки для контура данной формы:

```
rect.setFill(Color.TRANSPARENT);
rect.setStroke(Color.BLACK);
```

Методы `setFill()` и `setStroke()` принимают параметр типа `Paint`. Класс `Color` является производным от класса `Paint`, как, впрочем, и классы, предназначенные для воспроизведения градиентов и растровых изображений, которые здесь не обсуждаются. Для наименований всех 147 цветов по стандарту CSS3 имеются соответствующие константы от `Color.ALICEBLUE` до `Color.YELLOWGREEN`.

В листинге 13.2 представлен исходный код примера программы, рисующей геометрические формы, приведенные на рис. 13.4.

Листинг 13.2. Исходный код из файла `draw/DrawTest.java`

```
1 package draw;
2
```

```
3 import javafx.application.*;
4 import javafx.scene.*;
5 import javafx.scene.layout.*;
6 import javafx.scene.paint.*;
7 import javafx.scene.shape.*;
8 import javafx.stage.*;
9
10 /**
11     @version 1.4 2017-12-23
12     @author Cay Horstmann
13 */
14 public class DrawTest extends Application
15 {
16     private static final int PREFERRED_WIDTH = 400;
17     private static final int PREFERRED_HEIGHT = 400;
18
19     public void start(Stage stage)
20     {
21         double leftX = 100;
22         double topY = 100;
23         double width = 200;
24         double height = 150;
25
26         Rectangle rect = new Rectangle(leftX, topY, width,
27                                     height);
28         rect.setFill(Color.TRANSPARENT);
29         rect.setStroke(Color.BLACK);
30         // эллипс, вписанный в прямоугольник
31         double centerX = leftX + width / 2;
32         double centerY = topY + height / 2;
33         Ellipse ellipse = new Ellipse(centerX, centerY,
34                                     width / 2, height / 2);
35         ellipse.setFill(Color.PEACHPUFF);
36         // диагональная линия
37         Line diagonal = new Line(leftX, topY, leftX + width,
38                                topY + height);
39         // окружность с таким же центром, как и у эллипса
40         double radius = 150;
41         Circle circle = new Circle(centerX, centerY,
42                                   radius);
43         circle.setFill(Color.TRANSPARENT);
44         circle.setStroke(Color.RED);
45         Pane root = new Pane(rect, ellipse, diagonal,
46                             circle);
47         root.setPrefSize(PREFERRED_WIDTH, PREFERRED_HEIGHT);
48         stage.setScene(new Scene(root));
49         stage.setTitle("DrawTest");
50         stage.show();
51     }
52 }
```

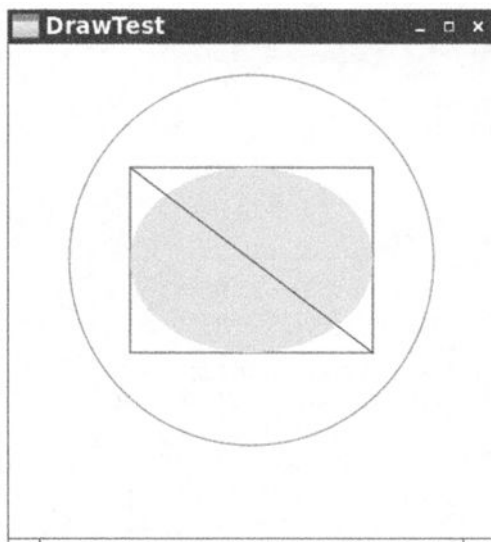


Рис. 13.4. Рисование геометрических форм

`javafx.scene.shape.Rectangle`

- `Rectangle(double x, double y, double width, double height)`
Конструирует прямоугольник по заданному верхнему левому углу, ширине и высоте.

`javafx.scene.shape.Circle`

- `Circle(double centerX, double centerY, double radius)`
Конструирует окружность по заданному центру и радиусу.

`javafx.scene.shape.Ellipse`

- `Ellipse(double centerX, double centerY, double radiusX, double radiusY)`
Конструирует эллипс по заданному центру и радиусам.

`javafx.scene.shape.Line`

- `Line(double startX, double startY, double endX, double endY)`
Конструирует линию по заданным начальной и конечной точкам.

```
class javafx.scene.layout.Pane
```

- `ObservableList<Node> getChildren()`
Выдает изменяемый список всех узлов, порожденных панелью.

```
javafx.scene.shape.Shape
```

- `void setStroke(Paint value)`
Задаёт раскраску для рисования границы данной геометрической формы или же самой формы, если она относится к типу `Line`, `Polyline` и `Path`.
- `void setFill(Paint value)`
Задаёт раскраску для рисования внутренней части геометрической формы.

13.2.3. Текст и изображения

Программа из листинга 13.1 выводила символьную строку шрифтом `System` стандартного размера. Но зачастую выводимый текст требуется воспроизвести другим шрифтом. Чтобы получить шрифт, следует сначала вызвать статический метод `Font.font()`, а затем метод `setFont()` для объекта типа `Text`, чтобы задать шрифт.

```
message.setFont(Font.font("Times New Roman", 36));
```

Этот фабричный метод из класса `Font` создает объект, представляющий шрифт с заданным наименованием семейства шрифтов и размером в пунктах. Полуужирное и наклонное начертание шрифта можно указать, сделав следующий вызов:

```
Font.font("Times New Roman", FontWeight.BOLD,  
        FontPosture.ITALIC, 36);
```

Постоянно доступны шрифты из следующих семейств:

```
System  
Serif  
SansSerif  
Monospaced
```

А в состав JDK входят три перечисленные ниже семейства шрифтов. Список всех доступных семейств шрифтов выдает статический метод `Font.getFamilies()`.

```
Lucida Bright  
Lucida Sans  
Lucida Sans Typewriter
```



ВНИМАНИЕ! Любое количество шрифтов может разделять общее наименование семейства шрифтов. Например, в семейство шрифтов `Lucida Bright` входят шрифты `Lucida Bright Regular`, `Lucida Bright Demibold` и `Lucida Bright Demibold Italic`. Но эти наименования находят ограниченное применение, поскольку в прикладном интерфейсе JavaFX API не допускается выбирать шрифт по наименованию.

Дело усложняется тем, что в перечислении `FontWeight` имеются значения `THIN`, `EXTRA_LIGHT`, `LIGHT`, `NORMAL`, `MEDIUM`, `SEMI_BOLD`, `BOLD`, `EXTRA_BOLD`, и чтобы выбрать из этого перечисления значение, соответствующее, например, насыщенности шрифта `Demibold`, приходится действовать методом проб и ошибок.

Положение текстового узла типа `Text` по оси *y* обозначает *базовую линию* текста (рис. 13.5). Чтобы выяснить протяженность текста, необходимо сделать сначала следующий вызов:

```
Bounds messageBounds = message.getBoundsInParent();
```

а затем вычислить подъем — расстояние от базовой линии до верхнего края буквы (например, *b* или *k*), а также спуск — расстояние от базовой линии до нижнего края буквы (например, *p* или *q*), как показано ниже.

```
double ascent = message.getY() - messageBounds.getMinY();
double descent = messageBounds.getMaxY() - message.getY();
double width = messageBounds.getWidth();
```



ВНИМАНИЕ! В классе `Node` имеются три метода для определения границ узла: `getLayoutBounds()`, `getBoundsInLocal()` и `getBoundsInParent()`. Но только в методе `getBoundsInParent()` во внимание принимается ширина обводки, эффекты и преобразования. Этим методом следует пользоваться всякий раз, когда требуется выяснить протяженность узла, когда он фактически воспроизводится.

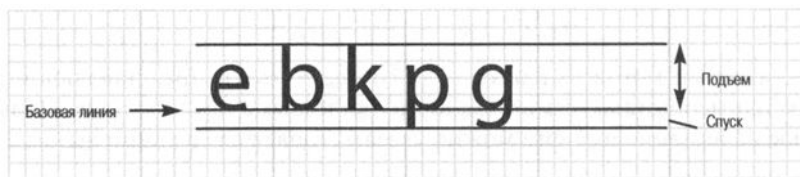


Рис. 13.5. Основные термины, применяемые при воспроизведении текста

В примере программы из листинга 13.3 демонстрируется порядок точного расположения текстового узла. Сначала текст располагается в начале координат, а также определяется его подъем, спуск и ширина. Затем текст центруется по горизонтали, а его базовая линия располагается в нужном положении с помощью метода `relocate()` из класса `Node`. Этот метод изменяет положение верхнего левого угла, а не базовой линии, поэтому необходимо откорректировать ее положение по подъему вдоль оси *y*.

Чтобы продемонстрировать точность вычислений, в данной программе рисуется ограничивающий прямоугольник и базовая линия (рис. 13.6). На экран выводится текст “Здравствуй, мир!” по-французски, чтобы наглядно показать, каким образом в нем воспроизводится буква с подстрочным элементом.

Прямо под текстом располагается изображение. С этой целью конструируется объект типа `ImageView`, исходя из того изображения, которое доступно по заданному пути или URL. В конструкторе данного объекта нельзя указать верхний левый угол, поэтому вызывается метод `relocate()`, чтобы переместить представление изображения.

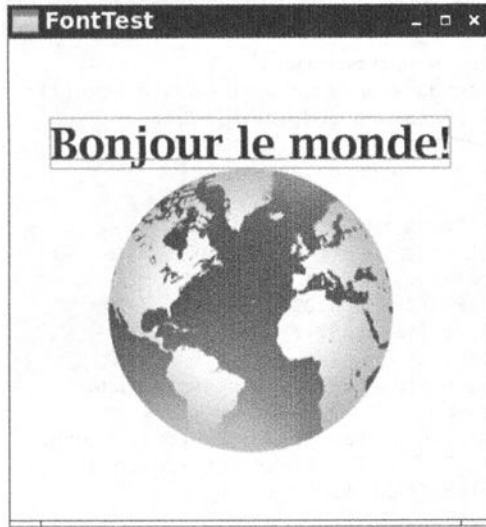


Рис. 13.6. Рисование ограничивающего прямоугольника и базовой линии

Листинг 13.3. Исходный код из файла **font/FontTest.java**

```
1 package font;
2
3 import javafx.application.*;
4 import javafx.geometry.*;
5 import javafx.scene.*;
6 import javafx.scene.image.*;
7 import javafx.scene.layout.*;
8 import javafx.scene.paint.*;
9 import javafx.scene.shape.*;
10 import javafx.scene.text.*;
11 import javafx.stage.*;
12
13 /**
14     @version 1.4 2017-12-23
15     @author Cay Horstmann
16 */
17 public class FontTest extends Application
18 {
19     private static final int PREFERRED_WIDTH = 400;
20     private static final int PREFERRED_HEIGHT = 400;
21
22     public void start(Stage stage)
23     {
24         // сконструировать текстовое сообщение в точке
25         // с координатами (0,0)
26         Text message = new Text("Bonjour le monde!");
27         Font f = Font.font("Lucida Bright",
28                           FontWeight.BOLD, 36);
```



```
29     message.setFont(f);
30
31     // получить размеры изображения
32     Bounds messageBounds = message.getBoundsInParent();
33     double ascent = -messageBounds.getMinY();
34     double descent = messageBounds.getMaxY();
35     double width = messageBounds.getWidth();
36
37     // отцентровать сообщение по горизонтали
38     double baseY = 100;
39     double topY = baseY - ascent;
40     double leftX = (PREFERRED_WIDTH - width) / 2;
41     message.relocate(leftX, topY);
42
43     // сконструировать ограничивающий прямоугольник
44     // и базовую линию
45     Rectangle rect = new Rectangle(leftX, topY, width,
46                                     ascent + descent);
47     rect.setFill(Color.TRANSPARENT);
48     rect.setStroke(Color.GRAY);
49     Line baseline = new Line(leftX, baseY,
50                             leftX + width, baseY);
51     baseline.setStroke(Color.GRAY);
52
53     // отцентровать изображение прямо под сообщением
54     ImageView image = new ImageView("font/world.png");
55     Bounds imageBounds = image.getBoundsInParent();
56     image.relocate((PREFERRED_WIDTH
57                     - imageBounds.getWidth()) / 2,
58                     baseY + descent);
59
60     Pane root = new Pane(message, rect,
61                           baseline, image);
62     root.setPrefSize(PREFERRED_WIDTH, PREFERRED_HEIGHT);
63     stage.setScene(new Scene(root));
64     stage.setTitle("FontTest");
65     stage.show();
66 }
67 }
```

javafx.scene.text.Font

- **static Font.font(double size)**
- **static Font.font(String family)**
- **static Font.font(String family, double size)**
- **static Font.font(String family, FontWeight weight, double size)**
- **static Font.font(String family, FontWeight weight, FontPosture posture, double size)**

Получают шрифт по заданному наименованию семейства (или System), насыщенности и положению (**FontWeight.NORMAL** и **FontPosture.REGULAR**), а также размеру в пунктах.

javafx.scene.text.Text

- **void setFont(Font value)**
Выделяет текст заданным шрифтом.
- **double getX()**
- **double getY()**
Получают положение точки базовой линии данного текстового узла по осям *x* и *y*.

javafx.scene.Node

- **Bounds getBoundsInParent()**
Получает границы данного текстового узла после применения любых обводок, обтравок и эффектов и преобразований.
- **void relocate(double x, double y)**
Изменяет положение данного текстового узла таким образом, чтобы верхний левый угол оказался в точке с заданными координатами *x* и *y*.

javafx.geometry.Bounds

- **double getMinX()**
- **double getMinY()**
- **double getMaxX()**
- **double getMaxY()**
Получают наименьшее или наибольшее значение координат *x* и *y* текущих границ.
- **double getWidth()**
- **double getHeight()**
Получают ширину и высоту текущих границ.

javafx.scene.image.ImageView

- **ImageView(String url)**
Конструирует изображение по заданному URL. В качестве параметра *url* должен быть указан допустимый параметр конструктора класса `java.net.URL` или путь к ресурсу. (Подробнее о ресурсах см. в главе 5.)

13.3. Обработка событий

События, наступающие в устройствах ввода (например, нажатия клавиш или щелчки кнопками мыши), постоянно контролируются в среде GUI и направляются

на обработку соответствующей программе. А в самой программе принимается решение, какому именно элементу пользовательского интерфейса следует передать событие на обработку. При этом низкоуровневые события надлежащим образом преобразуются в семантические. Так, если пользователь щелкнет на экранной кнопке, в библиотеке JavaFX организуется обработка последовательности событий, состоящих из нажатия и отпускания кнопки мыши над областью, занимаемой элементом управления экранной кнопкой. В итоге эта последовательность событий интерпретируется как “щелчок”. Чтобы организовать реакцию прикладной программы на подобные события, программист должен зарегистрировать *обработчик событий* с тем элементом управления пользовательского интерфейса, где наступило данное событие.

13.3.1. Реализация обработчиков событий

Если щелчок произведен на экранной кнопке, то обработчик событий должен реализовать интерфейс `EventHandler<ActionEvent>`. У функционального интерфейса `EventHandler<T>` имеется следующий единственный метод:

```
void handle(T event)
```

Чтобы указать действие над экранной кнопкой, достаточно воспользоваться лямбда-выражением следующим образом:

```
Button button = new Button("Click me!");
button.setOnAction(event ->
    System.out.println("I was clicked."));
```

В данном случае параметр `event` не находит применения в лямбда-выражении. У события действия не так уж и много интересных свойств. Самым полезным из них, вероятно, является источник события, т.е. тот элемент управления, где происходит действие. Но в данном случае известно, что источником события является экранная кнопка. Если же для нескольких событий служит один общий обработчик, то, вызвав метод `event.getSource()`, можно выяснить, какое из них наступило.

Обработка событий находит применение и в том случае, если интерес представляет объект события. Когда, например, пользователь закрывает окно, оно получает объект события с запросом на закрытие. Если же требуется отклонить такой запрос, то можно установить обработчик, в котором событие *потребляется*, как показано ниже.

```
stage.setOnCloseRequest(event ->
{
    if (не годится, чтобы закрыть окно) event.consume()
});
```



НА ЗАМЕТКУ! Если в узле требуется обрабатывать не одно событие, следует воспользоваться методом `addEventHandler()` следующим образом:

```
button.addEventHandler(javafx.event.ActionEvent.ACTION,
    event -> System.out.println("I was clicked"));
```

13.3.2. Реагирование на изменения свойств

Многие элементы управления из библиотеки JavaFX предоставляют разные механизмы для обработки событий. Рассмотрим в качестве пример ползунок, введенный на рис. 13.7. Когда ползунок настраивается, его значение изменяется. Но для реагирования на подобные изменения совсем не обязательно принимать

низкоуровневые события, отправляемые ползунком. Вместо этого можно обратиться к *свойству* ползунка, которое называется `value` в библиотеке JavaFX и предназначено для отправки событий при подобных изменениях. Более подробно свойства рассматриваются далее в разделе 13.6. Ниже показано, как организовать прием событий от свойства и откорректировать шрифт выводимого сообщения.

```
slider.valueProperty().addListener(property ->
message.setFont(Font.font(family, slider.getValue())));
```

Но в данном случае пользы от параметра `property` не особенно много. Ведь было бы проще получить обновленное значение от самого ползунка.



Рис. 13.7. Обработка событий действия и изменения свойства

Прием событий от свойств весьма распространен в библиотеке JavaFX. Так, если требуется изменить часть пользовательского интерфейса, когда пользователь вводит текст в текстовом поле, для этого достаточно ввести приемник событий от свойства `text`.

В примере программы из листинга 13.4 демонстрируется порядок обработки событий действия и изменения свойства. Так, если щелкнуть на кнопке `Random font` (Произвольный шрифт), в текстовом узле типа `Text` установится наименование произвольного шрифта, которым будет выделено выводимое сообщение (см. рис. 13.7). Размер шрифта корректируется при перемещении ползунка.

Когда окно закрывается, в приемнике событий проверяется, находится ли ползунок на исходной отметке 100%. И если это именно так, то данная программа отказывается закрыть окно, полагая, что пользователь еще (не?) опробовал ползунок. Экранная кнопка, ползунок и текстовый узел располагаются по вертикали на панели компоновки типа `VBox`. Подробнее об этом речь пойдет в разделе 13.4.

Листинг 13.4. Исходный код из файла `event/EventTest.java`

```
1 package event;
2
3 import java.util.*;
4
5 import javafx.application.*;
6 import javafx.scene.*;
7 import javafx.scene.control.*;
8 import javafx.scene.control.Alert.*;
9 import javafx.scene.layout.*;
10 import javafx.scene.text.*;
11 import javafx.stage.*;
12
13 /**
14  * @version 1.0 2017-12-23
15  * @author Cay Horstmann
```

```
16 */
17 public class EventTest extends Application
18 {
19     public void start(Stage stage)
20     {
21         Button button = new Button("Random font");
22         Text message = new Text("Times New Roman");
23         message.setFont(Font.font(
24             "Gloucester MT Extra Condensed", 100));
25         List<String> families = Font.getFamilies();
26         Random generator = new Random();
27         button.setOnAction(event ->
28             {
29                 String newFamily = families.get(
30                     generator.nextInt(families.size()));
31                 message.setText(newFamily);
32                 message.setFont(Font.font(
33                     newFamily, message.getFont().getSize()));
34             });
35
36         Slider slider = new Slider();
37         slider.setValue(100);
38         slider.valueProperty().addListener(property ->
39             {
40                 double newSize = slider.getValue();
41                 message.setFont(Font.font(
42                     message.getFont().getFamily(), newSize));
43             });
44
45         VBox root = new VBox(button, slider, message);
46         Scene scene = new Scene(root);
47
48         stage.setTitle("EventTest");
49         stage.setScene(scene);
50         stage.setOnCloseRequest(event ->
51             {
52                 if (slider.getValue() == 100)
53                 {
54                     event.consume(); // препятствует закрытию окна
55                     Alert alert = new Alert(AlertType.INFORMATION,
56                         "Move the slider before quitting.");
57                     alert.showAndWait();
58                 }
59             });
60         stage.show();
61     }
62 }
```

EventHandler<T extends Event>

- **void handle(T event)**

Этот метод следует переопределить, чтобы обработать заданное событие.

avafx.scene.control.ButtonBase

- **void setOnAction(EventHandler<ActionEvent> value)**
- Устанавливает приемник событий действия для данного элемента управления.

javafx.event.Event

- **void consume()**
Отмечает данное событие как потребляемое.
- **boolean isConsumed()**
Возвращает логическое значение **true**, если событие было отмечено как потребляемое.

java.util.EventObject 1.1

- **Object getSource()**
Получает объект, отвечающий за отправку данного события.

javafx.stage.Window

- **public final void setOnCloseRequest(EventHandler<WindowEvent> value)**
Устанавливает обработчик запросов на закрытие окна. Этот обработчик должен потребить событие, чтобы отклонить запрос на закрытие окна.

13.3.3. События от мыши и клавиатуры

События от мыши совсем не обязательно обрабатывать вручную, если требуется лишь предоставить пользователю возможность щелкнуть на экранной кнопке или переместить ползунок. Подобные события обрабатываются автоматически в различных элементах управления пользовательского интерфейса. Но если требуется предоставить пользователю возможность рисовать мышью, то придется перехватывать события, наступающие, когда пользователь перемещает мышь или щелкает ее кнопками.

В этом разделе рассматривается пример применения простого графического редактора, в котором пользователь может размещать, перемещать и удалять точки на холсте (рис. 13.8).

Когда пользователь щелкает кнопкой мыши, генерируются три события: от нажатия кнопки мыши, отпускания кнопки мыши и щелчка кнопкой мыши после нажатия и отпускания этой кнопки. В рассматриваемом здесь примере фиксируются лишь нажатия кнопки мыши, поскольку было бы желательно не задерживать визуальную реакцию до тех пор, пока не будет отпущена кнопка мыши.

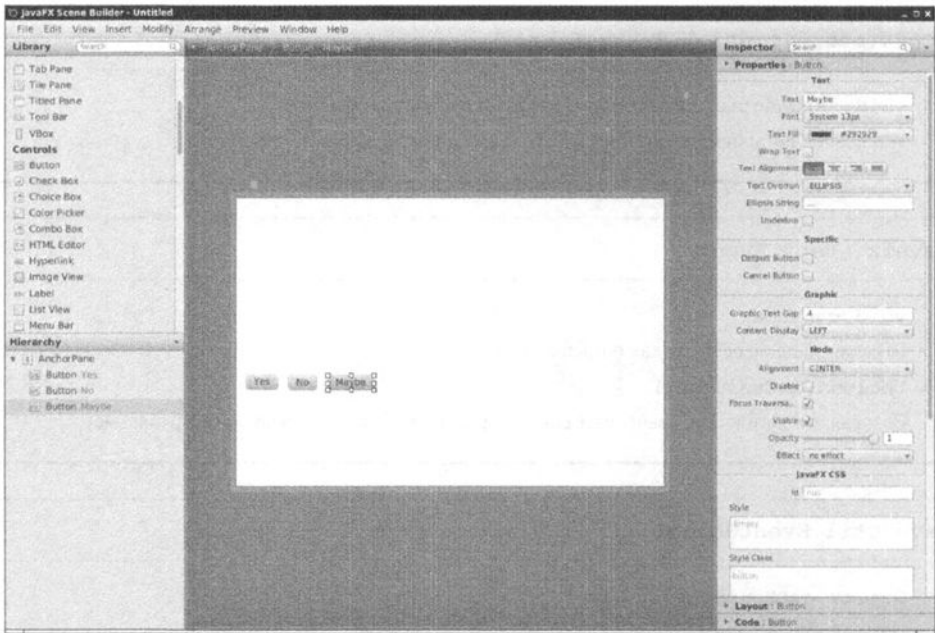


Рис. 13.8. Расстановка точек на холсте с помощью мыши и клавиатуры

Вызывая методы `getX()` и `getY()` для аргументов типа `MouseEvent`, можно получить координаты x и y курсора мыши, как показано ниже.

```
root.setOnMousePressed(event ->
{
    double x = event.getX();
    double y = event.getY();
    Circle dot = new Circle(x, y, RADIUS);
    root.getChildren().add(dot);
});
```

Чтобы различать одиночные, двойные и тройные (!) щелчки кнопкой мыши, следует вызвать метод `getClickCount()`. А для того чтобы выявить нажатую экранную кнопку, следует вызвать метод `getButton()`, как показано ниже.

```
if (event.getButton() == MouseButton.SECONDARY)
    . . . // щелчок правой кнопкой мыши
```

Некоторые разработчики GUI вынуждают пользователей употреблять модифицирующие клавиши в сочетании со щелчками кнопками мыши, как, например, `<Control+Shift+щелчок>`. Мы считаем такую практику предосудительной, но если вы не согласны, то воспользуйтесь одним из перечисленных ниже методов.

```
isShiftDown()
isControlDown()
isAltDown()
isMetaDown()
isShortcutDown()
```

Модифицирующей клавише `<Alt>` в Windows соответствует клавиша `<Option>` в Mac OS, а модифицирующей клавише `<Meta>` — клавиша `<Command>` в Mac OS. В




















методе `isShortcutDown()` проверяется избранная на данной платформе модифицирующая клавиша: `<Control>` в Linux и Windows или `<Meta>` в Mac OS.

При перемещении мыши генерируется устойчивый поток событий от перемещения мыши. Любой узел может запросить уведомлять его, когда мышь проходит над ним. В рассматриваемой здесь прикладной программе перехватываются события от перемещений мыши с целью изменить вид курсора (перекрестье), когда он оказывается над точкой:

```
dot.setOnMouseEntered(event ->
    scene.setCursor(Cursor.CROSSHAIR));
dot.setOnMouseExited(event ->
    scene.setCursor(Cursor.DEFAULT));
```

Доступные виды курсора мыши перечислены в табл. 13.1.

Таблица 13.1. Виды курсоров и соответствующие константы

Пиктограмма	Константа	Пиктограмма	Константа
	DEFAULT		N_RESIZE
	CROSSHAIR_CURSOR		NE_RESIZE
	HAND		E_RESIZE
	OPEN_HAND		SE_RESIZE
	CLOSED_HAND		S_RESIZE
	MOVE		SW_RESIZE
	TEXT		W_RESIZE
	WAIT		NW_RESIZE
	DISAPPEAR		H_RESIZE
NONE			V_RESIZE



НА ЗАМЕТКУ! Имеется также возможность определить собственный курсор, воспользовавшись классом `ImageCursor` следующим образом:

```
Image img = new Image("dynamite.gif");
Cursor dynamiteCursor = new ImageCursor(img, 10, 10);
```

В качестве второго и третьего параметра конструктора задается смещение “активной точки” курсора мыши, обозначающей его положение на экране.

Если пользователь нажимает кнопку мыши, когда она находится в движении, вместо событий перемещения мыши генерируются события ее перетаскивания. И когда курсор мыши оказывается над точкой в рассматриваемом здесь примере графического редактора, положение этой точки обновляется, чтобы отцентровать ее под курсором:

```
dot.setOnMouseDragged(event ->
{
    dot.setCenterX(event.getX());
    dot.setCenterY(event.getY());
});
```


Событие от мыши отправляется тому узлу, который находится под курсором. Но если наступает событие от клавиатуры, то какой узел следует уведомить о нем? Такое событие получает особый узел, имеющий *фокус ввода с клавиатуры*. Этот узел может запросить фокус ввода с клавиатуры с помощью метода `requestFocus()`.

Подобно событиям от щелчков кнопками мыши, события от клавиатуры наступают, когда нажимаются или отпускаются клавиши. Последовательное нажатие и отпускание нескольких клавиш приводит к вводу набранного на клавиатуре текста и генерированию события от нажатия клавиши. Так, если пользователь нажмет сначала клавишу <Shift>, а затем нажмет и отпустит клавишу <A>, то на клавиатуре будет набрана буква А. В частности, метод `getCharacter()` получает входные данные в виде символьной строки (на тот случай, если с клавиатуры вводятся символы эмодикона, требующие кодовых единиц в кодировке UTF-16, отдельные знаки ударения или какие-нибудь другие входные данные, не вписывающиеся в единое значение типа `char`).

```
source.setOnKeyTyped(event ->
{
    String input = event.getCharacter();
    . . .
});
```

Но если требуется обработать нажатия клавиш управления курсором или функциональных клавиш, то придется принять события от нажатия клавиш и вызвать метод `getCode()`. В итоге получается значение константы из перечисления `KeyCode`, где насчитывается более 200 констант для всех мыслимых клавиш на клавиатуре, в том числе `KeyCode.A`, `KeyCode.DELETE` и `KeyCode.EURO_SIGN`.

```
source.setOnKeyPressed(event ->
{
    KeyCode code = event.getCode();
    if (code == KeyCode.DELETE) . . . ;
});
```

В примере программы из листинга 13.5 демонстрируется порядок обработки событий от мыши и клавиатуры. Чтобы ввести новую точку на холсте, следует щелкнуть кнопкой мыши на пустом месте или перетащить существующую точку в новое положение. А для того чтобы удалить точку, следует дважды щелкнуть на ней. Точка с фокусом ввода с клавиатуры выделяется красным цветом. Ее можно удалить, нажав клавишу <Delete>, а также передвинуть с помощью клавиш перемещения курсора. Если же нажать одновременно клавишу <Shift>, точка будет передвигаться быстрее.

Следует также иметь в виду, что в каждой точке принимаются события наведения, отведения, нажатия и перетаскивания курсора мыши, а также нажатия клавиш на клавиатуре. Доставка этих событий в те узлы, которые должны на них реагировать, осуществляется средствами JavaFX. А событие нажатия кнопок мыши дополнительно обрабатывается на корневой панели, что дает возможность вводить новые точки на холсте.

Событие от мыши доставляется сначала порожденным, а затем родительским узлам. Чтобы исключить создание еще одной точки, обработчик событий нажатия кнопок мыши потребляет принятое событие.

С другой стороны, можно было бы реализовать единый обработчик событий нажатия кнопок мыши для корневой панели. Но тогда пришлось бы проверять, находится ли курсор мыши на данном узле. В библиотеке JavaFX принят следующий подход: как можно больше оперировать графом сцены, т.е. узлами и их отношениями “родитель–потомок”.

Листинг 13.5. Исходный код из файла `mouse/MouseTest.java`

```
1  package mouse;
2
3  import javafx.application.*;
4  import javafx.scene.*;
5  import javafx.scene.input.*;
6  import javafx.scene.layout.*;
7  import javafx.scene.paint.*;
8  import javafx.scene.shape.*;
9  import javafx.stage.*;
10
11 /**
12     @version 1.40 2017-12-27
13     @author Cay Horstmann
14 */
15 public class MouseTest extends Application
16 {
17     private static final int PREFERRED_WIDTH = 300;
18     private static final int PREFERRED_HEIGHT = 200;
19     private static final int RADIUS = 5;
20     private Scene scene;
21     private Pane root;
22     private Circle selected;
23
24     private Circle makeDot(double x, double y)
25     {
26         Circle dot = new Circle(x, y, RADIUS);
27         dot.setOnMouseEntered(event ->
28             scene.setCursor(Cursor.CROSSHAIR));
29         dot.setOnMouseExited(event ->
30             scene.setCursor(Cursor.DEFAULT));
31         dot.setOnMouseDragged(event ->
32             {
33                 dot.setCenterX(event.getX());
34                 dot.setCenterY(event.getY());
35             });
36         dot.setOnMousePressed(event ->
37             {
38                 if (event.getClickCount() > 1)
39                 {
40                     root.getChildren().remove(selected);
41                     select(null);
42                 }
43                 else
44                 {
45                     select(dot);
46                 }
47                 event.consume();
48             });
49
50         dot.setOnKeyPressed(event ->
51             {
52                 KeyCode code = event.getCode();
53                 int distance = event.isShiftDown() ? 10 : 1;
```

```
54         if (code == KeyCode.DELETE)
55             root.getChildren().remove(dot);
56         else if (code == KeyCode.UP)
57             dot.setCenterY(dot.getCenterY() - distance);
58         else if (code == KeyCode.DOWN)
59             dot.setCenterY(dot.getCenterY() + distance);
60         else if (code == KeyCode.LEFT)
61             dot.setCenterX(dot.getCenterX() - distance);
62         else if (code == KeyCode.RIGHT)
63             dot.setCenterX(dot.getCenterX() + distance);
64     });
65
66     return dot;
67 }
68
69 private void select(Circle dot)
70 {
71     if (selected == dot) return;
72     if (selected != null)
73         selected.setFill(Color.BLACK);
74     selected = dot;
75     if (selected != null)
76     {
77         selected.requestFocus();
78         selected.setFill(Color.RED);
79     }
80 }
81
82 public void start(Stage stage)
83 {
84     root = new Pane();
85     root.setOnMousePressed(event ->
86     {
87         double x = event.getX();
88         double y = event.getY();
89         Circle dot = makeDot(x, y);
90         root.getChildren().add(dot);
91         select(dot);
92     });
93     scene = new Scene(root);
94     root.setPrefSize(PREFERRED_WIDTH,
95                     PREFERRED_HEIGHT);
96     stage.setScene(scene);
97     stage.setTitle("MouseTest");
98     stage.show();
99 }
100 }
```

javafx.scene.Node

- **void setOnMousePressed(EventHandler<? super MouseEvent> value)**
- **void setOnMouseReleased(EventHandler<? super MouseEvent> value)**

javafx.scene.Node *(окончание)*

- **void setOnMouseClicked**(EventHandler<? super MouseEvent> value)
- **void setOnMouseEntered**(EventHandler<? super MouseEvent> value)
- **void setOnMouseExited**(EventHandler<? super MouseEvent> value)
- **void setOnMouseMoved**(EventHandler<? super MouseEvent> value)
- **void setOnMouseDragged**(EventHandler<? super MouseEvent> value)
Устанавливают обработки событий от мыши заданного типа.
- **void setOnKeyPressed**(EventHandler<? super KeyEvent> value)
- **void setOnKeyReleased**(EventHandler<? super KeyEvent> value)
- **void setOnKeyTyped**(EventHandler<? super KeyEvent> value)
Устанавливают обработки событий от клавиатуры заданного типа.
- **void requestFocus**()
Запрашивает установку фокуса ввода с клавиатуры в данном узле.

javafx.scene.input.MouseEvent

- **double getX**()
- **double getY**()
Выдают координаты x и y текущего положения курсора мыши в системе координат источника событий.
- **double getScreenX**() *
- **double getScreenY**()
Выдают координаты x и y текущего положения курсора мыши в экранной системе координат.
- **int getClickCount**()
Получает количество щелчков кнопками мыши (на небольшом участке и в течение короткого периода времени).
- **MouseButton getButton**()
Возвращает значения констант **PRIMARY**, **SECONDARY**, **MIDDLE** или **NONE** из перечисления **MouseButton**.
- **boolean isShiftDown**()
- **boolean isControlDown**()
- **boolean isAltDown**()
- **boolean isMetaDown**()
- **boolean isShortcutDown**()
Возвращают логическое значение **true**, если в течение данного события нажата модифицирующая клавиша <Shift>, <Control>, <Alt/Option>, <Windows/Command> или <Control/Command>.

javafx.scene.Scene

- **void setCursor(Cursor value)**

Устанавливает форму курсора мыши для данной сцены. Предопределенные формы курсора приведены в табл. 13.1.

javafx.scene.ImageCursor

- **ImageCursor(Image image, double hotspotX, double hotspotY)**

Конструирует курсор из заданного изображения. Курсор будет расположен с заданным смещением его активной точки.

javafx.scene.input.KeyEvent

- **String getCharacter()**

Получает набранные на клавиатуре входные данные, если это событие от нажатия клавиши.

- **KeyCode getCode()**

Получает код нажатой или отпущенной клавиши, если это событие от нажатия или отпускания клавиши.

- **boolean isShiftDown()**

- **boolean isControlDown()**

- **boolean isAltDown()**

- **boolean isMetaDown()**

- **boolean isShortcutDown()**

Возвращают логическое значение **true**, если в течение данного события нажата модифицирующая клавиша <Shift>, <Control>, <Alt/Option>, <Windows/Command> или <Control/Command>.

13.4. Компоновка

Если GUI содержит несколько элементов управления, их необходимо расположить на экране таким образом, чтобы это было функционально и привлекательно. Скомпоновать эти элементы можно, например, с помощью инструментального средства визуального конструирования GUI. Пользователем такого инструментального средства зачастую является художник-оформитель, который перетаскивает изображения элементов управления в представлении конструирования, располагая, изменяя размеры и конфигурируя их. Но такой подход может оказаться затруднительным, если изменятся размеры элементов, например, из-за длины интернационализированных версий прикладной программы.

С другой стороны, компоновка может быть достигнута программно написанием в методе установки кода, располагающего элементы управления на конкретных позициях в GUI. Именно так обычно поступают те, кто пользуется диспетчерами компоновки из библиотеки Swing.

Другой подход состоит в том, чтобы задать компоновку на декларативном языке. Например, веб-страницы компонуются средствами HTML и CSS. Аналогично язык XML применяется на платформе Android для задания различных компоновок.

Все три упомянутых выше подхода к компоновке GUI поддерживаются в библиотеке JavaFX. В частности, для визуального конструирования GUI в библиотеке JavaFX предоставляется инструментальное средство Scene Builder. Его можно загрузить по адресу <https://gluonhq.com/products/scene-builder>. А на рис. 13.9 показано, как оно выглядит на экране.

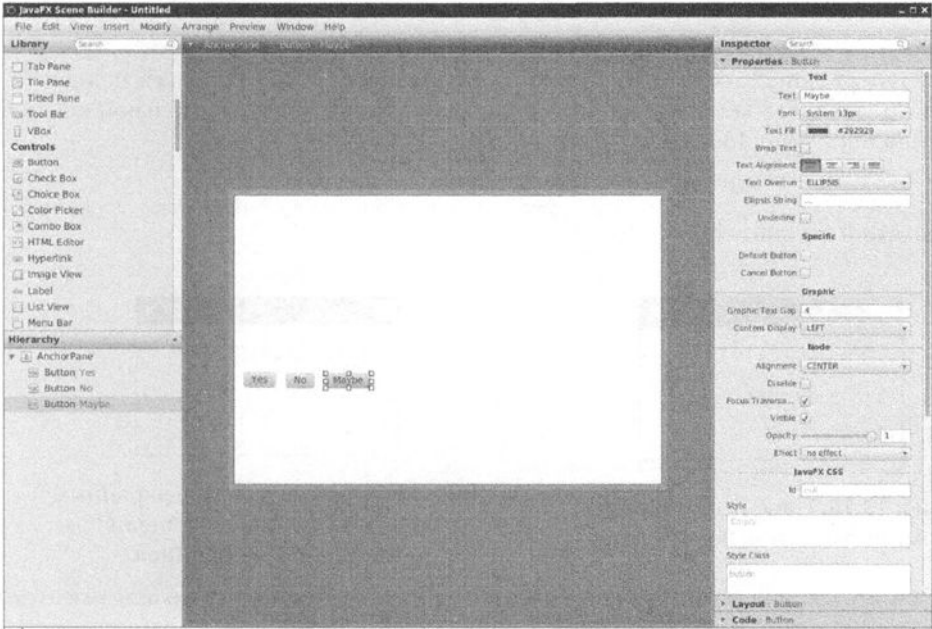


Рис. 13.9. Инструментальное средство Scene Builder из библиотеки JavaFX

Мы не станем обсуждать особенности инструментального средства Scene Builder. Если вы уясните понятия, рассматриваемые в последующих разделах, то воспользоваться этим инструментальным средством вам будет нетрудно. Поэтому далее речь пойдет о расположении элементов управления GUI с помощью панелей компоновки и языка разметки FXML.

13.4.1. Панели компоновки

В библиотеке JavaFX предоставляются специальные *панели* для компоновки элементов управления GUI. Они представляют собой родительские узлы, действующие по особым правилам компоновки. Например, панель граничной компоновки типа `BorderPane` состоит из верхней (Top), нижней (Bottom), левой (Left), правой (Right) и центральной (Center) областей. Ниже показано, как, например, расположить экранную кнопку в каждой из этих областей, а полученный результат приведен на рис. 13.10.

```
BorderPane pane = new BorderPane();
pane.setTop(new Button("Top"));
```

```
pane.setRight(new Button("Right"));
pane.setBottom(new Button("Bottom"));
pane.setLeft(new Button("Left"));
pane.setCenter(new Button("Center"));
stage.setScene(new Scene(pane));
```



НА ЗАМЕТКУ! При граничной компоновке с помощью компонента **BorderLayout** из библиотеки Swing экранные кнопки растягиваются таким образом, чтобы заполнить каждую область компоновки. А при граничной компоновке с помощью компонента **BorderPane** из библиотеки JavaFX экранные кнопки не растягиваются дальше своих естественных размеров.

А теперь допустим, что требуется расположить в нижней области не одну, а несколько экранных кнопок. Для этой цели служит панель горизонтальной компоновки типа **HBox** (рис. 13.11):

```
HBox buttons = new HBox(10, yesButton, noButton, maybeButton);
// промежуток 10 пикселей между элементами управления
pane.setBottom(buttons);
```

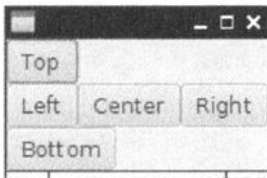


Рис. 13.10. Панель граничной компоновки типа **BorderPane**

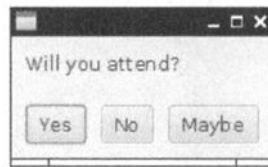


Рис. 13.11. Расположение экранных кнопок с помощью панели компоновки типа **HBox**

Разумеется, для расположения элементов управления по вертикали имеется панель компоновки типа **VBox**. В приведенном ниже примере кода демонстрируется, каким образом достигается такая компоновка.

```
VBox pane = new VBox(10, question, buttons);
pane.setPadding(new Insets(10));
```

Обратите внимание на установку свойства **padding** в приведенном выше примере кода. Без этого метка и экранные кнопки соприкасались бы с границей окна.



ВНИМАНИЕ! В библиотеке JavaFX размеры указываются в пикселях. В данном примере компоновочные блоки расставляются с промежутком и внутренним отступом 10 пикселей. Но в настоящее время, когда плотность сильно разнится, это уже не годится. В качестве выхода из этого затруднительного положения можно, например, вычислить размеры в корневых круглых шпациях, как это делается в CSS3 и показано ниже. (Корневая круглая шпация (**rem**) обозначает высоту стандартного шрифта, которым набран корневой элемент документа.)

```
final double rem = new Text("").getBoundsInParent().getHeight();
pane.setPadding(new Insets(0.8 * rem));
```

И это все, чего удастся добиться с помощью блоков горизонтальной и вертикальной компоновки. Подобно тому, как диспетчер компоновки типа **GridBagLayout** служит основой для всех остальных диспетчеров компоновки, в библиотеке JavaFX

аналогичной цели служит панель сеточной компоновки типа `GridPane`. Эта панель равнозначна HTML-таблице, где можно установить выравнивание всех ячеек по горизонтали и по вертикали. А при желании ячейки можно расставить рядами и столбцами. Примером сеточной компоновки служит диалоговое окно регистрации, приведенное на рис. 13.12.

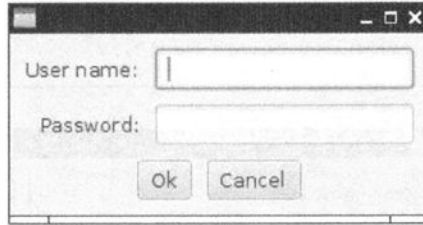


Рис. 13.12. Расположение элементов управления в диалоговом окне регистрации с помощью панели компоновки типа `GridPane`

Анализируя сеточную компоновку в данном примере, необходимо принимать во внимание следующее.

- Метки `User name` (Имя пользователя) и `Password` (Пароль) выровнены по правому краю.
- Экранные кнопки расположены на панели горизонтальной компоновки типа `HBox`, охватывающей два столбца.

Если вводится элемент, порожденный от панели компоновки типа `GridPane`, то индекс его столбцов и рядов указывается в таком же порядке, как и координаты x и y .

```
pane.add(usernameLabel, 0, 0);  
pane.add(username, 1, 0);  
pane.add(passwordLabel, 0, 1);  
pane.add(password, 1, 1);
```

Если порожденный элемент охватывает несколько столбцов и рядов, охватываемое пространство следует указать после заданных позиций. В качестве примера ниже показано, как ввести панель экранных кнопок, охватывающую два столбца и один ряд. Если же требуется, чтобы порожденный элемент охватывал все оставшиеся ряды и столбцы, следует указать константу `GridPane.REMAINING`.

```
pane.add(buttons, 0, 2, 2, 1);
```

Чтобы установить выравнивание порожденного элемента по горизонтали, следует вызвать метод `setHalignment()`, передав ему ссылку на порожденный элемент, а также константу `LEFT`, `CENTER` или `RIGHT` из перечисления `HPos`, как показано ниже. Аналогично для выравнивания порожденного элемента по вертикали следует вызвать метод `setValignment()`, передав ему константу `TOP`, `CENTER`, `BASELINE` или `BOTTOM` из перечисления `Vpos`.

```
GridPane.setHalignment(usernameLabel, HPos.RIGHT);
```



НА ЗАМЕТКУ! И хотя подобные вызовы статических методов выглядят довольно неуклюже в исходном коде на Java, они имеют свой смысл в языке разметки FXML, как поясняется в следующем разделе.



СОВЕТ. В целях отладки удобно отображать границы ячеек (рис. 13.13). Для этого достаточно сделать следующий вызов:

```
pane.setGridLinesVisible(true);
```

Если требуется отобразить границы отдельного порожденного элемента с целью выяснить, например, насколько он разросся, чтобы заполнить всю ячейку, достаточно установить его границы. И это проще всего сделать средствами CSS:

```
buttons.setStyle("-fx-border-color: red;");
```

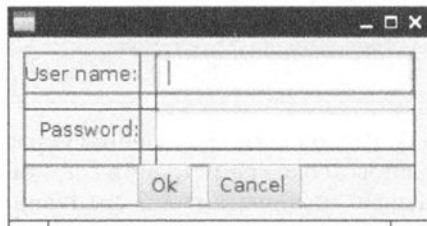


Рис. 13.13. Отображение видимых линий сетки при отладке на панели компоновки типа **GridPane**



ВНИМАНИЕ! Ни в коем случае не центруйте панель компоновки типа **HBox** с экранными кнопками, расположенными в сетке. Эта панель компоновки растянута до полного размера по горизонтали, и поэтому центровка никак не изменит ее положение. Вместо этого отцентрируйте содержимое панели компоновки типа **HBox**, как показано ниже.

```
buttons.setAlignment(Pos.CENTER);
```

Вполне возможно, что придется установить также некоторый внешний отступ вокруг рядов и столбцов и внутренний отступ во всей таблице:

```
pane.setHgap(0.8 * rem);
pane.setVgap(0.8 * rem);
pane.setPadding(new Insets(0.8 * rem));
```

Панелей компоновки, рассмотренных в этом разделе, должно быть достаточно для большинства примеров применения. В табл. 13.2 перечислены все виды панелей компоновки, предоставляемых в библиотеке JavaFX.

Таблица 13.2. Панели компоновки из библиотеки JavaFX

Класс панели	Описание
HBox, VBox	Выравнивают порожденные элементы по горизонтали и по вертикали
GridPane	Компонует порожденные элементы в табличную сетку аналогично компоненту GridBagLayout из библиотеки Swing
TilePane	Компонует порожденные элементы в табличную сетку, придавая им одинаковые размеры, аналогично компоненту GridBagLayout из библиотеки Swing
BorderPane	Предоставляет северную (North), восточную (East), западную (West) и центральную (Center) области для компоновки аналогично компоненту BorderLayout из библиотеки Swing

Класс панели	Описание
FlowPane	Располагает порожденные элементы поточными рядами, создавая новый ряд, если недостаточно места, аналогично компоненту FlowLayout из библиотеки Swing
AnchorPane	Располагает порожденные элементы на абсолютных позициях или относительно границ родительской панели. Этот вид компоновки выбирается по умолчанию в инструментальном средстве Scene Builder
StackPane	Располагает порожденные элементы стопкой. Это может быть удобно для декоративного оформления элементов управления, где экранная кнопка располагается, например, над окрашенным прямоугольником

В листинге 13.6 приведен весь исходный код программы для рассмотренной выше компоновки. Более подробно элементы управления GUI, употребляемые в данной программе, обсуждаются далее, в разделе 13.5.1.



COBET. В языках Scala, Kotlin и Groovy предоставляются привязки библиотеки JavaFX (<http://www.scalafx.org>, <http://tornadofx.io>, <http://groovyfx.org>) к удобному предметно-ориентированному языку для построения GUI, напоминающим синтаксис прежнего языка FX Script.

Листинг 13.6. Исходный код из файла `gridPane/GridPaneDemo.java`

```

1 package gridPane;
2
3 import javafx.application.*;
4 import javafx.geometry.*;
5 import javafx.scene.*;
6 import javafx.scene.control.*;
7 import javafx.scene.layout.*;
8 import javafx.scene.text.*;
9 import javafx.stage.*;
10
11 public class GridPaneDemo extends Application
12 {
13     public void start(Stage stage)
14     {
15         final double rem = new Text("").getLayoutBounds()
16                                     .getHeight();
17
18         GridPane pane = new GridPane();
19         // снять комментарии для отладки
20         // pane.setGridLinesVisible(true);
21
22         pane.setHgap(0.8 * rem);
23         pane.setVgap(0.8 * rem);
24         pane.setPadding(new Insets(0.8 * rem));
25         Label usernameLabel = new Label("User name:");
26         Label passwordLabel = new Label("Password:");
27         TextField username = new TextField();
28         PasswordField password = new PasswordField();
29
30         Button okButton = new Button("Ok");
31         Button cancelButton = new Button("Cancel");
32

```

```
33   HBox buttons = new HBox(0.8 * rem);
34   buttons.getChildren().addAll(okButton,
35                               cancelButton);
36   buttons.setAlignment(Pos.CENTER);
37   // снять комментарии для отладки
38   // buttons.setStyle("-fx-border-color: red;");
39
40   pane.add(usernameLabel, 0, 0);
41   pane.add(username, 1, 0);
42   pane.add(passwordLabel, 0, 1);
43   pane.add(password, 1, 1);
44   pane.add(buttons, 0, 2, 2, 1);
45
46   GridPane.setHalignment(usernameLabel, HPos.RIGHT);
47   GridPane.setHalignment(passwordLabel, HPos.RIGHT);
48   stage.setScene(new Scene(pane));
49   stage.show();
50 }
51 }
```

javafx.scene.layout.BorderPane

- **BorderPane()**

Конструирует пустую панель граничной компоновки.

- **void setTop(Node value)**
- **void setRight(Node value)**
- **void setBottom(Node value)**
- **void setLeft(Node value)**
- **void setCenter(Node value)**

Размещают узел в области данной панели граничной компоновки.

javafx.scene.layout.HBox

- **HBox(double spacing, Node... children)**

Конструирует горизонтальную панель с указанными порожденными элементами, разделяемыми количеством пикселей, определяемых параметром *spacing*.

- **void setAlignment(Pos pos)**

Устанавливает выравнивание для порожденных элементов. Вид выравнивания указывается с помощью одной из следующих констант, определяемых в перечислении **Pos**: **TOP_LEFT**, **TOP_CENTER**, **TOP_RIGHT**, **CENTER_LEFT**, **CENTER**, **CENTER_RIGHT**, **BASELINE_LEFT**, **BASELINE_CENTER**, **BASELINE_RIGHT**, **BOTTOM_LEFT**, **BOTTOM_CENTER**, **BOTTOM_RIGHT**.

javafx.scene.layout.VBox

- **VBox(double spacing, Node... children)**

Конструирует вертикальную панель с указанными порожденными элементами, разделяемыми количеством пикселей, определяемых параметром *spacing*.

class javafx.scene.layout.Region

- **void setPadding(Insets value)**

Устанавливает внутренний отступ вокруг содержимого данной области.

javafx.geometry.Insets

- **public Insets(double topRightBottomLeft)**
- **public Insets(double top, double right, double bottom, double left)**

Конструируют вставки с заданным количеством пикселей сверху, справа, снизу и слева.

javafx.scene.layout.GridPane

- **GridPane()**
Конструирует пустую панель сеточной компоновки.
- **void add(Node child, int columnIndex, int rowIndex)**
- **void add(Node child, int columnIndex, int rowIndex, int colspan, int rowspan)**

Вводят узел на указанной позиции. Второй метод определяет элемент управления, охватывающий несколько ячеек сетки. Чтобы охватить все остальные ряды или столбцы сетки, в качестве параметра **rowspan** этому методу следует передать константу **GridPane.REMAINING**.

- **static void setHalignment(Node child, HPos value)**
Устанавливает выравнивание заданного узла по горизонтали в ячейке своей сетки. Вид выравнивания указывается с помощью одной из следующих констант, определяемых в перечислении **HPos**: **LEFT**, **RIGHT** или **CENTER**.
- **static void setValignment(Node child, VPos value)**
Устанавливает выравнивание заданного узла по вертикали в ячейке своей сетки. Вид выравнивания указывается с помощью одной из следующих констант, определяемых в перечислении **VPos**: **TOP**, **CENTER**, **BASELINE** или **BOTTOM**.
- **void setHgap(double value)**
- **void setVgap(double value)**
Устанавливает промежуток между рядами или столбцами по горизонтали или по вертикали равным заданному количеству пикселей.

13.4.2. Язык FXML

Язык разметки, применяемый в библиотеке JavaFX для описания компоновок, называется FXML. Он обсуждается здесь потому, что его понятия представляют интерес не только в контексте JavaFX, а их реализация носит довольно общий характер.

Ниже приведен пример FXML-разметки для компоновки диалогового окна регистрации, упоминавшегося в предыдущем разделе.

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

```
<?import java.lang.*?>
```

```

<?import java.util.*?>
<?import javafx.geometry.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.text.*?>
<?import javafx.scene.layout.*?>

<GridPane hgap="10" vgap="10">
  <padding>
    <Insets top="10" right="10" bottom="10" left="10"/>
  </padding>
  <children>
    <Label text="User name:" GridPane.columnIndex="0"
      GridPane.rowIndex="0"
      GridPane.halignment="RIGHT"/>
    <Label text="Password:" GridPane.columnIndex="0"
      GridPane.rowIndex="1"
      GridPane.halignment="RIGHT"/>
    <TextField GridPane.columnIndex="1"
      GridPane.rowIndex="0"/>
    <PasswordField GridPane.columnIndex="1"
      GridPane.rowIndex="1"/>
    <HBox GridPane.columnIndex="0" GridPane.rowIndex="2"
      GridPane.columnSpan="2"
      alignment="CENTER" spacing="10">
      <children>
        <Button text="Ok"/>
        <Button text="Cancel"/>
      </children>
    </HBox>
  </children>
</GridPane>

```

Проанализируем вкратце приведенную выше FXML-разметку. Прежде всего обратите внимание на инструкции по обработке, заключенные в дескрипторы `<?import . . . ?>` и предназначенные для импорта пакетов Java. (Как правило, инструкции по обработке XML-документов служат “запасным выходом” для конкретных приложений.)

А теперь рассмотрим структуру документа. Прежде всего, вложение панели сеточной компоновки типа `GridPane`, меток и текстовых полей, панели горизонтальной компоновки типа `HBox` и ее порожденных экранных кнопок отражает то вложение, которое было реализовано непосредственно в коде Java и описано в предыдущем разделе. Большинство атрибутов разметки соответствуют операторам установки свойств. Например, приведенная ниже строка разметки означает конструирование панели компоновки типа `GridPane` и установку ее свойств `hgap` и `vgap`.

```
<GridPane hgap="10" vgap="10">
```

Если атрибут разметки начинается с имени класса и статического метода, тогда этот метод вызывается. Например, приведенная ниже строка разметки означает вызов методов `GridPane.setColumnIndex(thisTextField, 1)` и `GridPane.setRowIndex(thisTextField, 0)`.

```
<TextField GridPane.columnIndex="1" GridPane.rowIndex="0"/>
```

А если значение свойства оказывается слишком сложным, чтобы его можно было выразить в виде символьной строки, то вместо атрибутов употребляются вложенные

элементы разметки. Так, в приведенном ниже примере разметки свойство `padding` относится к типу `Insets`, а объект типа `Insets` конструируется с помощью порожденного элемента разметки `<Insets . . .>`, где указывается порядок установки свойств данного объекта.

```
<GridPane hgap="10" vgap="10">
  <padding>
    <Insets top="10" right="10" bottom="10" left="10"/>
  </padding>
  . . .
```

И, наконец, для списка свойств имеется особое правило. Например, `children` — это свойство из списка, и в следующем фрагменте разметки экранные кнопки вводятся в список, возвращаемый методом `getChildren()`:

```
<HBox . . .>
  <children>
    <Button text="Ok" />
    <Button text="Cancel" />
  </children>
</HBox>
```



НА ЗАМЕТКУ! Текстовые строки можно локализовать, используя ключи ресурсов, начинающиеся со знака `%` (например, с помощью разметки `<Button text="%ok">`). Но в таком случае необходимо предоставить комплект ресурсов, ставящий ключи ресурсов в соответствие локализованным значениям. Более подробно комплекты ресурсов рассматриваются в главе 7 второго тома настоящего издания.

FXML-файлы можно составлять вручную или же воспользоваться строителем GUI вроде `Scene Builder`. Имея в своем распоряжении такой файл, его можно загрузить, как показано ниже.

```
public void start(Stage stage)
{
    try
    {
        Parent root = FXMLLoader.load(getClass()
                                         .getResource("dialog.fxml"));
        stage.setScene(new Scene(root));
        stage.show();
    }
    catch (IOException e)
    {
        e.printStackTrace();
        System.exit(0);
    }
}
```

Безусловно, само по себе это не приносит особой пользы. И хотя пользовательский интерфейс отображается, тем не менее, предоставляемые пользователем значения недоступны прикладной программе. Чтобы установить связь между элементами управления пользовательского интерфейса и прикладной программой, можно, например, воспользоваться атрибутами `id`, как это обычно делается в сценариях JavaScript. Атрибут `id` можно внедрить в FXML-файл следующим образом:

```
<TextField id="username" GridPane.columnIndex="1"
           GridPane.rowIndex="0"/>
```

Но имеется более совершенный способ. Например, чтобы внедрить объекты элементов управления в класс *контроллера*, можно воспользоваться аннотацией `@FXML`. Класс контроллера должен реализовывать интерфейс `Initializable`, а в его методе `initialize()` привязываются обработчики событий. Контроллером может быть любой класс — даже само JavaFX-приложение.

Например, в следующем фрагменте определяется контроллер для упоминавшегося ранее диалогового окна регистрации:

```
public class LoginDialogController implements Initializable
{
    @FXML private TextField username;
    @FXML private PasswordField password;
    @FXML private Button okButton;
    @FXML private Button cancelButton;
    public void initialize(URL url, ResourceBundle rb)
    {
        okButton.setOnAction(event -> . . .);
        cancelButton.setOnAction(event ->
        {
            username.setText("");
            password.setText("");
        });
    }
}
```

Именами переменных экземпляра контроллера можно снабдить соответствующие элементы управления в FXML-файле, используя атрибут разметки `fx:id`, но не `id`, как показано ниже.

```
<TextField fx:id="username" GridPane.columnIndex="1"
           GridPane.rowIndex="0"/>
<PasswordField fx:id="password" GridPane.columnIndex="1"
               GridPane.rowIndex="1" />
<Button fx:id="okButton" text="Ok" />
```

Кроме того, в корневом элементе необходимо объявить класс контроллера, используя атрибут `fx:controller`, как демонстрируется в приведенном ниже примере. Обратите внимание на атрибут пространства имен `xmlns:fx`, предназначенный для внедрения пространства имен FXML.

```
<GridPane xmlns:fx=
           "http://javafx.com/fxml" hgap="10" vgap="10"
           fx:controller="LoginDialogController">
```

Когда FXML-файл загружается, конструируется граф сцены, и ссылки на именованные объекты элементов управления внедряются в аннотируемые поля объекта контроллера. А затем вызывается метод `initialize()`.



НА ЗАМЕТКУ! Если у контроллера отсутствует конструктор по умолчанию (например, потому, что он инициализируется по ссылке на услугу бизнес-логики), его можно установить программно, как показано ниже.

```
FXMLLoader loader =
    new FXMLLoader(getClass().getResource(. . .));
loader.setController(new Controller(service));
Parent root = loader.load();
```



ВНИМАНИЕ! Для программной установки контроллера рекомендуется использовать исходный код из предыдущего примечания. И хотя приведенный ниже фрагмент кода будет скомпилирован, в нем, тем не менее, вызывается статический метод `FXMLLoader.load()`, игнорируя сконструированный загрузчик.

```
FXMLLoader loader = new FXMLLoader();
loader.setController(new Controller(service));
Parent root = loader.load(getClass().getResource(. . .));
// ОШИБКА - вызывается статический метод!
```

Имеется даже возможность произвести большую часть инициализации в FXML-файле, определив простые привязки и установив аннотированные методы контроллера в виде приемников событий. Кроме того, в FXML-файл можно внедрить сценарии, написанные на JavaScript или других языках сценариев. Соответствующий синтаксис описывается в документации, оперативно доступной по адресу https://docs.oracle.com/javase/9/docs/api/javafx/fxml/doc-files/introduction_to_fxml.html. Но мы не будем останавливаться на этих возможностях, а лишь заметим, что визуальное конструирование, по-видимому, лучше отделить от поведения программы, чтобы разработчик пользовательского интерфейса оформил его, а программист реализовал соответствующее поведение. В листинге 13.7 демонстрируется применение аннотации `@FXML` в прикладном коде, реализующем диалоговое окно регистрации, а в листинге 13.8 — разметка соответствующего FXML-файла.

Листинг 13.7. Исходный код из файла `fxml/FXMLDemo.java`

```
1 package fxml;
2
3 import java.io.*;
4 import java.net.*;
5 import java.util.*;
6
7 import javafx.application.*;
8 import javafx.fxml.*;
9 import javafx.scene.*;
10 import javafx.scene.control.*;
11 import javafx.scene.control.Alert.*;
12 import javafx.stage.*;
13
14 /**
15     @version 1.0 2017-12-29
16     @author Cay Horstmann
17 */
18 public class FXMLDemo extends Application
19     implements Initializable
20 {
21     @FXML private TextField username;
22     @FXML private PasswordField password;
23     @FXML private Button okButton;
24     @FXML private Button cancelButton;
25
26     public void initialize(URL url, ResourceBundle rb)
27     {
28         okButton.setOnAction(event ->
```



```

29     {
30         Alert alert = new Alert(AlertType.INFORMATION,
31                                 "Verifying " + username.getText()
32                                 + ":" + password.getText());
33         alert.showAndWait();
34     });
35     cancelButton.setOnAction(event ->
36     {
37         username.setText("");
38         password.setText("");
39     });
40 }
41
42 public void start(Stage stage)
43 {
44     try
45     {
46         Parent root = FXMLLoader.load(
47             getClass().getResource("dialog.fxml"));
48         stage.setScene(new Scene(root));
49         stage.setTitle("FXMLDemo");
50         stage.show();
51     }
52     catch (IOException e)
53     {
54         e.printStackTrace();
55     }
56 }
57 }

```

Листинг 13.8. Код разметки из файла `fxml/dialog.fxml`

```

1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <?import java.lang.*?>
4  <?import java.util.*?>
5  <?import javafx.geometry.*?>
6  <?import javafx.scene.control.*?>
7  <?import javafx.scene.text.*?>
8  <?import javafx.scene.layout.*?>
9
10 <GridPane xmlns:fx="http://javafx.com/fxml"
11           hgap="10" vgap="10"
12           fx:controller="fxml.FXMLDemo">
13     <padding>
14         <Insets top="10" right="10" bottom="10" left="10"/>
15     </padding>
16     <children>
17         <Label text="User name:" GridPane.columnIndex="0"
18               GridPane.rowIndex="0"
19               GridPane.halignment="RIGHT"/>
20         <Label text="Password:" GridPane.columnIndex="0"
21               GridPane.rowIndex="1"
22               GridPane.halignment="RIGHT"/>
23         <TextField fx:id="username"
24                   GridPane.columnIndex="1"

```

```

25         GridPane.rowIndex="0"/>
26     <PasswordField fx:id="password"
27         GridPane.columnIndex="1"
28         GridPane.rowIndex="1"/>
29     <HBox GridPane.columnIndex="0"
30         GridPane.rowIndex="2"
31         GridPane.columnSpan="2"
32         alignment="CENTER" spacing="10">
33         <children>
34             <Button fx:id="okButton" text="Ok"/>
35             <Button fx:id="cancelButton" text="Cancel"/>
36         </children>
37     </HBox>
38 </children>
39 </GridPane>

```

javafx.fxml.FXMLLoader

- **static <T> T load(URL location)**
- **static <T> T load(URL location, ResourceBundle resources)**
Возвращают объект, описываемый в FXML-документе, находящемся в месте, определяемом параметром *location*. Во втором методе с помощью параметра *resources* указывается комплект ресурсов для разрешения ключей с префиксом %.
- **FXMLLoader(URL location)**
- **FXMLLoader(URL location, ResourceBundle resources)**
Конструируют загрузчик, выполняющий загрузку объекта из FXML-документа, находящегося в месте, определяемом параметром *location*. Во втором методе с помощью параметра *resources* указывается комплект ресурсов для разрешения ключей с префиксом %.
- **void setController(Object controller)**
Устанавливает контроллер для корневого элемента. Этот метод следует вызывать перед загрузкой FXML-документа.
- **<T> T load()**
Возвращает объект, описываемый в загружаемом FXML-документе.

javafx.fxml.Initializable

- **void initialize(URL location, ResourceBundle resources)**
- Этот метод вызывается, как только с помощью конструктора класса **FXMLLoader** будет построен корневой элемент и его контроллер. Местоположение FXML-документа и ресурсы такие же, как и у загрузчика.

13.4.3. Стилиевые таблицы CSS

В библиотеке JavaFX предоставляется возможность изменить внешний вид пользовательского интерфейса с помощью стилиевых таблиц CSS, и зачастую это оказывается удобнее сделать, чем предоставлять атрибуты FXML-разметки или вызывать методы Java.

Стилевую таблицу CSS можно загрузить программно и применить ее в графе сцен следующим образом:

```
Scene scene = new Scene(pane);
scene.getStylesheets().add("scene.css");
```

В стиливой таблице можно обращаться к любым элементам управления, имеющим свой идентификатор. В качестве примера покажем, как определить внешний вид панели сеточной компоновки типа `GridPane`. Сначала в прикладном коде задается идентификатор данной панели.

```
GridPane pane = new GridPane();
pane.setId("pane");
```

Вместо того чтобы устанавливать любые внутренние отступы или промежутки между элементами управления, это можно сделать в стиливой таблице CSS:

```
#pane {
    -fx-padding: 0.5em;
    -fx-hgap: 0.5em;
    -fx-vgap: 0.5em;
    fx-background-image: url("metal.jpg")
}
```

К сожалению, для стиливого оформления GUI нельзя воспользоваться общеизвестными атрибутами CSS, но необходимо знать характерные для библиотеки JavaFX атрибуты с префиксом **-fx-**. Имена атрибутов образуются из имен свойств, набранных строчными буквами, а также дефисов вместо обычного смешанного написания прописными и строчными буквами. Например, имя свойства `textAlignment` преобразуется в имя атрибута `-fx-text-alignment`. Перечень всех атрибутов CSS, поддерживаемых в библиотеке JavaFX, можно найти по адресу <https://docs.oracle.com/javase/9/docs/api/javafx/scene/doc-files/cssref.html>.

Пользоваться стиливыми таблицами CSS удобнее, чем загромождать прикладной код излишними деталями компоновки GUI. Безусловно, пользоваться стиливыми таблицами CSS можно как во благо, так и во зло. В связи с этим рекомендуется воздерживаться от искушения употреблять ничем не оправданные текстуры заднего плана в диалоговых окнах регистрации, как, например, показано на рис. 13.14.

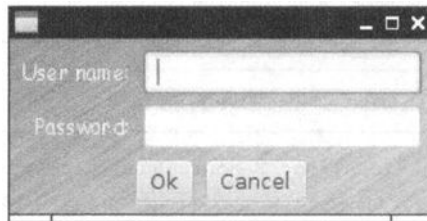


Рис. 13.14. Неудачный пример стиливого оформления пользовательского интерфейса средствами CSS

Вместо стиливого оформления элементов управления GUI по отдельным идентификаторам можно воспользоваться классами стилей. Для этого необходимо сначала ввести класс стиля в объект узла:

```
HBox buttons = new HBox();
buttons.getStyleClass().add("buttonrow");
```

а затем оформить его стилем, используя обозначение данного класса CSS:

```
.buttonrow {  
    -fx-spacing: 0.5em;  
}
```

Каждому классу элемента управления или геометрической формы в JavaFX соответствует класс CSS, имя которого составляется прописными буквами из имени класса Java. Например, у всех узлов типа `Label` имеется класс стиля `label`. В качестве примера ниже показано, каким образом можно изменить на `Comic Sans` шрифт всех меток, хотя делать этого не рекомендуется.

```
.label {  
    -fx-font-family: "Comic Sans MS";  
}
```

Стилевые таблицы можно также применять и в компоновках, размеченных на языке FXML. Например, стилевая таблица присоединяется к корневой панели следующим образом:

```
<GridPane id="pane" styleClass="scene.css">
```

Атрибуты `id` или `styleClass` вводятся в код FXML-разметки так, как показано ниже.

```
<HBox styleClass="buttonrow">
```

И тогда большую часть стилового оформления можно задать в таблице CSS, используя FXML-разметку только для компоновки. К сожалению, стилевое оформление нельзя полностью исключить из FXML-разметки. Например, в настоящее время нельзя никоим образом задать выравнивание ячеек сетки в стилевой таблице CSS.



НА ЗАМЕТКУ! Стили CSS можно применять и программно, как в следующем примере кода:

```
buttons.setStyle("-fx-border-color: red;");
```

Это может быть удобно для отладки, но, в общем, лучше все же пользоваться внешними таблицами стилей.

В примере программы из листинга 13.9 демонстрируется применение таблиц стилей. В частности, одна таблица стилей (из листинга 13.10) загружается непосредственно в данную программу, а другая (из листинга 13.11) загружается непосредственно из FXML-файла, приведенного в листинге 13.12.

Листинг 13.9. Исходный код из файла `css/CSSDemo.java`

```
1 package css;  
2  
3 import java.io.*;  
4 import javafx.application.*;  
5 import javafx.fxml.*;  
6 import javafx.scene.*;  
7 import javafx.scene.control.*;  
8 import javafx.stage.*;  
9  
10 public class CSSDemo extends Application  
11 {  
12     public void start(Stage stage)
```

```
13 {
14     try
15     {
16         Parent root = FXMLLoader.load(getClass()
17             .getResource("dialog.fxml"));
18         root.lookup("#username").getStyleClass()
19             .add("highlight");
20         Scene scene = new Scene(root);
21         scene.getStylesheets().add("css/scenel.css");
22         stage.setScene(scene);
23         stage.setTitle("CSSDemo");
24         stage.show();
25     }
26     catch (IOException ex)
27     {
28         ex.printStackTrace();
29         Platform.exit();
30     }
31 }
32 }
```

Листинг 13.10. Код стилевого оформления из файла `css/scenel.css`

```
1 .label {
2     -fx-text-fill: white;
3     -fx-font-family: "Comic Sans MS";
4 }
5
6 #pane {
7     -fx-padding: 0.5em;
8     -fx-hgap: 0.5em;
9     -fx-vgap: 0.5em;
10    -fx-background-image: url("metal.jpg");
11 }
12
13 .highlight:focused {
14     -fx-border-color: yellow;
15 }
```

Листинг 13.11. Код стилевого оформления из файла `css/scene2.css`

```
1 .buttonrow {
2     -fx-spacing: 0.5em;
3     -fx-alignment: center;
4 }
```

Листинг 13.12. Код разметки из файла `css/dialog.fxml`

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <?import java.lang.*?>
4 <?import java.util.*?>
5 <?import javafx.geometry.*?>
```

```

6  <?import javafx.scene.control.*?>
7  <?import javafx.scene.text.*?>
8  <?import javafx.scene.layout.*?>
9
10 <GridPane id="pane" xmlns:fx="http://javafx.com/fxml"
11         stylesheets="css/scene2.css">
12     <children>
13         <Label text="User name:" GridPane.columnIndex="0"
14             GridPane.rowIndex="0"
15             GridPane.halignment="RIGHT"/>
16         <Label text="Password: " GridPane.columnIndex="0"
17             GridPane.rowIndex="1"
18             GridPane.halignment="RIGHT"/>
19         <TextField id="username" GridPane.columnIndex="1"
20             GridPane.rowIndex="0"/>
21         <PasswordField GridPane.columnIndex="1"
22             GridPane.rowIndex="1"/>
23         <HBox styleClass="buttonrow"
24             GridPane.columnIndex="0"
25             GridPane.rowIndex="2"
26             GridPane.columnSpan="2">
27             <children>
28                 <Button text="Ok"/>
29                 <Button text="Cancel"/>
30             </children>
31         </HBox>
32     </children>
33 </GridPane>

```

javafx.scene.Scene

- **ObservableList<String> getStylesheets()**

Возвращает список символьных строк с URL для таблиц CSS стилового оформления данной сцены.

javafx.scene.Node

- **void setId(String value)**

Задаёт идентификатор данного узла. Этим идентификатором можно пользоваться в стиливых таблицах CSS.

- **void setStyle(String value)**

Задаёт стиль CSS оформления данного узла.

javafx.css.Styleable

- **ObservableList<String> getStyleClass()**

Возвращает список имен классов CSS для стилового оформления данного узла.

13.5. Элементы управления пользовательского интерфейса

В последующих разделах описываются наиболее употребительные элементы управления пользовательского интерфейса из библиотеки JavaFX: элементы управления вводом текста, флажки, кнопки-переключатели, комбинированные списки, меню и простые диалоговые окна.

13.5.1. Элементы управления вводом текста

Итак, начнем с элементов управления, дающих пользователю возможность вводить и редактировать текст. В частности, компонент `TextField`, реализующий текстовое поле, может принять лишь одну текстовую строку, тогда как компонент `TextArea` — несколько текстовых строк. Как показано на рис. 13.15, эти компоненты являются подклассами, производными от класса `TextInputControl`, который, в свою очередь, является производным от класса `Control`, от которого наследуют классы всех элементов управления пользовательского интерфейса в библиотеке JavaFX. А компонент `PasswordField` является подклассом, производным от класса `TextField`, и принимает одну текстовую строку, не отображая ее содержимое.

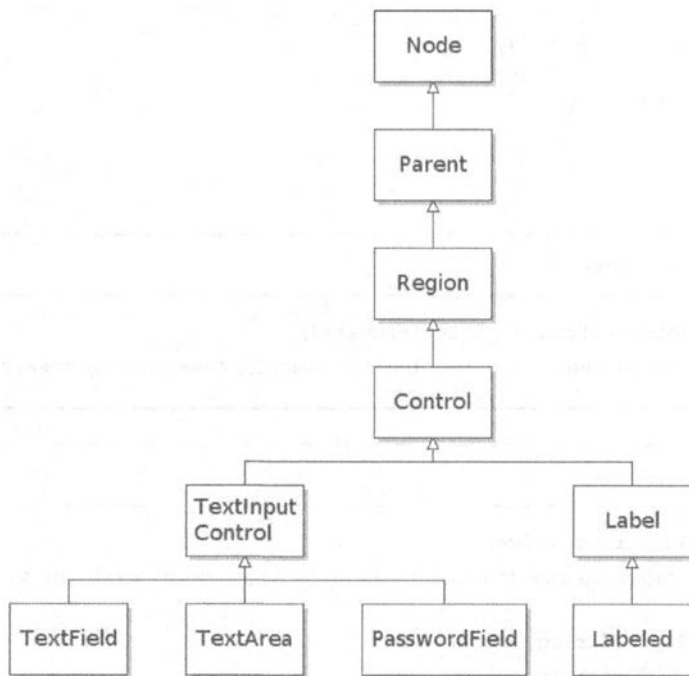


Рис. 13.15. Иерархия наследования элементов управления вводом текста в библиотеке JavaFX

В классе `TextInputControl` имеются методы `getText()`, `setText()` и `appendText()` для получения, установки или присоединения текста. Вызвав метод `setPrefColumnCount()` из класса `TextField`, можно дать указание на предпочтительный размер текстового поля. Предполагаемая ширина столбца должна соответствовать ширине одного символа того шрифта, которым набран текст. Впрочем,

пользователь может вводить и более длинные текстовые строки, но если длина вводимого текста превышает ширину поля, то такой текст будет прокручиваться. И для указания предпочтительного размера текстовой области предусмотрен метод `setPrefRowCount()`.



НА ЗАМЕТКУ! Полосы прокрутки появляются автоматически, если вводится больше текста, чем может отобразиться в текстовой области. Они снова исчезают, если текст удаляется, а оставшаяся его часть полностью помещается в текстовой области.

Чтобы активизировать режим автоматического переноса длинных строк в текстовой области, достаточно вызвать приведенный ниже метод, и тогда горизонтальные полосы прокрутки не будут употребляться в текстовой области.

```
textArea.setWrapText(true);
```

Все элементы управления вводом текста могут быть сделаны доступными только для чтения, как показано ниже.

```
textArea.setEditable(false);
```

В частности, доступную только для чтения текстовую область можно использовать для отображения крупного массива данных. При этом внешний вид текстовой области не претерпевает никаких изменений. А недоступный элемент управления вводом текста приобретает особый вид, указывающий на состояние его недоступности для пользователя.

Если задать *текст подсказки*, как показано ниже, он будет выделен светлым шрифтом, если элемент управления вводом текста пуст (рис. 13.16).

```
username.setPromptText("Choose a user name");
```



Рис. 13.16. Элементы управления вводом текста

В отличие от экранных кнопок, элементы управления вводом текста должны быть снабжены метками, размещаемыми рядом с ними. Это делается с помощью экземпляра внешнего узла типа `Label` следующим образом:

```
Label usernameLabel = new Label("User name:");
```


Узел метки типа `Label` подобен текстовому узлу типа `Text`, хотя последний относится к категории геометрических форм (`Shape`), предназначенных для рисования, тогда как первый — к категории элементов управления (`Control`). Класс `Label` наследует методы из суперкласса `Labeled` для решения таких задач, как графическое оформление и усечение текста, если недостаточно места для его отображения.

В примере программы из листинга 13.13 демонстрируется применение различных элементов управления вводом текста, включая поля ввода текста и пароля и текстовую область с полосами прокрутки. Поля ввода текста и пароля снабжены метками. Чтобы заполнить текстовую область содержимым обоих полей, следует щелкнуть на кнопке `Ok`.

Листинг 13.13. Исходный код из файла `text/TextControlTest.java`

```
1 package text;
2
3 import javafx.application.*;
4 import javafx.geometry.*;
5 import javafx.scene.*;
6 import javafx.scene.control.*;
7 import javafx.scene.layout.*;
8 import javafx.scene.text.*;
9 import javafx.stage.*;
10
11 /**
12  * @version 1.5 2017-12-29
13  * @author Cay Horstmann
14  */
15 public class TextControlTest extends Application
16 {
17     public void start(Stage stage)
18     {
19         final double rem = new Text("").getLayoutBounds()
20             .getHeight();
21
22         GridPane pane = new GridPane();
23         pane.setHgap(0.8 * rem);
24         pane.setVgap(0.8 * rem);
25         pane.setPadding(new Insets(0.8 * rem));
26
27         Label usernameLabel = new Label("User name:");
28         Label passwordLabel = new Label("Password:");
29
30
31         TextField username = new TextField();
32         username.setPromptText("Choose a user name");
33         PasswordField password = new PasswordField();
34         password.setPromptText("Choose a password");
35         TextArea textArea = new TextArea();
36         textArea.setPrefRowCount(10);
37         textArea.setPrefColumnCount(20);
38         textArea.setWrapText(true);
39         textArea.setEditable(false);
40
41         Button okButton = new Button("Ok");
```

```

42     okButton.setOnAction(event -> textArea.appendText(
43         "User name: " + username.getText()
44         + "\nPassword: " + password.getText() + "\n"));
45
46     pane.add(usernameLabel, 0, 0);
47     pane.add(username, 1, 0);
48     pane.add(passwordLabel, 0, 1);
49     pane.add(password, 1, 1);
50     pane.add(textArea, 0, 2, 2, 1);
51     pane.add(okButton, 0, 3, 2, 1);
52
53     GridPane.setHalignment(usernameLabel, HPos.RIGHT);
54     GridPane.setHalignment(passwordLabel, HPos.RIGHT);
55     GridPane.setHalignment(okButton, HPos.CENTER);
56
57     stage.setScene(new Scene(pane));
58     stage.setTitle("TextControlTest");
59     stage.show();
60 }
61 }

```

javafx.scene.Node

- **void setDisable(boolean value)**

Дезактивирует или активизирует данный узел и его порожденные узлы.

javafx.scene.control.TextInputControl

- **String getText()**
- **void setText(String value)**
Получают или устанавливают текст в данном элементе управления.
- **void appendText(String text)**
Присоединяет заданный текст к уже содержащемуся в данном элементе управления.
- **void setEditable(boolean value)**
Делает данный элемент управления доступным для редактирования или только для чтения.
- **void setPromptText(String value)**
Задаёт текст подсказки, отображаемый в данном элементе управления.

javafx.scene.control.TextField

- **void setPrefColumnCount(int value)**
Устанавливает предпочтительное количество столбцов в данном текстовом поле.

javafx.scene.control.TextArea

- **void setPrefRowCount(int value)**
- **void setPrefColumnCount(int value)**

Устанавливают предпочтительное количество рядов и столбцов в данной текстовой области.

- **void setWrapText(boolean value)**

Активизирует или деактивизирует автоматический перенос текстовых строк, превышающих данную текстовую область по длине.

javafx.scene.control.Label

- **Label(String text)**

Конструирует метку с заданным текстом.

javafx.scene.control.Labeled

- **void setGraphic(Node value)**
- **void setTextOverrun(OverrunStyle value)**

Вводит декоративный узел в данный помеченный элемент управления.

Устанавливает правило набегания текста, предписывающее указывать многоточие (...), когда текст метки следует усечь: вообще не указывать многоточие, указывать его в начале, посередине или в конце данного текста, на границах символов или слов. Для этой цели служат следующие константы из перечисления **OverrunStyle**: **CLIP**, **ELLIPSIS**, **WORD_ELLIPSIS**, **LEADING_ELLIPSIS**, **LEADING_WORD_ELLIPSIS**, **CENTER_ELLIPSIS**, **CENTER_WORD_ELLIPSIS**.

13.5.2. Элементы управления выбором разных вариантов

Выше пояснялось, как собирать текст, вводимый пользователем, но имеется немало случаев, когда ему лучше предоставить возможность выбрать один из готовых вариантов, чем вводить текст в элементе управления. Такие варианты выбора можно предоставить пользователю в виде кнопок-переключателей или списка, а себя — избавить от необходимости проверять ошибки во вводимых пользователем данных. В этом разделе поясняется, как программировать флажки, кнопки-переключатели и комбинированные списки из библиотеки JavaFX.

В рассматриваемом здесь примере программы пользователю предоставляется возможность выбрать начертание, размер и семейство шрифтов (рис. 13.17).

Если требуется лишь двухзначный ввод данных, можно воспользоваться флажком. С помощью флажка можно, например, предоставить возможность выбрать наклонное начертание шрифта или не выбирать его. Флажки сопровождаются обозначающими их метками. Текст метки задается в конструкторе при создании флажка следующим образом:

```
CheckBox italic = new CheckBox("Italic");
```

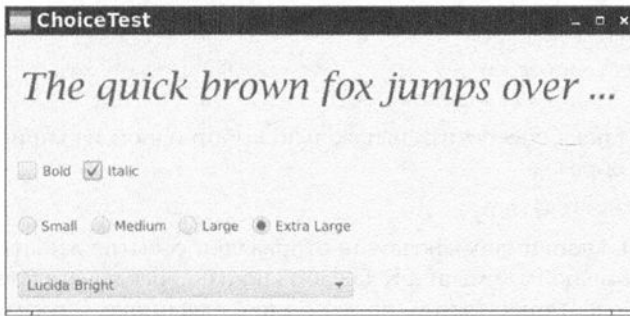


Рис. 13.17. Элементы управления выбором шрифта

Пользователь может установить флажок, щелкнув на его пустом квадратном поле, или сбросить флажок, снова щелкнув на галочке в его квадратном поле. Состояние флажка изменяется также нажатием клавиши пробела, когда фокус ввода находится на флажке.

Чтобы установить или сбросить флажок программно, следует воспользоваться методом `setSelected()`, как показано в следующем примере кода:

```
italic.setSelected(true);
```

Метод `setSelected()` принимает текущее состояние флажка. Если задается логическое значение `false`, то флажок сброшен, а если логическое значение `true`, то флажок установлен.

Когда пользователь щелкает на флажке, инициируется событие действия. Как всегда в подобных случаях, для присоединения обработчика событий к флажку вызывается метод `setOnAction()`. Класс `CheckBox` наследует этот метод от суперкласса `ButtonBase`.

В рассматриваемом здесь примере программы предоставляются флажки для выбора полужирного или наклонного начертания шрифта. Пользователь может установить любой из этих флажков, оба флажка или вообще не устанавливать их.

Зачастую требуется, чтобы пользователь установил лишь один из нескольких флажков. Так, если устанавливается следующий флажок, то предыдущий автоматически сбрасывается. Такая совокупность элементов управления выбором называется *группой кнопок-переключателей*, поскольку они напоминают переключатели диапазонов на старомодных радиоприемниках — при нажатии одной из таких кнопок ранее нажатая кнопка возвращается в исходное состояние. (Такая аналогия была уместна лет тридцать назад, но эти элементы управления выбором до сих пор называются “радиокнопками”).

В рассматриваемом здесь примере программы демонстрируется типичное приращение группы кнопок-переключателей. В частности, пользователь может выбрать один из следующих размеров шрифта: `Small` (Малый), `Medium` (Средний), `Large` (Крупный) и `Extra Large` (Очень крупный).

Сконструировать кнопку-переключатель совсем не трудно, как показано ниже.

```
RadioButton small = new RadioButton("Small");
```

Чтобы добиться требуемого поведения кнопок-переключателей, когда предыдущая кнопка выключается при выборе следующей, все кнопки-переключатели должны находиться в одной и той же *группе переключения*, как демонстрируется в следующем фрагменте кода:

```
ToggleGroup group = new ToggleGroup();
small.setToggleGroup(group);
medium.setToggleGroup(group);
. . .
```

Необходимо также обеспечить изначально выбор одной из кнопок-переключателей следующим образом:

```
medium.setSelected(true);
```

Как и флажки, кнопки-переключатели отправляют события действия, когда они выбираются щелчком кнопкой мыши. К каждой кнопке-переключателю можно, конечно, присоединить обработчик событий, но это было бы излишне. Вместо этого лучше организовать единый общий обработчик событий, но не включать в него код, проверяющий, является ли источником события кнопка-переключатель Small, Medium, Large или Extra Large. Ведь нас интересует лишь требующийся размер шрифта.

Пользовательские данные предоставляют удобный механизм для передачи произвольных данных от элемента управления к общему обработчику событий. В качестве пользовательских данных для элемента управления можно задать произвольные объекты. Так, требующиеся размеры шрифта можно установить следующим образом:

```
small.setUserData(8);
medium.setUserData(14);
. . .
```

Пользовательские данные извлекаются в общем обработчике событий из кнопки-переключателя, выбранной в группе в настоящий момент:

```
int size = (int) group.getSelectedToggle().getUserData();
```

Если обратиться снова к рис. 13.17, то можно заметить, что кнопки-переключатели внешне отличаются от флажков. В частности, флажки имеют квадратные поля с галочками, когда они установлены, а кнопки-переключатели — круглые поля с жирными точками, когда они выбраны.

Если имеется больше десятка разных вариантов, то кнопки-переключатели не годятся для их выбора, поскольку занимают слишком много места на экране. Вместо этого лучше воспользоваться комбинированным списком. Когда пользователь щелкает кнопкой мыши на таком элементе управления, раскрывается список вариантов, из которых он может сделать свой выбор (рис. 13.18).

Если раскрывающийся комбинированный список установлен в режим редактирования, то текущий вариант выбора можно отредактировать, как текст в поле. Данный элемент управления называется комбинированным списком потому, что он сочетает в себе удобство текстового поля с предварительно заданными вариантами выбора. А реализуется элемент управления комбинированным списком в классе `ComboBox`.

Чтобы сделать комбинированный список редактируемым, достаточно вызвать метод `setEditable()`. Однако редактирование оказывает воздействие только на выбранный элемент списка, не затрагивая остальные его элементы в частности и состав вариантов выбора вообще.

В рассматриваемом здесь примере программы пользователь может выбрать семейство шрифтов из списка всех имеющихся семейств. Все варианты выбора вводятся в список, возвращаемый методом `getItems()`, как показано ниже.

```
ComboBox<String> families = new ComboBox<>();
families.getItems().addAll(Font.getFontFamilies());
```

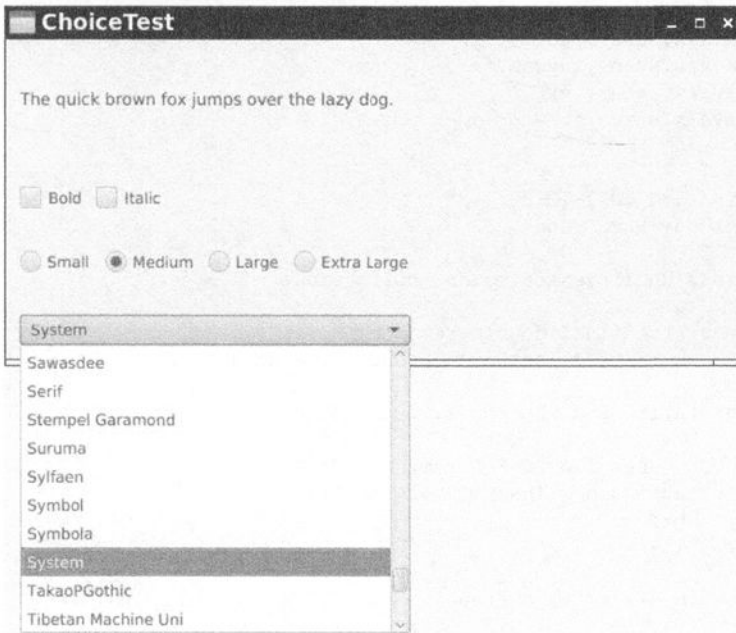


Рис. 13.18. Комбинированный список



НА ЗАМЕТКУ! Класс `ComboBox<T>` является обобщенным. Параметр типа обозначает в нем тип элементов комбинированного списка. Так, если элементы данного списка относятся к типу `String`, потребуется взаимное преобразование типа элементов списка и символьных строк, которые пользователь может видеть и потенциально редактировать. Более подробно этот вопрос рассматривается в главе 11 второго тома настоящего издания.

Вызвав метод `getValue()`, можно получить текущий вариант выбора, который мог быть отредактирован, если комбинированный список сделан редактируемым. А для того чтобы задать первоначальное значение варианта выбора, достаточно вызвать метод `setValue()`.

Подобно другим элементам управления, комбинированный список инициирует событие действия, когда пользователь делает свой выбор. В рассматриваемом здесь примере программы, исходный код которой приведен в листинге 13.14, не имеет смысла организовывать отдельные обработчики событий для каждого элемента управления. Вместо этого для всех элементов управления организуется единый общий обработчик событий, в котором шрифт определяется, исходя из значений, выбранных из флажков, кнопок-переключателей и комбинированного списка.

Листинг 13.14. Исходный код из файла `choices/ChoiceTest.java`

```
1 package choices;
2
3 import javafx.application.*;
4 import javafx.event.*;
5 import javafx.geometry.*;
```

```
6  import javafx.scene.*;
7  import javafx.scene.control.*;
8  import javafx.scene.layout.*;
9  import javafx.scene.text.*;
10 import javafx.stage.*;
11
12 /**
13     @version 1.4 2017-12-29
14     @author Cay Horstmann
15 */
16 public class ChoiceTest extends Application
17 {
18     private static final double rem =
19         new Text("").getLayoutBounds().getHeight();
20
21     private static HBox hbox(Node... children)
22     {
23         HBox box = new HBox(0.8 * rem, children);
24         box.setPadding(new Insets(0.8 * rem));
25         return box;
26     }
27
28     public void start(Stage stage)
29     {
30         Label sampleText = new Label("The quick brown fox "
31             + "jumps over the lazy dog.");
32         sampleText.setPrefWidth(40 * rem);
33         sampleText.setPrefHeight(5 * rem);
34         sampleText.setFont(Font.font(14));
35
36         CheckBox bold = new CheckBox("Bold");
37         CheckBox italic = new CheckBox("Italic");
38
39         RadioButton small = new RadioButton("Small");
40         RadioButton medium = new RadioButton("Medium");
41         RadioButton large = new RadioButton("Large");
42         RadioButton extraLarge =
43             new RadioButton("Extra Large");
44
45         small.setUserData(8);
46         medium.setUserData(14);
47         large.setUserData(18);
48         extraLarge.setUserData(36);
49
50         ToggleGroup group = new ToggleGroup();
51         small.setToggleGroup(group);
52         medium.setToggleGroup(group);
53         large.setToggleGroup(group);
54         extraLarge.setToggleGroup(group);
55         medium.setSelected(true);
56
57         ComboBox<String> families = new ComboBox<>();
58         families.getItems().addAll(Font.getFamilies());
59         families.setValue("System");
60
61         EventHandler<ActionEvent> listener = event ->
```

```
62     {
63         int size = (int) group.getSelectedToggle()
64             .getUserData();
65         Font font = Font.font(families.getValue(),
66             bold.isSelected()
67             ? FontWeight.BOLD :
68             FontWeight.NORMAL,
69             italic.isSelected()
70             ? FontPosture.ITALIC :
71             FontPosture.REGULAR, size);
72
73         sampleText.setFont(font);
74     };
75     small.setOnAction(listener);
76     medium.setOnAction(listener);
77     large.setOnAction(listener);
78     extraLarge.setOnAction(listener);
79     bold.setOnAction(listener);
80     italic.setOnAction(listener);
81     families.setOnAction(listener);
82
83     VBox root = new VBox(0.8 * rem,
84         hbox(sampleText),
85         hbox(bold, italic),
86         hbox(small, medium, large, extraLarge),
87         hbox(families));
88
89     stage.setScene(new Scene(root));
90     stage.setTitle("ChoiceTest");
91     stage.show();
92 }
93 }
```

javafx.scene.control.CheckBox

- **CheckBox(String text)**
Конструирует флажок с заданным текстом метки.
- **boolean isSelected()**
- **void setSelected(boolean value)**
Получают или устанавливают выбранное состояние флажка.

javafx.scene.control.RadioButton

- **RadioButton(String text)**
Конструирует кнопку-переключатель с заданным текстом метки.

javafx.scene.control.ToggleButton

- **void setToggleGroup(ToggleGroup value)**
Устанавливает группу переключения для данной кнопки-переключателя.
- **boolean isSelected()**
- **void setSelected(boolean value)**
Получают или устанавливают выбранное состояние данной кнопки-переключателя.

javafx.scene.control.ToggleGroup

- **ToggleGroup()**
Конструирует пустую группу переключения.
- **Toggle getSelectedToggle()**
Получает выбранный элемент из данной группы переключения.

javafx.scene.control.Toggle

- **void setUserData(Object value)**
- **Object getUserData()**
Получают или устанавливают элемент данных, связанный с данной кнопкой-переключателем.

javafx.scene.control.ComboBox<T>

- **ComboBox()**
Конструирует комбинированный список без элементов.
- **ObservableList<T> getItems()**
Выдает изменяемый список элементов, присутствующих в данном комбинированном списке.

javafx.scene.control.ComboBoxBase<T>

- **void setValue(T)**
- **T getValue()**
Получают или устанавливают значение, выбранное в настоящий момент.
- **void setOnAction(EventHandler<ActionEvent>)**
Устанавливает приемник событий действия для данного элемента управления.

13.5.3. Меню

Рассмотрение компонентов GUI было начато с тех элементов управления, которые чаще всего располагаются в окне, включая различные виды кнопок, текстовые поля, флажки, кнопки-переключатели и комбинированные списки. Но в библиотеке JavaFX поддерживается и другой тип элементов пользовательского интерфейса — ниспадающие меню, широко употребляемые в приложениях с GUI.

Строка меню в верхней части окна содержит названия ниспадающих меню. Щелкая на таком имени кнопкой мыши, пользователь открывает меню, состоящее из пунктов и подменю. Если пользователь щелкнет на пункте меню, все меню закроются и программе будет отправлено соответствующее уведомление. На рис. 13.19 показано типичное меню, состоящее из пунктов и подменю.

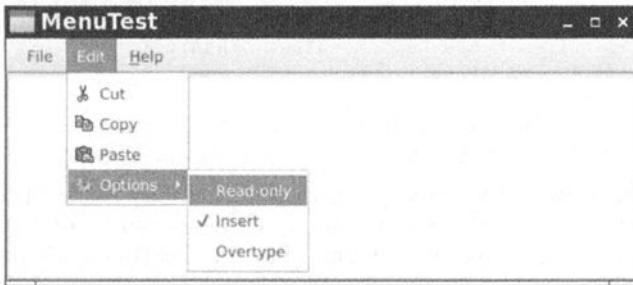


Рис. 13.19. Меню, состоящее из пунктов и подменю

Чтобы создать меню, следует предоставить его текст и дополнительное графическое оформление. В качестве графического оформления может служить любой узел типа `Node`, но чаще всего для этой цели предоставляется изображение, как показано ниже.

```
MenuItem newItem = new MenuItem("New");
MenuItem cutItem = new MenuItem("Cut",
    new ImageView("menu/cut.gif"));
```

Аналогичным образом конструируется объект типа `Menu`:

```
Menu editMenu = new Menu("Edit");
```

Пункты меню, их разделители и подменю вводятся в объект меню как элементы списка.

```
editMenu.getItems().addAll(cutItem, copyItem, pasteItem);
```

А с другой стороны, при создании меню можно предоставить его порожденные элементы. Если же графическое оформление меню не требуется, в качестве второго аргумента конструктора следует указать пустое значение `null`.

```
Menu optionsMenu = new Menu("Options",
    new ImageView("menu/options.gif"),
    readOnlyItem, insertItem, overtypеItem);
```

Меню верхнего уровня размещаются в строке меню, представленной объектом типа `MenuBar`, а сама строка меню — на корневой панели.

```
MenuBar bar = new MenuBar(fileMenu, editMenu, helpMenu);
VBox root = new VBox(bar, . . .);
```

Когда пользователь выбирает пункт меню, инициируется событие действия. Следовательно, для каждого пункта меню следует установить обработчик, как показано ниже.

```
exitItem.setOnAction(event -> Platform.exit());
```

С помощью компонентов `CheckMenuItem` и `RadioMenuItem` рядом с пунктами меню можно отобразить соответствующие отметки, как показано ниже. Они действуют аналогично флажкам и кнопкам-переключателям, но последние следует ввести в группу переключения. Если выбирается одна из кнопок в группе, выбор всех остальных автоматически отменяется.

```
CheckMenuItem readOnlyItem =  
    new CheckMenuItem("Read-only");  
  
ToggleGroup group = new ToggleGroup();  
RadioMenuItem insertItem = new RadioMenuItem("Insert");  
insertItem.setToggleGroup(group);  
insertItem.setSelected(true);  
RadioMenuItem overtypeItem = new RadioMenuItem("Overtypе");  
overtimeItem.setToggleGroup(group);
```

В рассматриваемом здесь примере прикладной программы при выборе пункта меню `Read-only` (Только для чтения) становятся недоступными пункты меню `Save` и `Save as` (Сохранить как). Для этого достаточно установить обработчик событий следующим образом:

```
readOnlyItem.setOnAction(event ->  
{  
    saveItem.setDisable(readOnlyItem.isSelected());  
    saveAsItem.setDisable(readOnlyItem.isSelected());  
});
```

С другой стороны, можно поступить более изящно, привязав соответствующие свойства, как показано ниже и поясняется далее, в разделе 13.6.2.

```
saveItem.disableProperty()  
    .bind(readOnlyItem.selectedProperty());  
saveAsItem.disableProperty()  
    .bind(readOnlyItem.selectedProperty());
```

В итоге недоступные пункты меню выделяются светлым шрифтом, как показано на рис. 13.20.

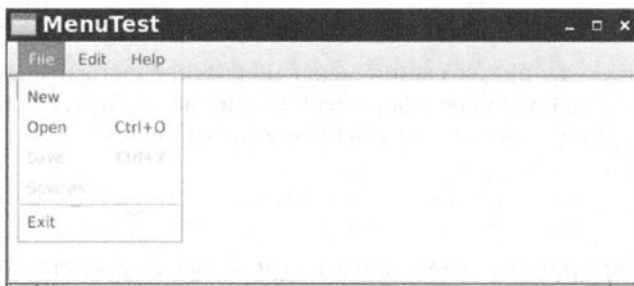


Рис. 13.20. Недоступные пункты меню

Оперативные клавиши позволяют выбрать пункт, не открывая меню. Например, во многих прикладных программах предусмотрены комбинации клавиш <Ctrl+O> и <Ctrl+S> для пунктов Open (Открыть) и Save (Сохранить) меню File (Файл). Для связывания оперативных клавиш с пунктом меню служит метод `setAccelerator()`:

```
openItem.setAccelerator(  
    KeyCombination.keyCombination("Shortcut+O"));
```

В приведенном выше примере кода `Shortcut` обозначает модифицирующую клавишу <Control> в Windows/Linux или <Command> в Mac OS. Оперативная клавиша отображается рядом с пунктом меню, как показано на рис. 13.21.

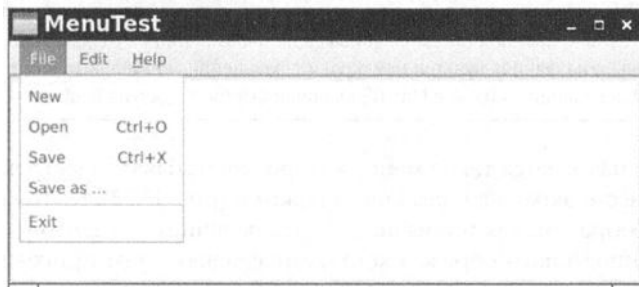


Рис. 13.21. Оперативные клавиши

При нажатии оперативных клавиш автоматически выбирается соответствующий пункт меню и событие инициируется таким же образом, как и при выборе пункта меню вручную. Оперативные клавиши можно связывать только с пунктами меню, но не с меню в целом.

Число оперативных клавиш ограничено, даже если употребить дополнительно модифицирующие клавиши. Хотя некоторые пользователи способны запоминать такие комбинации клавиш, как, например, <Control+Shift+F11>. В Linux и Windows поддерживаются клавиши быстрого доступа, с помощью которых пользователи могут быстро перемещаться по меню, нажимая на клавишу <Alt> и буквы, подчеркнутые в названиях пунктов меню. Например, в меню, приведенном на рис. 13.22, пункты Help (Справка) и About Core Java (О книге "Java. Библиотека профессионала") последовательно выбираются при нажатии сначала комбинации клавиш <Alt+H>, а затем клавиши <C>. В отличие от клавиатурных сокращений, клавиши быстрого доступа позволяют выбирать все пункты меню с помощью очень легко запоминаемых комбинаций клавиш.

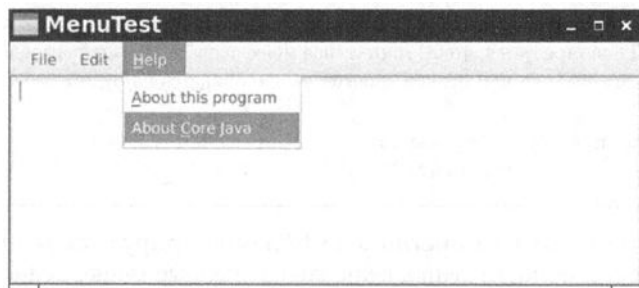


Рис. 13.22. Клавиши быстрого доступа

Чтобы назначить клавиши быстрого доступа для меню или его пунктов, достаточно указать знак подчеркивания перед мнемонической буквой в названии меню или его пункта при вызове соответствующего конструктора:

```
MenuItem aboutProgramItem =  
    new MenuItem("_About this program");  
MenuItem aboutCoreJavaItem =  
    new MenuItem("About _Core Java");  
Menu helpMenu = new Menu("_Help", null, aboutProgramItem,  
    aboutCoreJavaItem);
```



НА ЗАМЕТКУ! В большинстве сред, где поддерживается пользовательский интерфейс, включая и библиотеку JavaFX, предпринимаются сбивающие пользователя с толку усилия скрывать мнемонические буквы в названиях пунктов меню ради сокращения "визуальных помех" до тех пор, пока не будет нажата клавиша <Alt>. А в Mac OS клавиши быстрого доступа вообще не поддерживаются.

Контекстным называется такое меню, которое не привязано к строке меню, а появляется в любом месте экрана в плавающем режиме (рис. 13.23). Контекстное меню создается таким же образом, как и обычное, за исключением того, что у него отсутствует заголовок. Созданное таким образом контекстное меню затем присоединяется к соответствующему элементу управления, как показано ниже. Контекстное меню всплывает, если щелкнуть правой кнопкой мыши (в Mac OS нажать клавишу <Control> и щелкнуть кнопкой мыши) на том элементе управления, к которому оно присоединено.

```
ContextMenu contextMenu =  
    new ContextMenu(cutItem, copyItem, pasteItem);  
textArea.setContextMenu(contextMenu);
```

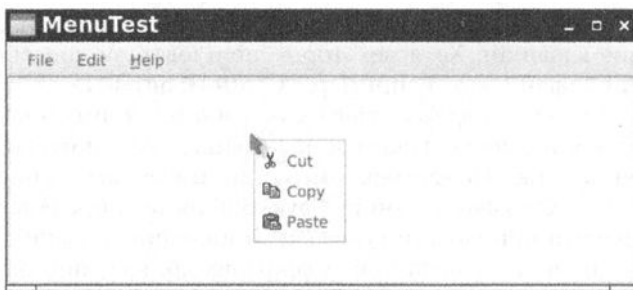


Рис. 13.23. Контекстное меню



НА ЗАМЕТКУ! Если требуется, чтобы контекстное меню всплывало над произвольным узлом, когда пользователь щелкает на нем правой кнопкой мыши, придется потрудиться чуть больше, как показано ниже.

```
node.setOnContextMenuRequested(e -> contextMenu.show(  
    node, e.getScreenX(), e.getScreenY()));
```

В примере программы из листинга 13.15 демонстрируются все рассмотренные разновидности и особенности меню, включая вложенные меню, недоступные пункты меню, помеченные флажками и кнопками-переключателями пункты меню, контекстное меню, а также клавиши быстрого вызова и оперативные клавиши.

Листинг 13.15. Исходный код из файла `menu/MenuTest.java`

```
1  package menu;
2
3  import javafx.application.*;
4  import javafx.event.*;
5  import javafx.scene.*;
6  import javafx.scene.control.*;
7  import javafx.scene.image.*;
8  import javafx.scene.input.*;
9  import javafx.scene.layout.*;
10 import javafx.stage.*;
11
12 /**
13     @version 1.3 2017-12-29
14     @author Cay Horstmann
15 */
16 public class MenuTest extends Application
17 {
18     private TextArea textArea = new TextArea();
19
20     /**
21         Вынуждает данный пункт меню или порожденные
22         от меню пункты выполнять действие по умолчанию,
23         если конкретное действие еще не определено
24         @param item Пункт меню (или же само меню)
25         @param action Действие по умолчанию
26     */
27     private void defaultAction(MenuItem item,
28                               EventHandler<ActionEvent> action)
29     {
30         if (item instanceof Menu)
31             for (MenuItem child : ((Menu) item).getItems())
32                 defaultAction(child, action);
33         else if (item.getOnAction() == null)
34             item.setOnAction(action);
35     }
36
37     public void start(Stage stage)
38     {
39         Menu fileMenu = new Menu("File");
40         MenuItem exitItem = new MenuItem("Exit");
41         exitItem.setOnAction(event -> Platform.exit());
42
43         // продемонстрировать оперативные клавиши
44
45         MenuItem newItem = new MenuItem("New");
46         MenuItem openItem = new MenuItem("Open ...");
47         openItem.setAccelerator(
48             KeyCombination.keyCombination("Shortcut+O"));
49         MenuItem saveItem = new MenuItem("Save");
50         saveItem.setAccelerator(
51             KeyCombination.keyCombination("Shortcut+S"));
52         MenuItem saveAsItem = new MenuItem("Save as ...");
53     }
54 }
```

```
54     fileMenu.getItems().addAll(newItem,
55                                 openItem,
56                                 saveItem,
57                                 saveAsItem,
58                                 new SeparatorMenuItem(),
59                                 exitItem);
60
61     // продемонстрировать пункты меню, помеченные
62     // флажками и кнопками-переключателями
63
64     CheckMenuItem readOnlyItem =
65         new CheckMenuItem("Read-only");
66     readOnlyItem.setOnAction(event ->
67     {
68         saveItem.setDisable(readOnlyItem.isSelected());
69         saveAsItem.setDisable(
70             readOnlyItem.isSelected());
71     });
72     /*
73     С другой стороны, воспользоваться привязкой:
74     saveItem.disableProperty()
75         .bind(readOnlyItem.selectedProperty());
76     saveAsItem.disableProperty()
77         .bind(readOnlyItem.selectedProperty());
78     */
79
80     ToggleGroup group = new ToggleGroup();
81     RadioMenuItem insertItem =
82         new RadioMenuItem("Insert");
83     insertItem.setToggleGroup(group);
84     insertItem.setSelected(true);
85     RadioMenuItem overtypeItem =
86         new RadioMenuItem("Overtypе");
87     overtypeItem.setToggleGroup(group);
88
89     Menu editMenu = new Menu("Edit");
90
91     // продемонстрировать пиктограммы
92
93     MenuItem cutItem = new MenuItem("Cut",
94         new ImageView("menu/cut.gif"));
95     MenuItem copyItem = new MenuItem("Copy",
96         new ImageView("menu/copy.gif"));
97     MenuItem pasteItem = new MenuItem("Paste",
98         new ImageView("menu/paste.gif"));
99
100    // продемонстрировать контекстное меню
101
102    ContextMenu contextMenu = new ContextMenu(
103        cutItem, copyItem, pasteItem);
104    textArea.setContextMenu(contextMenu);
105
106    editMenu.getItems().addAll(cutItem, copyItem,
107        pasteItem);
108    // Ошибка или ограничение - необходимо ввести
109    // сначала в контекстное меню
```

```

110         // http://bugs.java.com/bugdatabase
111         //         /view_bug.do?bug_id=JDK-8194270
112
113         // продемонстрировать вложенные меню
114
115         Menu optionsMenu = new Menu("Options",
116             new ImageView("menu/options.gif"),
117             readOnlyItem, insertItem, overtypeItem);
118
119         editMenu.getItems().add(optionsMenu);
120
121         // продемонстрировать клавиши быстрого выбора
122
123         MenuItem aboutProgramItem =
124             new MenuItem("_About this program");
125         MenuItem aboutCoreJavaItem =
126             new MenuItem("About _Core Java");
127         Menu helpMenu = new Menu("_Help", null,
128             aboutProgramItem, aboutCoreJavaItem);
129
130         // ввести строку меню
131
132         MenuBar bar = new MenuBar(fileMenu, editMenu,
133             helpMenu);
134         VBox root = new VBox(bar, textArea);
135         for (Menu menu : bar.getMenus())
136             defaultAction(menu, event ->
137             {
138                 MenuItem item = (MenuItem) event.getSource();
139                 textArea.appendText(item.getText()
140                     + " selected\n");
141             });
142
143         stage.setScene(new Scene(root));
144         stage.setTitle("MenuTest");
145         stage.show();
146     }
147 }

```

javafx.scene.control.MenuItem

- **MenuItem(String text)**
- **MenuItem(String text, Node graphic)**
Конструируют пункты меню с заданным текстом и графикой.
- **public void setOnAction(EventHandler<ActionEvent> value)**
Устанавливает обработчик событий, вызываемый при выборе данного пункта меню.
- **Menu getParentMenu()**
- Возвращает меню, в котором содержится данный пункт.
- **void setAccelerator(KeyCombination value)**
Устанавливает оперативную клавишу для выбора данного пункта меню.

javafx.scene.input.KeyCombination

- **static KeyCombination keyCombination(String name)**

Выдает комбинации клавиш для заданного описания. Такая комбинация составляется из модифицирующих клавиш <Shift>, <Ctrl>, <Alt>, <Meta>, <Shortcut>, знака + и наименования клавиши, возвращаемого методом **KeyCode.getName()**. Например: "Ctrl+Shift+F11".

javafx.scene.control.Menu

- **Menu(String text)**
- **Menu(String text, Node graphic)**
- **Menu(String text, Node graphic, MenuItem... items)**
Конструируют меню с заданным текстом, графикой и пунктами.
- **ObservableList<MenuItem> getItems()**

Возвращает изменяемый список пунктов данного меню.

javafx.scene.control.MenuBar

- **MenuBar(Menu... menus)**
Конструирует строку с заданными меню.
- **ObservableList<Menu> getMenus()**

Возвращает изменяемый список меню из данной строки меню.

javafx.scene.control.CheckMenuItem

- **CheckMenuItem(String text)**
- **CheckMenuItem(String text, Node graphic)**
Конструируют помечаемый флажком пункт меню с заданным текстом и графикой.
- **boolean isSelected()**

Возвращает логическое значение **true**, если выбран данный пункт меню.

javafx.scene.control.RadioMenuItem

- **RadioMenuItem(String text)**
- **RadioMenuItem(String text, Node graphic)**
Конструируют помечаемый кнопкой-переключателем пункт меню с заданным текстом и графикой.
- **boolean isSelected()**
Возвращает логическое значение **true**, если выбран данный пункт меню.
- **void setToggleGroup(ToggleGroup value)**
Устанавливает группу переключения для данного пункта меню.

javafx.scene.control.Control

- **void setContextMenu(ContextMenu value)**

Устанавливает контекстное меню для данного элемента управления.

javafx.scene.Node

- **void setOnContextMenuRequested(EventHandler<? super ContextMenuEvent> value)**

Устанавливает обработчик событий от щелчка правой кнопкой мыши на данном узле.

13.5.4. Простые диалоговые окна

В библиотеке JavaFX предоставляется ряд готовых простых диалоговых окон для отображения сообщений и получения разово введенных данных. Чтобы отобразить сообщение в таком диалоговом окне, следует создать сначала объект типа `Alert`, предоставив тип и текст сообщения (в данном случае — аварийное), а затем вызвать метод `showAndWait()`, как показано ниже.

```
Alert alert = new Alert(Alert.AlertType.INFORMATION,  
                        "Everything is fine.");  
alert.showAndWait();
```

Диалоговое окно отображается до тех пор, пока пользователь не удалит его с экрана, щелкнув на экранной кнопке **OK**, закрыв окно или нажав клавишу `<Esc>`. Изменив тип сообщения на подтверждающее (`CONFIRMATION`), предупреждающее (`WARNING`) или об ошибке (`ERROR`), можно изменить текст и изображение в заголовке окна, как показано на рис. 13.24.



Рис. 13.24. Разновидности простых диалоговых окон с предупреждающими сообщениями

В диалоговом окне для вывода предупреждающих сообщений типа `CONFIRMATION` имеются две экранные кнопки. Чтобы выяснить, подтвердил или отменил свои действия пользователь, необходимо зафиксировать значение, возвращаемое методом `showAndWait()`. Это значение относится к типу `Optional<ButtonType>`. В перечислении `ButtonType` определены типы экранных кнопок, которые допускаются в предупреждающем диалоговом окне. А класс `Optional`, обсуждаемый в главе 1 второго тома настоящего издания, представляет необязательное значение, которое может присутствовать или отсутствовать. Подтверждающее диалоговое окно всегда возвращает значение `ButtonType.OK` или `ButtonType.CANCEL`, заключаемое в оболочку класса `Optional`. А поскольку необязательное значение типа `Optional` не может быть пустым, то, для того чтобы выяснить, щелкнул ли пользователь на экранной кнопке `OK`, достаточно вызвать метод `get()`:

```
if (alert.showAndWait().get() == ButtonType.OK)
{
    // пользователь подтвердил свои действия
    . . .
}
```

В предупреждающее диалоговое окно можно ввести любые поддерживаемые экранные кнопки (рис. 13.25). Их достаточно перечислить в конструкторе этого диалогового окна, как показано ниже.

```
Alert alert = new Alert(Alert.AlertType.NONE,
    "Now what?",
    ButtonType.NEXT,
    ButtonType.PREVIOUS,
    ButtonType.FINISH);
```

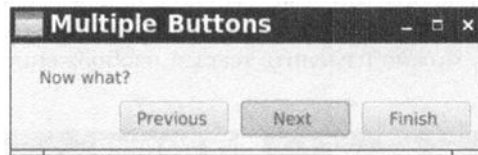


Рис. 13.25. Предупреждающее диалоговое окно с несколькими экранными кнопками



НА ЗАМЕТКУ! Если в простом диалоговом окне имеется несколько экранных кнопок, но отсутствует экранная кнопка `Cancel`, его нельзя закрыть. Чтобы продолжить дальше, пользователь должен щелкнуть на одной из имеющихся в окне экранных кнопок.

Предупреждающее диалоговое окно, приведенное на рис. 13.25, относится к типу `NONE`. Заголовок в нем не отображается, поскольку его текст и графика имеют пустое значение `null`. Впрочем, свой заголовок и графику можно предоставить, как показано в приведенном ниже примере кода и на рис. 13.26.

```
alert.setHeaderText("Exception");
alert.setGraphic(new ImageView("dialogs/bomb.png"));
```

В таком диалоговом окне отображается узел *расширяемого содержимого*, которое обнаруживается после щелчка на экранной кнопке `Show Details` (Показать подробности).

В данном случае это текстовая область, содержащая результат трассировки стека исключений. Ниже показано, как установить такой узел.

```
TextArea stackTrace = new TextArea();
alert.getDialogPane().setExpandableContent(stackTrace);
```

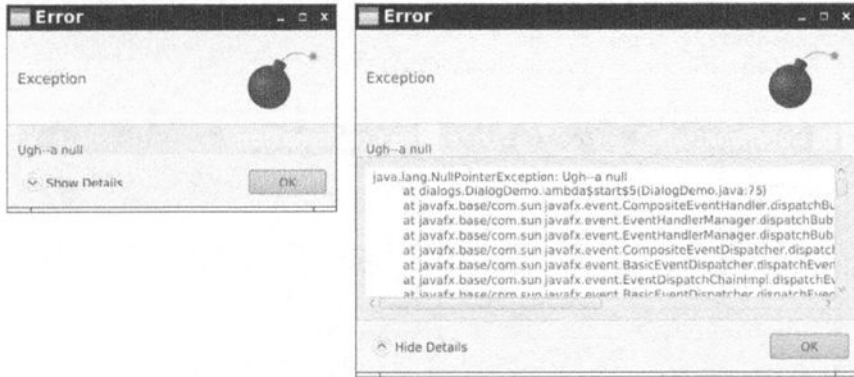


Рис. 13.26. Расширяемое содержимое простого диалогового окна

Если же требуется принять вводимые пользователем текстовые данные, следует создать соответствующее диалоговое окно, воспользовавшись классом `TextInputDialog`, как показано на рис. 13.27 и поясняется ниже.

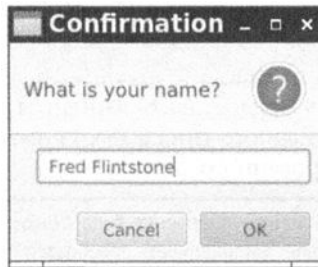


Рис. 13.27. Простое диалоговое окно для ввода текста

Метод `showAndWait()` возвращает значение типа `Optional<String>`, которое окажется пустым, если пользователь щелкнул на экранной кнопке `Cancel` или удалил окно с экрана. Ниже показан один из способов обработки такого значения.

```
TextInputDialog dialog = new TextInputDialog();
dialog.setHeaderText("What is your name?");
dialog.showAndWait().ifPresentOrElse(
    result -> { обработать полученный результат },
    () -> { обработать состояние отмены }
);
```

Аналогичное назначение имеет класс `ChoiceDialog`. С его помощью реализуется простое диалоговое окно, в котором предоставляется вариант выбора по умолчанию и массив или список других вариантов выбора, как показано ниже. Если вариант

выбора по умолчанию не пустой (`null`), то при отображении диалогового окна выбирается именно он (рис. 13.28). Метод `showAndWait()` возвращает значение типа `Optional`, содержащее выбранный элемент или оказывающееся пустым, если в диалоговом окне произошла отмена.

```
ChoiceDialog<String> dialog =
    new ChoiceDialog<>("System", Font.getFontFamilies());
dialog.setHeaderText("Pick a font.");
Optional<String> choice = dialog.showAndWait();
```

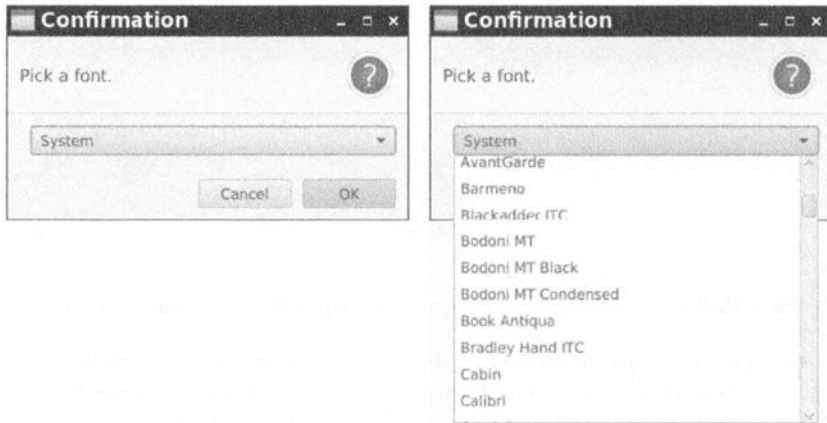


Рис. 13.28. Простое диалоговое окно для выбора вариантов

Параметр типа обобщенного класса `ChoiceDialog<T>` определяет тип вариантов выбора. Имеется возможность предоставить варианты выбора любого типа, при условии, что метод `toString()` данного типа выдает символьную строку, имеющую какое-то значение для комбинированного списка.



ВНИМАНИЕ! В отличие от элемента управления типа `ComboBox<T>`, элемент управления типа `ChoiceDialog<T>` не позволяет устанавливать преобразователь символьных строк.

И, наконец, имеется класс `FileChooser`, с помощью которого реализуется диалоговое окно для выбора файлов. Это окно является родственным применяемой операционной системе и не похоже на остальные диалоговые окна из библиотеки JavaFX. В этом окне можно указать каталог, выбираемый по умолчанию, а также фильтры для расширений файлов. При отображении диалогового окна для выбора файлов следует предоставить подмости, как показано ниже. Если же вместо этого указать пустое значение `null`, то ввод данных на подмостках будет заблокирован.

```
FileChooser dialog = new FileChooser();
dialog.setInitialDirectory(new File("images"));
dialog.getExtensionFilters().addAll(
    new FileChooser.ExtensionFilter("GIF images", "*.gif"),
    new FileChooser.ExtensionFilter("JPEG images", "*.jpg"));
File result = dialog.showSaveDialog(stage);
```

Метод `showSaveDialog()` отображает диалоговое окно, в котором предоставляется возможность выбрать существующий файл или ввести новое его имя. А метод

`showOpenDialog()` разрешает выбирать только существующие файлы. Имеется также метод `showOpenMultipleDialog()`, возвращающий список типа `List` всех выбранных файлов. Если же в диалоговом окне для выбора файлов происходит отмена, то все три упомянутых метода возвращают пустое значение `null`.

А для того чтобы выбрать каталог, следует воспользоваться классом `DirectoryChooser`:

```
DirectoryChooser dialog = new DirectoryChooser();  
File result = dialog.showDialog(stage);
```

В листинге 13.16 приведен пример программы, позволяющей поэкспериментировать со всеми описанными в этом разделе разновидностями простых диалоговых окон.

Листинг 13.16. Исходный код из файла `dialogs/DialogDemo.java`

```
1  package dialogs;  
2  
3  import java.io.*;  
4  
5  import javafx.application.*;  
6  import javafx.geometry.*;  
7  import javafx.scene.*;  
8  import javafx.scene.control.*;  
9  import javafx.scene.image.*;  
10 import javafx.scene.layout.*;  
11 import javafx.scene.text.*;  
12 import javafx.stage.*;  
13  
14 /**  
15  @version 1.0 2017-12-29  
16  @author Cay Horstmann  
17  */  
18 public class DialogDemo extends Application  
19 {  
20     public void start(Stage stage)  
21     {  
22         TextArea textArea = new TextArea();  
23  
24         Button information = new Button("Information");  
25         information.setOnAction(event ->  
26             {  
27                 Alert alert = new Alert(  
28                     Alert.AlertType.INFORMATION,  
29                     "Everything is fine.");  
30                 alert.showAndWait();  
31             });  
32  
33         Button warning = new Button("Warning");  
34         warning.setOnAction(event ->  
35             {  
36                 Alert alert = new Alert(  
37                     Alert.AlertType.WARNING,  
38                     "We have a problem.");  
39                 alert.showAndWait();  
40             });  
41     }  
42 }
```

```
42 Button error = new Button("Error");
43 error.setOnAction(event ->
44 {
45     Alert alert = new Alert(
46         Alert.AlertType.ERROR,
47         "This looks really bad.");
48     alert.showAndWait();
49 });
50
51 Button confirmation = new Button("Confirmation");
52 confirmation.setOnAction(event ->
53 {
54     Alert alert = new Alert(
55         Alert.AlertType.CONFIRMATION,
56         "Are you sure?");
57     if (alert.showAndWait().get() == ButtonType.OK)
58         textArea.appendText("Confirmed\n");
59     else
60         textArea.appendText("Canceled\n");
61 });
62
63 Button multipleButtons =
64     new Button("Multiple Buttons");
65 multipleButtons.setOnAction(event ->
66 {
67     Alert alert = new Alert(Alert.AlertType.NONE,
68         "Now what?",
69         ButtonType.NEXT,
70         ButtonType.PREVIOUS,
71         ButtonType.FINISH);
72     alert.setTitle("Multiple Buttons");
73     textArea.appendText(alert.showAndWait() + "\n");
74 });
75
76 Button expandableContent =
77     new Button("Expandable Content");
78 expandableContent.setOnAction(event ->
79 {
80     Throwable t = new NullPointerException(
81         "Ugh--a null");
82     Alert alert = new Alert(Alert.AlertType.ERROR,
83         t.getMessage());
84     alert.setHeaderText("Exception");
85     alert.setGraphic(new ImageView(
86         "dialogs/bomb.png"));
87
88     TextArea stackTrace = new TextArea();
89     StringWriter out = new StringWriter();
90     t.printStackTrace(new PrintWriter(out));
91     stackTrace.setText(out.toString());
92     alert.getDialogPane()
93         .setExpandableContent(stackTrace);
94
95     textArea.appendText(alert.showAndWait() + "\n");
96 });
97
98 Button textInput = new Button("Text input");
```

```
99     textInput.setOnAction(event ->
100     {
101         TextInputDialog dialog = new TextInputDialog();
102         dialog.setHeaderText("What is your name?");
103         dialog.showAndWait().ifPresentOrElse(
104             result -> textArea.appendText("Name: "
105                 + result + "\n"),
106             () -> textArea.appendText("Canceled\n"));
107     });
108
109     Button choiceDialog = new Button("Choice Dialog");
110     choiceDialog.setOnAction(event ->
111     {
112         ChoiceDialog<String> dialog =
113             new ChoiceDialog<>("System",
114                 Font.getFamilies());
115         dialog.setHeaderText("Pick a font.");
116         dialog.showAndWait().ifPresentOrElse(
117             result -> textArea.appendText("Selected: "
118                 + result + "\n"),
119             () -> textArea.appendText("Canceled\n"));
120     });
121
122     Button fileChooser = new Button("File Chooser");
123     fileChooser.setOnAction(event ->
124     {
125         FileChooser dialog = new FileChooser();
126         dialog.setInitialDirectory(new File("menu"));
127         dialog.setInitialFileName("untitled.gif");
128         dialog.getExtensionFilters().addAll(
129             new FileChooser.ExtensionFilter(
130                 "GIF images", "*.gif"),
131             new FileChooser.ExtensionFilter(
132                 "JPEG images", "*.jpg", "*.jpeg"));
133         File result = dialog.showSaveDialog(stage);
134         if (result == null)
135             textArea.appendText("Canceled\n");
136         else
137             textArea.appendText("Selected: "
138                 + result + "\n");
139     });
140
141     Button directoryChooser =
142         new Button("Directory Chooser");
143     directoryChooser.setOnAction(event ->
144     {
145         DirectoryChooser dialog =
146             new DirectoryChooser();
147         File result = dialog.showDialog(stage);
148         if (result == null)
149             textArea.appendText("Canceled\n");
150         else
151             textArea.appendText("Selected: "
152                 + result + "\n");
153     });
154
155     final double rem = new Text("").getLayoutBounds()
```



```

156                                     .getHeight();
157     VBox buttons = new VBox(0.8 * rem, information,
158                             warning, error, confirmation,
159                             multipleButtons, expandableContent,
160                             textInput, choiceDialog,
161                             fileChooser, directoryChooser);
162     buttons.setPadding(new Insets(0.8 * rem));
163
164     HBox root = new HBox(textArea, buttons);
165
166     stage.setScene(new Scene(root));
167     stage.setTitle("DialogDemo");
168     stage.show();
169 }
170 }

```

`javafx.scene.control.Alert`

- **`Alert(Alert.AlertType alertType)`**

Конструирует предупреждающее диалоговое окно заданного типа. Для обозначения разных типов диалоговых окон в перечислении **`Alert.AlertType`** определены константы **`INFORMATION`**, **`WARNING`**, **`ERROR`**, **`CONFIRMATION`** и **`NONE`**.

- **`Alert(Alert.AlertType alertType, String contentText, ButtonType... buttons)`**

Конструирует предупреждающее диалоговое окно заданного типа с указанным сообщением и экранными кнопками. Для обозначения разных типов экранных кнопок в перечислении **`ButtonType`** имеются константы **`OK`**, **`CANCEL`**, **`YES`**, **`NO`**, **`NEXT`**, **`PREVIOUS`**, **`FINISH`**, **`APPLY`** и **`CLOSE`**, причем обе константы, **`CANCEL`** и **`CLOSE`**, обозначают отмену в диалоговом окне.

`javafx.scene.control.Dialog<T>`

- **`Optional<T> showAndWait()`**

Отображает диалоговое окно и ожидает от пользователя ввода данных. Возвращает объект типа **`T`**, представляющий введенные пользователем данные и заключаемый в оболочку объекта типа **`Optional`**, или же пустой объект типа **`Optional`**, если диалоговое окно удалено.

- **`void setHeaderText(String value)`**

Задаёт текст, отображаемый в заголовке данного диалогового окна.

- **`void setGraphic(Node value)`**

Задаёт графику, отображаемую в данном диалоговом окне.

- **`DialogPane getDialogPane()`**

Получает панель, содержащую все элементы управления в данном диалоговом окне.

`javafx.scene.control.DialogPane`

- **`void setExpandableContent(Node content)`**

Устанавливает узел, который может быть развернут или свернут.

javafx.scene.control.TextInputDialog

- **TextInputDialog()**

Конструирует диалоговое окно для ввода символьной строки.

javafx.scene.control.ChoiceDialog

- **ChoiceDialog(T defaultChoice, T... choices)**

- **ChoiceDialog(T defaultChoice, Collection<T> choices)**

Конструируют диалоговое окно для выбора элемента типа **T**. Если параметр **defaultChoice** не принимает пустое значение **null**, то выбирается вариант по умолчанию.

javafx.stage.FileChooser

- **FileChooser()**

Конструирует диалоговое окно для выбора файлов.

- **File showOpenDialog(Window ownerWindow)**

- **List<File> showOpenMultipleDialog(Window ownerWindow)**

Возвращают выбранный файл или пустое значение **null**, если диалоговое окно удалено. На момент отображения диалогового окна для выбора файлов окно владельца, определяемое параметром **ownerWindow**, блокируется.

- **File showSaveDialog(Window ownerWindow)**

Отображает диалоговое окно для выбора существующего файла или ввода нового файла и возвращает выбранный файл или пустое значение **null**, если диалоговое окно удалено. На момент отображения диалогового окна для выбора файлов окно владельца, определяемое параметром **ownerWindow**, блокируется.

- **void setInitialDirectory(File value)**

Задаёт первоначальный каталог в данном диалоговом окне для выбора файлов.

FileChooser.ExtensionFilter

- **ExtensionFilter(String description, String... extensions)**

- **ExtensionFilter(String description, List<String> extensions)**

Конструируют фильтр расширений файлов, принимающий файлы с любыми заданными расширениями. Символьные строки расширения имеют форму ***.расширение**.

javafx.stage.DirectoryChooser

- **DirectoryChooser()**

Конструирует диалоговое окно для выбора каталогов.

javafx.stage.DirectoryChooser (окончание)

- **File showDialog(Window ownerWindow)**

Отображает диалоговое окно для выбора существующего каталога или создания нового каталога и возвращает выбранный каталог или пустое значение `null`, если диалоговое окно удалено. На момент отображения диалогового окна для выбора каталогов окно владельца, определяемое параметром `ownerWindow`, блокируется.

- **void setInitialDirectory(File value)**

Задаёт первоначальный каталог в данном диалоговом окне для выбора каталогов.

13.5.5. Специальные элементы управления

Как и в библиотеке Swing, в библиотеке JavaFX, безусловно, поддерживаются панели, деревья и таблицы, а также такие отсутствующие в Swing специальные элементы управления, как селектор данных и “меню-гармошка”. В этом разделе развешаются любые остатки ностальгии по библиотеке Swing путем демонстрации трех специальных элементов управления, функциональные возможности которых намного превосходят все, что способна предложить библиотека Swing.

На рис. 13.29 демонстрируется одна из многочисленных диаграмм, которые могут быть построены готовыми средствами библиотеки JavaFX. И для этого не придется устанавливать сторонние библиотеки.

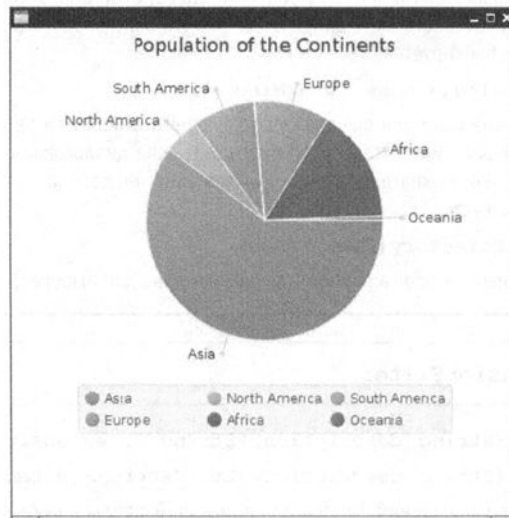


Рис. 13.29. Круговая диаграмма, построенная средствами библиотеки JavaFX

Построить круговую диаграмму совсем не трудно, как показано ниже.

```
PieChart chart = new PieChart();
chart.getData().addAll(
    new PieChart.Data("Asia", 4298723000.0),
    new PieChart.Data("North America", 355361000.0),
```

```
new PieChart.Data("South America", 616644000.0),  
new PieChart.Data("Europe", 742452000.0),  
new PieChart.Data("Africa", 1110635000.0),  
new PieChart.Data("Oceania", 38304000.0));  
chart.setTitle("Population of the Continents");
```

В общем, в библиотеке JavaFX насчитывается с полдесятка типов диаграмм, которыми можно воспользоваться или специально настроить. Подробнее об этом см. в документации, доступной по адресу <https://docs.oracle.com/javase/8/javafx/user-interface-tutorial/charts.htm>.

В библиотеке Swing имеется компонент JEditorPane для отображения HTML-страниц, хотя большинство реальных HTML-страниц все же воспроизводится плохо. И это понятно, поскольку реализовать браузер совсем не просто. На самом деле большинство браузеров созданы на основе механизма WebKit с открытым исходным кодом. То же самое сделано и в библиотеке JavaFX. Так, с помощью класса WebView отображается встроенное в WebKit платформенно-ориентированное окно (рис. 13.30).



Рис. 13.30. Просмотр веб-страницы.
Фото автора книги Кей Хорстманна

Ниже приведен исходный код для отображения веб-страницы.

```
String location = "http://horstmann.com";  
WebView browser = new WebView();  
WebEngine engine = browser.getEngine();  
engine.load(location);
```

В активном браузере имеется возможность щелкнуть на ссылках обычным образом, а также выполнять сценарии JavaScript. Но если требуется отобразить строку состояния или всплывающие сообщения из сценария JavaScript, то придется установить обработчики уведомлений и реализовать своими силами строку состояния и всплывающие сообщения.



НА ЗАМЕТКУ! В классе `WebView` не поддерживаются подключаемые модули, и поэтому с его помощью нельзя воспроизводить Flash-анимацию и документы в формате PDF. Ведь в этом классе не поддерживается отображение апплетов.

До появления библиотеки JavaFX воспроизводить мультимедийную информацию на языке Java было затруднительно. И хотя для этой можно было загрузить каркас Java Media Framework, он все же не нашел широкого распространения среди разработчиков. Безусловно, реализовать воспроизведение аудио- и видеозаписей еще труднее, чем написать браузер. Поэтому в библиотеке JavaFX выгодно используется уже существующий набор инструментальных средств, предоставляемый в каркасе GStreamer с открытым исходным кодом.

Чтобы воспроизвести видеозапись, достаточно сконструировать сначала объект типа `Media`, исходя из символьной строки с URL, а затем объект типа `MediaPlayer` для его воспроизведения и далее воспользоваться классом `MediaView` для отображения мультимедийного проигрывателя, как показано ниже.

```
Path path = Paths.get("moonlanding.mp4");
Media media = new Media(path.toUri().toString());
MediaPlayer player = new MediaPlayer(media);
player.setAutoplay(true);
MediaView video = new MediaView(player);
video.setOnError(System.out::println);
```

Как показано на рис. 13.31, видеозапись воспроизводится, но, к сожалению, без элементов управления ее воспроизведением. И хотя имеется возможность добавить эти элементы управления самостоятельно (см. документацию по адресу <https://docs.oracle.com/javase/8/javafx/media-tutorial/playercontrol.htm>), было бы неплохо, если бы элементы управления воспроизведением видеозаписей в библиотеке JavaFX предоставлялись по умолчанию.

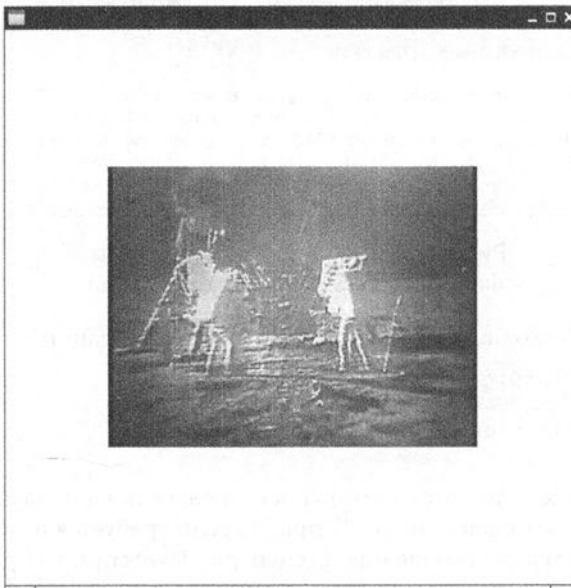


Рис. 13.31. Воспроизведение видеозаписи. Из архивов НАСА



НА ЗАМЕТКУ! Зачастую средствами каркаса GStreamer нельзя обработать конкретный видеофайл. Поэтому в обработчике ошибок из приведенного ниже примера программы выводятся сообщения из каркаса GStreamer о невозможности диагностировать ошибки воспроизведения.

В листинге 13.17 приведен исходный код программы, демонстрирующей применение всех специальных элементов управления, упоминавшихся в этом разделе.

Листинг 13.17. Исходный код из файла **fancy/FancyControls.java**

```
1 package fancy;
2
3 import java.nio.file.*;
4 import javafx.application.*;
5 import javafx.geometry.*;
6 import javafx.scene.*;
7 import javafx.scene.chart.*;
8 import javafx.scene.layout.*;
9 import javafx.scene.media.*;
10 import javafx.scene.web.*;
11 import javafx.stage.*;
12
13 /**
14  * @version 1.0 2017-12-29
15  * @author Cay Horstmann
16  */
17 public class FancyControls extends Application
18 {
19     public void start(Stage stage)
20     {
21         PieChart chart = new PieChart();
22         chart.getData().addAll(
23             new PieChart.Data("Asia", 4298723000.0),
24             new PieChart.Data("North America", 355361000.0),
25             new PieChart.Data("South America", 616644000.0),
26             new PieChart.Data("Europe", 742452000.0),
27             new PieChart.Data("Africa", 1110635000.0),
28             new PieChart.Data("Oceania", 38304000.0));
29         chart.setTitle("Population of the Continents");
30
31         String location = "http://horstmann.com";
32         WebView browser = new WebView();
33         WebEngine engine = browser.getEngine();
34         engine.load(location);
35
36         Path path = Paths.get("fancy/moonlanding.mp4");
37         Media media = new Media(path.toUri().toString());
38         MediaPlayer player = new MediaPlayer(media);
39         player.setAutoPlay(true);
40         MediaView video = new MediaView(player);
41         video.setOnError(ex -> System.out.println(ex));
42
43         stage.setWidth(500);
44         stage.setHeight(500);
```

```
45     stage.setScene(new Scene(browser));
46     stage.show();
47
48     Stage stage2 = new Stage();
49     stage2.setWidth(500);
50     stage2.setHeight(500);
51     stage2.setX(stage.getX() + stage.getWidth());
52     stage2.setY(stage.getY());
53     stage2.setScene(new Scene(chart));
54     stage2.show();
55
56     HBox box = new HBox(video);
57     box.setAlignment(Pos.CENTER);
58     Stage stage3 = new Stage();
59     stage3.setWidth(500);
60     stage3.setHeight(500);
61     stage3.setX(stage.getX());
62     stage3.setY(stage.getY() + stage.getHeight());
63     stage3.setScene(new Scene(box));
64     stage3.show();
65 }
66 }
```

13.6. Свойства и привязки

Свойство является атрибутом класса, доступным для чтения или записи. Обычно свойство поддерживается полем, а также методами получения и установки для чтения и записи данных в данном поле. Но в методах получения и установки можно выполнять и другие действия, включая чтение значений из базы данных или отправки уведомлений об изменениях.

В библиотеке JavaFX свойства играют особую роль, поскольку их легко “привязать”, чтобы обновить одно свойство, когда изменяется другое. В последующих разделах свойства и привязки в библиотеке JavaFX рассматриваются более подробно.

13.6.1. Свойства в библиотеке JavaFX

Во многих языках программирования имеется удобный синтаксис для вызова методов установки и получения значений свойств. Чтобы вызвать метод установки и получения значения свойства, достаточно указать имя свойства в правой части операции присваивания:

```
value = obj.property; // вызывается метод получения значения
                        // свойства во многих языках, но не в Java
obj.property = value; // а здесь вызывается метод установки
                        // значения свойства
```

К сожалению, в языке Java такой синтаксис отсутствует. Тем не менее, начиная с версии 1.1, в Java принято поддерживать свойства. Так, в спецификации JavaBeans утверждается, что свойство должно быть выведено из пары методов получения и установки. Например, в классе с методами `String getText()` и `void setText(String newValue)` подразумевается наличие свойства `text`. Классы `Introspector` и `BeanInfo` из пакета `java.beans` позволяют перечислить все свойства отдельного класса.

В спецификации JavaBeans определяются также *привязки свойств*, где объекты отправляют события изменения свойств при вызове методов установки. А в библиотеке JavaFX данная часть спецификации JavaBeans не применяется. Вместо этого в библиотеке JavaFX, помимо методов получения и установки, предоставляется еще и третий метод, возвращающий объект класса, реализующего интерфейс `Property`. Например, у свойства `text` в библиотеке JavaFX имеются следующие методы доступа:

```
String getText()  
void setText(String value)  
Property<String> textProperty()
```



НА ЗАМЕТКУ! В классе `Node` из библиотеки JavaFX имеется свыше 80 свойств, а в таких его подклассах, как `Rectangle` или `Button`, их больше сотни. Все эти свойства перечислены в документации на прикладной интерфейс Java API прежде конструкторов и методов.

К объекту свойства можно присоединить приемник событий. И в этом состоит заметное отличие от устаревших компонентов JavaBeans. В библиотеке JavaFX объект свойства, а не компонент `JavaBean` посылает уведомления, на что имеются веские основания. Ведь в реализацию привязываемых свойств JavaBeans требуется внедрять стереотипный код для ввода, удаления и запуска приемников событий. А в библиотеке JavaFX это делается намного проще, поскольку в ней предоставляются классы, берущие на себя все необходимые хлопоты.

Покажем, каким образом свойство `text` реализуется, например, в классе `Greeting`. Ниже приведен простейший способ сделать это.

```
public class Greeting  
{  
    private StringProperty text =  
        new SimpleStringProperty("");  
  
    public final StringProperty textProperty()  
    { return text; }  
    public final void setText(String newValue)  
    { text.set(newValue); }  
    public final String getText()  
    { return text.get(); }  
}
```

Символьная строка заключается в оболочку класса `StringProperty`. В этом классе имеются методы для получения и установки заключаемого в оболочку значения свойства `text`, а также для манипулирования приемниками событий.

Как видите, для реализации свойства в библиотеке JavaFX все-таки требуется некоторый стереотипный код, который, к сожалению, нельзя сгенерировать в Java автоматически. Но, по крайней мере, не нужно писать код для манипулирования приемниками событий. И хотя объявлять методы установки и получения как конечные (`final`) совсем не обязательно, разработчики библиотеки JavaFX все же рекомендуют это делать.



НА ЗАМЕТКУ! Если следовать такому образцу, то у каждого свойства должен быть свой объект, независимо от того, принимаются ли от него какие-нибудь события. Так, если реализовать класс со многими свойствами, предполагая построение многих его экземпляров, то создавать объекты свойств придется по требованию. Для хранения конкретного значения свойств рекомендуется использовать обычное поле, а обращаться к объекту свойства следует лишь в том случае, если вызывается метод `xxxProperty()`.

В приведенном выше примере кода был определен объект типа `StringProperty`. Что же касается свойства примитивного типа, то следует воспользоваться классом `IntegerProperty`, `LongProperty`, `DoubleProperty`, `FloatProperty` или `BooleanProperty`. Для этой цели имеются также классы `ListProperty`, `MapProperty` и `SetProperty`, а во всех остальных случаях следует воспользоваться классом `ObjectProperty<T>`. Все эти классы являются абстрактными и обладают конкретными подклассами `SimpleIntegerProperty`, `SimpleObjectProperty<T>` и т.д.



НА ЗАМЕТКУ! Если требуется лишь манипулировать приемниками событий и привязками, то из методов свойств могут быть возвращены объекты типа `ObjectProperty<T>` или даже типа `Property<T>`. Для выполнения вычислений с помощью свойств следует воспользоваться специализированными классами, как поясняется в следующем разделе.

К свойствам могут быть присоединены две разновидности приемников событий. Так, приемник событий, определяемый в интерфейсе `ChangeListener`, уведомляется об изменении значения в свойстве, а приемник событий, определяемый в функциональном интерфейсе `InvalidationListener`, вызывается, если значение в свойстве *могло* быть изменено. Отличия обоих приемников событий имеют значение лишь для свойства с отложенным вычислением. Как поясняется в следующем разделе, одни свойства вычисляются из других свойств, причем вычисление выполняется лишь по мере необходимости. Обратный вызов приемника событий типа `ChangeListener` позволяет получить прежние и новые значения свойства, и это означает, что в нем должно быть вычислено новое значение. А в приемнике событий типа `InvalidationListener` новое значение не вычисляется, но это означает, что его обратный вызов может быть произведен даже в том случае, если значение свойства фактически не изменилось. Но, как правило, подобные отличия обоих приемников событий несущественны.

В качестве примера ниже показано, как ввести приемник событий недостоверности, вызываемый всякий раз, когда изменяется значение в свойстве `text` объекта приветствия.

```
Greeting greeting = new Greeting();
greeting.setText("Hello");
greeting.textProperty().addListener(event ->
{
    System.out.println("greeting is now "
        + greeting.getText());
});
greeting.setText("Goodbye"); // теперь вызывается
                             // приемник событий
```

С другой стороны, ниже показано, как присоединить к свойству `text` приемник событий изменения этого свойства.

```
greeting.textProperty().addListener((property,
    oldValue, newValue) ->
{
    System.out.println("greeting is now " + newValue);
});
```



ВНИМАНИЕ! Применять интерфейс `ChangeListener` к числовым свойствам немного труднее. В частности, было бы желательно сделать следующий вызов:

```
slider.valueProperty().addListener((property,  
    oldValue, newValue) ->  
    message.setFont(Font.font(newValue)));
```

Но такой прием не сработает. Ведь в классе **DoubleProperty** реализуется интерфейс **Property<Number>**, а не **Property<Double>**. Поэтому параметры **oldValue** и **newValue** метода **addListener()** относятся к типу **Number**, а не к типу **Double**. Это означает, что их значения придется распаковать вручную, как показано ниже.

```
slider.valueProperty().addListener((property,  
    oldValue, newValue) ->  
    message.setFont(Font.font(newValue.doubleValue())));
```

13.6.2. Привязки

Разумным основанием для применения свойств в библиотеке JavaFX служит понятие привязки, предназначенной для автоматического обновления одного свойства, когда изменяется другое свойство. Рассмотрим в качестве примера прикладную программу, приведенную на рис. 13.32. Когда пользователь редактирует адрес доставки в верхней части окна, в нижней его части обновляется также адрес выставления счета.

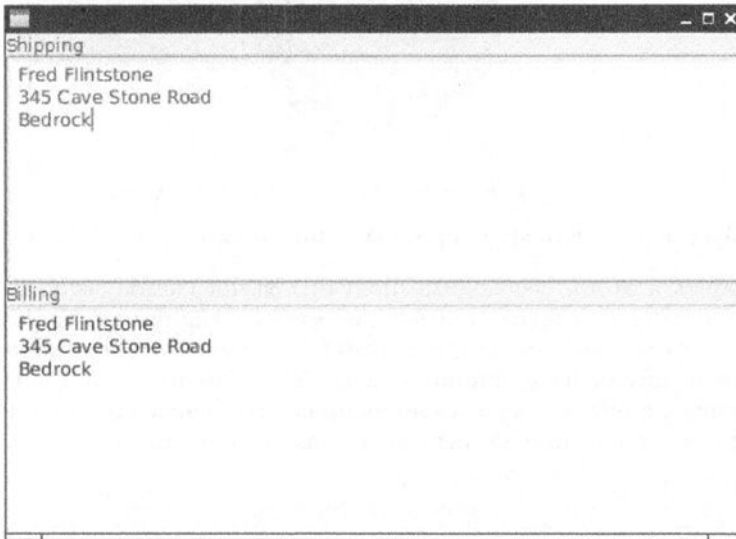


Рис. 13.32. Привязанное текстовое свойство обновляется автоматически

Такой результат достигается благодаря привязке одного свойства к другому. Подспудно приемник событий изменения свойства присоединяется к свойству **text** объекта **shipping**, где устанавливается свойство **text** объекта **billing**.

```
billing.textProperty().bind(shipping.textProperty());
```

Имеется также возможность сделать следующий вызов:

```
billing.textProperty().bindBidirectional(  
    shipping.textProperty());
```

Если теперь изменится любое из этих свойств, то обновится и другое свойство. Чтобы отменить привязку, достаточно вызвать метод `unbind()` или `unbindBidirectional()`.

Механизм привязки решает типичное затруднение, возникающее при программировании пользовательского интерфейса. Рассмотрим в качестве примера поле даты и селектор дат из календаря. В частности, когда пользователь выбирает дату из календаря, поле даты должно автоматически обновиться, как, впрочем, и свойство даты в модели.

Как правило, одно свойство зависит от другого, хотя их взаимосвязь оказывается более сложной. Рассмотрим в качестве примера прикладную программу, приведенную на рис. 13.33. Круг требуется, как всегда, расположить по центру сцены. Это означает, что значение свойства `centerX` должно быть наполовину меньше значения свойства `width` данной сцены.

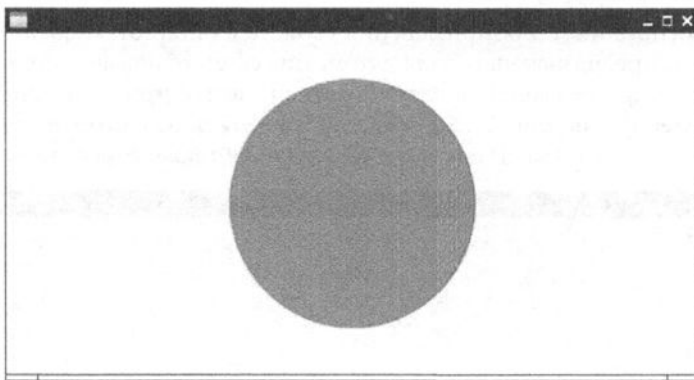


Рис. 13.33. Центр данного круга привязан к половине ширины и высоты сцены

Чтобы добиться этого, необходимо получить вычисляемое свойство. Для этой цели в классе `Bindings` имеются статические методы. Например, при вызове метода `Bindings.divide(scene.widthProperty(), 2)` вычисляется свойство, значение которого наполовину меньше ширины сцены. Когда изменяется ширина сцены, изменяется и данное свойство. Остается лишь привязать данное вычисляемое свойство к свойству `centerX` рисуемого на сцене круга, как показано ниже.

```
circle.centerXProperty().bind(Bindings.divide(
    scene.widthProperty(), 2));
```



НА ЗАМЕТКУ! С другой стороны, можно вызвать метод `scene.widthProperty().divide(2)`. Но если для вычисления свойства применяются более сложные выражения, то методы из класса **Bindings**, по-видимому, становятся более удобочитаемыми, особенно если в прикладном коде употребляется оператор.

```
import static javafx.beans.binding.Bindings.*;
```

и делается такой вызов:

```
divide(scene.widthProperty(), 2)
```

Рассмотрим более реалистичный пример, где требуется сделать недоступными экранные кнопки **Smaller** (Меньше) и **Larger** (Больше), когда индикатор показывает

слишком много или же слишком мало. Так, если ширина £ 0, недоступной становится экранная кнопка **Smaller**. А если ширина ³ 100, недоступной становится экранная кнопка **Larger** (рис. 13.34):

```
smaller.disableProperty().bind(Bindings.lessThanOrEqualTo(
    gauge.widthProperty(), 0));
larger.disableProperty().bind(Bindings.greaterThanOrEqualTo(
    gauge.widthProperty(), 100));
```

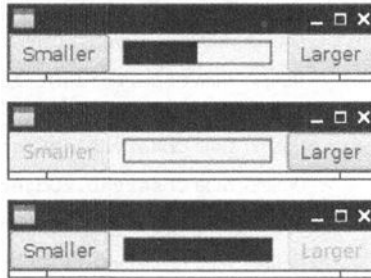


Рис. 13.34. Когда индикатор достигает одного из крайних пределов, соответствующая экранная кнопка становится недоступной

В табл. 13.3 перечислены все методы, предоставляемые в классе `Bindings` для выполнения операций над свойствами. В одном или обоих аргументах этих методов указывается объект класса, реализующего интерфейс `Observable` или один из его подчиненных интерфейсов. В интерфейсе `Observable` предоставляются методы для ввода и удаления приемника событий типа `InvalidationListener`, в интерфейсе `ObservableValue` — методы для манипулирования приемником событий типа `ChangeListener` и получения текущего значения соответствующего типа. Например, метод `get()` из интерфейса `ObservableStringValue` возвращает объект типа `String`, а метод `get()` из интерфейса `ObservableIntegerValue` — значение типа `int`. Типы значений, возвращаемые методами из класса `Bindings`, относятся к интерфейсам, подчиненным интерфейсу `Binding`, который, в свою очередь, подчинен интерфейсу `Observable`. При этом интерфейсу `Binding` известны все свойства, от которых он зависит.

На практике все упомянутые выше интерфейсы не применяются непосредственно в коде. Чтобы привязать одно свойство к другому, достаточно объединить их вместе.

Таблица 13.3. Методы из класса `Bindings` для операций над свойствами

Методы	Аргументы
<code>add()</code> , <code>subtract()</code> , <code>multiply()</code> , <code>divide()</code> , <code>max()</code> , <code>min()</code>	Объект типа <code>ObservableNumberValue</code> (в качестве первого или второго аргумента), а также значение типа <code>int</code> , <code>long</code> , <code>float</code> , <code>double</code> или еще один объект типа <code>ObservableNumberValue</code>
<code>negate()</code>	Объект типа <code>ObservableNumberValue</code>
<code>greaterThan()</code> , <code>greaterThanOrEqualTo()</code> , <code>lessThan()</code> , <code>lessThanOrEqualTo()</code>	Объект типа <code>ObservableNumberValue</code> , а также значение типа <code>int</code> , <code>long</code> , <code>float</code> , <code>double</code> , или объект типа <code>ObservableNumberValue</code> , или объект типа <code>ObservableStringValue</code> и объект типа <code>String</code> , или же объект типа <code>ObservableStringValue</code>

Методы	Аргументы
<code>equal()</code> , <code>notEqual()</code>	Объект типа <code>ObservableNumberValue</code> , а также значение типа <code>int</code> , <code>long</code> , <code>float</code> , <code>double</code> , тили объект типа <code>ObservableNumberValue</code> , или объект типа <code>ObservableStringValue</code> и объект типа <code>String</code> , или объект типа <code>ObservableStringValue</code> , или объект типа <code>ObservableObjectValue</code> и объект типа <code>Object</code> , или же объект типа <code>ObservableObjectValue</code>
<code>equalIgnoreCase()</code> , <code>notEqualIgnoreCase()</code>	Объект типа <code>ObservableStringValue</code> и объект типа <code>String</code> или объект типа <code>ObservableStringValue</code>
<code>isEmpty()</code> , <code>isNotEmpty()</code>	Объект типа <code>Observable (List Map Set StringValue)</code>
<code>isNull()</code> , <code>isNotNull()</code>	Объект типа <code>ObservableObjectValue</code>
<code>length()</code>	Объект типа <code>ObservableStringValue</code>
<code>size()</code>	Объект типа <code>Observable (List Map Set)</code>
<code>and()</code> , <code>or()</code>	Два объекта типа <code>ObservableBooleanValue</code>
<code>not()</code>	Объект типа <code>ObservableBooleanValue</code>
<code>convert()</code>	Объект типа <code>ObservableValue</code> , преобразуемый в привязку символьных строк
<code>concat()</code>	Последовательность объектов, значения которых сцепляются в виде символьных строк, возвращаемых методом <code>toString()</code> . Если изменяются любые объекты типа <code>ObservableValue</code> , то изменяется и результат сцепления их строковых представлений
<code>format()</code>	Необязательные региональные настройки, строка форматирования типа <code>MessageFormat</code> , а также любая последовательность форматируемых объектов. Если изменяются любые объекты типа <code>ObservableValue</code> , то изменяется и отформатированная строка
<code>valueAt()</code> , <code>[double float integer long]</code> <code>ValueAt()</code> , <code>stringValueAt()</code>	Объект типа <code>ObservableList</code> и индекс или объект типа <code>ObservableMap</code> и ключ
<code>create[Boolean Double Float Integer Long Object String]</code> <code>Binding()</code>	Объект типа <code>Callable</code> и список зависимостей
<code>select()</code> , <code>select[Boolean Double Float Integer Long String]</code>	Объект типа <code>Object</code> или <code>ObservableValue</code> , а также последовательность имен открытых свойств, дающая свойство $obj.p_1.p_2 \dots p_n$
<code>when()</code>	Выдает построитель для условного оператора. Так, привязка <code>when(b).then(v₁).otherwise(v₂)</code> выдает значение v_1 или v_2 в зависимости от того, принимает ли объект b типа <code>ObservableBooleanValue</code> логическое значение <code>true</code> , где v_1 или v_2 могут быть обычными или наблюдаемыми значениями. Условное значение вычисляется повторно всякий раз, когда изменяется наблюдаемое значение

Построение вычисляемого значения с помощью методов из класса `Bindings` может оказаться довольно сложным делом. Впрочем, имеется другой способ получения вычисляемых привязок, который может быть более простым. Он состоит в том,

чтобы ввести вычисляемое выражение непосредственно в лямбда-выражение и предоставить список зависимых свойств. Когда изменяются любые свойства, лямбда-выражение вычисляется повторно, как демонстрируется в следующем примере кода:

```
larger.disableProperty().bind(
    createBooleanBinding(
        () -> gauge.getWidth() >= 100,
        // это выражение вычисляется...
        gauge.widthProperty()));
// ... когда изменяется данное свойство
```



НА ЗАМЕТКУ! В языке JavaFX Script компилятор анализировал выражения привязки и автоматически выявлял зависимые свойства. Для этого было достаточно сделать объявление **larger.disable bind gauge.width >= 100**, чтобы компилятор присоединил приемник событий к свойству **gauge.width**. Программирующий на Java должен, конечно, предоставить подобную информацию.

В листингах 13.18–13.20 приведен весь исходный код программ для трех примеров применения привязок, рассмотренных в этом разделе.

Листинг 13.18. Исходный код из файла `binding/BindingDemo1.java`

```
1 package binding;
2
3 import javafx.application.*;
4 import javafx.scene.*;
5 import javafx.scene.control.*;
6 import javafx.scene.layout.*;
7 import javafx.stage.*;
8
9 /**
10  * @version 1.0 2017-12-29
11  * @author Cay Horstmann
12  */
13 public class BindingDemo1 extends Application
14 {
15     public void start(Stage stage)
16     {
17         TextArea shipping = new TextArea();
18         TextArea billing = new TextArea();
19         billing.textProperty().bindBidirectional(
20             shipping.textProperty());
21         VBox root = new VBox(new Label("Shipping"),
22                               shipping,
23                               new Label("Billing"),
24                               billing);
25         Scene scene = new Scene(root);
26         stage.setScene(scene);
27         stage.show();
28     }
29 }
```

Листинг 13.19. Исходный код из файла `binding/BindingDemo2.java`

```
1 package binding;
2
3 import javafx.application.*;
4 import javafx.beans.binding.*;
5 import javafx.scene.*;
6 import javafx.scene.layout.*;
7 import javafx.scene.paint.*;
8 import javafx.scene.shape.*;
9 import javafx.stage.*;
10
11 /**
12     @version 1.0 2017-12-29
13     @author Cay Horstmann
14 */
15 public class BindingDemo2 extends Application
16 {
17     public void start(Stage stage)
18     {
19         Circle circle = new Circle(100, 100, 100);
20         circle.setFill(Color.RED);
21         Pane pane = new Pane(circle);
22         Scene scene = new Scene(pane);
23         circle.centerXProperty().bind(
24             Bindings.divide(scene.widthProperty(), 2));
25         circle.centerYProperty().bind(
26             Bindings.divide(scene.heightProperty(), 2));
27         stage.setScene(scene);
28         stage.show();
29     }
30 }
```

Листинг 13.20. Исходный код из файла `binding/BindingDemo3.java`

```
1 package binding;
2
3 import static javafx.beans.binding.Bindings.*;
4
5 import javafx.application.*;
6 import javafx.scene.*;
7 import javafx.scene.control.*;
8 import javafx.scene.layout.*;
9 import javafx.scene.paint.*;
10 import javafx.scene.shape.*;
11 import javafx.stage.*;
12
13 /**
14     @version 1.0 2017-12-29
15     @author Cay Horstmann
16 */
17 public class BindingDemo3 extends Application
18 {
19     public void start(Stage stage)
```

```

20 {
21     Button smaller = new Button("Smaller");
22     Button larger = new Button("Larger");
23     Rectangle gauge = new Rectangle(0, 5, 50, 15);
24     Rectangle outline = new Rectangle(0, 5, 100, 15);
25     outline.setFill(null);
26     outline.setStroke(Color.BLACK);
27     Pane pane = new Pane(gauge, outline);
28
29     smaller.setOnAction(
30         event -> gauge.setWidth(gauge.getWidth() - 10));
31     larger.setOnAction(
32         event -> gauge.setWidth(gauge.getWidth() + 10));
33
34     // применение метода из класса Bindings
35
36     smaller.disableProperty().bind(
37         lessThanOrEqual(gauge.widthProperty(), 0));
38
39     // создание привязки из лямбда-выражения
40
41     larger.disableProperty().bind(
42         createBooleanBinding(
43             () -> gauge.getWidth() >= 100,
44             // это выражение вычисляется...
45             gauge.widthProperty())); // ... когда
46             // изменяется данное свойство
47
48     Scene scene = new Scene(new HBox(10, smaller,
49                                     pane, larger));
50     stage.setScene(scene);
51     stage.show();
52 }

```

13.7. Длительные задачи в обратных вызовах пользовательского интерфейса

Применение потоков исполнения объясняется, в частности, тем, что прикладные программы оперативно реагируют на действия пользователя. И это особенно важно в прикладных программах с графическим пользовательским интерфейсом. Когда в прикладной программе требуется выполнить какую-нибудь операцию, потребляющую немало времени, этого нельзя сделать в потоке исполнения пользовательского интерфейса, иначе его работа застопорится. Вместо этого следует запустить еще один рабочий поток исполнения.

Так, если требуется прочитать содержимое файла, как только пользователь щелкнет на экранной кнопке, то не следует поступать так, как показано ниже.

```

var open = new JButton("Open");
open.setOnAction(event ->
{ // НЕУДАЧНО - длительное действие выполняется
  // в потоке исполнения пользовательского интерфейса
  Scanner in = new Scanner(file);
  while (in.hasNextLine())
  {
    String line = in.nextLine();

```



```
    . . .  
  }  
});
```

Вместо этого длительное действие лучше выполнить в отдельном потоке следующим образом:

```
open.setOnAction(event ->  
{ // УДАЧНО - длительное действие выполняется  
  // в отдельном потоке исполнения  
  Runnable task = () ->  
  {  
    Scanner in = new Scanner(file);  
    while (in.hasNextLine())  
    {  
      String line = in.nextLine();  
      . . .  
    }  
  };  
  executor.execute(task);  
});
```

Но пользовательский интерфейс нельзя обновить непосредственно из рабочего потока исполнения, в котором выполняется длительная задача. Пользовательские интерфейсы, построенные на основе библиотеки JavaFX, Swing или платформы Android, не являются потокобезопасными. Манипулировать элементами пользовательского интерфейса из нескольких потоков исполнения нельзя без риска нарушить их нормальное функционирование. На самом деле в JavaFX и Android подобные попытки получить доступ к пользовательскому интерфейсу из другого потока исполнения проверяются, а в итоге генерируется соответствующее исключение.

Таким образом, любые обновления, которые должны происходить в пользовательском интерфейсе, необходимо планировать. В каждой библиотеке для построения пользовательского интерфейса предоставляется свой механизм, позволяющий планировать выполнение задачи, представленной объектом типа `Runnable`, в потоке исполнения пользовательского интерфейса. Например, в библиотеке JavaFX с этой целью делается следующий вызов:

```
Platform.runLater(() -> content.appendText(line + "\n"));
```

Реализовать реакцию пользователя в рабочем потоке исполнения непросто, и поэтому в каждой библиотеке для построения пользовательского интерфейса предоставляется особый вспомогательный класс, берущий на себя все хлопоты, в том числе класс `SwingWorker` в Swing или класс `AsyncTask` в Android. А программисту достаточно указать действия для выполнения длительной задачи (в отдельном потоке) и порядок обновлений по ходу выполнения задачи и окончательное решение (в потоке исполнения пользовательского интерфейса).

Для выполнения длительных задач в библиотеке JavaFX служит класс `Task<V>`. Этот класс удобно расширяет класс `FutureTask<V>`, избавляя от необходимости изучать еще одну специальную языковую конструкцию. В классе `Task` предоставляются методы для обновления определенных свойств задач в рабочем потоке исполнения. Эти свойства сначала привязываются к элементам пользовательского интерфейса, а затем обновляются в потоке исполнения пользовательского интерфейса. И для этой цели имеются следующие свойства:

```
String message
double progress
double workDone
double totalWork
String title
V value
```

В следующей строке кода свойство `message` привязывается к метке состояния:

```
status.textProperty().bind(task.messageProperty());
```

А в рабочем потоке исполнения вызывается метод `updateMessage()`, но не метод `setMessage()`, как показано ниже. Этот метод объединяет изменения в свойстве. Так, если изменения происходят в нескольких свойствах подряд в ускоренном темпе, то в потоке исполнения пользовательского интерфейса оказывается только последнее изменение, произошедшее в соответствующем свойстве.

```
task = new Task<>()
{
    public Integer call()
    {
        while (. . .)
        {
            . . .
            lines++;
            updateMessage(lines + " lines read");
        }
    }
};
```

Чтобы отменить выполнение задачи, следует вызвать метод `cancel` из класса `Task`. В итоге выполнение задачи в потоке будет прервано. Чтобы проверить, отменена ли задача, достаточно периодически вызывать метод `isCanceled()`:

```
task = new Task<>()
{
    public Integer call()
    {
        while (!isCanceled() && !done)
        {
            . . .
        }
    }
};
```

В прикладном коде следует также установить обработчики событий, чтобы уведомлять их, когда запланировано, начинается и завершается (удачно, вследствие исключения или отмены) выполнение задачи, как показано ниже. Все эти обработчики действуют в потоке исполнения пользовательского интерфейса.

```
task.isScheduled(event -> cancel.setDisable(false));
task.setOnRunning(event -> status.setText("Running"));
task.setOnSucceeded(event -> status.setText("Read "
                                + task.getValue()
                                + " lines"));
task.setOnFailed(event -> status.setText("Failed due to "
                                + task.getException()));
task.setOnCancelled(event -> status.setText("Canceled"));
```

В примере программы из листинга 13.21 предоставляются команды для загрузки текстового файла и отмены процесса загрузки. Попробуйте запустить ее вместе с крупным текстовым файлом, например, файлом, содержащим полный англоязычный текст романа “Граф Монте-Кристо” и находящимся в каталоге `gutenberg` загружаемого исходного кода примеров к этой книге. Такой файл загружается в отдельном потоке исполнения, и в этот момент пункт меню `Open` (Открыть) остается недоступным, а пункт меню `Cancel` (Отмена), наоборот, доступным (рис. 13.35). После ввода из файла каждой текстовой строки обновляется счетчик в строке состояния. По завершении процесса загрузки пункт меню `Open` становится доступным, тогда как пункт меню `Cancel` — недоступным, а в строке состояния появляется сообщение “Done” (Готово). Текстовые строки вводятся из файла по очереди с задержкой 10 мс, чтобы стали ясно видны обновления в строке состояния, но на практике этого не следует делать в своих прикладных программах.

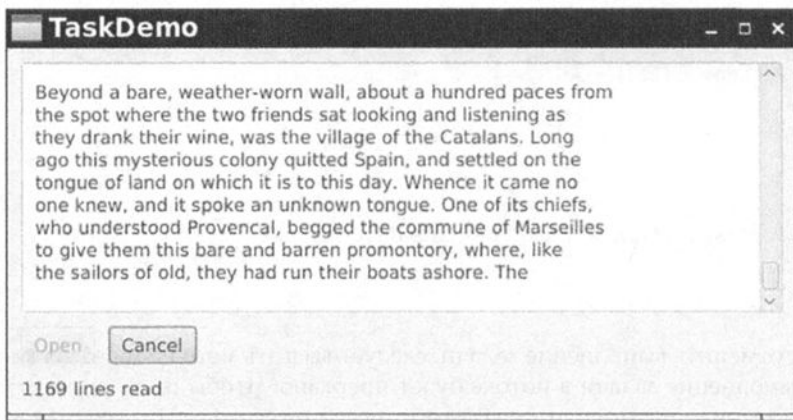


Рис. 13.35. Чтение текстовых строк из файла в рабочем потоке исполнения

Обратите особое внимание на то, что в рабочем потоке исполнения и в потоке исполнения пользовательского интерфейса происходит следующее.

- Метод `call()` вызывается в рабочем потоке исполнения.
- Лямбда-выражение, передаваемое обработчику событий `Platform.runLater()`, выполняется в потоке исполнения пользовательского интерфейса.
- Обработчики событий по планированию, выполнению, неудачному завершению, отмене и удачному завершению задачи действуют в потоке исполнения пользовательского интерфейса.
- В результате вызова метода `updateMessage()` происходит изменение соответствующего свойства в потоке исполнения пользовательского интерфейса и вызывается метод установки значения в привязанном свойстве.

Листинг 13.21. Исходный код из файла `uitask/TaskDemo.java`

```
1 package uitask;
2
3 import java.io.*;
4 import java.nio.charset.*;
```

```
5  import java.util.*;
6  import java.util.concurrent.*;
7
8  import javafx.application.*;
9  import javafx.concurrent.*;
10 import javafx.geometry.*;
11 import javafx.scene.*;
12 import javafx.scene.control.*;
13 import javafx.scene.layout.*;
14 import javafx.stage.*;
15
16 public class TaskDemo extends Application
17 {
18     private TextArea content = new TextArea("");
19     private Label status = new Label();
20     private ExecutorService executor =
21         Executors.newCachedThreadPool();
22     private Task<Integer> task;
23     private Button open = new Button("Open");
24     private Button cancel = new Button("Cancel");
25
26     public void start(Stage stage)
27     {
28         open.setOnAction(event -> read(stage));
29         cancel.setOnAction(event ->
30             {
31                 if (task != null) task.cancel();
32             });
33         cancel.setDisable(true);
34         stage.setOnCloseRequest(event ->
35             {
36                 if (task != null) task.cancel();
37                 executor.shutdown();
38                 Platform.exit();
39             });
40
41         HBox box = new HBox(10, open, cancel);
42         VBox pane = new VBox(10, content, box, status);
43         pane.setPadding(new Insets(10));
44         stage.setScene(new Scene(pane));
45         stage.setTitle("TaskDemo");
46         stage.show();
47     }
48
49     private void read(Stage stage)
50     {
51         if (task != null) return;
52         FileChooser chooser = new FileChooser();
53         chooser.setInitialDirectory(new File(".."));
54         File file = chooser.showOpenDialog(stage);
55         if (file == null) return;
56         content.setText("");
57         task = new Task<>()
58             {
59                 public Integer call()
60                 {
```

```
61         int lines = 0;
62         try (Scanner in = new Scanner(file,
63             StandardCharsets.UTF_8))
64         {
65             while (!isCancelled() && in.hasNextLine())
66             {
67                 Thread.sleep(10); // симитировать задачу
68                 String line = in.nextLine();
69                 Platform.runLater(() ->
70                     content.appendText(line + "\n"));
71                 lines++;
72                 updateMessage(lines + " lines read");
73             }
74         }
75         catch (InterruptedException e)
76         {
77             // задача отменена во время ожидания
78         }
79         catch (IOException e)
80         {
81             throw new UncheckedIOException(null, e);
82         }
83         return lines;
84     }
85 };
86 executor.execute(task);
87 task.setOnScheduled(event ->
88     {
89         cancel.setDisable(false);
90         open.setDisable(true);
91     });
92 task.setOnRunning(event ->
93     {
94         status.setText("Running");
95         status.textProperty().bind(
96             task.messageProperty());
97     });
98 task.setOnFailed(event ->
99     {
100         cancel.setDisable(true);
101         status.textProperty().unbind();
102         status.setText("Failed due to "
103             + task.getException());
104         task = null;
105         open.setDisable(false);
106     });
107 task.setOnCancelled(event ->
108     {
109         cancel.setDisable(true);
110         status.textProperty().unbind();
111         status.setText("Canceled");
112         task = null;
113         open.setDisable(false);
114     });
115 task.setOnSucceeded(event ->
116     {
```

```

117         cancel.setDisable(true);
118         status.textProperty().unbind();
119         status.setText("Done reading "
120             + task.getValue() + " lines");
121         task = null;
122         open.setDisable(false);
123     });
124 }
125 }

```

javafx.application.Platform

- **static void runLater(Runnable runnable)**

Вызывает метод `runnable.run()` в потоке исполнения пользовательского интерфейса.

javafx.concurrent.Task<V>

- **protected abstract V call()**

Переопределяет данный метод для выполнения задачи.

- **boolean cancel()**

Отменяет данную задачу.

- **V getValue()**

Выдает значение, устанавливаемое в методе `updateValue()`, а после удачного завершения задачи — значение, возвращаемое методом `call()`.

- **Throwable getException()**

Выдает исключение, которым завершилось выполнение метода, или же пустое значение `null`, если это исключение не было сгенерировано.

- **void setOnCancelled(EventHandler<WorkerStateEvent> value)**

- **void setOnFailed(EventHandler<WorkerStateEvent> value)**

- **void setOnRunning(EventHandler<WorkerStateEvent> value)**

- **void setOnScheduled(EventHandler<WorkerStateEvent> value)**

- **void setOnSucceeded(EventHandler<WorkerStateEvent> value)**

Устанавливают обработчик событий, связанных с указанным состоянием рабочего потока исполнения.

- **protected void updateMessage(String message)**

- **protected void updateProgress(double workDone, double max)**

- **protected void updateProgress(long workDone, long max)**

- **protected void updateTitle(String title)**

- **protected void updateValue(V value)**

Обновляют указанное свойство. В частности, метод `updateProgress()` устанавливает в свойствах `workDone` и `totalWork` значения его аргументов, а в свойстве `progress` — отношение значений его аргументов. Обновления свойств объединяются для последующей обработки в потоке исполнения пользовательского интерфейса.

Итак, краткое введение в библиотеку JavaFX в этой главе завершено. В библиотеке JavaFX все еще имеются некоторые шероховатости, обусловленные ее ускоренным преобразованием из первоначального языка сценариев. Но пользоваться ею, безусловно, не труднее, чем библиотекой Swing, и в ней имеется намного больше удобных и привлекательных элементов управления, чем когда-либо присутствовало в библиотеке Swing.

На этом первый том настоящего издания завершается. Он был посвящен основам языка программирования Java и тем компонентам стандартной библиотеки, которые требуются в большинстве программных проектов. Надеемся, что этот экскурс в основы Java оказался интересным для вас, и вы извлекли из него немало полезного. А более сложные вопросы, в том числе модульная система на платформе Java, сетевое программирование, расширенные средства программирования GUI, безопасность и интернационализация, будут обсуждаться во втором томе.

Ключевые слова Java

В этом приложении перечислены все ключевые слова языка Java (табл. А.1). Одни ключевые слова имеют ограниченное применение, т.е. имеют специальное назначение только в определенных обстоятельствах (например, в объявлениях модулей), а в остальном могут служить в качестве идентификаторов.

Таблица А.1. Ключевые слова языка Java

Ключевое слово	Что означает	См. главу
abstract	Абстрактный класс или метод	5
assert	Поиск внутренних ошибок в программе	7
boolean	Логический тип данных	3
break	Прерывание оператора switch или цикла	3
byte	8-разрядный целочисленный тип	3
case	Ветвь перехода в операторе switch	3
catch	Оператор блока try для перехвата исключений	7
char	Символьный тип в Юникоде	3
class	Тип класса	4
const	Не используется	
continue	Продолжение выполнения кода в конце цикла	3
default	Ветвь перехода по умолчанию в операторе switch	3, 6
do	Начало цикла do-while	3
double	Числовой тип данных с плавающей точкой двойной точности	3
else	Альтернативная ветвь условного оператора if	3
enum	Перечислимый тип	3
exports	Экспорт пакета из модуля (ограниченное применение)	9 (второй том)
extends	Родительский класс для данного класса	4
final	Константа, класс или метод, который нельзя переопределить	5
finally	Часть блока try , которая выполняется всегда	7
float	Числовой тип данных с плавающей точкой одинарной точности	3
for	Разновидность цикла	3
goto	Не используется	

Окончание табл. А.1

Ключевое слово	Что означает	См. главу
if	Условный оператор	3
implements	Один или несколько интерфейсов, реализуемых в классе	6
import	Импорт пакета	4
instanceof	Проверка, является ли объект экземпляром класса	5
int	32-разрядный целочисленный тип	3
interface	Абстрактный тип с методами, который может быть реализован в классе	6
long	64-разрядный длинный целочисленный тип	3
module	Объявление модуля (ограниченное применение)	9 (второй том)
native	Метод, реализуемый на уровне операционной системы	9 (второй том)
new	Выделение памяти для нового объекта или массива	3
null	Пустая ссылка (формально null является константой, а не ключевым словом)	3
open	Модификация объявления модуля (ограниченное применение)	9 (второй том)
opens	Открытие пакета из модуля (ограниченное применение)	9 (второй том)
package	Пакет классов	4
private	Модификатор доступа только для методов данного класса	4
protected	Модификатор доступа только для методов данного класса, производных от него классов и других классов из того же самого пакета	5
provides	Указание на применение службы в модуле (ограниченное применение)	9 (второй том)
public	Модификатор открытого доступа для методов всех классов	4
return	Возврат из метода	3
short	16-разрядный короткий целочисленный тип	3
static	Модификатор однозначности компонентов класса, а не его объектов	3, 6
strictfp	Строгие правила для вычислений с плавающей точкой	2
super	Объект или конструктор суперкласса	5
switch	Оператор выбора	3
synchronized	Метод или блок кода, атомарный для потока исполнения	12
this	Неявный аргумент метода или конструктора класса	4
throw	Генерирование исключения	7
to	Часть объявления exports или opens (ограниченное применение)	9 (второй том)
throws	Исключения, которые может генерировать метод	7
transient	Обозначение данных, которые не должны быть постоянными	2 (второй том)
transitive	Модификация объявления requires (ограниченное применение)	9 (второй том)
try	Блок кода, в котором могут возникнуть обрабатываемые исключения	7
uses	Указание на применение службы в модуле (ограниченное применение)	9 (второй том)
var	Объявление переменной, тип которой выводится (ограниченное применение)	3
void	Обозначение метода, не возвращающего никаких значений	3
volatile	Указание на то, что поле может быть доступно из нескольких потоков исполнения	14
with	Определение класса службы в операторе provides (ограниченное применение)	9 (второй том)
while	Разновидность цикла	3

Предметный указатель

А

Автоупаковка и распаковка примитивных типов, 239

Алгоритмы обработки

коллекций, реализация, 490

параллельных массивов, 720

решето Эратосфена, реализация, 506

сжатия данных, 182

Аплеты

выполнение в браузере, 28

назначение, 28

Архитектура вилочного соединения

алгоритм перехвата работы, 734

назначение, 733

Б

Библиотеки

AWT

компоненты для обработки

событий, 559

механизм обработки событий, 538

назначение, 510

недостатки, 510

события, разновидности, 558

IFC, назначение, 510

Java 2D, применение, 522

JavaFX

геометрические формы, рисование, 763

история развития, 511

компоновка GUI, 783

назначение, 759

обработка событий, порядок, 772

панели компоновки, назначение, 783

пользовательские данные, задание, 806

свойства, описание, 833

специальные элементы управления,

описание, 828

стилевые таблицы CSS, применение,

795; 797

сцены и подмостки, назначение

и расположение, 760

текст и изображения,

воспроизведение, 767

текст, отображение, 759

типичные элементы управления,

описание, 800; 819

узлы, назначение и расположение, 760

язык FXML, применение, 789–793

Swing

архитектура компонентов по шаблону

MVC, 572

выполнение компонентов в потоке

диспетчеризации событий, 512

история развития, 510

отличия от AWT, 510

разработка, 510

реализация шаблона MVC, 568

коллекций Java

итераторы, особенности, 442

классы, 448

разделение интерфейсов

и реализации, 438

протоколирования, разновидности, 369

рефлексии, назначение, 245

Блоки

вложенные, 93

инициализации, назначение, 167

назначение, 93

область видимости, 93

синхронизированные, назначение, 693

Большие числа

математические операции, 109

определение, 109

Будущие действия

завершение, 736

завершаемые

определение, 737

составление, 738

В

Ввод-вывод

консольный, 84

файловый, 91

чтение вводимых данных, 84

Взаимная блокировка

возникновение, 686

как явление, 699

условия возникновения, 700

Встраивание кода, назначение, 208

Вычисления

асинхронные, организация, 735

параллельные, организация, 680

Г

Глобальные параметры настройки

описание в таблицах свойств, 501

сохранение

в центральном хранилище, 561

потребность, 560

прикладной интерфейс API, 561

Границы
компоновка, 592
назначение, 592
стили оформления, 592

Д

Двухмерные формы
окрашивание цветом
выбор цвета, 529
механизм, 529
установка цвета фона и переднего
плана, 529
построение, 524
Действия
активизация и запрет, 546
изменяющие цвет фона, внедрение, 547
назначение, 546
предопределенные имена, 547
привязка, механизм, 550
Дескрипторы
переменные, применение, 259
процессов, описание, 755
Диалоговые окна
для выбора разных вариантов
вывод сообщений, 637
компоненты, 636
создание, 638
типы вариантов, 637
для выбора файлов
вспомогательный компонент, 656
создание, 652
для обмена данными, создание, 645
кнопка по умолчанию, установка, 647
модальные
назначение, 636
создание, 641
немодальные, назначение, 636
признак модальности, указание, 642
простые
назначение, 819
разновидности, 819
расширяемое содержимое,
отображение, 820
селектора шрифтов, компоновка, 622
фрейм-владелец, указание, 642
Диспетчеры
компоновки
границной, назначение, 574
групповой, назначение, 622
поточной, назначение, 572
по умолчанию, назначение, 574
сеточно-контейнерной, назначение, 622
собственные, установка, 574
специальные, создание, 632
функции, 573
протоколирования
инициализация, 372
смена, 372
Документация, составление
и комментирование, 187

З

Загрузчики служб
инициализация, 330
назначение, 329
реализация, 329

И

Изображения
ввод из файлов, 536
вывод
в окне, 536
рядами, 537
Импорт
классов, 173
статический, 174
Инкапсуляция
основной принцип, 127
поля экземпляра и методы, 127
преимущества, 148
Интегрированные среды разработки
Eclipse
выявление ошибок компиляции, 47
применение, 45
NetBeans
групповая компоновка GUI, 622
применение, 37
назначение, 45
Интерфейсы
Action
методы, 546
назначение, 546
реализация, 547
ActionListener
метод actionPerformed(),
реализация, 540
объявление, 286
реализация, 286; 538
AutoCloseable, реализация и метод, 355
Binding, назначение, 837
ButtonModel
методы, 591
реализация, 570
свойства, 570
Callable, назначение и метод, 723
ChangeListener, назначение, 834
ChangeListener, реализация и метод, 598
Cloneable
назначение, 292
реализация, 293
Collection
методы, 440; 445
обобщенные служебные методы, 443
расширение, 441
Comparable
метод compareTo(), назначение, 276
обобщение, 274
определение, 274
Comparator
методы, 311

- назначение и реализация, 289
- CompletionStage, реализация и методы, 741
- Condition, методы, 689
- Deque, реализация и методы, 468
- Enumeration, реализация и методы, 500
- ExecutorService
 - методы, 726–732
 - реализация, 725
- Filter, реализация и метод, 378
- Future, назначение и методы, 723
- InvalidationListener, назначение, 834
- InvocationHandler
 - метод обработки вызовов, 332
 - реализация, 332
- Iterable, реализация и метод, 441
- Iterator
 - методы, 441; 446
 - обобщенные служебные методы, 443
 - реализация, 440
- LayoutManager2, реализация и методы, 632
- LayoutManager, реализация и методы, 632
- ListIterator, реализация и методы, 452; 458
- List, реализация и методы, 446; 457; 458
- Lock, методы, 683; 689
- Map
 - методы, 472
 - реализация, 470
- MouseListener, назначение и методы, 614
- MouseListener, реализация и методы, 554
- MouseMotionListener, реализация и методы, 553
- NavigableMap, реализация и методы, 448; 489
- NavigableSet, реализация и методы, 467; 483; 489
- ObservableValue, назначение и методы, 837
- Observable, назначение и методы, 837
- Predicate, назначение и определение, 301
- ProcessHandle
 - методы, 755
 - назначение, 755
- Queue
 - методы, 467
 - реализация, способы, 439
- RandomAccess, назначение и реализация, 447
- Runnable
 - метод, 662; 667
 - реализация, 662
- ScheduledExecutorService, реализация и методы, 726
- Set, реализация и методы, 448
- Shape, реализация, 522
- SortedMap
 - методы, 489
 - назначение, 448
- SortedSet
 - методы, 466; 483; 489
 - назначение, 448
- Supplier<T>, назначение и определение, 302
- SwingConstants, реализация и константы, 580
- Thread.UncaughtExceptionHandler, реализация и метод, 674
- TransferQueue
 - методы, 712
 - реализация, 707
- Type, иерархия наследования, 427
- WindowListener, реализация и методы, 544
- коллекций
 - применение, 490
 - разделение, 438
 - разновидности, 446
- константы, объявление, 280
- маркерные, назначение, 292
- методы с реализацией по умолчанию
 - назначение, 283
 - разрешение конфликтов, 284
- очереди, назначение, 438
- порядок реализаций, 275
- приемников событий, определение, 538
- функциональные
 - аннотирование, преимущества, 311
 - для примитивных типов, 310
 - наиболее употребительные, 309
 - определение, 300
 - преобразование, 300
- Исключения
 - возникновение, 342
 - генерирование
 - объявление, 343
 - повторное, 351
 - порядок, 346
 - условия, 248; 343
 - заклчение в оболочку, 352
 - классификация, 341
 - необработываемые, обработчики, 674
 - непроверяемые, определение, 248; 343
 - обработка
 - механизм, назначение, 340
 - организация, 249
 - описание, 344
 - освобождение ресурсов в блоке оператора finally, 353
 - передача на обработку, 363
 - перехват
 - в блоке операторов try/catch, 348
 - нескольких исключений, 350
 - одного исключения, 348
 - проверяемые
 - объявление, 343
 - определение, 248; 343
 - преодоление ограничений на обработку, 413

собственных типов, генерирование, 347
цепочки, связывание, 351
Исполнители
 применение, 725
 управление группами задач, 727

К

Клавиши
 быстрого доступа, назначение, 611; 813
 оперативные, назначение, 612; 813
Классы
 AbstractAction, назначение и методы, 547
 AbstractCollection
 методы абстрактные и служебные, 444
 расширение, 444
 реализуемые методы, 455
 Array
 методы, 263
 назначение, 263
 ArrayDeque, конструкторы, 468
 ArrayList
 методы, 233
 назначение, 232
 применение, 459
 Arrays, назначение и методы, 115; 116;
 263; 488
 BasicButtonUI, назначение, 572
 BigDecimal, применение, 109
 BigInteger
 применение, 109
 Bindings
 методы, 837
 назначение, 836
 BorderFactory, конструкторы
 и методы, 593
 BorderLayout, константы и методы, 575
 BoxLayout, назначение, 621
 ButtonUIListener, назначение, 572
 ChoiceDialog, назначение, 821
 Class
 методы, 246; 252; 425; 435
 назначение, 246
 обобщение, 425
 Collections
 методы, 483; 487; 493; 494
 реализуемые алгоритмы, 495
 Collections, методы, 495
 Color
 константы для выбора цвета, 529
 применение, 529
 ComboBox<T>, назначение, 807
 CompletableFuture, назначение, 735
 CompletableFuture, назначение
 и методы, 738
 Component
 иерархия наследования, 573
 методы, 514
 ConcurrentHashMap, назначение
 и конструкторы, 713

ConcurrentSkipListSet, назначение
 и конструкторы, 713
Console, применение, 85
Constructor
 назначение и методы, 252; 426
 обобщение, 426
Container, назначение, 573
CopyOnWriteArrayList, назначение, 720
CopyOnWriteArraySet, назначение, 720
Date
 методы, 131
 назначение, 131
DefaultButtonModel, назначение, 570
DirectoryChooser, назначение, 823
Employee
 анализ реализации, 142
 методы, 141
 наследование, 196
 поля и методы, 147
 создание, 139
EnumMap
 конструктор, 480
 назначение, 479
EnumSet
 методы, 480
 назначение, 478
Error, иерархия наследования, 341
Exception, иерархия наследования, 342
ExecutorCompletionService
 конструктор и методы, 732
 назначение, 728
Executors
 исполнители пулов потоков, 725
 методы, 726
 фабричные методы, 725
Field, назначение и методы, 251
FileChooser
 методы, 822
 назначение, 822
FileFilter, назначение и методы, 654
FileHandler, конструкторы, 386
FileView, назначение и методы, 654
Font
 задание параметров шрифта
 в конструкторе, 531
 объекты для выделения
 шрифтами, 531
Formatter, методы, 387
FutureTask
 конструкторы, 724
 назначение, 724
Graphics
 методы, 519; 536
 объекты для рисования графики, 518
Graphics2D
 методы, 522; 528
 применение, 522
GregorianCalendar, назначение
 и методы, 136
GridBagConstraints

- конструктор, 631
- параметры компоновки, 631
- GridLayout, методы, 577
- HashMap
 - конструкторы, 472
 - назначение, 470
- HashSet
 - конструкторы, 462
 - методы, 461
 - назначение, 461
- IdentityHashMap
 - конструкторы, 480
 - назначение, 479
- InputMap, назначение, 550
- Integer
 - методы, 240
 - объектной оболочки типа int, 239
- JButton
 - как оболочка, 572
 - обращение к модели кнопки, 572
- JCheckBoxMenuItem, конструкторы, 609
- JCheckBox, конструкторы и методы, 588
- JComboBox
 - методы, 595
 - назначение, 594
 - обобщение, 594
- JComponent
 - методы, 518; 579
 - привязки ввода, 549
 - расширение, 518
- JDialog, конструктор, 645
- JFileChooser
 - методы, 657
- JFileChooser, назначение и методы, 651
- JFrame
 - иерархия наследования, 514
 - методы, 513
 - назначение, 511
- JLabel
 - конструкторы, 581
 - методы, 581
 - параметры конструктора, 580
- JMenuItem, конструкторы и методы, 606; 613
- JMenu, конструктор и методы, 606
- JOptionPane
 - методы, 636; 639
 - назначение, 636
- JPasswordField
 - конструктор и методы, 581
 - назначение, 581
- JPopupMenu, методы, 610
- JRadioButtonMenuItem, конструкторы, 609
- JRadioButton, конструкторы и методы, 591
- JScrollPane, назначение и конструктор, 585
- JSlider, конструкторы и методы, 603
- JTextArea, конструкторы и методы, 584
- JTextComponent, назначение и методы, 578
- JTextField
 - конструктор, задание параметров, 578
 - методы, 579
- JToolBar, конструкторы и методы, 620
- KeyStroke, назначение, 548
- LineBorder, конструктор, 594
- LinkedHashMap
 - конструкторы и методы, 480
 - назначение, 477
- LinkedHashSet
 - конструкторы, 479
 - назначение, 477
- LinkedList
 - методы, 452
 - назначение, 450
- LocalDate
 - методы, 134
 - назначение, 134
- LogRecord, методы, 386
- LongAccumulator, назначение, конструктор и методы, 698
- LongAdder, назначение и методы, 698
- Manager
 - конструкторы, 199
 - определение, 196
 - поля и методы, 197
- Math
 - инкапсуляция только функциональных возможностей, 131
 - константы, 66
 - математические функции, 65
 - методы, 65; 67
- Method, назначение и методы, 252; 435
- Node
 - методы, 768
 - назначение, 760
 - свойства, 833
- Object
 - методы, 218; 225; 291; 693
 - назначение, 217
- Pair
 - методы, 397
 - назначение, 396
 - объявление переменных типа, 396
- PasswordChooser, назначение, 646
- Point2D, назначение и подклассы, 524
- Preferences
 - методы, 561; 565
 - реализация центрального хранилища, 561
- PriorityQueue, конструкторы, 469
- Process
 - методы, 754; 756
 - назначение, 751
- ProcessBuilder
 - методы, 756
 - назначение, 751
- Properties

- конструкторы, 503
- методы, 503
- Properties, назначение и методы, 502
- Proxy
 - методы, 332
 - расширение прокси-классами, 336
- Rectangle2D
 - иерархия наследования, 523
 - методы, 523
- RectangularShape
 - иерархия наследования, 524
 - методы, 524
- ReentrantLock
 - защита блока кода, 681
 - конструкторы, 684
- RuntimeException, исключения и ошибки, 342
- Scanner, методы, 84
- ServiceLoader, назначение, 329
- SoftBevelBorder, конструкторы и методы, 594
- StackTraceElement, методы, 360
- StackWalker, назначение, 357
- StrictMath, математические функции, 67
- String
 - для символьных строк, 72
 - методы, 77
- StringBuilder
 - методы, 83
 - применение, 82
- StringProperty, назначение и методы, 833
- SwingWorker
 - методы, 745; 751
 - назначение, 745
- Task, назначение и методы, 842
- TextArea, назначение, 800
- TextField, назначение, 800
- TextInputControl, назначение и методы, 800
- TextInputDialog, назначение, 821
- Thread
 - конструкторы, 666
 - методы, 666; 670; 673; 675; 676
 - не рекомендованные к применению
 - методы, 703
 - расширение, 664
- ThreadGroup, назначение, 674
- ThreadLocal
 - методы, 703
 - назначение, 702
- ThreadLocalRandom, назначение и методы, 702
- ThreadPoolExecutor
 - методы, 727
 - назначение, 725
- Throwable
 - иерархия наследования, 341
 - конструкторы и методы, 347; 358
- Timer
 - конструктор, применение, 287
 - назначение, 286
- Toolkit, назначение и методы, 515
- TreeMap
 - конструкторы, 473
 - назначение, 470
- TreeSet
 - конструкторы, 466
 - назначение, 463
- WeakHashMap
 - конструкторы, 479
 - назначение, 476
- WebView, назначение, 829
- Window, назначение и методы, 515
- абстрактные, назначение, 211
- адаптеров
 - назначение, 544
 - создание, 545
- внутренние
 - анонимные, особенности, 322
 - локальные, особенности, 320; 321
 - назначение, 312
 - ссылки на внешние классы, 314
 - статические, особенности, 325
- для рисования двумерных форм, 522; 524
- идентификация, 128
- импорт, 172
- исключений
 - иерархия наследования, 341
 - создание, 347
- коллекций
 - абстрактные, назначение, 440
 - из библиотеки Java, 448
 - унаследованные, 499–505
- конечные
 - назначение, 207
 - определение, 150
- назначение, 52
- наследование, механизм, 196
- низкоуровневых событий, 559
- обобщенные
 - правила наследования типов, 416
 - применение, 395
 - реализация, трудности, 395
- объектных оболочек, разновидности, 239
- отношения, 129
- перечислений
 - назначение, 243
 - наследование и методы, 243
 - получение экземпляров, 126
 - порядок именования, 52
- потокобезопасные, синхронизирующая оболочка, 721
- пути
 - определение, 179
 - указание, 181
- расширение, 127
- свойств, описание, 834
- семантических событий, 559
- суперклассы и подклассы, 196

Ключевые слова

abstract, назначение, 212
assert, назначение и основные формы, 364
class, назначение, 52
extends, назначение, 196; 400
final, назначение, 63; 208
implements, назначение, 275
import, применение, 172
static, назначение, 153
strictfp, назначение, 65
super
 назначение, 198
 применение, 199
synchronized, назначение, 681; 689
this
 назначение, 146; 166
 применение, 167; 199
var, применение, 144
volatile, назначение, 696
языка Java, перечень, 849
Кнопки-переключатели
внешние отличия от флажков, 589
группы, 588
компоновка, 588
применение, 805
Кнопки экранные
анализ по шаблону MVC, 570
компоновка на панели, 572
модель, описание, 570
Коллекции
алгоритмы
 двоичного поиска, применение, 494
 простые, применение, 495
 реализация, 490
 собственные, написание, 498
 сортировки и перетасовки, применение, 491
выбор реализации, 440
групповые операции, применение, 497
и массивы, взаимное преобразование, 497
интерфейсы, разновидности, 440; 446
итераторы
 особенности, 442
 применение, 441
мелкие, разновидности и применение, 481
необязательные операции, методика, 486
ограниченные, 440
отсортированные, назначение, 463
перебор элементов, 442
поддиапазоны
 операции, 482
 формирование, 482
поиск элементов, 459
построение, 440
потокобезопасные
 поддержка читающих и записывающих потоков, 712
 разновидности, 705

слабо совместные итераторы, возврат, 712
устаревшие, 721
представления
немодифицируемые, получение, 483
определение, 481
поддиапазонов, 482
применение, 481
проверяемые, назначение, 485
синхронизированные, назначение, 484
удаление элементов, 443
упорядоченные
 организация, 446
 особенности доступа, 447
Командная строка
выполнение графического приложения, 43
компиляция и запуск программ, 41
параметры, описание, 186
режим работы, особенности, 42
Комментарии
документирующие
 гипертекстовые ссылки, 191
 дескрипторы, разновидности, 189; 190
 извлечение в каталог, 191
 порядок составления, 188
к классам, составление, 188
к методам, составление, 189
к полям, составление, 189
обзорные, составление, 192
способы выделения в коде, 55
Компараторы
обращение, 312
определение, 289
связывание в цепочку, 311
создание, 311
Компиляторы
Java, порядок вызова, 41
динамические, назначение, 26; 208
обнаружение классов в пакетах, 174
традиционные, назначение, 27
Комплект JDK
загрузка, 36
задание пути к исполняемым файлам, 38
установка, 37
Комплекты ресурсов
ввод сопоставлений, 373
включение в состав программ, 373
назначение, 373
Компоненты GUI
автоматическая перерисовка, 519
блочная компоновка, особенности, 621
границы, задание, 592
граничная компоновка, 574
групповая компоновка, 622
диалоговые окна, назначение, 636
для ввода текста, описание, 577; 800
для выбора разных вариантов, описание, 585; 804
для рисования графики, создание, 518

М

- комбинированные списки, создание, 594
 - меню, назначение, 604
 - панели
 - инструментов, назначение, 618
 - прокрутки, 583
 - пометка, 580
 - поточная компоновка, 572
 - пружинная компоновка, особенности, 621
 - расположение в контейнерах, 573
 - регулируемые ползунки, разновидности, 598
 - сегочная компоновка, назначение, 576
 - сеточно-контейнерная компоновка
 - весовые поля, определение, 624
 - внешнее и внутреннее заполнение, параметры, 625
 - заполнение и привязка компонентов, параметры, 624
 - наложение ограничений, вспомогательный класс, 626
 - описание, 623
 - особенности, 621
 - принцип действия, 622
 - расположение компонентов, параметры, 624
 - рекомендации, 625
 - составляющие и свойства, 568
 - Константы
 - класса, назначение, 63
 - назначение, 61
 - обозначение, 63
 - статические, назначение, 152
 - Конструкторы
 - без аргументов, назначение, 164
 - вызов
 - одних из других, 167
 - порядок действий, 168
 - именование параметров, 166
 - локальные переменные, 144
 - особенности, 143
 - перегрузка, 163
 - Курсоры
 - в JavaFX, формы, 777
 - для Windows, виды, 553
- Л**
- Литералы типов, применение, 432
 - Лямбда-выражения
 - захват значений переменных, 307
 - как замыкания, 307
 - определение, 296
 - отложенное выполнение, реализация, 308
 - преобразование в функциональные интерфейсы, 301
 - применение, 298
 - синтаксис, 298
 - составляющие, 307
- Манифест
 - главный класс, указание, 184
 - назначение, 183
 - разделы, разновидности, 183
 - редактирование, 183
 - файл, назначение, 183
 - Массивы
 - анонимные, 112
 - доступ к элементам, порядок, 113
 - доступ по индексу, 112
 - копирование, 115
 - копируемые при записи, 720
 - многомерные, описание, 119
 - назначение, 111
 - неровные, описание, 123
 - объявление, 112
 - определение, 112
 - параллельные операции, особенности, 720
 - размер
 - выделение памяти, 233
 - задание, 231
 - свойство length, назначение, 113
 - сортировка, 116
 - списочные
 - добавление элементов, 233
 - доступ к элементам, 234
 - емкость, 233
 - обобщенные, 232
 - применение, 459
 - создание, 232
 - Меню
 - всплывающие
 - построение, 609
 - триггер, назначение, 610
 - контекстные
 - назначение, 814
 - создание, 814
 - подменю и пункты, назначение, 604; 811
 - построение, 605; 811
 - пункты
 - разрешение и запрет доступа, 613
 - с кнопками-переключателями, 608
 - с пиктограммами, 607
 - с флажками, 608
 - строка, назначение, 604; 811
 - Метки
 - назначение, 580
 - размещение в контейнере, 580
 - с пиктограммой и текстом надписи, задание, 580
 - с текстом надписи, задание, 580
 - Методы
 - clone()
 - применение, 293
 - реализация, 293
 - equals()
 - назначение, 218

- рекомендации по реализации, 221
- характеристики, 220
- hashCode()
 - назначение, 223
 - усовершенствование, 224
- main()
 - назначение, 53
 - объявление, 53
 - особенности, 154
- printf()
 - назначение, 87
 - переменное число параметров, 242
- setVisible(), назначение и вызов, 645
- toString()
 - назначение, 225
 - основания для реализации, 226
 - универсальные, реализация, 259
- абстрактные, назначение, 212
- вызов
 - обозначение, 54
 - по значению, 157
 - по имени, 157
 - по ссылке, 157
- доступа
 - к полям, 147
 - назначение, 136
- закрытые, применение в интерфейсах, 283
- защищенные, применение, 217
- идентификация, 129
- именование параметров, 166
- ковариантные возвращаемые типы, 206
- конечные, определение, 207
- модифицирующие, назначение, 136
- мостовые, назначение, 405
- обобщенные
 - вызов, 398
 - определение, 398
 - служебные, назначение, 443
 - сопоставление типов, 427
- открытые и закрытые, отличия, 150
- параметры и аргументы, 54
- перегрузка, 164
- переопределение, 197
- платформенно-ориентированные, назначение и применение, 152
- сигнатура
 - назначение, 205
 - определение, 164
- синхронизированные, назначение, 690
- с переменным числом параметров, 242
- с реализацией по умолчанию
 - назначение, 283
 - применение, 284
- статические
 - в интерфейсах, применение, 282
 - назначение, 153
 - применение, 153
- тело, обозначение, 54
- типы параметров, разновидности, 158
- фабричные, назначение, 154
- явные и неявные параметры, 146

- Множества
 - битовые, реализация, 505
 - древовидные
 - ввод элементов, 463
 - как отсортированные коллекции, 463
 - структура красно-черного дерева, 463
 - на основе хеш-таблиц, 461
 - определение, 461
 - параллельные, представления, 719
 - перечислимые, реализация, 478
- Модификаторы доступа
 - default, назначение, 283
 - final, назначение, 150; 207; 697
 - private, назначение, 150
 - protected, назначение, 216
 - public, назначение, 150
 - static, назначение, 151
 - volatile, назначение, 697
 - назначение, 52
 - разновидности, 217
- Мониторы
 - принцип, реализация в Java, 695
 - свойства, 695

Н

- Наследование
 - иерархия и цепочки, 203
 - как принцип ООП, 195
 - обозначение, 196
 - предотвращение, 207
 - приведение типов, правила, 210
 - признак, 196
 - применение, 196
 - принцип подстановки, 204
 - рекомендации по применению, 270

О

- Обобщения
 - захват подстановок, механизм, 423
 - и виртуальная машина, 402
 - накладываемые ограничения, 407; 415; 416
 - неограниченные подстановки, назначение, 422
 - ограничения супертипа
 - на подстановки, 419
 - параметры типа, назначение, 394
 - подстановочные типы, назначение, 395
 - подстановочные типы, описание, 417
 - преобразование, особенности, 405
 - стирание типов, механизм, 402
 - экземпляры обобщенного типа, получение, 397
- Обратные вызовы, назначение, 286
- Объектные оболочки
 - классы, 239
 - применение, 240
 - примитивных типов, 239
- Объекты
 - блокировки, назначение, 681

действия, свойства, 547
 исключений, назначение, 350
 клонирование, особенности, 292
 ключевые свойства, 128
 копирование
 неполное, 291
 полное, 292
 назначение, 126
 отличия от мониторов, 695
 присваивание переменным, 131
 событий, назначение, 537; 557
 создание экземпляров, 131
 уничтожение, 171
 условий, назначение, 684
Операторы
 break
 без метки, применение, 106
 с меткой, применение, 107
 continue
 применение, 108
 с меткой, назначение, 108
 import, применение, 173
 package, применение, 175
 switch
 метки ветвей case, 105
 принцип действия, 104
 throws, назначение, 343
 throw, назначение, 346
 try с ресурсами
 назначение, 355
 формы, 355
 задания блоков, применение, 94
 ромбовидные, назначение, 232
 составные, применение, 94
Операции
 instanceof, назначение, 209
 new, назначение, 133
 арифметические, 64
 атомарность, обеспечение, 697
 инкремента и декремента, 69
 логические, 70
 назначение, 64
 отношения, 70
 поразрядные, логические, 70
 приоритетность, 71
 сравнения, 70
 тернарные, 70
Отладка программ
 в среде JUnit, 388
 отладчики, применение в IDE, 387
 рекомендации, 388
Отложенное вычисление, поставщики, 302
Отображения
 ввод элементов, 470
 назначение, 470
 обновление записей, 473
 параллельные
 атомарное обновление записей, 714
 групповые операции,
 разновидности, 717

порог параллелизма, указание, 717
 перечислимые, реализация, 479
 представления
 получение, 474
 разновидности, 475
 применение, 471
 разновидности реализации, 470
 связные хеш-отображения,
 назначение, 478
 слабые хеш-отображения,
 назначение, 476
 удаление элементов, 471
 хеш-отображения идентичности,
 построение, 479
Очереди
 блокирующие
 классы, описание, 706
 методы, разновидности, 705
 назначение, 705
 интерфейс, 438
 односторонние и двусторонние, 467
 по приоритету
 организация в виде 'кучи', 468
 применение, 469
 применение, 438
 принцип действия, 438
 реализация, 438
Ошибки
 возврат кода или признака, 341
 инкапсуляция в объектах, 341
 порядок обработки, 340
 причины появления, 340
 программные, разновидности, 342
 разновидности, 340

П

Пакеты
 java.awt.event, 559
 java.beans, 832
 java.lang.reflect, 251
 java.util.concurrent, 691; 706; 712; 722
 java.util.concurrent.atomic, 697
 java.util.function, 301
 java.util.preferences, 560
 javax.swing, 512
 javax.swing.filechooser, 654
 безымянные, назначение, 175
 ввод классов, 174
 доступ к классам, способы, 172
 именование, порядок, 172
 назначение, 172
 область видимости, 177
 размещение, 175
Панели
 инструментов
 всплывающие подсказки, 620
 назначение, 618
 обособленные, 619
 перетаскивание, 619
 построение, 619

- компоновки
 - назначение, 783
 - применение, 784
 - разновидности, 786
- прокрутки
 - ввод компонентов GUI, 583
 - назначение, 583
- Пароли
 - ввод с консоли, 85
 - поля для ввода, 581
- Перегрузка
 - назначение, 163
 - разрешение, 163; 205
- Переменные
 - действительно конечные
 - назначение, 308
 - объявление, 321
 - инициализация, 62
 - локальные
 - инициализация, 164
 - объявление с помощью ключевого слова var, 144
 - назначение, 61
 - объектные
 - инициализация, 132
 - особенности, 132
 - полиморфные, 204
 - объявление, 61
 - разделяемые, способы доступа, 697
 - типа
 - именование, 397
 - объявление, 397
 - ограничения, 399
- Перечисления, реализация, 500
- Полиморфизм
 - как принцип ООП, 204
 - механизм, 204
 - определение, 200
- Поля
 - для ввода пароля, компоновка, 581
 - защищенные, применение, 217
 - инициализация
 - явная, 165
 - конечные
 - назначение, 150
 - определение, 208
 - открытые и закрытые, назначение, 147
 - разделяемые, способы доступа, 696
 - статические, назначение, 151
 - текстовые
 - ввод и расположение, 578
 - задание размеров, 578
 - правка содержимого, 579
 - экземпляра, назначение, 127
- Потоки исполнения
 - блокированные, состояние, 668
 - блокировка по условию, 685
 - в GUI, применение, 744; 841
 - временно ожидающие, состояние, 668
 - встроенная блокировка, 689
 - группы, назначение, 674
 - завершенные, состояние, 669
 - именование, 674
 - исполняемые, состояние, 667
 - клиентская блокировка, 694
 - набор ожидания, 685
 - новые, состояние, 667
 - ожидающие, состояние, 668
 - определение, 661
 - отличие от процессов, 661
 - плановое исполнение, 726
 - превращение в потоковые демоны, 673
 - прерывание, 670
 - приоритеты, 675
 - пулы, назначение, 723
 - равноправная блокировка, 684
 - реентрабельная блокировка, 682
 - синхронизация
 - по принципу монитора, 695
 - путем блокировок и условий, 682
 - состояния, разновидности, 667
 - счетчик захватов блокировки, 682
 - установка и проверка состояния прерывания, 670
 - установка и снятие блокировки, 682
- Предисловие
 - наложение, 367
 - нарушение, 367
 - определение, 367
- Приведение типов
 - объектов, процесс, 209
 - определение и обозначение, 68
 - примитивных, процесс, 68
 - при наследовании, правила, 210
- Привязки
 - ввода
 - назначение, 549
 - получение, 549
 - порядок проверки условий, 549
 - условия, 549
 - действий к компонентам GUI, 550
- Примеры программ
 - загрузка, 19
 - установка, 40
- Программирование
 - обобщенное
 - назначение, 394
 - реализация, способы, 394
 - уровни квалификации, 395
 - объектно-ориентированное
 - инкапсуляция, 127
 - наследование, 128; 195
 - основные понятия, 126
 - особенности, 126
 - полиморфизм, механизм, 204
 - параллельное, особенности, 661
 - структурное, особенности, 126
- Проектный шаблон MVC
 - взаимодействие составляющих, 570

назначение, 569
преимущества, 569
Прокси-классы
методы, 332
назначение, 331
обработчики вызовов, назначение, 332
прокси-объекты, создание
и применение, 332
протоколирующие прокси-объекты,
назначение, 388
свойства, 336
Протоколирование
вывод и хранение протокольных
записей, 375
иерархия регистраторов, 369
конфигурирование, 372
легковесная система, описание, 369
локализация протокольных
сообщений, 373
обработчики протоколов
в файлах, параметры настройки, 376
собственные, определение, 376
уровни, 374
установка, 374
прикладной интерфейс API,
преимущества, 368
рекомендации по выполнению
операций, 378
удобные методы, 370
уровни, описание, 370
усовершенствованное, организация, 369
файлы протоколов
именование по шаблону, 375
ротация, 376
форматирование протокольных
записей, 378
элементарное, организация, 369
Процессы
выполнение, 753
дескрипторы
применение, 755
способы получения, 755
отличие от потоков исполнения, 662
построение, 752
потоки ввода-вывода, назначение, 752
рабочий каталог, назначение, 752
уничтожение, 754
Пулы потоков исполнения
организация, 726
применение, 723

Р

Развертывание приложений
JAR-файлы
запуск приложений, особенности, 184
многоверсионные, назначение, 185
назначение, 182
создание, 182
ресурсы
применение, 250

размещение, 250
рановидности, 249
Регулируемые ползунки
дополнение отметками и привязка
к ним, 598
компоновка, 598
назначение, 598
Ресурсы, порядок освобождения, 171; 355
Рефлексия
анализ
объектов во время выполнения, 257
структуры классов, 251
вызов произвольных методов
и конструкторов, 266
классы и методы, 251
манипулирование массивами, 263
механизм, 245
обобщенных типов, 427
применение, 245

С

Свойства
в JavaFX
методы доступа, 833
реализация, 833
в Java, поддержка, 832
задач, назначение и привязка, 842
назначение, 832
привязки
в JavaBeans, определение, 833
в JavaFX, назначение, 835
механизм, 836
приемники событий, разновидности, 834
применение в прикладных
программах, 503
таблицы
назначение, 501
применение, 501
Связывание
динамическое, определение, 200; 206
статическое, определение, 206
Символьные строки
в Java, особенности, 72
выделение подстрок, 72
построение, 82
принцип постоянства, 73
пустые и нулевые, 76
сцепление, 73
Системные сбои, механизмы
обработки, 366
Смешанное написание, назначение, 52
События
в окне, обработка, 544
действия, назначение, 538
инкапсуляция в объектах, 537
источники и приемники, 537
краткое обозначение приемников, 543
механизм обработки, 538
низкоуровневые, 558
от клавиатуры

- обработка, 778
- фокус ввода, запрос, 778
- от мыши
 - обработка, 551; 554; 775
 - разновидности, 554
- от щелчков на экранных кнопках, обработка, 539
- семантические, 558
- Состояние гонок
 - как явление, 676
 - причины возникновения, 679
- Списки
 - комбинированные
 - компоновка, 595
 - назначение, 594
 - применение, 806
 - раскрывающиеся
 - назначение, 594
 - редактируемые, назначение, 594
 - связные
 - ввод элементов, 452
 - двунаправленные, 450
 - итераторы, особенности, 454
 - как упорядоченные коллекции, 452
 - организация, 450
 - применение, 456
 - реализация, 450
 - структурные модификации, отслеживание, 454
 - удаление элементов, 453
- Ссылки
 - на конструкторы
 - массивов, применение, 306
 - назначение, 305
 - применение, 306
 - на методы
 - назначение, 302
 - разновидности, 303
 - пустые, обработка, способы, 145
- Стеки, реализация, 504

Т

- Таблицы
 - методов, создание, 206
 - свойств, особенности, 501
- Таймеры
 - передача объекта для обратного вызова, 286
 - установка, 286
- Текстовые области
 - автоматический перенос строк, 582
 - компоновка, 582
 - назначение, 582
 - на панели прокрутки, 583
- Типы данных
 - boolean, 60
 - char, 58
 - динамическая идентификация, механизм, 246
 - контейнерные, применение, 241

- обобщенные и базовые, соответствие, 402
- перечислимые, применение, 64; 243
- примивные, 55
- целочисленные, 56
- числовые
 - с плавающей точкой, 57
- Трассировка стека
 - определение, 356
 - получение, 357

У

- Управляющие последовательности специальных символов, 58
- Условные операторы if
 - общая форма, 95
 - повторяющиеся, 96
- Утверждения
 - документирование
 - предположений, 367
 - назначение, 364
 - проверка параметров метода, 366
 - разрешение и запрет, 365
 - условия для применения, 366
- Утилиты
 - jar
 - параметры, 182
 - применение, 179; 182
 - javac, применение, 41
 - javadoc, применение, 187
 - javap, применение, 318
 - java, применение, 41
 - jconsole, применение, 373; 391
 - JShell
 - запуск, 48
 - назначение, 48
 - применение, 49

Ф

- Фигурные скобки
 - назначение, 53
 - стиль расстановки, 53
- Флажки
 - компоновка, 586
 - применение, 804
 - установка и сброс, 585
- Фокус ввода
 - отображение, 549
 - перемещение, 549
 - с клавиатуры, 549; 778
- Форматирование
 - выводимых данных, 86
 - даты и времени, 88
 - символы преобразования, 87
 - спецификатор формата, 87
 - флаги, 87
- Фреймы
 - вывод данных в компоненте, 517
 - задание размеров, 515
 - определение, 511

отображение, 513
расположение, 513
свойства, 515
создание, 511
структура, описание, 517

Х

Хеш-коды
 вычисление, алгоритм, 223; 459
 определение, 459
 порождение, 459
Хеш-множества
 ввод и вывод элементов, 461
 итераторы, применение, 461
Хеш-таблицы
 группы, 460
 коэффициент загрузки, 461
 назначение, 459
 повторное хеширование, 461
 реализация, 460
 связные, 477
хеш-конфликты, явление, 460

Ц

Циклы
 do-while, принцип действия, 97
 for, принцип действия, 100
 while, принцип действия, 97
 в стиле for each
 организация, 441
 применение, 114; 441
 принцип действия, 114

Ш

Шрифтовое оформление
 выравнивание текста

 по центру, 531
 рамка, ограничивающая текст, 532
 типографские характеристики, 532
Шрифты
 гарнитура, определение, 530
 доступные в системе, 530
 логические названия, 531
 начертание, определение, 530
 размеры в пунктах, 531
 семейства, наименования, 767

Ю

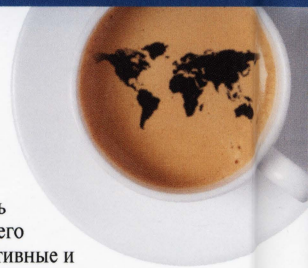
Юникод
 кодировка
 UTF-16, 60
 назначение, 59
 кодовые
 единицы, 60
 плоскости, 60
 точки, 60
 область подстановки, 60

Я

Язык Java
 версии, описание, 31
 компилятор, 41
 краткая история развития, 29
 особенности, 22; 27
 отсутствие поддержки множественного
 наследования, причины, 282
 программная платформа, 22
 программные средства, обозначение, 36
 распространенные заблуждения, 32
 система безопасности, 24
 строго типизированный, 55
 учет регистра букв, 52

Исчерпывающее руководство по Java для серьезных программистов!

Полностью обновлено по версиям Java SE 9, 10 и 11



Это одиннадцатое издание исчерпывающего руководства по написанию надежного, удобно сопровождаемого прикладного кода. Какой бы версией вы ни пользовались, будь то Java SE 9, 10 или 11, эта книга дает глубокое и практическое понимание языка Java и его интерфейса API, а сотни приведенных в ней реальных примеров демонстрируют эффективные и действенные способы решения практических задач прикладного программирования.

Обновленные в этой книге примеры кода отражают долгожданные возможности модуляризации, а также наглядно показывают, как писать легко расширяемый и сопровождаемый прикладной код на языке Java. Читая книгу, вы узнаете, как пользоваться новой утилитой JShell, реализующей цикл REPL для ускоренного освоения языка Java и экспериментальной разработки прикладных программ, практического применения усовершенствований в прикладном интерфейсе Process API, состязательной блокировке, протоколировании и компиляции.

В первом томе настоящего двухтомного издания главное внимание уделяется основным понятиям языка Java и средствам программирования пользовательского интерфейса, включая принципы ООП, обобщения, коллекции, лямбда-выражения, разработку графических программ средствами библиотеки Swing, а также методики параллельного и функционального программирования. Если у вас имеется достаточный опыт программирования на Java и вы собираетесь перейти к версии Java SE 9, 10 или 11, то лучшего руководства по компетентному подходу к решению практических задач программирования на Java вам не найти.

ОСНОВНЫЕ ТЕМЫ КНИГИ

- Быстрое освоение основных методик и норм наилучшей практики для написания высококачественного кода на Java
- Владение интерфейсами, внутренними классами и лямбда-выражениями для функционального программирования
- Повышение надежности прикладных программ благодаря обработке исключений и эффективной отладке
- Написание более безопасного и удобочитаемого исходного кода приложений с использованием обобщений
- Повышение производительности и эффективности прикладных программ с помощью стандартных коллекций в Java
- Построение современных межплатформенных графических приложений с использованием стандартных компонентов библиотеки Swing
- Использование в полной мере вычислительных ресурсов многоядерных процессоров с помощью усовершенствованных в Java функциональных средств параллелизма

Подробное рассмотрение более развитых языковых средств Java, включая функциональные средства корпоративного уровня в версиях Java SE 9, 10 и 11, модульную систему, работу в сети, вопросы безопасности и усовершенствованного программирования графических приложений, предлагается во втором томе настоящего издания.

КЕЙ ХОРСТМАНН — профессор факультета вычислительной техники в Университете Сан-Хосе, обладатель звания “Чемпион по Java” и частый докладчик на многих отраслевых конференциях. Автор настоящего двухтомного издания, а также книг *Scala for Impatient, Second Edition* (издательство Addison-Wesley, 2017 г.) и *Core Java SE 9 for the Impatient* (издательство Addison-Wesley, 2018 г.; в русском переводе книга вышла под названием *Java SE 9. Базовый курс* в издательстве “Диалектика”, 2018 г.). Он написал также более десятка других книг специально для профессиональных программистов и студентов, изучающих курсы по компьютерным наукам.

Категория: программирование

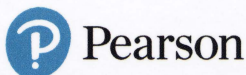
Предмет рассмотрения: язык Java, версии SE 9, 10 и 11

Уровень: промежуточный/продвинутый

www.informit.com/java/horstmann.com



www.williamspublishing.com



ISBN 978-5-907114-79-1



9 785907 114791