Содержание

Предисловие 29

Часть I. Язык Java 33

Глава 1. История и эволюция языка Java 34

Происхождение Java 34
Зарождение современного программирования: язык C 35
C++: следующий шаг 37
Условия для появления языка Java 38
Создание языка Java 38
Связь с языком C# 41
Влияние языка Java на Интернет 41
Аплеты Java 41
Безопасность 42
Переносимость 42
Магия Java: байт-код 43
Выход за рамки аплетов 44
Более быстрый график выпуска 45
Сервлеты: Java на серверной стороне 46
Терминология языка Java 47
Простота 47
Объектная ориентация 47
Надежность 48
Многопоточность 49
Нейтральность к архитектуре 49
Интерпретируемость и высокая производительность 49
Распределенность 49
Динамичность 50
Эволюция языка Java 50
Культура инноваций 57

Глава 2. Краткий обзор языка Java 58

Объектно-ориентированное программирование 58
Две парадигмы 58
Абстракция 59
Три принципа ООП 60
Первая простая программа 66
Ввод кода программы 66
Компиляция программы 67
Подробный анализ первого примера программы 68
Вторая простая программа 70
Два управляющих оператора 72
Оператор `if` 72
Цикл `for` 73

Использование блоков кода 75
Лексические вопросы 76
Пробельные символы 76
Идентификаторы 76
Литералы 77
Комментарии 77
Разделители 77
Ключевые слова Java 77
Библиотеки классов Java 79

Глава 3. Типы данных, переменные и массивы 80
Java — строго типизированный язык 80
Примитивные типы 80
Целые числа 81
Тип `byte` 82
Тип `short` 82
Тип `int` 82
Тип `long` 83
Типы с плавающей точкой 83
Тип `float` 84
Тип `double` 84
Символы 85
Булевские значения 86
Подробный анализ литералов 87
Целочисленные литералы 87
Литералы с плавающей точкой 88
Булевские литералы 89
Символьные литералы 89
Строковые литералы 90
Переменные 91
Объявление переменной 91
Динамическая инициализация 92
Область видимости и время жизни переменных 92
Преобразование и приведение типов 95
Автоматические преобразования в Java 95
Приведение несовместимых типов 95
Автоматическое повышение типов в выражениях 97
Правила повышения типов 98
Массивы 99
Одномерные массивы 99
Многомерные массивы 101
Альтернативный синтаксис объявления массивов 105
Знакомство с выведением типов локальных переменных 106
Некоторые ограничения `var` 108
Несколько слов о строках 109

Глава 4. Операции 110
Арифметические операции 110
Основные арифметические операции 111
Операция деления по модулю 112
Составные арифметические операции присваивания 112
Операции инкремента и декремента 113

Побитовые операции 115
Побитовые логические операции 116
Сдвиг влево 119
Сдвиг вправо 120
Беззнаковый сдвиг вправо 121
Составные побитовые операции присваивания 123
Операции отношения 124
Булевские логические операции 125
Короткозамкнутые логические операции 126
Операция присваивания 127
Операция ? 128
Старшинство операций 129
Использование круглых скобок 130

Глава 5. Управляющие операторы 131
Операторы выбора Java 131
Оператор if 131
Традиционный оператор switch 134
Операторы итерации 140
Цикл while 140
Цикл do-while 141
Цикл for 144
Версия цикла for в стиле “for-each” 148
Выведение типов локальных переменных в цикле for 153
Вложенные циклы 154
Операторы перехода 155
Использование оператора break 155
Использование оператора continue 159
Оператор return 161

Глава 6. Введение в классы 162
Основы классов 162
Общая форма класса 162
Простой класс 163
Объявление объектов 166
Подробный анализ операции new 166
Присваивание для переменных ссылок на объекты 168
Введение в методы 169
Добавление метода в класс Box 169
Возвращение значения 171
Добавление метода, принимающего параметры 173
Конструкторы 175
Параметризованные конструкторы 177
Ключевое слово this 178
Сокрытие переменных экземпляра 178
Сборка мусора 179
Класс Stack 180

Глава 7. Подробный анализ методов и классов 183
Перегрузка методов 183
Перегрузка конструкторов 186
Использование объектов в качестве параметров 188

Подробный анализ передачи аргументов 190
Возвращение объектов 192
Рекурсия 193
Введение в управление доступом 195
Ключевое слово `static` 199
Ключевое слово `final` 201
Снова о массивах 201
Вложенные и внутренние классы 203
Исследование класса `String` 206
Использование аргументов командной строки 208
Аргументы переменной длины 209
Перегрузка методов с аргументами переменной длины 212
Аргументы переменной длины и неоднозначность 214
Выведение типов локальных переменных для ссылочных типов 215

Глава 8. Наследование 217

Основы наследования 217
Доступ к членам и наследование 219
Более реалистичный пример 220
Переменная типа суперкласса может ссылаться на объект подкласса 222
Использование ключевого слова `super` 223
Использование ключевого слова `super` для вызова конструкторов суперкласса 223
Использование второй формы ключевого слова `super` 226
Создание многоуровневой иерархии 227
Когда конструкторы выполняются 230
Переопределение методов 231
Динамическая диспетчеризация методов 233
Зачем нужны переопределенные методы? 235
Применение переопределения методов 236
Использование абстрактных классов 237
Использование ключевого слова `final` с наследованием 240
Использование ключевого слова `final` для предотвращения переопределения 240
Использование ключевого слова `final` для предотвращения наследования 241
Выведение типов локальных переменных и наследование 241
Класс `Object` 243

Глава 9. Пакеты и интерфейсы 245

Пакеты 245
Определение пакета 246
Поиск пакетов и `CLASSPATH` 247
Краткий пример пакета 247
Пакеты и доступ к членам классов 248
Пример, демонстрирующий использование модификаторов доступа 250
Импортирование пакетов 252
Интерфейсы 254
Определение интерфейса 255
Реализация интерфейсов 256
Вложенные интерфейсы 259

Применение интерфейсов 260
Переменные в интерфейсах 263
Интерфейсы можно расширять 265
Стандартные методы интерфейса 266
Основы стандартных методов 267
Более реалистичный пример 269
Проблемы множественного наследования 269
Использование статических методов в интерфейсе 271
Закрытые методы интерфейса 271
Заключительные соображения по поводу пакетов и интерфейсов 273

Глава 10. Обработка исключений 274
Основы обработки исключений 274
Типы исключений 275
Неперехваченные исключения 276
Использование `try` и `catch` 277
Отображение описания исключения 279
Использование нескольких конструкций `catch` 279
Вложенные операторы `try` 281
Оператор `throw` 283
Конструкция `throws` 284
Конструкция `finally` 286
Встроенные исключения Java 287
Создание собственных подклассов `Exception` 289
Сцепленные исключения 292
Три дополнительных средства в системе исключений 294
Использование исключений 295

Глава 11. Многопоточное программирование 296
Потоковая модель Java 297
Приоритеты потоков 298
Синхронизация 299
Обмен сообщениями 300
Класс `Thread` и интерфейс `Runnable` 300
Главный поток 301
Создание потока 303
Реализация интерфейса `Runnable` 303
Расширение класса `Thread` 305
Выбор подхода 306
Создание множества потоков 306
Использование `isAlive()` и `join()` 308
Приоритеты потоков 310
Синхронизация 311
Использование синхронизированных методов 312
Оператор `synchronized` 314
Взаимодействие между потоками 316
Взаимоблокировка 320
Приостановка, возобновление и останов потоков 322
Получение состояния потока 325
Использование фабричных методов для создания и запуска потока 326
Использование многопоточности 327

Глава 12. Перечисления, автоупаковка и аннотации 328
Перечисления 328
Основы перечислений 329
Методы `values()` и `valueOf()` 331
Перечисления Java являются типами классов 332
Перечисления унаследованы от `Enum` 334
Еще один пример перечисления 336
Оболочки типов 337
Класс `Character` 338
Класс `Boolean` 338
Оболочки числовых типов 339
Автоупаковка 341
Автоупаковка и методы 342
Автоупаковка/автораспаковка и выражения 342
Автоупаковка/автораспаковка типов `Boolean` и `Character` 344
Автоупаковка/автораспаковка помогает предотвратить ошибки 345
Предостережение 346
Аннотации 346
Основы аннотаций 346
Указание политики хранения 347
Получение аннотаций во время выполнения с использованием рефлексии 348
Интерфейс `AnnotatedElement` 353
Использование стандартных значений 354
Маркерные аннотации 355
Одноэлементные аннотации 356
Встроенные аннотации 357
Аннотации типов 360
Повторяющиеся аннотации 364
Некоторые ограничения 366

Глава 13. Ввод-вывод, оператор `try` с ресурсами и другие темы 367
Основы ввода-вывода 367
Потоки данных 368
Потоки байтовых и символьных данных 368
Предопределенные потоки данных 371
Чтение консольного ввода 372
Чтение символов 373
Чтение строк 374
Запись консольного вывода 376
Класс `PrintWriter` 376
Чтение файлов и запись в файлы 378
Автоматическое закрытие файла 384
Модификаторы `transient` и `volatile` 388
Введение в `instanceof` 388
Модификатор `strictfp` 391
Собственные методы 391
Использование `assert` 392
Параметры включения и отключения проверки утверждений 395
Статическое импортирование 395
Вызов перегруженных конструкторов через `this()` 398
Несколько слов о классах, основанных на значениях 400

Глава 14. Обобщения 401
Что такое обобщения? 402
Простой пример обобщения 402
Обобщения работают только со ссылочными типами 406
Обобщенные типы различаются на основе их аргументов типов 407
Каким образом обобщения улучшают безопасность в отношении типов? 407
Обобщенный класс с двумя параметрами типов 409
Общая форма обобщенного класса 411
Ограниченные типы 411
Использование аргументов с подстановочными знаками 413
Ограниченные аргументы с подстановочными знаками 416
Создание обобщенного метода 421
Обобщенные конструкторы 424
Обобщенные интерфейсы 424
Низкоуровневые типы и унаследованный код 427
Иерархии обобщенных классов 429
Использование обобщенного суперкласса 429
Обобщенный подкласс 431
Сравнение типов в обобщенной иерархии во время выполнения 432
Приведение 434
Переопределение методов в обобщенном классе 435
Выведение типов и обобщения 436
Выведение типов локальных переменных и обобщения 437
Стирание 438
Мостовые методы 438
Ошибки неоднозначности 440
Некоторые ограничения обобщений 441
Невозможность создать экземпляры параметров типов 441
Ограничения, касающиеся статических членов 442
Ограничения, касающиеся обобщенных массивов 442
Ограничения, касающиеся обобщенных исключений 443

Глава 15. Лямбда-выражения 444
Введение в лямбда-выражения 444
Основы лямбда-выражений 445
Функциональные интерфейсы 446
Примеры лямбда-выражений 447
Блочные лямбда-выражения 451
Обобщенные функциональные интерфейсы 453
Передача лямбда-выражений в качестве аргументов 454
Лямбда-выражения и исключения 457
Лямбда-выражения и захват переменных 458
Ссылки на методы 459
Ссылки на статические методы 459
Ссылки на методы экземпляра 461
Ссылки на методы и обобщения 464
Ссылки на конструкторы 467
Предопределенные функциональные интерфейсы 471

Глава 16. Модули 473
Основы модулей 473
Простой пример модуля 474
Компиляция и запуск первого примера модуля 478
Более подробный анализ операторов requires и exports 480
Модуль java.base и модули платформы 481
Унаследованный код и неименованные модули 482
Экспортирование в конкретный модуль 483
Использование requires transitive 485
Использование служб 489
Основы служб и поставщиков служб 489
Ключевые слова, связанные со службами 490
Пример службы, основанной на модулях 491
Графы модулей 497
Три специальных характерных черты модулей 498
Открытые модули 498
Оператор opens 499
Оператор requires static 499
Введение в jlink и файлы модулей JAR 499
Связывание файлов в развернутом каталоге 500
Связывание модульных файлов JAR 500
Файлы JMOD 501
Кратко об уровнях и автоматических модулях 502
Заключительные соображения по поводу модулей 502

Глава 17. Выражения switch, записи и прочие недавно добавленные средства 503
Расширения оператора switch 504
Использование списка констант case 505
Появление выражения switch и оператора yield 506
Появление стрелки в операторе case 509
Подробный анализ оператора case со стрелкой 510
Еще один пример выражения switch 514
Текстовые блоки 514
Основы текстовых блоков 515
Ведущие пробельные символы 516
Использование двойных кавычек в текстовом блоке 517
Управляющие последовательности в текстовых блоках 518
Записи 519
Основы записей 520
Создание конструкторов записи 522
Еще один пример конструктора записи 526
Создание методов получения для записи 528
Сопоставление с образцом в операции instanceof 530
Шаблонные переменные в логических выражениях “И” 531
Сопоставление с образцом в других операторах 532
Запечатанные классы и запечатанные интерфейсы 533
Запечатанные классы 533
Запечатанные интерфейсы 535
Будущие направления развития 537

Часть II. Библиотека Java 539

Глава 18. Обработка строк 540

Конструкторы класса String 541
Длина строки 543
Специальные строковые операции 543
Строковые литералы 543
Конкатенация строк 544
Конкатенация строк с другими типами данных 544
Преобразование в строку и toString() 545
Извлечение символов 546
charAt() 546
getChars() 547
getBytes() 547
toCharArray() 547
Сравнение строк 548
equals() и equalsIgnoreCase() 548
regionMatches() 549
startsWith() и endsWith() 549
equals() или == 549
compareTo() 550
Поиск в строках 552
Модификация строк 553
substring() 553
concat() 554
replace() 554
trim() и strip() 555
Преобразование данных с использованием valueOf() 556
Изменение регистра символов внутри строк 556
Соединение строк 557
Дополнительные методы класса String 558
Класс StringBuffer 561
Конструкторы класса StringBuffer 561
length() и capacity() 561
ensureCapacity() 562
setLength() 562
charAt() и setCharAt() 562
getChars() 563
append() 563
insert() 564
reverse() 564
delete() и deleteCharAt() 565
replace() 566
substring() 566
Дополнительные методы класса StringBuffer 566
Класс StringBuilder 568

Глава 19. Исследование пакета java.lang 569

Оболочки примитивных типов 570
Number 570
Double и Float 570
Методы isInfinite() и isNaN() 576

Byte, Short, Integer и Long 576
Character 591
Дополнения класса Character для поддержки кодовых точек Unicode 594
Boolean 596
Void 598
Process 598
Runtime 599
Выполнение других программ 601
Runtime.Version 602
ProcessBuilder 604
System 608
Использование currentTimeMillis() для хронометража выполнения программы 610
Использование arraycopy() 611
Свойства среды 612
System.Logger и System.LoggerFinder 612
Object 612
Использование метода clone() и интерфейса Cloneable 613
Class 615
ClassLoader 621
Math 621
Тригонометрические функции 621
Экспоненциальные функции 622
Функции округления 623
Прочие методы Math 625
StrictMath 628
Compiler 628
Thread, ThreadGroup и Runnable 628
Интерфейс Runnable 628
Класс Thread 628
Класс ThreadGroup 632
ThreadLocal и InheritableThreadLocal 636
Package 636
Module 638
ModuleLayer 639
RuntimePermission 639
Throwable 639
SecurityManager 639
StackTraceElement 640
StackWalker и StackWalker.StackFrame 641
Enum 641
Record 642
ClassValue 643
Интерфейс CharSequence 643
Интерфейс Comparable 644
Интерфейс Appendable 644
Интерфейс Iterable 644
Интерфейс Readable 645
Интерфейс AutoCloseable 645
Интерфейс Thread.UncaughtExceptionHandler 646

Подпакеты java.lang 646
java.lang.annotation 646
java.lang.constant 646
java.lang.instrument 646
java.lang.invoke 647
java.lang.management 647
java.lang.module 647
java.lang.ref 647
java.lang.reflect 647

Глава 20. Пакет java.util, часть 1: Collections Framework 648

Обзор Collections Framework 649
Интерфейсы коллекций 651
Интерфейс Collection 652
Интерфейс List 655
Интерфейс Set 658
Интерфейс SortedSet 659
Интерфейс NavigableSet 660
Интерфейс Queue 662
Интерфейс Deque 663
Классы коллекций 666
Класс ArrayList 667
Класс LinkedList 671
Класс HashSet 672
Класс LinkedHashSet 674
Класс TreeSet 674
Класс PriorityQueue 675
Класс ArrayDeque 676
Класс EnumSet 677
Доступ в коллекцию через итератор 679
Использование итератора 680
Альтернатива итераторам в виде цикла for в стиле “for-each” 682
Сплитераторы 683
Хранение объектов пользовательских классов в коллекциях 686
Интерфейс RandomAccess 688
Работа с картами 688
Интерфейсы карт 688
Классы карт 697
Компараторы 703
Использование компаратора 705
Алгоритмы коллекций 711
Массивы 719
Унаследованные классы и интерфейсы 724
Интерфейс Enumeration 725
Класс Vector 725
Класс Stack 730
Класс Dictionary 732
Класс Hashtable 733
Класс Properties 737
Использование методов store() и load() 740
Заключительные соображения по поводу коллекций 742

Глава 21. Пакет java.util, часть 2: дополнительные служебные классы 743
Класс StringTokenizer 743
BitSet 745
Optional, OptionalDouble, OptionalInt и OptionalLong 749
Date 753
Calendar 755
GregorianCalendar 759
TimeZone 761
SimpleTimeZone 762
Locale 763
Random 764
Timer и TimerTask 767
Currency 770
Formatter 771
Конструкторы класса Formatter 772
Методы класса Formatter 772
Основы форматирования 773
Форматирование строк и символов 776
Форматирование чисел 776
Форматирование времени и даты 777
Спецификаторы %n и %% 779
Указание минимальной ширины поля 779
Указание точности 781
Использование флагов формата 782
Выравнивание выводимых данных 782
Флаги пробела, +, 0 и (783
Флаг запятой 784
Флаг # 784
Версии в верхнем регистре 784
Использование индекса аргумента 785
Закрытие объекта Formatter 787
Альтернативный вариант: метод printf() 787
Scanner 787
Конструкторы класса Scanner 788
Основы сканирования 789
Примеры использования класса Scanner 794
Установка разделителей 798
Дополнительные средства класса Scanner 799
ResourceBundle, ListResourceBundle и PropertyResourceBundle 800
Смешанные служебные классы и интерфейсы 805
Подпакеты пакета java.util 807
java.util.concurrent, java.util.concurrent.atomic и java.util.concurrent.locks 807
java.util.function 807
java.util.jar 811
java.util.logging 812
java.util.prefs 812
java.util.random 812
java.util.regex 812
java.util.spi 812
java.util.stream 812
java.util.zip 812

Глава 22. Ввод-вывод: исследование пакета java.io 813
Классы и интерфейсы ввода-вывода 814
File 814
Каталоги 818
Использование интерфейса FilenameFilter 819
Альтернативные методы listFiles() 820
Создание каталогов 820
Интерфейсы AutoCloseable, Closeable и Flushable 821
Исключения ввода-вывода 821
Два способа закрытия потока данных 822
Классы потоков данных 824
Байтовые потоки 824
InputStream 824
OutputStream 826
FileInputStream 827
FileOutputStream 829
ByteArrayInputStream 831
ByteArrayOutputStream 832
Фильтрующие байтовые потоки 834
Буферизованные байтовые потоки 834
SequenceInputStream 838
PrintStream 840
DataOutputStream и DataInputStream 842
RandomAccessFile 844
Символьные потоки 845
Reader 845
Writer 847
FileReader 848
FileWriter 848
CharArrayReader 850
CharArrayWriter 851
BufferedReader 852
BufferedWriter 853
PushbackReader 854
PrintWriter 855
Класс Console 856
Сериализация 858
Serializable 859
Externalizable 859
ObjectOutput 859
ObjectOutputStream 860
ObjectInput 862
ObjectInputStream 862
Пример сериализации 864
Преимущества потоков 867

Глава 23. Исследование системы NIO 868
Классы NIO 868
Основы NIO 869
Буферы 869
Каналы 873
Наборы символов и селекторы 875

Усовершенствования, появившиеся в NIO.2 875
Интерфейс `Path` 875
Класс `Files` 878
Класс `Paths` 881
Интерфейсы для файловых атрибутов 882
Классы `FileSystem`, `FileSystems` и `FileStore` 885
Использование системы NIO 885
Использование системы NIO для ввода-вывода, основанного на каналах 886
Использование системы NIO для ввода-вывода, основанного на потоках 896
Использование системы NIO для операций с путями и файловой системой 898

Глава 24. Работа в сети 907

Основы работы в сети 907
Классы и интерфейсы пакета `java.net` для работы в сети 909
`InetAddress` 910
Фабричные методы 910
Методы экземпляра 911
`Inet4Address` и `Inet6Address` 912
Клиентские сокеты TCP/IP 912
`URL` 916
`URLConnection` 917
`HttpURLConnection` 920
Класс `URI` 922
Cookie-наборы 923
Серверные сокеты TCP/IP 923
Дейтаграммы 924
`DatagramSocket` 924
`DatagramPacket` 925
Пример использования дейтаграмм 926
Введение в пакет `java.net.http` 928
Три ключевых элемента 928
Простой пример клиента HTTP 931
Что еще рекомендуется изучить в `java.net.http` 933

Глава 25. Обработка событий 934

Два механизма обработки событий 935
Модель делегирования обработки событий 935
События 936
Источники событий 936
Прослушиватели событий 937
Классы событий 937
Класс `ActionEvent` 938
Класс `AdjustmentEvent` 940
Класс `ComponentEvent` 941
Класс `ContainerEvent` 942
Класс `FocusEvent` 942
Класс `InputEvent` 943
Класс `ItemEvent` 944
Класс `KeyEvent` 945
Класс `MouseEvent` 946
Класс `MouseWheelEvent` 947

Класс `TextEvent` 948
Класс `WindowEvent` 949
Источники событий 950
Интерфейсы прослушивателей событий 951
Интерфейс `ActionListener` 952
Интерфейс `AdjustmentListener` 952
Интерфейс `ComponentListener` 952
Интерфейс `ContainerListener` 952
Интерфейс `FocusListener` 953
Интерфейс `ItemListener` 953
Интерфейс `KeyListener` 953
Интерфейс `MouseListener` 953
Интерфейс `MouseMotionListener` 954
Интерфейс `MouseWheelListener` 954
Интерфейс `TextListener` 954
Интерфейс `WindowFocusListener` 954
Интерфейс `WindowListener` 954
Использование модели делегирования обработки событий 955
Основные концепции графических пользовательских интерфейсов AWT 955
Обработка событий мыши 956
Обработка событий клавиатуры 960
Классы адаптеров 963
Внутренние классы 966
Анонимные внутренние классы 968

Глава 26. Введение в AWT: работа с окнами, графикой и текстом 970

Классы AWT 971
Основы окон 974
`Component` 974
`Container` 975
`Panel` 975
`Window` 975
`Frame` 975
`Canvas` 975
Работа с окнами `Frame` 976
Установка размеров окна 976
Скрытие и отображение окна 976
Установка заголовка окна 976
Закрытие фреймового окна 977
Метод `paint()` 977
Отображение строки 977
Установка цветов фона и переднего плана 978
Запрос перерисовки 978
Создание приложения на основе `Frame` 980
Введение в графику 980
Вычерчивание линий 981
Вычерчивание прямоугольников 981
Вычерчивание эллипсов и окружностей 981
Вычерчивание дуг 982
Вычерчивание многоугольников 982
Демонстрация работы методов вычерчивания 982
Установка размеров графики 983

Работа с цветом 985
Методы класса Color 986
Установка текущего цвета графики 987
Программа, демонстрирующая работу с цветом 987
Установка режима рисования 988
Работа со шрифтами 990
Выяснение доступных шрифтов 992
Создание и выбор шрифта 993
Получение информации о шрифте 995
Управление выводом текста с использованием FontMetrics 996

Глава 27. Использование элементов управления, диспетчеров компоновки и меню AWT 1001
Основы элементов управления AWT 1002
Добавление и удаление элементов управления 1002
Реагирование на события, генерируемые элементами управления 1003
Исключение HeadlessException 1003
Метки 1003
Использование кнопок 1005
Обработка событий для кнопок 1005
Использование флажков 1009
Обработка событий для флажков 1010
Группы флажков 1012
Элементы управления выбором 1014
Обработка событий для списков выбора 1015
Использование списков 1016
Обработка событий для списков 1018
Управление полосами прокрутки 1019
Обработка событий для полос прокрутки 1021
Использование текстовых полей 1023
Обработка событий для текстовых полей 1024
Использование текстовых областей 1026
Понятие диспетчеров компоновки 1028
FlowLayout 1029
BorderLayout 1030
Использование вставок 1031
GridLayout 1033
CardLayout 1034
GridBagLayout 1037
Меню и панели меню 1043
Диалоговые окна 1048
Несколько слов о переопределении метода paint() 1052

Глава 28. Изображения 1053
Форматы файлов 1053
Основы работы с изображениями: создание, загрузка и отображение 1054
Создание объекта изображения 1054
Загрузка изображения 1055
Отображение изображения 1055
Двойная буферизация 1057
ImageProducer 1060
MemoryImageSource 1060

ImageConsumer 1062
PixelGrabber 1062
ImageFilter 1065
CropImageFilter 1065
RGBImageFilter 1067
Дополнительные классы для обработки изображений 1078

Глава 29. Утилиты параллелизма 1079

Пакеты параллельного API 1080
java.util.concurrent 1081
java.util.concurrent.atomic 1082
java.util.concurrent.locks 1082
Использование объектов синхронизации 1082
Semaphore 1083
CountDownLatch 1088
CyclicBarrier 1090
Exchanger 1092
Phaser 1095
Использование исполнителя 1103
Простой пример использования исполнителя 1104
Использование интерфейсов Callable и Future 1105
Перечисление TimeUnit 1108
Параллельные коллекции 1109
Блокировки 1110
Атомарные операции 1113
Параллельное программирование с помощью Fork/Join Framework 1114
Главные классы Fork/Join Framework 1115
Стратегия “разделяй и властвуй” 1119
Простой пример использования Fork/Join Framework 1121
Влияние уровня параллелизма 1123
Пример использования RecursiveTask<V> 1126
Выполнение задачи асинхронным образом 1129
Отмена задачи 1129
Определение состояния завершения задачи 1130
Перезапуск задачи 1130
Дальнейшие исследования 1130
Советы по использованию Fork/Join Framework 1132
Сравнение утилит параллелизма и традиционного подхода к многопоточности в Java 1133

Глава 30. Потоковый API-интерфейс 1134

Основы потоков 1134
Потоковые интерфейсы 1135
Получение потока 1138
Простой пример использования потока 1139
Операции редукции 1142
Использование параллельных потоков 1145
Сопоставление 1147
Накопление 1151
Итераторы и потоки 1155
Использование итератора с потоком 1155
Использование сплитератора 1156
Дальнейшее исследование потокового API 1159

Глава 31. Регулярные выражения и другие пакеты 1160
Обработка регулярных выражений 1160
Класс `Pattern` 1161
Класс `Matcher` 1161
Синтаксис регулярных выражений 1162
Демонстрация сопоставления с шаблоном 1163
Два варианта сопоставления с шаблоном 1168
Дальнейшее исследование регулярных выражений 1169
Рефлексия 1169
Удаленный вызов методов 1174
Простое клиент-серверное приложение, использующее удаленный вызов методов 1174
Форматирование даты и времени с помощью пакета `java.text` 1178
Класс `DateFormat` 1178
Класс `SimpleDateFormat` 1180
Пакеты `java.time`, поддерживающие API даты и времени 1182
Фундаментальные классы для поддержки даты и времени 1182
Форматирование даты и времени 1184
Разбор строк с датой и временем 1187
Дальнейшее исследование пакета `java.time` 1188

Часть III. Введение в программирование графических пользовательских интерфейсов с помощью Swing Java 1189

Глава 32. Введение в Swing 1190
Происхождение инфраструктуры Swing 1190
Инфраструктура Swing построена на основе AWT 1191
Две ключевые особенности Swing 1191
Компоненты Swing являются легковесными 1192
Инфраструктура Swing поддерживает подключаемый внешний вид 1192
Связь с архитектурой MVC 1192
Компоненты и контейнеры 1194
Компоненты 1194
Контейнеры 1195
Панели контейнеров верхнего уровня 1195
Пакеты Swing 1196
Простое приложение Swing 1196
Обработка событий 1201
Рисование в Swing 1204
Основы рисования 1205
Вычисление области рисования 1206
Пример программы рисования 1206

Глава 33. Исследование Swing 1210
`JLabel` и `ImageIcon` 1210
`JTextField` 1212
Кнопки Swing 1214
`JButton` 1214
`JToggleButton` 1217
Флажки 1219
Взаимоисключающие переключатели 1221

JTabbedPane 1223
JScrollPane 1226
JList 1227
JComboBox 1231
Деревья 1233
JTable 1236

Глава 34. Введение в меню Swing 1240

Основы меню 1240
Обзор JMenuBar, JMenu и JMenuItem 1242
JMenuBar 1242
JMenu 1243
JMenuItem 1244
Создание главного меню 1245
Добавление мнемонических символов и клавиатурных сочетаний к пунктам меню 1249
Добавление изображений и всплывающих подсказок к пунктам меню 1252
Использование JRadioButtonMenuItem и JCheckBoxMenuItem 1253
Создание всплывающего меню 1255
Создание панели инструментов 1259
Использование действий 1261
Построение окончательной программы MenuDemo 1267
Продолжение исследования Swing 1273

Часть IV. Применение Java 1275

Глава 35. Архитектура JavaBeans 1276

Что собой представляет Bean-компонент 1276
Преимущества Bean-компонентов 1277
Самоанализ 1277
Паттерны проектирования для свойств 1278
Паттерны проектирования для событий 1279
Методы и паттерны проектирования 1280
Использование интерфейса BeanInfo 1280
Связанные и ограниченные свойства 1281
Постоянство 1281
Настройщики 1281
JavaBeans API 1282
Introspector 1285
PropertyDescriptor 1285
EventSetDescriptor 1285
MethodDescriptor 1285
Пример Bean-компонента 1286

Глава 36. Введение в сервлеты 1289

Происхождение сервлетов 1289
Жизненный цикл сервлета 1290
Варианты разработки сервлетов 1291
Использование Tomcat 1291
Простой сервлет 1293
Создание и компиляция исходного кода сервлета 1293

Запуск Tomcat 1294
Запуск веб-браузера и запрашивание сервлета 1294
Servlet API 1294
Пакет jakarta.servlet 1295
Интерфейс Servlet 1296
Интерфейс ServletConfig 1297
Интерфейс ServletContext 1297
Интерфейс ServletRequest 1298
Интерфейс ServletResponse 1299
Класс GenericServlet 1299
Класс ServletInputStream 1300
Класс ServletOutputStream 1300
Классы исключений сервлетов 1300
Чтение параметров сервлета 1300
Пакет jakarta.servlet.http 1302
Интерфейс HttpServletRequest 1302
Интерфейс HttpServletResponse 1304
Интерфейс HttpSession 1305
Класс Cookie 1306
Класс HttpServlet 1307
Обработка запросов и ответов HTTP 1308
Обработка HTTP-запросов GET 1309
Обработка HTTP-запросов POST 1310
Использование cookie-наборов 1311
Отслеживание сеансов 1313

Часть V. Приложения 1315

Приложение А. Использование документирующих комментариев Java 1316
Дескрипторы javadoc 1316
@author 1318
{@code} 1318
@deprecated 1318
{@docRoot} 1318
@exception 1319
@hidden 1319
{@index} 1319
{@inheritDoc} 1319
{@link} 1319
{@linkplain} 1320
{@literal} 1320
@param 1320
@provides 1320
@return 1320
@see 1321
@serial 1321
@serialData 1321
@serialField 1321
@since 1322
{@summary} 1322
{@systemProperty} 1322

`@throws` 1322
`@uses` 1322
`{@value}` 1323
`@version` 1323
Общая форма документирующего комментария 1323
Вывод утилиты `javadoc` 1323
Пример использования документирующих комментариев 1324

Приложение Б. Введение в JShell 1325
Основы JShell 1325
Просмотр, редактирование и повторного выполнение кода 1328
Добавление метода 1329
Создание класса 1330
Использование интерфейса 1331
Вычисление выражений и использование встроенных переменных 1332
Импортирование пакетов 1333
Исключения 1334
Другие команды JShell 1334
Дальнейшее исследование JShell 1335

Приложение В. Компиляция и запуск простых однофайловых программ за один шаг 1336

Предметный указатель 1338

ГЛАВА

10 Обработка исключений

В настоящей главе рассматривается механизм обработки исключений Java. *Исключение* — это ненормальное состояние, которое возникает в кодовой последовательности во время выполнения. Другими словами, исключение является ошибкой времени выполнения. В языках программирования, не поддерживающих обработку исключений, ошибки необходимо проверять и обрабатывать вручную — обычно с помощью кодов ошибок и т.д. Такой подход столь же громоздкий, сколь и хлопотный. Обработка исключений в Java позволяет избежать проблем подобного рода и попутно переносит управление ошибками во время выполнения в объектно-ориентированный мир.

Основы обработки исключений

Исключение Java представляет собой объект, описывающий исключительное (т.е. ошибочное) состояние, которое произошло внутри фрагмента кода. При возникновении исключительного состояния в методе, вызвавшем ошибку, *генерируется* объект, представляющий это исключение. Метод может обработать исключение самостоятельно или передать его дальше. Так или иначе, в какой-то момент исключение *перехватывается* и *обрабатывается*. Исключения могут быть сгенерированы исполняющей средой Java или вручную в вашем коде. Исключения, генерируемые Java, относятся к фундаментальным ошибкам, которые нарушают правила языка Java или ограничения исполняющей среды Java. Исключения, сгенерированные вручную, обычно используются для сообщения об ошибке вызывающей стороне метода.

Обработка исключений в Java управляется пятью ключевыми словами: `try`, `catch`, `throw`, `throws` и `finally`. Давайте кратко рассмотрим, как они работают. Операторы программы, которые вы хотите отслеживать на наличие исключений, содержатся в блоке `try`. Если внутри блока `try` возникает исключение, тогда оно генерируется. Ваш код может перехватить это исключение (с помощью `catch`) и обработать его рациональным образом. Системные исключения автоматически генерируются исполняющей средой Java. Для ручной генерации исключения используйте ключевое слово `throw`. Любое исключение, генерируемое в методе, должно быть указано как таковое с помощью конструкции `throws`. Любой код, который обязательно должен быть выполнен после завершения блока `try`, помещается в блок `finally`.

Ниже показана общая форма блока обработки исключений:

```
try {
  // блок кода, где отслеживаются ошибки
}
catch (ТипИсключения1 объектИсключения) {
  // обработчик исключений для ТипИсключения1
}
catch (ТипИсключения2 объектИсключения) {
  // обработчик исключений для ТипИсключения2
}
// ...
finally {
  // блок кода, подлежащий выполнению после окончания блока try
}
```

Здесь *ТипИсключения* — это тип возникшей исключительной ситуации. В оставшихся материалах главы демонстрируется применение приведенной выше структуры.

На заметку! Существует еще одна форма оператора `try`, которая поддерживает *автоматическое управление ресурсами*. Она называется *`try` с ресурсами* и описана в главе 13 в контексте управления файлами, поскольку файлы являются одним из наиболее часто используемых ресурсов.

Типы исключений

Все типы исключений являются подклассами встроенного класса `Throwable`. Таким образом, класс `Throwable` расположен на вершине иерархии классов исключений. Непосредственно под `Throwable` находятся два подкласса, которые разделяют исключения на две отдельные ветви. Одну ветвь возглавляет класс `Exception`, используемый для представления исключительных условий, которые должны перехватываться пользовательскими программами. Он также будет служить подклассом для создания собственных специальных типов исключений. Кроме того, у класса `Exception` имеется важный подкласс, который называется `RuntimeException`. Исключения такого типа автоматически определяются для разрабатываемых программ и охватывают такие ситуации, как деление на ноль и недопустимое индексирование массивов.

Другую ветвь возглавляет класс `Error`, определяющий исключения, которые не должны перехватываться программой в обычных условиях. Исключения типа `Error` применяется исполняющей средой Java для указания ошибок, связанных с самой средой. Примером такой ошибки является переполнение стека. Исключения типа `Error` здесь не рассматриваются, т.к. они обычно создаются в ответ на катастрофические отказы, которые обычно не могут быть обработаны создаваемой программой.

Иерархия исключений верхнего уровня показана на рис. 10.1.

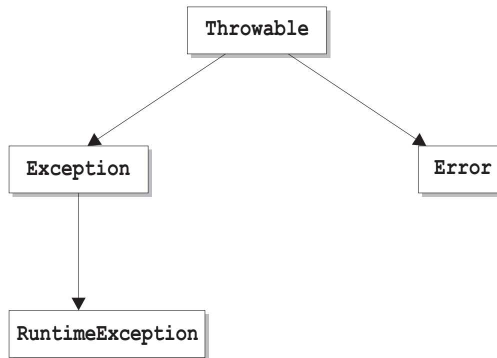

Рис. 10.1. Иерархия исключений верхнего уровня

Неперехваченные исключения

Прежде чем вы научитесь обрабатывать исключения в своей программе, полезно посмотреть, что происходит, когда их не обрабатывать. Следующая небольшая программа содержит выражение, которое намеренно вызывает ошибку деления на ноль:

```
class Exc0 {
  public static void main(String[] args) {
    int d = 0;
    int a = 42 / d;
  }
}
```

Когда исполняющая среда Java обнаруживает попытку деления на ноль, она создает новый объект исключения и затем *генерирует* это исключение. В результате выполнение класса Exc0 останавливается, поскольку после генерации исключение должно быть *перехвачено* обработчиком исключений и немедленно обработано. В приведенном примере не было предусмотрено никаких собственных обработчиков исключений, поэтому исключение перехватывается стандартным обработчиком, предоставляемым исполняющей средой Java. Любое исключение, которое не перехвачено вашей программой, в конечном итоге будет обработано стандартным обработчиком. Стандартный обработчик отображает строку с описанием исключения, выводит трассировку стека от точки, где произошло исключение, и прекращает работу программы.

Вот какое исключение генерируется при выполнении примера:

```
java.lang.ArithmeticException: / by zero
    at Exc0.main(Exc0.java:4)
java.lang.ArithmeticException: деление на ноль
    в Exc0.main(Exc0.java:4)
```

Обратите внимание, что в простой трассировки стека присутствует имя класса — Exc0, имя метода — main(), имя файла — Exc0.java и номер строки — 4. Кроме того, как видите, типом сгенерированного исключения является подкласс Exception по имени ArithmeticException, который более

конкретно описывает тип возникшей ошибки. Далее в главе будет показано, что Java предлагает несколько встроенных типов исключений, соответствующих различным типам ошибок времени выполнения, которые могут быть сгенерированы. Еще одно замечание: точный вывод, который вы видите при запуске этого и других примеров программ в главе, использующих встроенные исключения Java, может немного отличаться от показанного здесь из-за различий между версиями JDK.

В трассировке стека всегда показана последовательность вызовов методов, которые привели к ошибке. Например, вот еще одна версия предыдущей программы, которая вызывает ту же ошибку, но в методе, отдельном от main():

```
class Exc1 {
  static void subroutine() {
    int d = 0;
    int a = 10 / d;
  }
  public static void main(String[] args) {
    Exc1.subroutine();
  }
}
```

В полученной трассировке стека из стандартного обработчика исключений видно, что отображается весь стек вызовов:

```
java.lang.ArithmeticException: / by zero
    at Exc1.subroutine(Exc1.java:4)
    at Exc1.main(Exc1.java:7)
```

Как видите, в нижней части стека указана строка 7 метода main(), в которой вызывается метод subroutine(), ставший причиной исключения в строке 4. Стек вызовов весьма полезен для отладки, поскольку он указывает точную последовательность шагов, которые привели к ошибке.

Использование try и catch

Хотя стандартный обработчик исключений, предоставляемый исполняющей средой Java, удобен при отладке, обычно вы пожелаете обрабатывать исключение самостоятельно, что дает два преимущества. Самостоятельная обработка, во-первых, позволяет исправить ошибку и, во-вторых, предотвращает автоматическое прекращение работы программы. Большинство пользователей будут (по меньшей мере) сбиты с толку, если ваша программа перестанет работать, начав выводить трассировку стека всякий раз, когда возникает ошибка! К счастью, предотвратить это довольно легко.

Чтобы защититься от ошибки времени выполнения и обработать ее, просто поместите код, который хотите отслеживать, в блок try. Сразу после блока try добавьте конструкцию catch с указанием типа исключения, которое желательно перехватить. В целях иллюстрации того, насколько легко это делать, в следующей программе определен блок try и конструкция catch, обрабатывающая исключение ArithmeticException, которое генерируется ошибкой деления на ноль:

```
class Exc2 {
public static void main(String[] args) {
  int d, a;

  try {  // отслеживать блок кода
    d = 0;
    a = 42 / d;
    System.out.println("Это выводиться не будет.");
    } catch (ArithmeticException e) {  // перехватить ошибку деления на ноль
      System.out.println("Деление на ноль.");
    }
    System.out.println("После оператора catch.");
  }
}
```

Программа выдает такой вывод:

```
Деление на ноль.
После оператора catch.
```

Обратите внимание, что вызов `println()` внутри блока `try` никогда не выполняется. После генерации исключения управление передается из блока `try` в блок `catch`. Другими словами, блок `catch` не "вызывается" и потому управление никогда не "возвращается" в блок `try` из `catch`. Таким образом, строка `"Это выводиться не будет."` не отображается. После блока `catch` выполнение продолжается со строки программы, следующей за всем механизмом `try/catch`.

Оператор `try` и его конструкция `catch` образуют единицу. Область действия `catch` ограничена операторами, которые относятся к непосредственно предшествующему оператору `try`. Конструкция `catch` не может перехватывать исключение, сгенерированное другим оператором `try` (за исключением описанного ниже случая вложенных операторов `try`). Операторы, защищенные с помощью `try`, должны быть заключены в фигурные скобки (т.е. находиться внутри блока). Применять `try` для одиночного оператора нельзя.

Целью большинства хорошо построенных конструкций `catch` должно быть разрешение исключительной ситуации и продолжение работы, как если бы ошибка вообще не возникала. Например, в приведенной далее программе на каждой итерации цикла `for` получаются два случайных целых числа, одно из которых делится на другое, а результат используется для деления значения `12345`. Окончательный результат помещается в переменную `a`. Если какая-либо операция вызывает ошибку деления на ноль, то она перехватывается, значение `a` устанавливается равным нулю и выполнение программы продолжается.

```
// Обработать исключение и продолжить работу.
import java.util.Random;

class HandleError {
  public static void main(String[] args) {
    int a=0, b=0, c=0;
    Random r = new Random();
```

```
      for(int i=0; i<32000; i++) {
        try {
          b = r.nextInt();
          c = r.nextInt();
          a = 12345 / (b/c);
        } catch (ArithmeticException e) {
          System.out.println("Деление на ноль.");
          a = 0;  // установить а в ноль и продолжить
        }
        System.out.println("a: " + a);
      }
    }
  }
```

Отображение описания исключения

В классе Throwable переопределен метод toString() (определенный в Object), так что он возвращает строку, содержащую описание исключения. Для отображения этого описания в операторе println() нужно просто передать исключение в качестве аргумента. Например, блок catch из предыдущей программы можно переписать так:

```
catch (ArithmeticException e) {
  System.out.println("Исключение: " + e);
  a = 0;  // установить а в ноль и продолжить
}
```

Тогда в случае ошибки деления на ноль будет отображаться следующее сообщение:

```
Исключение: java.lang.ArithmeticException: / by zero
```

Хотя в таком контексте это не имеет особой ценности, возможность отображать описание исключения полезна в других обстоятельствах, особенно когда вы экспериментируете с исключениями или занимаетесь отладкой.

Использование нескольких конструкций catch

В некоторых случаях один фрагмент кода может генерировать более одного исключения. Чтобы справиться с ситуацией такого рода, можно указать две или более конструкции catch, каждая из которых будет перехватывать разные типы исключений. При возникновении исключения все конструкции catch проверяются по порядку, и выполняется первая из них, в которой указанный тип совпадает с типом сгенерированного исключения. После выполнения одной конструкции catch остальные игнорируются, и выполнение продолжается после блока try/catch. В следующем примере перехватываются два разных типа исключений:

```
// Демонстрация применения нескольких конструкций catch.
class MultipleCatches {
  public static void main(String[] args) {
    try {
```

```
      int a = args.length;
      System.out.println("a = " + a);
      int b = 42 / a;
      int[] c = { 1 };
      c[42] = 99;
    } catch(ArithmeticException e) {
      System.out.println("Деление на ноль: " + e);
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Выход за допустимые пределы индекса в массиве: " + e);
    }
    System.out.println("После блоков try/catch.");
  }
}
```

Программа вызовет исключение деления на ноль, если будет запущена без аргументов командной строки, т.к. значение a будет равно нулю. Деление пройдет успешно в случае предоставления аргумента командной строки, который приведет к установке a во что-то большее, чем ноль. Но это станет причиной генерации исключения ArrayIndexOutOfBoundsException, поскольку целочисленный массив c имеет длину 1, а программа пытается присвоить значение несуществующему элементу c[42].

Ниже показан вывод программы, выдаваемый в обеих ситуациях:

```
C:\>java MultipleCatches a = 0
Деление на ноль: java.lang.ArithmeticException: / by zero
После блоков try/catch.

C:\>java MultipleCatches TestArg a = 1
Выход за допустимые пределы индекса в массиве:
java.lang.ArrayIndexOutOfBoundsException:
Index 42 out of bounds for length 1
После блоков try/catch.
```

При использовании нескольких конструкций catch важно помнить о том, что подклассы исключений должны предшествовать любым из своих суперклассов. Дело в том, что конструкция catch, в которой применяется суперкласс, будет перехватывать исключения указанного типа плюс любых его подклассов. В итоге конструкция catch с подклассом никогда не будет достигнута, если она находится после конструкции catch с суперклассом. Кроме того, недостижимый код в Java является ошибкой. Например, рассмотрим следующую программу:

```
/* Эта программа содержит ошибку.

   В последовательности конструкций catch подкласс должен
   предшествовать своему суперклассу. В противном случае будет создан
   недостижимый код, что приведет к ошибке на этапе компиляции.
*/
class SuperSubCatch {
  public static void main(String[] args) {
    try {
      int a = 0;
      int b = 42 / a;
```

```
    } catch(Exception e) {
      System.out.println("Перехват обобщенного исключения Exception.");
    }
    /* Эта конструкция catch недостижима, потому что
       ArithmeticException является подклассом Exception. */
    catch(ArithmeticException e) {  // ОШИБКА - недостижимый код
      System.out.println("Это никогда не будет достигнуто.");
    }
  }
}
```

Попытка компиляции этой программы приводит к получению сообщения об ошибке, указывающего на то, что вторая конструкция catch недостижима, т.к. исключение уже было перехвачено. Поскольку ArithmeticException является подклассом Exception, первая конструкция catch будет обрабатывать все ошибки, связанные с Exception, в том числе ArithmeticException. Таким образом, вторая конструкция catch никогда не выполнится. Чтобы решить проблему, понадобится изменить порядок следования конструкций catch.

Вложенные операторы try

Оператор try может быть вложенным, т.е. находиться внутри блока другого оператора try. Каждый раз, когда происходит вход в try, контекст этого исключения помещается в стек. Если внутренний оператор try не имеет обработчика catch для определенного исключения, тогда стек раскручивается, и на предмет совпадения проверяются обработчики catch следующего оператора try. Процесс продолжается до тех пор, пока не будет найдена подходящая конструкция catch либо исчерпаны все вложенные операторы try. Если ни одна из конструкций catch не дает совпадения, то исключение будет обработано исполняющей средой Java. Ниже приведен пример использования вложенных операторов try:

```
// Пример применения вложенных операторов try.
class NestTry {
  public static void main(String[] args) {
    try {
      int a = args.length;

      /* Если аргументы командной строки отсутствуют, то следующий
         оператор сгенерирует исключение деления на ноль. */
      int b = 42 / a;

      System.out.println("a = " + a);

      try {  // вложенный блок try
        /* Если используется один аргумент командной строки, тогда
           исключение деления на ноль сгенерирует следующий код. */
        if(a==1) a = a/(a-a);  // деление на ноль

        /* Если используется один аргумент командной строки,
           тогда генерируется исключение выхода за допустимые
           пределы индекса в массиве. */
```

```
        if(a==2) {
          int[] c = { 1 };
          c[42] = 99;  // генерирует исключение ArrayIndexOutOfBoundsException
        }
      } catch(ArrayIndexOutOfBoundsException e) {
       System.out.println("Выход за допустимые пределы индекса в массиве: "+ e);
      }
    } catch(ArithmeticException e) {
      System.out.println("Деление на ноль: " + e);
    }
  }
}
```

Как видите, в программе один блок `try` вложен в другой. Программа работает следующим образом. При запуске программы без аргументов командной строки внешний блок `try` генерирует исключение деления на ноль. Запуск программы с одним аргументом командной строки приводит к генерации исключения деления на ноль внутри вложенного блока `try`. Поскольку внутренний блок `try` не перехватывает это исключение, оно передается внешнему блоку `try`, где и обрабатывается. При запуске программы с двумя аргументами командной строки во внутреннем блоке `try` генерируется исключение выхода за допустимые пределы индекса в массиве. Вот примеры запуска, иллюстрирующие каждый случай:

```
C:\>java NestTry
Деление на ноль: java.lang.ArithmeticException: / by zero

C:\>java NestTry One a = 1
Деление на ноль: java.lang.ArithmeticException: / by zero

C:\>java NestTry One Two a = 2
Выход за допустимые пределы индекса в массиве:
java.lang.ArrayIndexOutOfBoundsException:
Index 42 out of bounds for length 1
```

Когда задействованы вызовы методов, вложение операторов `try` может происходить менее очевидным образом. Например, если вызов метода заключен в блок `try` и внутри этого метода находится еще один оператор `try`, то оператор `try` внутри метода будет вложен во внешний блок `try`, где метод вызывается. Ниже представлена предыдущая программа, в которой вложенный блок `try` перемещен внутрь метода `nesttry()`:

```
/* Операторы try могут быть неявно вложенными через вызовы методов. */
class MethNestTry {
  static void nesttry(int a) {
    try {  // вложенный блок try
      /* Если используется один аргумент командной строки, тогда
         исключение деления на ноль сгенерирует следующий код. */
      if(a==1) a = a/(a-a);  // деление на ноль

      /* Если используются два аргумента командной строки, тогда генерируется
         исключение выхода за допустимые пределы индекса в массиве. */
      if(a==2) {
        int[] c = { 1 };
```

```
        c[42] = 99;  // генерирует исключение ArrayIndexOutOfBoundsException
      }
    } catch(ArrayIndexOutOfBoundsException e) {
     System.out.println("Выход за допустимые пределы индекса в массиве: " + e);
    }
  }

  public static void main(String[] args) {
    try {
      int a = args.length;

      /* Если аргументы командной строки отсутствуют, то следующий
         оператор сгенерирует исключение деления на ноль. */
      int b = 42 / a;
      System.out.println("a = " + a);
      nesttry(a);
    } catch(ArithmeticException e) {
        System.out.println("Деление на ноль: " + e);
    }
  }
}
```

Вывод программы идентичен выводу в предыдущем примере.

Оператор throw

До сих пор перехватывались только те исключения, которые генерируются исполняющей средой Java. Однако программа может генерировать исключение явно с применением оператора throw со следующей общей формой:

```
throw ThrowableInstance;
```

Здесь ThrowableInstance должен быть объектом типа Throwable или подклассом Throwable. Примитивные типы вроде int или char, а также классы, отличающиеся от Throwable, такие как String и Object, не могут использоваться в качестве исключений. Есть два способа получить объект Throwable: указывая параметр в конструкции catch или создавая его с помощью операции new.

Поток выполнения останавливается сразу после оператора throw; любые последующие операторы не выполняются. Ближайший охватывающий блок try проверяется на предмет наличия в нем конструкции catch, соответствующей типу исключения. Если совпадение найдено, то управление передается этому оператору, а если нет, тогда проверяется следующий охватывающий оператор try и т.д. Если соответствующая конструкция catch не найдена, то стандартный обработчик исключений останавливает работу программы и выводит трассировку стека.

Ниже приведен пример программы, в которой создается и генерируется исключение. Обработчик, перехватывающий исключение, повторно генерирует его для внешнего обработчика.

```
// Демонстрация применения throw.
class ThrowDemo {
```

```
    static void demoproc() {
      try {
        throw new NullPointerException("демонстрация");
      } catch(NullPointerException e) {
        System.out.println("Перехвачено внутри demoproc().");
        throw e;          // повторно сгенерировать исключение
      }
    }

    public static void main(String[] args) {
      try {
        demoproc();
      } catch(NullPointerException e) {
        System.out.println("Повторно перехвачено: " + e);
      }
    }
  }
```

В программе есть два шанса справиться с одной и той же ошибкой. Сначала в `main()` устанавливается контекст исключения и вызывается `demoproc()`. Затем метод `demoproc()` устанавливает другой контекст обработки исключений и немедленно создает новый экземпляр `NullPointerException`, который перехватывается в следующей строке. Далее исключение генерируется повторно. Вот результат:

```
Перехвачено внутри demoproc().
Повторно перехвачено: java.lang.NullPointerException: демонстрация
```

В программе также демонстрируется создание одного из стандартных объектов исключений Java. Обратите особое внимание на следующую строку:

```
throw new NullPointerException("демонстрация");
```

Здесь с применением операции `new` создается экземпляр `NullPointerException`. Многие встроенные исключения времени выполнения Java имеют как минимум два конструктора: один без параметров и один принимающий строковый параметр. Когда используется вторая форма, аргумент указывает строку, описывающую исключение. Эта строка отображается, когда объект передается как аргумент в `print()` или `println()`. Его также можно получить, вызвав метод `getMessage()`, который определен в `Throwable`.

Конструкция `throws`

Если метод способен приводить к исключению, которое он не обрабатывает, то метод должен сообщить о таком поведении, чтобы вызывающий его код мог защитить себя от этого исключения. Задача решается добавлением к объявлению метода конструкции `throws`, где перечисляются типы исключений, которые может генерировать метод. Поступать так необходимо для всех исключений, кроме исключений типа `Error`, `RuntimeException` или любых их подклассов. Все остальные исключения, которые может генерировать метод, должны быть объявлены в конструкции `throws`. В противном случае возникнет ошибка на этапе компиляции.

Вот общая форма объявления метода, которая содержит конструкцию throws:

```
тип имя-метода(список-параметров) throws список-исключений
{
  // тело метода
}
```

Здесь *список-исключений* представляет собой список разделяемых запятыми исключений, которые метод может сгенерировать.

Ниже приведен пример некорректной программы, пытающейся сгенерировать исключение, которое она не перехватывает. Из-за того, что в программе не указана конструкция throws для объявления данного факта, программа не скомпилируется.

```
// Эта программа содержит ошибку и компилироваться не будет.
class ThrowsDemo {
  static void throwOne() {
    System.out.println("Внутри throwOne().");
    throw new IllegalAccessException("демонстрация");
  }

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

Чтобы пример скомпилировался, в него понадобится внести два изменения. Во-первых, вам нужно объявить, что метод throwOne() генерирует исключение IllegalAccessException. Во-вторых, в методе main() должен быть определен оператор try/catch, который перехватывает это исключение.

Далее показан исправленный пример:

```
// Теперь программа компилируется.
class ThrowsDemo {
  static void throwOne() throws IllegalAccessException {
    System.out.println("Внутри throwOne().");
    throw new IllegalAccessException("демонстрация");
  }

  public static void main(String[] args) {
    try {
      throwOne();
    } catch (IllegalAccessException e) {
      System.out.println("Перехвачено " + e);
    }
  }
}
```

Программа выдает следующий вывод:

```
Внутри throwOne().
Перехвачено java.lang.IllegalAccessException: демонстрация
```

Конструкция finally

Когда генерируются исключения, поток выполнения в методе направляется по довольно резкому нелинейному пути, изменяющем нормальный ход выполнения метода. В зависимости от того, как закодирован метод, исключение может даже привести к преждевременному возврату из метода, что в некоторых методах может стать проблемой. Например, если метод открывает файл при входе и закрывает его при выходе, то пропуск кода, закрывающего файл, механизмом обработки исключений нельзя считать приемлемым. Для такой нештатной ситуации и предназначено ключевое слово `finally`.

Ключевое слово `finally` позволяет создать блок кода, который будет выполняться после завершения блока `try/catch` и перед кодом, следующим после `try/catch`. Блок `finally` будет выполняться независимо от того, сгенерировано исключение или нет. В случае генерации исключения блок `finally` будет выполняться, даже если исключение не соответствует ни одной конструкции `catch`. Каждый раз, когда метод собирается вернуть управление вызывающему коду из блока `try/catch` через неперехваченное исключение или явно посредством оператора `return`, блок `finally` тоже выполняется непосредственно перед возвратом из метода. Таким образом, с помощью блока `finally` удобно закрывать файловые дескрипторы и освобождать любые другие ресурсы, которые могли быть выделены в начале метода с намерением освобождения их перед возвратом. Конструкция `finally` является необязательной. Тем не менее, для каждого оператора `try` требуется хотя бы одна конструкция `catch` или `finally`.

Ниже приведен пример программы с тремя методами, которые завершаются разными способами, не пропуская выполнение конструкции `finally`:

```
// Демонстрация применения finally.
class FinallyDemo {
  // Сгенерировать исключение внутри метода.
  static void procA() {
    try {
      System.out.println("Внутри метода procA()");
      throw new RuntimeException("демонстрация");
    } finally {
      System.out.println("Блок finally метода procA()");
    }
  }
  // Возвратить управление изнутри блока try.
  static void procB() {
    try {
      System.out.println("Внутри метода procB()");
      return;
    } finally {
      System.out.println("Блок finally метода procB()");
    }
  }
  // Выполнить блок try обычным образом.
  static void procC() {
```

```
    try {
      System.out.println("Внутри метода procC()");
    } finally {
      System.out.println("Блок finally метода procC()");
    }
  }
  public static void main(String[] args) {
    try {
      procA();
    } catch (Exception e) {
      System.out.println("Исключение перехвачено");
    }
    procB();
    procC();
  }
}
```

Оператор try в методе procA() преждевременно прерывается генерацией исключения. Блок finally выполняется при выходе. Оператор try в методе procB() завершается оператором return. Блок finally выполняется до возврата из procB(). В методе procC() оператор try выполняется нормально, без ошибок. Однако блок finally все равно выполняется.

Помните! Если с оператором try ассоциирован блок finally, то этот блок будет выполнен по завершении try.

Вот вывод, полученный в результате запуска предыдущей программы:

```
Внутри метода procA()
Блок finally метода procA()
Исключение перехвачено
Внутри метода procB()
Блок finally метода procB()
Внутри метода procC()
Блок finally метода procC()
```

Встроенные исключения Java

Внутри стандартного пакета java.lang определено несколько классов исключений Java. Некоторые из них использовались в предшествующих примерах. Наиболее общие из них являются подклассами стандартного типа RuntimeException. Как объяснялось ранее, такие исключения не нужно включать в список throws любого метода. На языке Java они называются *непроверяемыми исключениями*, потому что компилятор не проверяет, обрабатывает метод подобные исключения или же генерирует их. Непроверяемые исключения, определенные в java.lang, описаны в табл. 10.1. В табл. 10.2 перечислены те исключения, определенные в java.lang, которые должны помещаться в список throws метода, если метод может генерировать одно из исключений и не обрабатывает его самостоятельно. Они называются *проверяемыми исключениями*. Помимо исключений из java.lang в Java определено еще несколько, которые относятся к другим стандартным пакетам.

Таблица 10.1. Подклассы `RuntimeException` непроверяемых исключений Java, определенные в `java.lang`

Исключение	Описание
`ArithmeticException`	Арифметическая ошибка, такая как деление на ноль
`ArrayIndexOutOfBoundsException`	Выход за допустимые пределы индекса в массиве
`ArrayStoreException`	Присваивание элементу массива значения несовместимого типа
`ClassCastException`	Недопустимое приведение
`EnumConstantNotPresentException`	Попытка использования неопределенного значения перечисления
`IllegalArgumentException`	Использование недопустимого аргумента при вызове метода
`IllegalCallerException`	Метод не может быть законно выполнен вызывающим кодом
`IllegalMonitorStateException`	Недопустимая операция монитора, такая как ожидание неблокированного потока
`IllegalStateException`	Некорректное состояние среды или приложения
`IllegalThreadStateException`	Несовместимость запрошенной операции с текущим состоянием потока
`IndexOutOfBoundsException`	Выход за допустимые пределы индекса некоторого вида
`LayerInstantiationException`	Невозможность создания уровня модуля
`NegativeArraySizeException`	Создание массива с отрицательным размером
`NullPointerException`	Недопустимое использование ссылки `null`
`NumberFormatException`	Недопустимое преобразование строки в числовой формат
`SecurityException`	Попытка нарушения безопасности
`StringIndexOutOfBoundsException`	Попытка индексации за границами строки
`TypeNotPresentException`	Тип не найден
`UnsupportedOperationException`	Обнаружение неподдерживаемой операции

Таблица 10.2. Классы проверяемых исключений Java, определенные в `java.lang`

Исключение	Описание
`ClassNotFoundException`	Класс не найден
`CloneNotSupportedException`	Попытка клонирования объекта, который не реализует интерфейс `Cloneable`
`IllegalAccessException`	Доступ к классу запрещен
`InstantiationException`	Попытка создания объекта абстрактного класса или интерфейса
`InterruptedException`	Один поток был прерван другим потоком
`NoSuchFieldException`	Запрошенное поле не существует
`NoSuchMethodException`	Запрошенный метод не существует
`ReflectiveOperationException`	Суперкласс исключений, связанных с рефлексией

Создание собственных подклассов `Exception`

Хотя встроенные исключения Java обрабатывают наиболее распространенные ошибки, вполне вероятно, что вы захотите создать собственные типы исключений, которые подходят для ситуаций, специфичных для ваших приложений. Делается это довольно легко: нужно просто определить подкласс `Exception` (который, конечно же, является подклассом `Throwable`). Вашим подклассам фактически ничего не придется реализовывать — одно их существование в системе типов позволяет использовать их как исключения. В самом классе `Exception` никаких методов не определено. Разумеется, он наследует методы, предоставляемые `Throwable`. Таким образом, все исключения, в том числе созданные вами, имеют доступные для них методы, которые определены в классе `Throwable` и описаны в табл. 10.3. Вы также можете переопределить один или несколько из этих методов в создаваемых классах исключений.

Таблица 10.3. Методы, определенные в классе `Throwable`

Метод	Описание
`final void addSuppressed(Throwable exc)`	Добавляет `exc` в список подавляемых исключений, ассоциированный с вызывающим исключением. Метод предназначен главным образом для использования в операторе `try` с ресурсами
`Throwable fillInStackTrace()`	Возвращает объект `Throwable`, который содержит полную трассировку стека. Этот объект может быть сгенерирован повторно

Окончание табл. 10.3

Метод	Описание
`Throwable getCause()`	Возвращает исключение, которое лежит в основе текущего исключения. Если лежащее в основе исключение отсутствует, тогда возвращается `null`
`String getLocalizedMessage()`	Возвращает локализованное описание исключения
`String getMessage()`	Возвращает описание исключения
`StackTraceElement[] getStackTrace()`	Возвращает массив объектов `StackTraceElement`, содержащий поэлементную трассировку стека. Метод на верхушке стека — это тот, который был вызван последним перед генерацией исключения. Он находится в первом элементе массива. Класс `StackTraceElement` предоставляет программе доступ к информации о каждом элементе в трассировке стека, такой как имя метода
`final Throwable[] getSuppressed()`	Получает подавляемые исключения, ассоциированные с вызываемым исключением, и возвращает массив, который содержит результат. Подавляемые исключения генерируются главным образом оператором `try` с ресурсами
`Throwable initCause( Throwable causeExc)`	Связывает `causeExc` с вызывающим исключением как причину его возникновения. Возвращает ссылку на исключение
`void printStackTrace()`	Отображает трассировку стека
`void printStackTrace( PrintStream stream)`	Посылает трассировку стека в указанный поток
`void printStackTrace( PrintWriter stream)`	Посылает трассировку стека в указанный поток
`void setStackTrace( StackTraceElement[] elements)`	Устанавливает трассировку стека в элементы, переданные в параметре `elements`. Предназначен для специализированного, а не нормального применения
`String toString()`	Возвращает объект `String`, содержащий описание исключения. Вызывается оператором `println()` при выводе объекта `Throwable`