Содержание

Предисловие 29
Книга для всех программистов 30
Структура книги 30
Исходный код примеров, доступный в Интернете 31
Особые благодарности 31
Дополнительная литература 31
От издательства 32

Часть I. Язык Java 33

Глава 1. История и развитие языка Java 35
Происхождение Java 35
Зарождение современного программирования: язык C 36
Следующий этап: язык C++ 37
Предпосылки к созданию Java 39
Создание языка Java 39
Связь с языком C# 42
Каким образом язык Java повлиял на Интернет 42
Аплеты на Java 42
Безопасность 43
Переносимость 43
Чудо Java: байт-код 44
Выход за пределы аплетов 45
Сервлеты: серверные программы на Java 46
Терминология Java 46
Простота 47
Объектная ориентированность 47
Надежность 48
Многопоточность 48
Архитектурная нейтральность 49
Интерпретируемость и высокая производительность 49
Распределенность 49
Динамичность 49
Эволюция языка Java 50
Версия Java SE 9 54
Культура нововведений 55

Глава 2. Краткий обзор Java 57
Объектно-ориентированное программирование 57
Две парадигмы 57

Абстракция 58
Три принципа ООП 59
Первый пример простой программы 64
Ввод кода программы 65
Компиляция программы 65
Подробный анализ первого примера программы 66
Второй пример короткой программы 69
Два управляющих оператора 71
Условный оператор `if` 71
Оператор цикла `for` 73
Применение блоков кода 74
Вопросы лексики 75
Пробелы 76
Идентификаторы 76
Литералы 76
Комментарии 76
Разделители 77
Ключевые слова Java 77
Библиотеки классов Java 78

Глава 3. Типы данных, переменные и массивы 79
Java — строго типизированный язык 79
Примитивные типы 79
Целые числа 80
Тип `byte` 81
Тип `short` 81
Тип `int` 81
Тип `long` 82
Числа с плавающей точкой 82
Тип `float` 83
Тип `double` 83
Символы 84
Логические значения 85
Подробнее о литералах 86
Целочисленные литералы 87
Литералы с плавающей точкой 88
Логические литералы 89
Символьные литералы 89
Строковые литералы 90
Переменные 90
Объявление переменной 91
Динамическая инициализация 91
Область видимости и срок действия переменных 92
Преобразование и приведение типов 95
Автоматическое преобразование типов в Java 95
Приведение несовместимых типов 95
Автоматическое продвижение типов в выражениях 97
Правила продвижения типов 98
Массивы 99
Одномерные массивы 99
Многомерные массивы 102
Альтернативный синтаксис объявления массивов 106
Введение в символьные строки 106

Глава 4. Операции 109
Арифметические операции 109
Основные арифметические операции 110
Операция деления по модулю 111
Составные арифметические операции с присваиванием 111
Операции инкремента и декремента 112
Поразрядные операции 114
Поразрядные логические операции 116
Сдвиг влево 118
Сдвиг вправо 120
Беззнаковый сдвиг вправо 121
Поразрядные составные операции с присваиванием 123
Операции отношения 124
Логические операции 125
Укороченные логические операции 126
Операция присваивания 127
Тернарная операция ? 128
Предшествование операций 129
Применение круглых скобок 130

Глава 5. Управляющие операторы 131
Операторы выбора 131
Условный оператор if 131
Оператор switch 135
Операторы цикла 140
Цикл while 140
Цикл do-while 142
Цикл for 145
Вложенные циклы 154
Операторы перехода 155
Применение оператора break 155
Применение оператора continue 159
Оператор return 161

Глава 6. Введение в классы 163
Основы классов 163
Общая форма класса 163
Простой класс 164
Объявление объектов 167
Подробное рассмотрение операции new 168
Присваивание переменным ссылок на объекты 169
Введение в методы 170
Ввод метода в класс Box 171
Возврат значений 173
Ввод метода, принимающего параметры 174
Конструкторы 177
Параметризированные конструкторы 179
Ключевое слово this 180
Сокрытие переменных экземпляра 180
Сборка “мусора” 181
Класс Stack 181

Глава 7. Подробное рассмотрение классов и методов 185
Перегрузка методов 185
Перегрузка конструкторов 188
Применение объектов в качестве параметров 191
Подробное рассмотрение особенностей передачи аргументов 193
Возврат объектов 195
Рекурсия 196
Введение в управление доступом 199
Ключевое слово `static` 202
Ключевое слово `final` 204
Еще раз о массивах 205
Вложенные и внутренние классы 207
Краткий обзор класса `String` 210
Применение аргументов командной строки 212
Аргументы переменной длины 213
Перегрузка методов с аргументами переменной длины 216
Аргументы переменной длины и неоднозначность 218

Глава 8. Наследование 221
Основы наследования 221
Доступ к членам класса и наследование 223
Практический пример наследования 224
Переменная из суперкласса может ссылаться на объект подкласса 226
Ключевое слово `super` 227
Вызов конструкторов суперкласса с помощью ключевого слова `super` 227
Другое применение ключевого слова `super` 231
Создание многоуровневой иерархии 232
Порядок вызова конструкторов 235
Переопределение методов 236
Динамическая диспетчеризация методов 239
Назначение переопределенных методов 240
Применение переопределения методов 241
Применение абстрактных классов 242
Ключевое слово `final` в сочетании с наследованием 246
Предотвращение переопределения с помощью ключевого слова `final` 246
Предотвращение наследования с помощью ключевого слова `final` 247
Класс `Object` 247

Глава 9. Пакеты и интерфейсы 249
Пакеты 249
Определение пакета 250
Поиск пакетов и переменная окружения `CLASSPATH` 251
Краткий пример пакета 251
Доступ к пакетам и его компонентам 252
Пример доступа к пакетам 254
Импорт пакетов 257
Интерфейсы 259
Объявление интерфейса 260
Реализация интерфейсов 261
Вложенные интерфейсы 263

Применение интерфейсов 264
Переменные в интерфейсах 268
Расширение интерфейсов 270
Методы с реализацией по умолчанию 271
Основы применения методов с реализацией по умолчанию 272
Прикладной пример 274
Вопросы множественного наследования 275
Применение статических методов в интерфейсе 276
Закрытые методы интерфейсов 277
Заключительные соображения по поводу пакетов и интерфейсов 278

Глава 10. Обработка исключений 279
Основы обработки исключений 279
Типы исключений 280
Необрабатываемые исключения 281
Применение блоков операторов `try` и `catch` 282
Вывод описания исключения 284
Применение нескольких операторов `catch` 284
Вложенные операторы `try` 286
Оператор `throw` 289
Оператор `throws` 290
Оператор `finally` 291
Встроенные в Java исключения 293
Создание собственных подклассов исключений 295
Цепочки исключений 298
Дополнительные средства для обработки исключений 299
Применение исключений 301

Глава 11. Многопоточное программирование 303
Модель потоков исполнения в Java 304
Приоритеты потоков 305
Синхронизация 306
Обмен сообщениями 307
Класс `Thread` и интерфейс `Runnable` 307
Главный поток исполнения 308
Создание потока исполнения 310
Реализация интерфейса `Runnable` 310
Расширение класса `Thread` 312
Выбор способа создания потоков исполнения 313
Создание многих потоков исполнения 314
Применение методов `isAlive()` и `join()` 315
Приоритеты потоков исполнения 318
Синхронизация 319
Применение синхронизированных методов 319
Оператор `synchronized` 322
Взаимодействие потоков исполнения 323
Взаимная блокировка 328
Приостановка, возобновление и остановка потоков исполнения 330
Получение состояния потока исполнения 333
Одновременное создание и запуск потоков исполнения фабричными методами 335
Применение многопоточности 336

Глава 12. Перечисления, автоупаковка и аннотации 337
Перечисления 337
Основные положения о перечислениях 338
Методы `values()` и `valueOf()` 340
Перечисления в Java относятся к типам классов 341
Перечисления наследуются от класса `Enum` 343
Еще один пример перечисления 346
Оболочки типов 347
Класс `Character` 348
Класс `Boolean` 348
Оболочки числовых типов 349
Автоупаковка 350
Автоупаковка и методы 351
Автоупаковка и автораспаковка в выражениях 352
Автоупаковка и распаковка значений из классов `Boolean` и `Character` 354
Автоупаковка и автораспаковка помогает предотвратить ошибки 355
Предупреждение 356
Аннотации 356
Основы аннотирования программ 357
Правила удержания аннотаций 358
Получение аннотаций во время выполнения с помощью рефлексии 358
Второй пример применения рефлексии 361
Получение всех аннотаций 362
Интерфейс `AnnotatedElement` 364
Использование значений по умолчанию 364
Маркерные аннотации 366
Одночленные аннотации 367
Встроенные аннотации 368
Типовые аннотации 371
Повторяющиеся аннотации 376
Некоторые ограничения на аннотации 378

Глава 13. Ввод-вывод, оператор `try` с ресурсами и прочие вопросы 379
Основы ввода-вывода 379
Потоки ввода-вывода 380
Потоки ввода-вывода байтов и символов 380
Предопределенные потоки ввода-вывода 382
Чтение данных, вводимых с консоли 383
Чтение символов 384
Чтение символьных строк 385
Запись данных, выводимых на консоль 386
Класс `PrintWriter` 387
Чтение и запись данных в файлы 388
Автоматическое закрытие файла 395
Модификаторы доступа `transient` и `volatile` 399
Применение операции `instanceof` 399
Модификатор доступа `strictfp` 402
Платформенно-ориентированные методы 402
Применение ключевого слова `assert` 403
Параметры включения и отключения режима проверки утверждений 406

Статический импорт 406
Вызов перегружаемых конструкторов по ссылке `this()` 409
Компактные профили Java API 411

Глава 14. Обобщения 413

Что такое обобщения 414
Простой пример обобщения 414
Обобщения оперируют только ссылочными типами 418
Обобщенные типы различаются по аргументам типа 419
Каким образом обобщения повышают типовую безопасность 419
Обобщенный класс с двумя параметрами типа 421
Общая форма обобщенного класса 423
Ограниченные типы 423
Применение метасимвольных аргументов 426
Ограниченные метасимвольные аргументы 429
Создание обобщенного метода 434
Обобщенные конструкторы 436
Обобщенные интерфейсы 437
Базовые типы и унаследованный код 439
Иерархии обобщенных классов 442
Применение обобщенного суперкласса 442
Обобщенный подкласс 444
Сравнение типов в обобщенной иерархии во время выполнения 446
Приведение типов 449
Переопределение методов в обобщенном классе 449
Выведение типов и обобщения 450
Стирание 451
Мостовые методы 452
Ошибки неоднозначности 454
Некоторые ограничения, присущие обобщениям 455
Получить экземпляр по параметру типа нельзя 456
Ограничения на статические члены 456
Ограничения на обобщенные массивы 456
Ограничения на обобщенные исключения 458

Глава 15. Лямбда-выражения 459

Введение в лямбда-выражения 459
Основные положения о лямбда-выражениях 460
Функциональные интерфейсы 461
Некоторые примеры лямбда-выражений 463
Блочные лямбда-выражения 466
Обобщенные функциональные интерфейсы 468
Передача лямбда-выражений в качестве аргументов 470
Лямбда-выражения и исключения 473
Лямбда-выражения и захват переменных 475
Ссылки на методы 476
Ссылки на статические методы 476
Ссылки на методы экземпляра 478
Ссылки на обобщенные методы 482
Ссылки на конструкторы 485
Предопределенные функциональные интерфейсы 490

Глава 16. Модули 493
Основные положения о модулях 493
Простой пример модуля 494
Компиляция и выполнение первого примера модульного приложения 499
Подробное рассмотрение операторов `requires` и `exports` 500
Модуль `java.base` и платформенные модули 502
Унаследованный код и безымянные модули 503
Экспорт в конкретный модуль 504
Применение оператора `requires transitive` 505
Применение служб 510
Основные положения о службах и поставщиках их услуг 511
Ключевые слова для поддержки служб 512
Пример модульной службы 512
Графы модулей 519
Специальные средства модулей 520
Открытые модули 520
Оператор `opens` 521
Оператор `requires static` 521
Утилита `jlink` и модульные архивные JAR-файлы 522
Связывание файлов в развернутом каталоге 522
Связывание модульных архивных JAR-файлов 523
Файлы формата JMOD 524
Об уровнях и автоматических модулях 524
Заключительные соображения по поводу модулей 525

Часть II. Библиотека Java 527

Глава 17. Обработка символьных строк 529
Конструкторы символьных строк 530
Длина символьной строки 532
Специальные строковые операции 532
Строковые литералы 533
Сцепление строк 533
Сцепление символьных строк с другими типами данных 534
Преобразование символьных строк и метод `toString()` 534
Извлечение символов 536
Метод `charAt()` 536
Метод `getChars()` 536
Метод `getBytes()` 537
Метод `toCharArray()` 537
Сравнение символьных строк 537
Методы `equals()` и `equalsIgnoreCase()` 538
Метод `regionMatches()` 539
Методы `startsWith()` и `endsWith()` 539
Метод `equals()` в сравнении с операцией == 540
Метод `compareTo()` 540
Поиск в символьных строках 542
Видоизменение символьных строк 544
Метод `substring()` 544
Метод `concat()` 545
Метод `replace()` 545
Метод `trim()` 546

Преобразование данных методом valueOf() 546
Смена регистра букв в строке 547
Соединение символьных строк 548
Дополнительные методы из класса String 549
Класс StringBuffer 550
Методы length() и capacity() 551
Метод ensureCapacity() 552
Метод setLength() 552
Методы charAt() и setCharAt() 552
Метод getChars() 553
Метод append() 554
Метод insert() 554
Метод reverse() 555
Методы delete() и deleteCharAt() 555
Метод replace() 556
Метод substring() 557
Дополнительные методы из класса StringBuffer 557
Класс StringBuilder 558
Глава 18. Пакет java.lang 559
Оболочки примитивных типов 560
Класс Number 560
Классы Double и Float 560
Методы isInfinite() и isNaN() 565
Классы Byte, Short, Integer и Long 565
Класс Character 578
Дополнения класса Character для поддержки кодовых точек в Юникоде 581
Класс Boolean 583
Класс Void 584
Класс Process 584
Класс Runtime 586
Управление памятью 588
Выполнение других программ 589
Класс Runtime.Version 590
Класс ProcessBuilder 591
Класс System 594
Измерение времени выполнения программы методом currentTimeMills() 596
Применение метода arraycopy() 597
Свойства окружения 598
Интерфейс System.Logger и класс System.LoggerFinder 599
Класс Object 599
Применение метода clone() и интерфейса Cloneable 600
Класс Class 602
Класс ClassLoader 605
Класс Math 606
Тригонометрические функции 606
Экспоненциальные функции 607
Функции округления 607
Прочие методы из класса Math 609
Класс StrictMath 612
Класс Compiler 612

Классы Thread, ThreadGroup и интерфейс Runnable 613
Интерфейс Runnable 613
Класс Thread 613
Класс ThreadGroup 616
Классы ThreadLocal и InheritableThreadLocal 621
Класс Package 621
Класс Module 623
Класс ModuleLayer 624
Класс RuntimePermission 624
Класс Throwable 624
Класс SecurityManager 624
Класс StackTraceElement 624
Класс StackWalker и интерфейс StackWalker.StackFrame 626
Класс Enum 626
Глава 19. Пакет java.util, часть I. Collections Framework 633
Краткий обзор коллекций 634
Интерфейсы коллекций 636
Интерфейс Collection 637
Интерфейс List 640
Интерфейс Set 642
Интерфейс SortedSet 643
Интерфейс NavigableSet 644
Интерфейс Queue 646
Интерфейс Dequeue 647
Классы коллекций 649
Класс ArrayList 650
Класс LinkedList 654
Класс HashSet 656
Класс LinkedHashSet 657
Класс TreeSet 658
Класс PriorityQueue 659
Класс ArrayDeque 660
Класс EnumSet 661
Доступ к коллекциям через итератор 662
Применение интерфейса Iterator 664
Цикл for в стиле for each как альтернатива итераторам 665
Итераторы-разделители 666
Сохранение объектов пользовательских классов в коллекциях 670
Интерфейс RandomAccess 672
Манипулирование отображениями 672
Интерфейсы отображений 672
Классы отображений 680
Компараторы 686
Применение компараторов 689
Алгоритмы коллекций 695
Массивы 702
Унаследованные классы и интерфейсы 708
Интерфейс Enumeration 709
Класс Vector 709
Класс Stack 714
Класс Dictionary 716

Класс Hashtable 717
Класс Properties 720
Применение методов store() и load() 724
Заключительные соображения по поводу коллекций 726

Глава 20. Пакет java.util, часть II. Прочие служебные классы 727

Класс StringTokenizer 727
Класс BitSet 729
Классы Optional, OptionalDouble, OptionalInt и OptionalLong 733
Класс Date 736
Класс Calendar 738
Класс GregorianCalendar 742
Класс TimeZone 743
Класс SimpleTimeZone 745
Класс Locale 746
Класс Random 747
Классы Timer и TimerTask 750
Класс Currency 753
Класс Formatter 754
Конструкторы класса Formatter 754
Методы из класса Formatter 755
Основы форматирования 756
Форматирование строк и символов 758
Форматирование чисел 758
Форматирование времени и даты 760
Спецификаторы формата %n и %% 761
Указание минимальной ширины поля 762
Указание точности 763
Применение признаков формата 764
Выравнивание выводимых данных 765
Признаки пробела, +, 0 и (765
Признак запятой 766
Признак # 767
Прописные формы спецификаторов формата 767
Применение индекса аргумента 768
Закрытие объекта типа Formatter 769
Аналог функции printf() в Java 770
Класс Scanner 770
Конструкторы класса Scanner 770
Основы сканирования 772
Некоторые примеры применения класса Scanner 775
Установка разделителей 779
Прочие средства класса Scanner 781
Классы ResourceBundle, ListResourceBundle и PropertyResourceBundle 782
Прочие служебные классы и интерфейсы 787
Подпакеты, входящие в состав пакета java.util 788
Пакеты java.util.concurrent, java.util.concurrent.atomic, java.util.concurrent.locks 789
Пакет java.util.function 789
Пакет java.util.jar 792

Пакет java.util.logging 793
Пакет java.util.prefs 793
Пакет java.util.regex 793
Пакет java.util.spi 793
Пакет java.util.stream 793
Пакет java.util.zip 793

Глава 21. Пакет java.io для ввода-вывода 795

Классы и интерфейсы ввода-вывода 796
Класс File 796
Каталоги 800
Применение интерфейса FilenameFilter 801
Альтернативный метод listFiles() 802
Создание каталогов 803
Интерфейсы AutoCloseable, Closeable и Flushable 803
Исключения ввода-вывода 804
Два способа закрытия потоков ввода-вывода 804
Классы потоков ввода-вывода 806
Потоки ввода-вывода байтов 807
Класс InputStream 807
Класс OutputStream 808
Класс FileInputStream 809
Класс FileOutputStream 811
Класс ByteArrayInputStream 813
Класс ByteArrayOutputStream 815
Фильтруемые потоки ввода-вывода байтов 817
Буферизованные потоки ввода-вывода байтов 817
Класс SequenceInputStream 821
Класс PrintStream 823
Классы DataOutputStream и DataInputStream 826
Класс RandomAccessFile 828
Потоки ввода-вывода символов 829
Класс Reader 829
Класс Writer 830
Класс FileReader 831
Класс FileWriter 832
Класс CharArrayReader 833
Класс CharArrayWriter 834
Класс BufferedReader 836
Класс BufferedWriter 837
Класс PushbackReader 838
Класс PrintWriter 839
Класс Console 840
Сериализация 842
Интерфейс Serializable 843
Интерфейс Externalizable 843
Интерфейс ObjectOutput 844
Класс ObjectOutputStream 844
Интерфейс ObjectInput 845
Класс ObjectInputStream 846
Пример сериализации 848
Преимущества потоков ввода-вывода 849

Глава 22. Система ввода-вывода NIO 851
Классы системы ввода-вывода NIO 851
Основные положения о системе ввода-вывода NIO 852
Буферы 852
Каналы 855
Наборы символов и селекторы 856
Усовершенствования в системе NIO.2 856
Интерфейс `Path` 857
Класс `Files` 858
Класс `Paths` 861
Интерфейсы атрибутов файлов 862
Классы `FileSystem`, `FileSystems` и `FileStore` 864
Применение системы ввода-вывода NIO 864
Применение системы NIO для канального ввода-вывода 865
Применение системы NIO для потокового ввода-вывода 876
Применение системы ввода-вывода NIO для операций в файловой системе 879

Глава 23. Работа в сети 889
Основы работы в сети 889
Сетевые классы и интерфейсы 891
Класс `InetAddress` 891
Фабричные методы 892
Методы экземпляра 893
Классы `Inet4Address` и `Inet6Address` 894
Клиентские сокеты по протоколу TCP/IP 894
Класс `URL` 898
Класс `URLConnection` 900
Класс `HttpURLConnection` 903
Класс `URI` 905
Cookie-файлы 905
Серверные сокеты по протоколу TCP/IP 905
Дейтаграммы 906
Класс `DatagramSocket` 907
Класс `DatagramPacket` 908

Глава 24. Обработка событий 911
Два подхода к обработке событий 912
Модель делегирования событий 912
События 912
Источники событий 913
Приемники событий 914
Классы событий 914
Класс `ActionEvent` 916
Класс `AdjustmentEvent` 916
Класс `ComponentEvent` 917
Класс `ContainerEvent` 918
Класс `FocusEvent` 919
Класс `ItemEvent` 921
Класс `KeyEvent` 922
Класс `MouseEvent` 923
Класс `MouseWheelEvent` 924

Класс TextEvent 925
Класс WindowEvent 926
Источники событий 927
Интерфейсы приемников событий 928
Интерфейс ActionListener 929
Интерфейс AdjustmentListener 929
Интерфейс ComponentListener 929
Интерфейс ContainerListener 929
Интерфейс FocusListener 929
Интерфейс ItemListener 930
Интерфейс KeyListener 930
Интерфейс MouseListener 930
Интерфейс MouseMotionListener 930
Интерфейс MouseWheelListener 931
Интерфейс TextListener 931
Интерфейс WindowFocusListener 931
Интерфейс WindowListener 931
Применение модели делегирования событий 931
Основные принципы обработки событий в ГПИ средствами AWT 932
Обработка событий от мыши 933
Обработка событий от клавиатуры 937
Классы адаптеров 940
Внутренние классы 943
Анонимные внутренние классы 945

Глава 25. Введение в библиотеку AWT: работа с окнами, графикой и текстом 947
Классы библиотеки AWT 948
Основные положения об окнах 950
Класс Component 951
Класс Container 951
Класс Panel 952
Класс Window 952
Класс Frame 952
Класс Canvas 952
Работа с обрамляющими окнами 952
Установка размеров окна 953
Сокрытие и отображение окна 953
Установка заголовка окна 953
Закрытие обрамляющего окна 953
Метод paint() 954
Отображение символьной строки 954
Установка цвета переднего и заднего плана 955
Запрос на повторное воспроизведение 955
Создание прикладной программы на основе класса Frame 957
Поддержка графики 957
Рисование линий 958
Рисование прямоугольников 958
Рисование эллипсов и окружностей 959
Рисование дуг 959
Рисование многоугольников 959
Демонстрация методов рисования 960
Изменение размеров графики 962

Работа с цветом 963
Методы из класса `Color` 964
Установка текущего цвета графики 965
Пример программы, демонстрирующий работу с цветом 965
Установка режима рисования 966
Работа со шрифтами 968
Определение доступных шрифтов 970
Создание и выбор шрифта 971
Получение сведений о шрифте 974
Управление форматированием выводимого текста 975

Глава 26. Применение элементов управления, диспетчеров компоновки и меню из библиотеки AWT 979

Основные положения об элементах управления 980
Ввод и удаление элементов управления 980
Реагирование на элементы управления 981
Исключение типа `HeadlessException` 981
Метки 981
Экранные кнопки 983
Обработка событий от кнопок 984
Флажки 988
Обработка событий от флажков 989
Кнопки-переключатели 991
Элементы управления выбором 992
Обработка событий от раскрывающихся списков 993
Использование списков 995
Обработка событий от списков 997
Управление полосами прокрутки 999
Обработка событий от полос прокрутки 1000
Текстовые поля 1002
Обработка событий в текстовых полях 1003
Текстовые области 1005
Диспетчеры компоновки 1007
Класс `FlowLayout` 1008
Класс `BorderLayout` 1010
Вставки 1011
Класс `GridLayout` 1013
Класс `CardLayout` 1015
Класс `GridBagLayout` 1018
Меню и строки меню 1023
Диалоговые окна 1029
О переопределении метода `paint()` 1033

Глава 27. Изображения 1035

Форматы файлов изображений 1035
Основы работы с изображениями: создание, загрузка и отображение 1036
Создание объекта класса `Image` 1036
Загрузка изображения 1037
Воспроизведение изображения 1037
Двойная буферизация 1039

Интерфейс ImageProducer 1042
Класс MemoryImageSource 1042
Интерфейс ImageConsumer 1044
Класс PixelGrabber 1045
Класс ImageFilter 1048
Фильтр класса CropImageFilter 1048
Фильтр класса RGBImageFilter 1050
Дополнительные классы для формирования изображений 1062

Глава 28. Служебные средства параллелизма 1063

Пакеты параллельного прикладного интерфейса API 1064
Пакет java.util.concurrent.atomic 1066
Пакет java.util.concurrent.locks 1066
Применение объектов синхронизации 1066
Класс Semaphore 1067
Класс CountDownLatch 1073
Класс CyclicBarrier 1075
Класс Exchanger 1078
Класс Phaser 1080
Применение исполнителя 1088
Простой пример исполнителя 1089
Применение интерфейсов Callable и Future 1091
Перечисление TimeUnit 1094
Параллельные коллекции 1095
Блокировки 1096
Атомарные операции 1099
Параллельное программирование средствами Fork/Join Framework 1101
Основные классы Fork/Join Framework 1102
Стратегия “разделяй и властвуй” 1106
Первый простой пример вилочного соединения 1107
Влияние уровня параллелизма 1110
Пример применения класса RecursiveTask<V> 1114
Асинхронное выполнение задач 1117
Отмена задачи 1117
Определение состояния завершения задачи 1118
Перезапуск задачи 1118
Предмет дальнейшего изучения 1118
Рекомендации относительно вилочного соединения 1120
Служебные средства параллелизма в сравнении с традиционным подходом к многозадачности в Java 1121

Глава 29. Потоковый прикладной интерфейс API 1123

Основные положения о потоках данных 1123
Потоковые интерфейсы 1124
Получение потока данных 1127
Простой пример потока данных 1128
Операции сведения 1132
Параллельные потоки данных 1135
Отображение 1138
Накопление 1143
Итераторы и потоки данных 1147
Применение итератора в потоке данных 1147

Применение итератора-разделителя 1149
Дальнейшее изучение потокового прикладного интерфейса API 1152

Глава 30. Регулярные выражения и другие пакеты 1153
Обработка регулярных выражений 1153
Класс `Pattern` 1154
Класс `Matcher` 1154
Синтаксис регулярных выражений 1155
Примеры, демонстрирующие совпадение с шаблоном 1156
Два варианта сопоставления с шаблоном 1162
Дальнейшее изучение регулярных выражений 1163
Рефлексия 1163
Удаленный вызов методов 1167
Простое приложение "клиент-сервер", использующее механизм RMI 1168
Форматирование даты и времени средствами пакета `java.text` 1171
Класс `DateFormat` 1172
Класс `SimpledateFormat` 1174
Пакеты из прикладного интерфейса API даты и времени 1176
Основные классы даты и времени 1177
Форматирование даты и времени 1178
Синтаксический анализ символьных строк даты и времени 1181
Дальнейшее изучение пакета `java.time` 1182

Часть III. Введение в программирование ГПИ средствами Swing 1183

Глава 31. Введение в библиотеку Swing 1185
Происхождение библиотеки Swing 1185
Построение библиотеки Swing на основе библиотеки AWT 1186
Главные особенности библиотеки Swing 1186
Легковесные компоненты Swing 1187
Подключаемый стиль оформления 1187
Связь с архитектурой MVC 1188
Компоненты и контейнеры 1189
Компоненты 1189
Контейнеры 1190
Панели контейнеров верхнего уровня 1190
Пакеты библиотеки Swing 1191
Простое Swing-приложение 1192
Обработка событий 1196
Рисование средствами Swing 1200
Основы рисования 1200
Вычисление области рисования 1202
Пример рисования 1203

Глава 32. Исследование библиотеки Swing 1207
Классы `JLabel` и `ImageIcon` 1207
Класс `JTextField` 1209
Кнопки из библиотеки Swing 1211
Класс `JButton` 1212
Класс `JToggleButton` 1214

Флажки 1217
Кнопки-переключатели 1218
Класс JTabbedPane 1221
Класс JScrollPane 1223
Класс JList 1225
Класс JComboBox 1229
Деревья 1232
Класс JTable 1236

Глава 33. Введение в меню Swing 1239
Основные положения о меню 1239
Краткий обзор классов JMenuBar, JMenu и JMenuItem 1241
Класс JMenuBar 1241
Класс JMenu 1243
Класс JMenuItem 1244
Создание главного меню 1245
Ввод мнемоники и оперативных клавиш в меню 1249
Ввод изображений и всплывающих подсказок в пункты меню 1251
Классы JRadioButtonMenuItem и JCheckBoxMenuItem 1253
Создание всплывающего меню 1255
Создание панели инструментов 1259
Действия 1262
Составление окончательного варианта программы MenuDemo 1268
Дальнейшее изучение библиотеки Swing 1274

Часть IV. Введение в программирование ГПИ средствами JavaFX 1275

Глава 34. Введение в JavaFX 1277
Основные понятия JavaFX 1278
Пакеты JavaFX 1278
Классы подмостков и сцены 1279
Узлы и графы сцены 1279
Компоновки 1279
Класс приложения и методы его жизненного цикла 1280
Запуск JavaFX-приложения 1280
Скелет JavaFX-приложения 1281
Компиляция и выполнение JavaFX-приложения 1285
Поток исполнения приложения 1286
Метка — простейший элемент управления в JavaFX 1286
Применение кнопок и событий 1288
Основы обработки событий в JavaFX 1289
Элемент управления экранной кнопкой 1290
Демонстрация обработки событий на примере экранных кнопок 1291
Рисование непосредственно на холсте 1294

Глава 35. Элементы управления JavaFX 1301
Классы Image и ImageView 1301
Ввод изображения в метку 1304
Ввод изображения в экранную кнопку 1306
Класс ToggleButton 1309
Класс RadioButton 1312
Обработка событий изменения в группе кнопок-переключателей 1316
Другой способ управления кнопками-переключателями 1317

Класс CheckBox 1320
Класс ListView 1325
Представление списка с полосами прокрутки 1329
Активизация режима одновременного выбора нескольких элементов из списка 1330
Класс ComboBox 1331
Класс TextField 1335
Класс ScrollPane 1338
Класс TreeView 1342
Эффекты и преобразования 1347
Эффекты 1348
Преобразования 1349
Демонстрация эффектов и преобразований 1350
Ввод всплывающих подсказок 1354
Отключение элементов управления 1354

Глава 36. Введение в меню JavaFX 1355

Основные положения о меню 1355
Краткий обзор классов MenuBar, Menu и MenuItem 1357
Класс MenuBar 1357
Класс Menu 1358
Класс MenuItem 1359
Создание главного меню 1360
Ввод мнемоники и оперативных клавиш в меню 1366
Ввод изображений в пункты меню 1368
Классы RadioMenuItem и CheckMenuItem 1369
Создание контекстного меню 1372
Создание панели инструментов 1375
Составление окончательного варианта приложения MenuDemo 1378
Дальнейшее изучение JavaFX 1385

Часть V. Применение Java 1387

Глава 37. Компоненты Java Beans 1389

Общее представление о компонентах Java Beans 1389
Преимущества компонентов Java Beans 1390
Самоанализ 1390
Проектные шаблоны для свойств компонентов Java Beans 1391
Проектные шаблоны для событий 1392
Методы и проектные шаблоны 1393
Применение интерфейса BeanInfo 1393
Привязанные и ограниченные свойства 1394
Сохраняемость компонентов Java Beans 1394
Настройщики 1395
Прикладной интерфейс Java Beans API 1395
Класс Introspector 1398
Класс PropertyDescriptor 1398
Класс EventSetDescriptor 1398
Класс MethodDescriptor 1398
Пример компонента Java Bean 1398

Глава 38. Введение в сервлеты 1403
Предпосылки для разработки сервлетов 1403
Жизненный цикл сервлета 1404
Варианты разработки сервлетов 1405
Применение контейнера сервлетов Tomcat 1406
Простой пример сервлета 1407
Создание и компиляция исходного кода сервлета 1408
Запуск контейнера сервлетов Tomcat на выполнение 1409
Запуск веб-браузера и запрос сервлета 1409
Прикладной интерфейс Servlet API 1409
Пакет `javax.servlet` 1410
Интерфейс `Servlet` 1410
Интерфейс `ServletConfig` 1411
Интерфейс `ServletContext` 1411
Интерфейс `ServletRequest` 1412
Интерфейс `ServletResponse` 1413
Класс `GenericServlet` 1413
Класс `ServletInputStream` 1414
Класс `ServletOutputStream` 1414
Класс `ServletException` 1414
Ввод параметров сервлета 1414
Пакет `javax.servlet.http` 1416
Интерфейс `HttpServletRequest` 1417
Интерфейс `HttpServletResponse` 1418
Интерфейс `HttpSession` 1419
Класс `Cookie` 1419
Класс `HttpServlet` 1421
Обработка HTTP-запросов и ответов 1422
Обработка HTTP-запросов типа `GET` 1422
Обработка HTTP-запросов типа `POST` 1424
Применение cookie-файлов 1425
Отслеживание сеансов связи 1427

Часть VI. Приложения 1429

Приложение А. Применение документирующих комментариев в Java 1431
Дескрипторы утилиты `javadoc` 1431
Дескриптор `@author` 1432
Дескриптор `{@code}` 1433
Дескриптор `@deprecated` 1433
Дескриптор `{@docRoot}` 1433
Дескриптор `@exception` 1433
Дескриптор `@hidden` 1433
Дескриптор `{@index}` 1434
Дескриптор `{@inheritDoc}` 1434
Дескриптор `{@link}` 1434
Дескриптор `{@linkplain}` 1434
Дескриптор `{@literal}` 1434
Дескриптор `@param` 1435
Дескриптор `@provides` 1435
Дескриптор `@return` 1435

Дескриптор @see 1435
Дескриптор @serial 1436
Дескриптор @serialData 1436
Дескриптор @serialField 1436
Дескриптор @since 1436
Дескриптор @throws 1436
Дескриптор @uses 1436
Дескриптор {@value} 1437
Дескриптор @version 1437
Общая форма документирующих комментариев 1437
Результаты, выводимые утилитой javadoc 1437
Пример применения документирующих комментариев 1438

Приложение Б. Краткий обзор Java Web Start 1440

Назначение Java Web Start 1440
Главные элементы Java Web Start 1441
Упаковка приложений Java Web Start в архивный JAR-файл 1441
Подписание приложений Java Web Start 1442
Запуск приложений Java Web Start с помощью JNLP-файла 1443
Связывание приложения Java Web Start с JNLP-файлом 1444
Экспериментирование с Java Web Start в локальной файловой системе 1445
Создание архивного JAR-файла для приложения ToggleButtonDemo 1446
Создание хранилища ключей и подписание архивного JAR-файла 1447
Создание JNLP-файла для запуска приложения ToggleButtonDemo 1448
Создание краткого HTML-файла StartTBD.html 1449
Ввод JNLP-файла в список Exception Site List на панели управления Java 1449
Выполнение приложения ToggleButtonDemo из браузера 1450
Выполнение приложений Java Web Start с помощью утилиты javaws 1450
Выполнение аплетов средствами Java Web Start 1450

Приложение В. Утилита JShell 1451

Основные положения об утилите JShell 1451
Перечисление, редактирование и повторное выполнение кода 1454
Ввод метода 1455
Создание класса 1456
Применение интерфейса 1457
Вычисление выражений и встроенных переменных 1458
Импорт пакетов 1459
Исключения 1460
Другие команды JShell 1460
Дальнейшее изучение JShell 1461

Приложение Г. Аплеты 1463

Два типа аплетов 1463
Основы разработки аплетов 1464
Класс Applet 1465
Архитектура аплетов 1465
Скелет аплета 1466
Инициализация и прекращение работы аплета 1468
Аплеты на основе библиотеки Swing 1469

Предметный указатель 1472

ГЛАВА

11 Многопоточное программирование

В отличие от некоторых языков программирования, в Java предоставляется встроенная поддержка *многопоточного программирования*. Многопоточная программа содержит две или несколько частей, которые могут выполняться одновременно. Каждая часть такой программы называется *потоком исполнения*, причем каждый поток задает отдельный путь исполнения кода. Следовательно, многопоточность — это особая форма многозадачности.

Вам, вероятнее всего, приходилось сталкиваться с многозадачностью на практике, поскольку она поддерживается почти во всех современных операционных системах. Тем не менее существуют два отдельных вида многозадачности: многозадачность на основе процессов и многозадачность на основе потоков. Важно понимать, в чем состоит отличие этих видов многозадачности. Большинству читателей лучше известна многозадачность на основе процессов. *Процесс*, по существу, является выполняющейся программой. Следовательно, *многозадачность на основе процессов* — это средство, которое позволяет одновременно выполнять две или несколько программ на компьютере. Так, многозадачность на основе процессов позволяет запускать компилятор Java, работая одновременно в текстовом редакторе или посещая веб-сайт. В среде многозадачности на основе процессов программа оказывается наименьшей единицей кода, которую может диспетчеризировать планировщик операционной системы.

В среде *многозадачности на основе потоков* наименьшей единицей диспетчеризируемого кода является поток исполнения. Это означает, что одна программа может выполнять две или несколько задач одновременно. Например, текстовый редактор может форматировать текст даже во время распечатки при условии, что оба эти действия выполняются в двух отдельных потоках исполнения. Таким образом, многозадачность на основе процессов имеет дело с "общей картиной", тогда как многозадачность на основе потоков — с отдельными с подробностями.

Многозадачные потоки исполнения требуют меньших издержек, чем многозадачные процессы. Процессы являются крупными задачами, каждой из которых требуется свое адресное пространство. Связь между процессами ограничена и обходится дорого. Переключение контекста с одного процесса на другой также обходится дорого. С другой стороны, потоки исполнения более просты. Они совместно используют одно и то же адресное пространство и один и тот же крупный процесс.

Связь между потоками исполнения обходится недорого, как, впрочем, и переключение контекста с одного потока исполнения на другой. Несмотря на то что программы на Java пользуются многозадачными средами на основе процессов, такая многозадачность в Java не контролируется, в отличие от многопоточной многозадачности.

Многопоточность позволяет писать эффективные программы, максимально использующие доступные вычислительные мощности в системе. Еще одним преимуществом многопоточности является сведение к минимуму времени ожидания. Это особенно важно для интерактивных сетевых сред, где работают программы на Java, поскольку простои в таких средах — обычное явление. Например, скорость передачи данных по сети намного ниже, чем скорость их обработки на компьютере. Даже чтение и запись ресурсов в локальной файловой системе выполняется намного медленнее, чем их обработка на центральном процессоре (ЦП). И, конечно, пользователь намного медленнее вводит данные с клавиатуры, чем их может обработать компьютер. В однопоточных средах прикладной программе приходится ожидать завершения таких задач, прежде чем переходить к следующей задаче, даже если бо́льшую часть времени программа простаивает, ожидая ввода. Многопоточность помогает сократить простои, поскольку в то время, как один поток исполнения ожидает, другой может выполняться.

Если у вас имеется опыт программирования для таких операционных систем, как Windows, значит, вы уже владеете основами многопоточного программирования. Но то обстоятельство, что в Java можно управлять потоками исполнения, делает многопоточность особенно удобной, поскольку многие мелкие вопросы ее организации решаются автоматически.

Модель потоков исполнения в Java

Исполняющая система Java во многом зависит от потоков исполнения, и все библиотеки классов разработаны с учетом многопоточности. По существу, потоки исполнения используются в Java для того, чтобы обеспечить асинхронность работы всей исполняющей среды. Благодаря предотвращению бесполезной траты циклов ЦП удается повысить эффективность выполнения всего кода в целом.

Ценность многопоточной среды позволяет лучше понять сравнение. В однопоточных системах применяется подход, называемый *циклом ожидания событий с опросом*. В этой модели единственный поток управления выполняется в бесконечном цикле, опрашивая единственную очередь событий, чтобы принять решение, что делать дальше. Как только этот механизм опроса возвращает, скажем, сигнал, что сетевой файл готов к чтению, цикл ожидания событий передает управление соответствующему обработчику событий. И до тех пор, пока тот не возвратит управление, в программе ничего не может произойти. На это расходует драгоценное время ЦП. Это может также привести к тому, что одна часть программы будет господствовать, не позволяя обрабатывать любые другие события. Вообще говоря, когда в однопоточной среде поток исполнения *блокируется* (т.е. приостанавливается) по причине ожидания некоторого ресурса, то приостанавливается выполнение всей программы в целом.

Выгода от многопоточности состоит в том, что основной механизм циклического опроса исключается. Один поток может быть приостановлен без остановки других частей программы. Например, время ожидания при чтении данных из сети или вводе пользователем данных может быть выгодно использовано в любом другом месте программы. Многопоточность позволяет переводить циклы анимации в состояние ожидания на секунду в промежутках между соседними кадрами, не приостанавливая работу всей системы в целом. Когда поток исполнения блокируется в программе на Java, приостанавливается только один заблокированный поток, а все остальные потоки продолжают выполняться.

За последние годы многоядерные системы стали обычным явлением. Безусловно, одноядерные системы все еще широко распространены и применяются. Следует, однако, иметь в виду, что многопоточные средства Java вполне работоспособны в обоих типах систем. В одноядерной системе одновременно выполняющиеся потоки совместно используют ЦП, получая каждый в отдельности некоторый квант времени ЦП. Поэтому в одноядерной системе два или больше потоков фактически не выполняются одновременно, но простаивают в ожидании своей очереди на использование вычислительных ресурсов ЦП. А в многоядерных системах два или больше потоков могут фактически выполняться одновременно. Как правило, это позволяет увеличить эффективность программы и повысить скорость выполнения некоторых операций.

На заметку! Помимо средств многопоточного программирования, описываемых в этой главе, необходимо также исследовать функциональные возможности каркаса Fork/Join Framework. Этот каркас предоставляет эффективные средства для создания многопоточных приложений с автоматическим масштабированием, позволяющим лучше использовать многоядерные системы. Каркас Fork/Join Framework является частью общей поддержки в Java *параллельного программирования*, под которым понимаются методики оптимизации некоторых видов алгоритмов параллельного выполнения в системах с несколькими процессорами. Подробнее о каркасе Fork/Join Framework и других служебных средствах параллелизма речь пойдет в главе 28, а здесь рассматриваются традиционные возможности многопоточного программирования на Java.

Потоки исполнения находятся в нескольких состояниях. Рассмотрим их вкратце. Поток может *выполняться*. Он может быть *готовым к выполнению*, как только получит время ЦП. Работающий поток может быть *приостановлен*, что приводит к временному прекращению его активности. Выполнение приостановленного потока может быть *возобновлено*, что позволяет продолжить его выполнение с того места, где он был приостановлен. Поток может быть *заблокирован* на время ожидания какого-нибудь ресурса. В любой момент поток может быть *прерван*, что приводит к немедленной остановке его исполнения. Однажды прерванный поток исполнения уже не может быть возобновлен.

Приоритеты потоков

Каждому потоку исполнения в Java присваивается свой приоритет, который определяет поведение данного потока по отношению к другим потокам. Приоритеты потоков исполнения задаются целыми числами, определяющими от-

носительный приоритет одного потока над другими. Абсолютное значение приоритета еще ни о чем не говорит, поскольку высокоприоритетный поток не выполняется быстрее, чем низкоприоритетный, когда он является единственным исполняемым потоком в данный момент. Вместо этого приоритет потока исполнения служит для принятия решения при переходе от одного потока к другому. Это так называемое *переключение контекста*. Правила, которые определяют, когда должно происходить переключение контекста, довольно просты и приведены ниже.

- **Поток может добровольно уступить управление.** Для этого достаточно явно уступить очередь на исполнение, приостановить или блокировать поток на время ожидания ввода-вывода. В этом случае все прочие потоки исполнения проверяются, а ресурсы ЦП передаются потоку, имеющему наибольший приоритет и готовому к выполнению.
- **Один поток исполнения может быть вытеснен другим, более приоритетным потоком.** В этом случае низкоприоритетный поток исполнения, который не уступает ЦП, просто вытесняется высокоприоритетным потоком, независимо от того, что он делает. По существу, высокоприоритетный поток выполняется, как только это ему потребуется. Это так называемая *вытесняющая многозадачность* (или *многозадачность с приоритетами*).

Дело усложняется, если два потока исполнения имеют одинаковый приоритет и одновременно претендуют на вычислительные ресурсы ЦП. В таких операционных системах, как Windows, потоки исполнения с одинаковым приоритетом разделяют общее время по кругу. А в операционных системах других типов потоки исполнения с одинаковым приоритетом должны принудительно передавать управление равноправным с ними потокам. Если они этого не делают, другие потоки исполнения не запускаются.

Внимание! В силу отличий в способах переключения потоковых контекстов в операционных системах могут возникать трудности переносимости.

Синхронизация

Многопоточность дает возможность для асинхронного поведения прикладных программ, поэтому при необходимости следует каким-то образом обеспечить синхронизацию. Так, если требуется, чтобы два потока исполнения взаимодействовали и совместно использовали сложную структуру данных вроде связного списка, необходимо найти способ предотвратить возможный конфликт между этими потоками. Это означает предотвратить запись данных в одном потоке исполнения, когда в другом потоке исполнения выполняется их чтение. Для этой цели в Java реализован изящный прием из старой модели межпроцессной синхронизации, называемый *монитором*. Монитор — это механизм управления, впервые определенный Чарльзом Энтони Ричардом Хоаром. Монитор можно рассматривать как маленький ящик, одновременно хранящий только один поток исполнения. Как только поток исполнения войдет в монитор, все другие потоки исполнения должны ожидать до тех пор, пока тот не покинет монитор. Таким образом, монитор может

служить для защиты общих ресурсов от одновременного использования несколькими потоками исполнения.

Для монитора в Java отсутствует отдельный класс вроде `Monitor`. Вместо этого у каждого объекта имеется свой неявный монитор, вход в который осуществляется автоматически, когда для этого объекта вызывается синхронизированный метод. Когда поток исполнения находится в теле синхронизированного метода, ни один другой поток исполнения не может вызвать какой-нибудь другой синхронизированный метод для того же самого объекта. Это позволяет писать очень ясный и краткий многопоточный код, поскольку поддержка синхронизации встроена в сам язык.

Обмен сообщениями

Разделив программу на отдельные потоки исполнения, необходимо организовать их взаимное общение. Для организации взаимодействия потоков исполнения в других языках программирования приходится опираться на средства операционной системы, а это, конечно, влечет за собой накладные расходы. Вместо этого Java обладает ясным и экономичным способом организации взаимодействия нескольких потоков исполнения через вызовы предопределенных методов, которые имеются у всех объектов. Система обмена сообщениями в Java позволяет потоку исполнения войти в синхронизированный метод объекта и ожидать до тех пор, пока какой-нибудь другой поток явно не уведомит его об освобождении требующихся ресурсов.

Класс `Thread` и интерфейс `Runnable`

Многопоточная система в Java построена на основе класса `Thread`, его методах и дополняющем его интерфейсе `Runnable`. Класс `Thread` инкапсулирует поток исполнения. Обратиться напрямую к нематериальному состоянию работающего потока исполнения нельзя, поэтому приходится иметь дело с его заместителем — экземпляром класса `Thread`, который и породил его. Чтобы создать новый поток исполнения, следует расширить класс `Thread` или же реализовать интерфейс `Runnable`.

В классе `Thread` определяется ряд методов, помогающих управлять потоками исполнения. Некоторые из тех методов, которые упоминаются в этой главе, перечислены в табл. 11.1.

Таблица 11.1. Методы управления потоками исполнения из класса `Thread`

Метод	Назначение
`getName`	Получает имя потока исполнения
`getPriority`	Получает приоритет потока исполнения
`isAlive`	Определяет, выполняется ли поток
`join`	Ожидает завершения потока исполнения
`run`	Задает точку входа в поток исполнения
`sleep`	Приостанавливает выполнение потока на заданное время
`start`	Запускает поток, вызывая его метод `run()`

Во всех упоминавшихся до сих пор примерах программ использовался единственный поток исполнения. А далее поясняется, как пользоваться классом `Thread` и интерфейсом `Runnable` для создания потоков исполнения и управления ими, начиная с главного потока, присутствующего в каждой программе на Java.

Главный поток исполнения

Когда программа на Java запускается на выполнение, сразу же начинает исполняться один поток. Он обычно называется *главным потоком исполнения* программы, потому что он запускается вместе с ней. Главный поток исполнения важен по двум причинам.

- От этого потока исполнения порождаются все дочерние потоки.
- Зачастую он должен быть последним потоком, завершающим выполнение программы, поскольку в нем производятся различные завершающие действия.

Несмотря на то что главный поток исполнения создается автоматически при запуске программы, им можно управлять через объект класса `Thread`. Для этого достаточно получить ссылку на него, вызвав метод `currentThread()`, который объявляется как открытый и статический (`public static`) в классе `Thread`. Его общая форма выглядит следующим образом:

```
static Thread currentThread()
```

Этот метод возвращает ссылку на тот поток исполнения, из которого он был вызван. Получив ссылку на главный поток, можно управлять им таким же образом, как и любым другим потоком исполнения. Рассмотрим следующий пример программы:

```
// Управление главным потоком исполнения
class CurrentThreadDemo {
  public static void main(String args[]) {
    Thread t = Thread.currentThread();

    System.out.println("Текущий поток исполнения: " + t);

    // изменить имя потока исполнения
    t.setName("My Thread");
    System.out.println("После изменения имени потока: "
                       + t);

    try {
      for(int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println(
             "Главный поток исполнения прерван");
    }
  }
}
```

В этом примере программы ссылка на текущий поток исполнения (в данном случае — главный поток) получается в результате вызова метода `currentThread()` и сохраняется в локальной переменной `t`. Затем выводятся сведения о потоке исполнения. Далее вызывается метод `setName()` для изменения внутреннего имени потока исполнения. После этого сведения о потоке исполнения выводятся заново. А в следующем далее цикле выводятся цифры в обратном порядке с задержкой на 1 секунду после каждой строки. Пауза организуется с помощью метода `sleep()`. Аргумент метода `sleep()` задает время задержки в миллисекундах. Обратите внимание на блок операторов `try/catch`, в котором находится цикл. Метод `sleep()` из класса `Thread` может сгенерировать исключение типа `InterruptException`, если в каком-нибудь другом потоке исполнения потребуется прервать ожидающий поток. В данном примере просто выводится сообщение, если поток исполнения прерывается, а в реальных программах подобную ситуацию придется обрабатывать иначе. Ниже приведен результат, выводимый данной программой.

```
Текущий поток исполнения: Thread[main,5,main]
После изменения имени потока: Thread[My Thread,5,main]
5
4
3
2
1
```

Обратите внимание на то, что вывод производится тогда, когда переменная `t` служит в качестве аргумента метода `println()`. Этот метод выводит по порядку имя потока исполнения, его приоритет и имя его группы. По умолчанию главный поток исполнения имеет имя `main` и приоритет, равный `5`. Именем `main` обозначается также группа потоков исполнения, к которой относится данный поток. *Группа потоков исполнения* — это структура данных, которая управляет состоянием всей совокупности потоков исполнения в целом. После изменения имени потока исполнения содержимое переменной `t` выводится снова — на этот раз новое имя потока исполнения.

Рассмотрим подробнее методы из класса `Thread`, используемые в приведенном выше примере программы. Метод `sleep()` вынуждает тот поток, из которого он вызывается, приостановить свое выполнение на указанное количество миллисекунд. Общая форма этого метода выглядит следующим образом:

```
static void sleep(long миллисекунд)
      throws InterruptedException
```

Количество миллисекунд, на которое требуется приостановить выполнение, задает аргумент *`миллисекунд`*. Метод `sleep()` может сгенерировать исключение типа `InterruptedException`. У него имеется и вторая, приведенная ниже форма, которая позволяет точнее задать время ожидания в миллисекундах и наносекундах. Вторая форма данного метода может применяться только в тех средах, где предусматривается задание промежутков времени в наносекундах.

```
static void sleep(long миллисекунд, long наносекунд)
      throws InterruptedException
```

Как продемонстрировано в предыдущем примере программы, установить имя потока исполнения можно с помощью метода `setName()`. А для того чтобы получить имя потока исполнения, достаточно вызвать метод `getName()`, хотя это в данном примере программы не показано. Оба эти метода являются членами класса `Thread` и объявляются так, как показано ниже, где *имя_потока* обозначает имя конкретного потока исполнения.

```
final void setName(String имя_потока)
final String getName()
```

Создание потока исполнения

В самом общем смысле для создания потока исполнения следует получить экземпляр объекта типа `Thread`. В языке Java этой цели можно достичь следующими двумя способами:

- реализовав интерфейс `Runnable`;
- расширив класс `Thread`.

В последующих разделах эти способы рассматриваются по очереди.

Реализация интерфейса `Runnable`

Самый простой способ создать поток исполнения состоит в том, чтобы объявить класс, реализующий интерфейс `Runnable`. Этот интерфейс предоставляет абстракцию единицы исполняемого кода. Поток исполнения можно создать из объекта любого класса, реализующего интерфейс `Runnable`. Для реализации интерфейса `Runnable` в классе должен быть объявлен единственный метод `run()`:

```
public void run()
```

В теле метода `run()` определяется код, который, собственно, и составляет новый поток исполнения. Но в методе `run()` можно также вызывать другие методы, использовать другие классы, объявлять переменные таким же образом, как и в главном потоке исполнения. Единственное отличие заключается в том, что в методе `run()` устанавливается точка входа в другой, параллельный поток исполнения в программе. Этот поток исполнения завершится, когда метод `run()` возвратит управление.

После создания класса, реализующего интерфейс `Runnable`, в этом классе следует получить экземпляр объекта типа `Thread`. Для этой цели в классе `Thread` определен ряд конструкторов. Тот конструктор, который должен использоваться в данном случае, выглядит в общей форме следующим образом:

```
Thread(Runnable объект_потока, String имя_потока)
```

В этом конструкторе параметр *объект_потока* обозначает экземпляр класса, реализующего интерфейс `Runnable`. Этим определяется место, где начинается выполнение потока. Имя нового потока исполнения передается данному конструктору в качестве параметра *имя_потока*.

После того как новый поток исполнения будет создан, он не запускается до тех пор, пока не будет вызван метод `start()`, объявленный в классе `Thread`. По существу, в методе `start()` вызывается метод `run()`. Ниже показано, каким образом объявляется метод `start()`.

```
void start()
```

Рассмотрим следующий пример программы, где демонстрируется создание и запуск нового потока на выполнение:

```
// Создать второй поток исполнения
class NewThread implements Runnable {
  Thread t;

  NewThread() {
    // создать новый, второй поток исполнения
    t = new Thread(this, "Демонстрационный поток");
    System.out.println("Дочерний поток создан: " + t);
    t.start(); // запустить поток исполнения
  }

  // Точка входа во второй поток исполнения
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Дочерний поток: " + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Дочерний поток прерван.");
    }
    System.out.println("Дочерний поток завершен.");
  }
}

class ThreadDemo {
  public static void main(String args[]) {
    new NewThread(); // создать новый поток

    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Главный поток: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Главный поток прерван.");
    }
    System.out.println("Главный поток завершен.");
  }
}
```

Новый объект класса `Thread` создается в следующем операторе из конструктора `NewThread()`:

```
t = new Thread(this, "Демонстрационный поток");
```

Передача ссылки this на текущий объект в первом аргументе данного конструктора означает следующее: в новом потоке исполнения для текущего объекта по ссылке this следует вызвать метод run(). Далее в приведенном выше примере программы вызывается метод start(), в результате чего поток исполнения запускается, начиная с метода run(). Это, в свою очередь, приводит к началу цикла for в дочернем потоке исполнения. После вызова метода start() конструктор NewThread() возвращает управление методу main(). Возобновляя свое исполнение, главный поток входит в свой цикл for. Далее потоки выполняются параллельно, совместно используя ресурсы процессора в одноядерной системе, вплоть до завершения своих циклов. Ниже приведен результат, выводимый данной программой (у вас он может оказаться иным в зависимости от конкретной исполняющей среды).

```
Дочерний поток: Thread[Демонстрационный поток, 5, main]
Главный поток: 5
Дочерний поток: 5
Дочерний поток: 4
Главный поток: 4
Дочерний поток: 3
Дочерний поток: 2
Главный поток: 3
Дочерний поток: 1
Дочерний поток завершен.
Главный поток: 2
Главный поток: 1
Главный поток завершен.
```

Как упоминалось ранее, в многопоточной программе главный поток исполнения зачастую должен завершаться последним. На самом же деле, если главный поток исполнения завершается раньше дочерних потоков, то исполняющая система Java может "зависнуть", что характерно для некоторых старых виртуальных машин JVM. В приведенном выше примере программы гарантируется, что главный поток исполнения завершится последним, поскольку главный поток исполнения находится в состоянии ожидания в течение 1000 миллисекунд в промежутках между последовательными шагами цикла, а дочерний поток исполнения — только 500 миллисекунд. Это заставляет дочерний поток исполнения завершиться раньше главного потока. Впрочем, далее будет показано, как лучше организовать ожидание завершения потоков исполнения.

Расширение класса Thread

Еще один способ создать поток исполнения состоит в том, чтобы сначала объявить класс, расширяющий класс Thread, а затем получить экземпляр этого класса. В расширяющем классе должен быть непременно переопределен метод run(), который является точкой входа в новый поток исполнения. Кроме того, в этом классе должен быть вызван метод start() для запуска нового потока на исполнение. Ниже приведена версия программы из предыдущего примера, переделенная с учетом расширения класса Thread.

```
// Создать второй поток исполнения, расширив класс Thread
class NewThread extends Thread {
```

```
  NewThread() {
    // создать новый поток исполнения
    super("Демонстрационный поток");
    System.out.println("Дочерний поток: " + this);
    start(); // запустить поток на исполнение
  }

  // Точка входа во второй поток исполнения
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Дочерний поток: " + i);
         Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Дочерний поток прерван.");
    }
    System.out.println("Дочерний поток завершен.");
  }
}

class ExtendThread {
  public static void main(String args[]) {
    new NewThread(); // создать новый поток исполнения

    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Главный поток: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Главный поток прерван.");
    }

    System.out.println("Главный поток завершен.");
  }
}
```

Эта версия программы выводит такой же результат, как и предыдущая ее версия. Как видите, дочерний поток исполнения создается при конструировании объекта класса `NewThread`, наследующего от класса `Thread`. Обратите внимание на метод `super()` в классе `NewThread`. Он вызывает конструктор `Thread()`, общая форма которого приведена ниже, где параметр *имя_потока* обозначает имя порождаемого потока исполнения.

```
public Thread(String имя_потока)
```

Выбор способа создания потоков исполнения

В связи с изложенным выше могут возникнуть следующие вопросы: почему в Java предоставляются два способа для создания порождаемых потоков исполнения и какой из них лучше? Ответы на эти вопросы взаимосвязаны. В классе `Thread` определяется ряд методов, которые могут быть переопределены в про-

изводных классах. И только один из них *должен* быть *непременно* переопределен: метод run(). Безусловно, этот метод требуется и в том случае, когда реализуется интерфейс Runnable. Многие программирующие на Java считают, что классы следует расширять только в том случае, если они должны быть усовершенствованы или каким-то образом видоизменены. Следовательно, если ни один из других методов не переопределяется в классе Thread, то лучше и проще реализовать интерфейс Runnable. Кроме того, при реализации интерфейса Runnable класс порождаемого потока исполнения не должен наследовать класс Thread, что освобождает его от наследования другого класса. В конечном счете выбор конкретного способа для создания потоков исполнения остается за вами. Тем не менее в примерах, приведенных далее в этой главе, потоки будут создаваться с помощью классов, реализующих интерфейс Runnable.

Создание многих потоков исполнения

В приведенных до сих пор примерах использовались только два потока исполнения: главный и дочерний. Но в прикладной программе можно порождать сколько угодно потоков исполнения. Например, в следующей программе создаются три дочерних потока исполнения:

```
// Создать несколько потоков исполнения
class NewThread implements Runnable {
  String name; // имя потока исполнения
  Thread t;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("Новый поток: " + t);
    t.start(); // запустить поток на исполнение
  }

  // Точка входа в поток исполнения
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println(name + " прерван");
    }
    System.out.println(name + " завершен.");
  }
}

class MultiThreadDemo {
  public static void main(String args[]) {
    new NewThread("Один"); // запустить потоки на исполнение
    new NewThread("Два");
    new NewThread("Три");
```

```
    try {
      // ожидать завершения других потоков исполнения
      Thread.sleep(10000);
    } catch (InterruptedException e) {
      System.out.println("Главный поток прерван");
    }

    System.out.println("Главный поток завершен.");
  }
}
```

Ниже приведен результат, выводимый данной программой (у вас он может оказаться иным в зависимости от конкретной исполняющей среды).

```
Новый поток: Thread[Один, 5, main]
Новый поток: Thread[Два, 5, main]
Новый поток: Thread[Три, 5, main]
Один: 5
Два: 5
Три: 5
Один: 4
Два: 4
Три: 4
Один: 3
Три: 3
Два: 3
Один: 2
Три: 2
Два: 2
Один: 1
Три: 1
Два: 1
Один завершен.
Два завершен.
Три завершен.
Главный поток завершен.
```

Как видите, после запуска на исполнение все три дочерних потока совместно используют общие ресурсы ЦП. Обратите внимание на вызов метода `sleep(10000)` в методе `main()`. Это вынуждает главный поток перейти в состояние ожидания на 10 секунд и гарантирует его завершение последним.

Применение методов `isAlive()` и `join()`

Как упоминалось ранее, нередко требуется, чтобы главный поток исполнения завершался последним. С этой целью метод `sleep()` вызывался в предыдущих примерах из метода `main()` с достаточной задержкой, чтобы все дочерние потоки исполнения завершились раньше главного. Но это неудовлетворительное решение, вызывающее следующий серьезный вопрос: откуда одному потоку исполнения известно, что другой поток завершился? Правда, в классе `Thread` предоставляется средство, позволяющее разрешить этот вопрос.

Определить, был ли поток исполнения завершен, можно двумя способами. Во-первых, для этого потока можно вызвать метод `isAlive()`, определенный в классе `Thread`. Ниже приведена общая форма этого метода.

```
final Boolean isAlive()
```

Метод `isAlive()` возвращает логическое значение `true`, если поток, для которого он вызван, еще исполняется. В противном случае он возвращает логическое значение `false`.

И во-вторых, в классе `Thread` имеется метод `join()`, который применяется чаще, чем метод `isAlive()`, чтобы дождаться завершения потока исполнения. Ниже приведена общая форма этого метода.

```
final void join() throws InetrruptedException
```

Этот метод ожидает завершения того потока исполнения, для которого он вызван. Его имя отражает следующий принцип: вызывающий поток ожидает, когда указанный поток *присоединится* к нему. Дополнительные формы метода `join()` позволяют указывать максимальный промежуток времени, в течение которого требуется ожидать завершения указанного потока исполнения.

Ниже приведена усовершенствованная версия программы из предыдущего примера, где с помощью метода `join()` гарантируется, что главный поток завершится последним. В данном примере демонстрируется также применение метода `isAlive()`.

```
// Применить метод join(), чтобы ожидать завершения
// потоков исполнения
class NewThread implements Runnable {
  String name; // имя потока исполнения
  Thread t;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("Новый поток: " + t);
    t.start(); // запустить поток исполнения
  }

  // Точка входа в поток исполнения
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println(name + " прерван.");
    }
    System.out.println(name + " завершен.");
  }
}

class DemoJoin {
```

```
  public static void main(String args[]) {
    NewThread ob1 = new NewThread("Один");
    NewThread ob2 = new NewThread("Два");
    NewThread ob3 = new NewThread("Три");

    System.out.println("Поток Один запущен: "
                       + ob1.t.isAlive());
    System.out.println("Поток Два запущен: "
                       + ob2.t.isAlive());
    System.out.println("Поток Три запущен: "
                       + ob3.t.isAlive());
    // ожидать завершения потоков исполнения
    try {
      System.out.println("Ожидание завершения потоков.");
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch (InterruptedException e) {
      System.out.println("Главный поток прерван");
    }

    System.out.println("Поток Один запущен: "
                       + ob1.t.isAlive());
    System.out.println("Поток Два запущен: "
                       + ob2.t.isAlive());
    System.out.println("Поток Три запущен: "
                       + ob3.t.isAlive());
    System.out.println("Главный поток завершен.");
  }
}
```

Ниже приведен результат, выводимый данной программой (у вас он может оказаться иным в зависимости от конкретной исполняющей среды). Как видите, потоки прекращают исполнение после того, как управление возвращается из вызовов метода `join()`.

```
Новый поток: Thread[Один, 5, main]
Новый поток: Thread[Два, 5, main]
Новый поток: Thread[Три, 5, main]
Поток Один запущен: true
Поток Два запущен: true
Поток Три запущен: true
Ожидание завершения потоков.
Один: 5
Два: 5
Три: 5
Один: 4
Два: 4
Три: 4
Один: 3
Два: 3
Три: 3
Один: 2
Два: 2
Три: 2
Один: 1
```

```
Два: 1
Три: 1
Два завершен.
Три завершен.
Один завершен.
Поток Один запущен: false
Поток Два запущен: false
Поток Три запущен: false
Главный поток завершен.
```

Приоритеты потоков исполнения

Планировщик потоков использует приоритеты потоков исполнения, чтобы принять решение, когда разрешить исполнение каждому потоку. Теоретически высокоприоритетные потоки исполнения получают больше времени ЦП, чем низкоприоритетные. А на практике количество времени ЦП, которое получает поток исполнения, нередко зависит не только от его приоритета, но и от ряда других факторов. (Например, особенности реализации многозадачности в операционной системе могут оказывать влияние на относительную доступность времени ЦП.) Высокоприоритетный поток исполнения может также вытеснять низкоприоритетный. Например, когда низкоприоритетный поток исполняется, а высокоприоритетный собирается возобновить свое исполнение, прерванное в связи с приостановкой или ожиданием завершения операции ввода-вывода, он вытесняет низкоприоритетный поток.

Теоретически потоки исполнения с одинаковым приоритетом должны получать равный доступ к ЦП. Но не следует забывать, что язык Java предназначен для применения в обширном ряде сред. В одних из этих сред многозадачность реализуется совершенно иначе, чем в других. В целях безопасности потоки исполнения с одинаковым приоритетом должны получать управление лишь время от времени. Этим гарантируется, что все потоки получат возможность выполняться в среде операционной системы с невытесняющей многозадачностью. Но на практике даже в средах с невытесняющей многозадачностью большинство потоков все-таки имеют шанс для исполнения, поскольку во всех потоках неизбежно возникают ситуации блокировки, например в связи с ожиданием ввода-вывода. Когда случается нечто подобное, исполнение заблокированного потока приостанавливается, а остальные потоки могут исполняться. Но если требуется добиться плавной работы многопоточной программы, то полагаться на случай лучше не стоит. К тому же в некоторых видах задач весьма интенсивно используется ЦП. Потоки, исполняющие такие задачи, стремятся захватить ЦП, поэтому передавать им управление следует изредка, чтобы дать возможность выполняться другим потокам.

Чтобы установить приоритет потока исполнения, следует вызвать метод `setPriority()` из класса `Thread`. Его общая форма выглядит следующим образом:

```
final void setPriority(int уровень)
```

Здесь аргумент *`уровень`* обозначает новый уровень приоритета для вызывающего потока исполнения. Значение аргумента *`уровень`* должно быть в пределах от `MIN_PRIORITY` до `MAX_PRIORITY`. В настоящее время эти значения равны со-

ответственно 1 и 10. Чтобы возвратить потоку исполнения приоритет по умолчанию, следует указать значение NORM_PRIORITY, которое в настоящее время равно 5. Эти приоритеты определены в классе Thread как статические конечные (static final) переменные. А для того чтобы получить текущее значение приоритета потока исполнения, достаточно вызвать метод getPriority() из класса Thread, как показано ниже.

```
final int getPriority()
```

Разные реализации Java могут вести себя совершенно иначе в отношении планирования потоков исполнения. Большинство несоответствий возникает при наличии потоков исполнения, опирающихся на вытесняющую многозадачность вместо совместного использования времени ЦП. Наиболее безопасный способ получить предсказуемое межплатформенное поведение многопоточных программ на Java состоит в том, чтобы использовать потоки исполнения, которые добровольно уступают управление ЦП.

Синхронизация

Когда несколько потоков исполнения имеют доступ к одному совместно используемому ресурсу, необходимо гарантировать, что ресурс будет одновременно использован только одним потоком. Процесс, обеспечивающий такое поведение потоков исполнения, называется *синхронизацией*. Как будет показано далее, в Java предоставляется особая поддержка синхронизации на уровне языка.

Ключом к синхронизации является понятие монитора. *Монитор* — это объект, используемый в качестве *взаимоисключающей блокировки*. Только один поток исполнения может в одно и то же время *владеть* монитором. Когда поток исполнения запрашивает блокировку, то говорят, что он *входит* в монитор. Все другие потоки исполнения, пытающиеся войти в заблокированный монитор, будут приостановлены до тех пор, пока первый поток не *выйдет* из монитора. Обо всех прочих потоках говорят, что они *ожидают* монитор. Поток, владеющий монитором, может, если пожелает, повторно войти в него. Синхронизировать прикладной код можно двумя способами, предусматривающими использование ключевого слова synchronized. Оба эти способа будут рассмотрены далее по очереди.

Применение синхронизированных методов

Синхронизация достигается в Java просто, поскольку у объектов имеются свои, неявно связанные с ними мониторы. Чтобы войти в монитор объекта, достаточно вызвать метод, объявленный с модификатором доступа synchronized. Когда поток исполнения оказывается в теле синхронизированного метода, все другие потоки исполнения или любые другие синхронизированные методы, пытающиеся вызвать его для того же самого экземпляра, вынуждены ожидать. Чтобы выйти из монитора и передать управление объектом другому ожидающему потоку исполнения, владелец монитора просто возвращает управление из синхронизированного метода.

Чтобы стала понятнее потребность в синхронизации, рассмотрим сначала простой пример, в котором синхронизация отсутствует, хотя и должна быть осуществлена. Приведенная ниже программа состоит из трех простых классов. Первый из них, `Callme`, содержит единственный метод `call()`. Этот метод принимает параметр `msg` типа `String` и пытается вывести символьную строку `msg` в квадратных скобках. Любопытно отметить, что после того, как метод `call()` выводит открывающую квадратную скобку и символьную строку `msg`, он вызывает метод `Thread.sleep(1000)`, который приостанавливает текущий поток исполнения на одну секунду.

Конструктор следующего класса, `Caller`, принимает ссылку на экземпляры классов `Callme` и `String`, которые сохраняются в переменных `target` и `msg` соответственно. В этом конструкторе создается также новый поток исполнения, в котором вызывается метод `run()` для данного объекта. Этот поток запускается немедленно. В методе `run()` из класса `Caller` вызывается метод `call()` для экземпляра `target` класса `Callme`, передавая ему символьную строку `msg`. И наконец, класс `Synch` начинается с создания единственного экземпляра класса `Callme` и трех экземпляров класса `Caller` с отдельными строками сообщения. Один и тот же экземпляр класса `Callme` передается каждому конструктору `Caller()`.

```
// Эта программа не синхронизирована
class Callme {
  void call(String msg) {
    System.out.print("[" + msg);
    try {
      Thread.sleep(1000);
    } catch(InterruptedException e) {
      System.out.println("Прервано");
    }
    System.out.println("]");
  }
}

class Caller implements Runnable {
  String msg;
  Callme target;
  Thread t;

  public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
    t.start();
  }
  public void run() {
    target.call(msg);
  }
}

class Synch {
  public static void main(String args[]) {
```

```
    Callme target = new Callme();
    Caller ob1 = new Caller(target, "Добро пожаловать");
    Caller ob2 = new Caller(target,
                            "в синхронизированный");
    Caller ob3 = new Caller(target, "мир!");

    // ожидать завершения потока исполнения
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Прервано");
    }
  }
}
```

Ниже приведен результат, выводимый данной программой.

```
[Добро пожаловать[в синхронизированный[мир!]
]
]
```

Как видите, благодаря вызову метода `sleep()` из метода `call()` удается переключиться на исполнение другого потока. Это приводит к смешанному выводу трех строк сообщений. В данной программе отсутствует механизм, предотвращающий одновременный вызов в потоках исполнения одного и того же метода для того же самого объекта, или так называемое *состояние гонок*, поскольку три потока соперничают за окончание метода. В данном примере применяется метод `sleep()`, чтобы добиться повторяемости и наглядности получаемого эффекта. Но, как правило, состояние гонок менее заметно и предсказуемо, поскольку трудно предугадать, когда именно произойдет переключение контекста. В итоге программа может быть выполнена один раз правильно, а другой раз — ошибочно.

Чтобы исправить главный недостаток данной программы, следует *упорядочить* доступ к методу `call()`. Это означает, что доступ к этому методу из потоков исполнения следует разрешить только по очереди. Для этого достаточно предварить объявление метода `call()` ключевым словом `synchronized`, как показано ниже.

```
class Callme {
  synchronized void call(String msg) {
  ...
```

Этим предотвращается доступ к методу `call()` из других потоков исполнения, когда он уже используется в одном потоке. После ввода модификатора доступа `synchronized` в объявление метода `call()` результат выполнения данной программы будет выглядеть следующим образом:

```
[Добро пожаловать]
[в синхронизированный]
[мир!]
```

Всякий раз, когда имеется метод или группа методов, манипулирующих внутренним состоянием объекта в многопоточной среде, следует употребить ключе-

вое слово synchronized, чтобы исключить состояние гонок. Напомним, что, как только поток исполнения входит в любой синхронизированный метод экземпляра, ни один другой поток исполнения не сможет войти в какой-нибудь другой синхронизированный метод того же экземпляра. Тем не менее несинхронизированные методы этого экземпляра по-прежнему остаются доступными для вызова.

Оператор synchronized

Несмотря на всю простоту и эффективность синхронизации, которую обеспечивает создание синхронизированных методов, такой способ оказывается пригодным далеко не всегда. Чтобы стало понятнее, почему так происходит, рассмотрим следующую ситуацию. Допустим, что требуется синхронизировать доступ к объектам класса, не предназначенного для многопоточного доступа. Это означает, что в данном классе не применяются синхронизированные методы. Более того, класс написан сторонним разработчиком, и его исходный код недоступен, а следовательно, в объявление соответствующих методов данного класса нельзя ввести модификатор доступа synchronized. Как же синхронизировать доступ к объектам такого класса? К счастью, существует довольно простое решение этого вопроса: заключить вызовы методов такого класса в блок оператора synchronized. Ниже приведена общая форма оператора synchronized.

```
synchronized(ссылка_на_объект) {
  // синхронизируемые операторы
}
```

Здесь *ссылка_на_объект* обозначает ссылку на синхронизируемый объект. Блок оператора synchronized гарантирует, что вызов метода, являющегося членом того же класса, что и синхронизируемый объект, на который делается указанная *ссылка_на_объект*, произойдет только тогда, когда текущий поток исполнения успешно войдет в монитор данного объекта.

Ниже приведена альтернативная версия программы из предыдущего примера, где в теле метода run() используется синхронизированный блок.

```
// В этой программе используется синхронизированный блок
class Callme {
  void call(String msg) {
    System.out.print("[" + msg);
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      System.out.println("Прервано");
    }
    System.out.println("]");
  }
}

class Caller implements Runnable {
  String msg;
  Callme target;
  Thread t;
```

```
  public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
    t.start();
  }

  // синхронизированные вызовы метода call()
  public void run() {
    synchronized(target) {
      // синхронизированный блок
      target.call(msg);
    }
  }
}

class Synch1 {
  public static void main(String args[]) {
    Callme target = new Callme();
    Caller ob1 = new Caller(target, "Добро пожаловать");
    Caller ob2 = new Caller(target,"в синхронизированный");
    Caller ob3 = new Caller(target, "мир!");

    // ожидать завершения потока исполнения
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Прервано");
    }
  }
}
```

В данном примере метод `call()` объявлен без модификатора доступа `synchronized`. Вместо этого используется оператор `synchronized` в теле метода `run()` из класса `Caller`. Благодаря этому получается тот же правильный результат, что и в предыдущем примере, поскольку каждый поток исполнения ожидает завершения предыдущего потока.

Взаимодействие потоков исполнения

В предыдущих примерах другие потоки исполнения, безусловно, блокировались от асинхронного доступа к некоторым методам. Такое применение неявных мониторов объектов в Java оказывается довольно эффективным, но более точного управления можно добиться, организовав взаимодействие потоков исполнения. Как будет показано ниже, добиться такого взаимодействия особенно просто в Java.

Как обсуждалось ранее, многопоточность заменяет программирование циклов ожидания событий благодаря разделению задач на дискретные, логически обособленные единицы. Еще одно преимущество предоставляют потоки исполнения, исключая опрос. Как правило, опрос реализуется в виде цикла, организуемого для периодической проверки некоторого условия. Как только условие оказывает-

ся истинным, выполняется определенное действие. Но на это расходуется время ЦП. Рассмотрим в качестве примера классическую задачу организации очереди, когда некоторые данные поставляются в одном потоке исполнения, а в другом потоке они потребляются. Чтобы сделать эту задачу более интересной, допустим, что поставщик данных должен ожидать завершения работы потребителя, прежде чем сформировать новые данные. В системах с опросом потребитель данных тратит немало циклов ЦП на ожидание данных от поставщика. Как только поставщик завершит работу, он должен начать опрос, напрасно расходуя лишние циклы ЦП в ожидании завершения работы потребителя данных, и т.д. Ясно, что такая ситуация нежелательна.

Чтобы избежать опроса, в Java внедрен изящный механизм взаимодействия потоков исполнения с помощью методов `wait()`, `notify()` и `notifyAll()`. Эти методы реализованы как конечные в классе `Object`, поэтому они доступны всем классам. Все три метода могут быть вызваны только из синхронизированного контекста. Правила применения этих методов достаточно просты, хотя с точки зрения вычислительной техники они принципиально прогрессивны. Эти правила состоят в следующем.

- Метод `wait()` вынуждает вызывающий поток исполнения уступить монитор и перейти в состояние ожидания до тех пор, пока какой-нибудь другой поток исполнения не войдет в тот же монитор и не вызовет метод `notify()`.
- Метод `notify()` возобновляет исполнение потока, из которого был вызван метод `wait()` для того же самого объекта.
- Метод `notifyAll()` возобновляет исполнение всех потоков, из которых был вызван метод `wait()` для того же самого объекта. Одному из этих потоков предоставляется доступ.

Все эти методы объявлены в классе `Object`, как показано ниже. Существуют дополнительные формы метода `wait()`, позволяющие указать время ожидания.

```
final void wait() throws InterruptedException
final void notify()
final void notifyAll()
```

Прежде чем рассматривать пример, демонстрирующий взаимодействие потоков исполнения, необходимо сделать одно важное замечание. Метод `wait()` обычно ожидает до тех пор, пока не будет вызван метод `notify()` или `notifyAll()`. Но вполне вероятно, хотя и в очень редких случаях, что ожидающий поток исполнения может быть возобновлен вследствие *ложной активизации*. При этом исполнение ожидающего потока возобновляется без вызова метода `notify()` или `notifyAll()`. (По существу, исполнение потока возобновляется без очевидных причин.) Из-за этой маловероятной возможности в компании Oracle рекомендуют вызывать метод `wait()` в цикле, проверяющем условие, по которому поток ожидает возобновления. И такой подход демонстрируется в представленном далее примере программы.

А до тех пор рассмотрим несложный пример программы, неправильно реализующей простую форму поставщика и потребителя данных. Эта программа состоит из четырех классов: Q — синхронизируемой очереди; Producer — поточного объекта, создающего элементы очереди; Consumer — поточного объекта, принимающего элементы очереди; а также PC — мелкого класса, в котором создаются объекты классов Q, Producer и Consumer. Ниже приведен исходный код этой программы.

```
// Неправильная реализация поставщика и потребителя
class Q {
  int n;

  synchronized int get() {
    System.out.println("Получено: " + n);
    return n;
  }

  synchronized void put(int n) {
    this.n = n;
    System.out.println("Отправлено: " + n);
  }
}

class Producer implements Runnable {
  Q q;

  Producer(Q q) {
    this.q = q;
    new Thread(this, "Поставщик").start();
  }

  public void run() {
    int i = 0;

    while(true) {
      q.put(i++);
    }
  }
}

class Consumer implements Runnable {
  Q q;

  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Потребитель").start();
  }

  public void run() {
    while(true) {
      q.get();
    }
  }
}
```

```
class PC {
  public static void main(String args[]) {
    Q q = new Q();
    new Producer(q);
    new Consumer(q);

    System.out.println("Для остановки нажмите Ctrl+C.");
  }
}
```

Несмотря на то что методы `put()` и `get()` синхронизированы в классе `Q`, ничто не остановит переполнение потребителя данными от поставщика, как и ничто не помешает потребителю дважды извлечь один и тот же элемент из очереди. В итоге будет выведен неверный результат, как показано ниже (конкретный результат может быть иным в зависимости от быстродействия и загрузки ЦП).

```
Отправлено: 1
Получено: 1
Получено: 1
Получено: 1
Получено: 1
Получено: 1
Отправлено: 2
Отправлено: 3
Отправлено: 4
Отправлено: 5
Отправлено: 6
Отправлено: 7
Получено: 7
```

Как видите, после того, как поставщик отправит значение `1`, запускается потребитель, который получает это значение пять раз подряд. Затем поставщик продолжает свою работу, поставляя значения от `2` до `7`, не давая возможности потребителю получить их. Чтобы правильно реализовать взаимодействие поставщика и потребителя в рассматриваемом здесь примере программы на Java, следует применить методы `wait()` и `notify()` для передачи уведомлений в обоих направлениях:

```
// Правильная реализация поставщика и потребителя
class Q {
  int n;
  boolean valueSet = false;

  synchronized int get() {
    while(!valueSet)
      try {
        wait();
      } catch(InterruptedException e) {
        System.out.println("Исключение типа "
               + "InterruptedException перехвачено");
      }

      System.out.println("Получено: " + n);
      valueSet = false;
```

```
      notify();
      return n;
  }

  synchronized void put(int n) {
    while(valueSet)
      try {
        wait();
      } catch(InterruptedException e) {
        System.out.println("Исключение типа "
               + "InterruptedException перехвачено");
      }

      this.n = n;
      valueSet = true;
      System.out.println("Отправлено: " + n);
      notify();
  }
}

class Producer implements Runnable {
  Q q;

  Producer(Q q) {
    this.q = q;
    new Thread(this, "Поставщик").start();
  }

  public void run() {
    int i = 0;
    while(true) {
      q.put(i++);
    }
  }
}

class Consumer implements Runnable {
  Q q;

  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Потребитель").start();
  }

  public void run() {
    while(true) {
      q.get();
    }
  }
}

class PCFixed {
  public static void main(String args[]) {
    Q q = new Q();
    new Producer(q);
    new Consumer(q);
```

```
    System.out.println("Для остановки нажмите Ctrl-C.");
  }
}
```

В методе `get()` вызывается метод `wait()`. В итоге исполнение потока приостанавливается до тех пор, пока объект класса `Producer` не уведомит, что данные прочитаны. Когда это произойдет, исполнение потока в методе `get()` возобновится. Как только данные будут получены, в методе `get()` вызывается метод `notify()`. Этим объект класса `Producer` уведомляется, что все в порядке и в очереди можно разместить следующий элемент данных. Метод `wait()` приостанавливает исполнение потока в методе `put()` до тех пор, пока объект класса `Consumer` не извлечет элемент из очереди. Когда исполнение потока возобновится, следующий элемент данных размещается в очереди и вызывается метод `notify()`. Этим объект класса `Consumer` уведомляется, что теперь он может извлечь элемент из очереди.

Ниже приведен результат, выводимый данной программой. Он наглядно показывает, что теперь синхронизация потоков исполнения действует правильно.

```
Отправлено: 1
Получено: 1
Отправлено: 2
Получено: 2
Отправлено: 3
Получено: 3
Отправлено: 4
Получено: 4
Отправлено: 5
Получено: 5
```

Взаимная блокировка

Следует избегать особого типа ошибок, имеющего отношение к многозадачности и называемого *взаимной блокировкой*, которая происходит в том случае, когда потоки исполнения имеют циклическую зависимость от пары синхронизированных объектов. Допустим, что один поток исполнения входит в монитор объекта `X`, а другой — в монитор объекта `Y`. Если поток исполнения в объекте `X` попытается вызвать любой синхронизированный метод для объекта `Y`, он будет блокирован, как и предполагалось. Но если поток исполнения в объекте `Y`, в свою очередь, попытается вызвать любой синхронизированный метод для объекта `X`, то этот поток будет ожидать вечно, поскольку для получения доступа к объекту `X` он должен снять свою блокировку с объекта `Y`, чтобы первый поток исполнения мог завершиться. Взаимная блокировка является ошибкой, которую трудно отладить, по двум следующим причинам.

- Взаимная блокировка возникает очень редко, когда исполнение двух потоков точно совпадает по времени.
- Взаимная блокировка может возникнуть, если в ней участвует больше двух потоков исполнения и двух синхронизированных объектов. (Это означает,

что взаимная блокировка может произойти в результате более сложной последовательности событий, чем в упомянутой выше ситуации.)

Чтобы полностью разобраться в этом явлении, его лучше рассмотреть в действии. В приведенном ниже примере программы создаются два класса, `A` и `B`, с методами `foo()` и `bar()` соответственно, которые приостанавливаются непосредственно перед попыткой вызвать метод из другого класса. Сначала в главном классе `Deadlock` получаются экземпляры классов `A` и `B`, а затем запускается второй поток исполнения, в котором устанавливается состояние взаимной блокировки. В методах `foo()` и `bar()` применяется метод `sleep()`, чтобы стимулировать появление взаимной блокировки.

```
// Пример взаимной блокировки
class A {
  synchronized void foo(B b) {
    String name = Thread.currentThread().getName();

    System.out.println(name + " вошел в метод A.foo()");

    try {
      Thread.sleep(1000);
    } catch(Exception e) {
      System.out.println("Класс A прерван");
    }
    System.out.println(name
               + " пытается вызвать метод B.last()");
    b.last();
  }

  synchronized void last() {
    System.out.println("В методе A.last()");
  }
}

class B {
  synchronized void bar(A a) {
    String name = Thread.currentThread().getName();
    System.out.println(name + " вошел в метод B.bar()");

    try {
      Thread.sleep(1000);
    } catch(Exception e) {
      System.out.println("Класс B прерван");
    }

    System.out.println(name +
               " пытается вызвать метод A.last()");
    a.last();
  }

  synchronized void last() {
    System.out.println("В методе A.last()");
  }
}
```

```
class Deadlock implements Runnable {
  A a = new A();
  B b = new B();

  Deadlock() {
    Thread.currentThread().setName("Главный поток");
    Thread t = new Thread(this, "Соперничающий поток");
    t.start();

    a.foo(b); // получить блокировку для объекта a
              // в данном потоке исполнения
    System.out.println("Назад в главный поток");
  }

  public void run() {
    b.bar(a); // получить блокировку для объекта b
              // в другом потоке исполнения
    System.out.println("Назад в другой поток");
  }

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

Если запустить эту программу на выполнение, то в конечном итоге будет получен следующий результат:

```
Главный поток вошел в метод A.foo()
Соперничающий поток вошел в метод B.bar()
Главный поток пытается вызвать метод B.last()
Соперничающий поток пытается вызвать метод A.last()
```

В связи со взаимной блокировкой придется нажать комбинацию клавиш <Ctrl+C>, чтобы завершить данную программу. Нажав комбинацию клавиш <Ctrl+Pause> на ПК, можно увидеть весь вывод из оперативной памяти (так называемый дамп) потока и кеша монитора. В частности, `Соперничающий поток` владеет монитором объекта `b`, тогда как он ожидает монитор объекта `a`. В то же время `Главный поток` владеет объектом `a` и ожидает получить объект `b`. Следовательно, программа никогда не завершится. Как демонстрирует данный пример, если многопоточная программа неожиданно зависла, то прежде всего следует проверить возможность взаимной блокировки.

Приостановка, возобновление и остановка потоков исполнения

Иногда возникает потребность в приостановке исполнения потоков. Например, отдельный поток исполнения может служить для отображения времени дня. Если пользователю не требуется отображение текущего времени, этот поток исполнения можно приостановить. Но в любом случае приостановить исполнение потока совсем не трудно. Выполнение приостановленного потока может быть легко возобновлено.

Механизм временной или окончательной остановки потока исполнения, а также его возобновления отличался в ранних версиях Java, например Java 1.0, от современных версий, начиная с Java 2. До версии Java 2 методы `suspend()` и `resume()`, определенные в классе `Thread`, использовались в программах для приостановки и возобновления потоков исполнения. На первый взгляд применение этих методов кажется вполне благоразумным и удобным подходом к управлению выполнением потоков. Тем не менее пользоваться ими в новых программах на Java не рекомендуется по следующей причине: метод `suspend()` из класса `Thread` несколько лет назад был объявлен не рекомендованным к употреблению, начиная с версии Java 2. Это было сделано потому, что иногда он способен порождать серьезные системные сбои. Допустим, что поток исполнения получил блокировки для очень важных структур данных. Если в этот момент приостановить исполнение данного потока, блокировки не будут сняты. Другие потоки исполнения, ожидающие эти ресурсы, могут оказаться взаимно блокированными.

Метод `resume()` также не рекомендован к употреблению. И хотя его применение не вызовет особых осложнений, тем не менее им нельзя пользоваться без метода `suspend()`, который его дополняет.

Метод `stop()` из класса `Thread` также объявлен устаревшим, начиная с версии Java 2. Это было сделано потому, что он может иногда послужить причиной серьезных системных сбоев. Допустим, что поток выполняет запись в критически важную структуру данных и успел произвести лишь частичное ее обновление. Если его остановить в этот момент, структура данных может оказаться в поврежденном состоянии. Дело в том, что метод `stop()` вызывает снятие любой блокировки, устанавливаемой вызывающим потоком исполнения. Следовательно, поврежденные данные могут быть использованы в другом потоке исполнения, ожидающем по той же самой блокировке.

Если методы `suspend()`, `resume()` или `stop()` нельзя использовать для управления потоками исполнения, то можно прийти к выводу, что теперь вообще нет никакого механизма для приостановки, возобновления или прерывания потока исполнения. К счастью, это не так. Вместо этого код управления выполнением потока должен быть составлен таким образом, чтобы в методе `run()` периодически проверялось, должно ли исполнение потока быть приостановлено, возобновлено или прервано. Обычно для этой цели служит флаговая переменная, обозначающая состояние потока исполнения. До тех пор, пока эта флаговая переменная содержит флаг "выполняется", метод `run()` должен продолжать выполнение. Если же эта переменная содержит флаг "приостановить", поток исполнения должен быть приостановлен. А если флаговая переменная получает флаг "остановить", то поток исполнения должен завершиться. Безусловно, имеются самые разные способы написать код управления выполнением потока, но основной принцип остается неизменным для всех программ.

В приведенном ниже примере программы демонстрируется применение методов `wait()` и `notify()`, унаследованных из класса `Object`, для управления исполнением потока. Рассмотрим подробнее работу этой программы. Класс `NewThread` содержит переменную экземпляра `suspendFlag` типа `boolean`, используемую

для управления выполнением потока. В конструкторе этого класса она инициализируется логическим значением `false`. Метод `run()` содержит блок оператора `synchronized`, где проверяется состояние переменной `suspendFlag`. Если она принимает логическое значение `true`, то вызывается метод `wait()` для приостановки выполнения потока. В методе `mysuspend()` устанавливается логическое значение `true` переменной `suspendFlag`, а в методе `myresime()` — логическое значение `false` этой переменной и вызывается метод `notify()`, чтобы активизировать поток исполнения. И наконец, в методе `main()` вызываются оба метода `mysuspend()` и `myresime()`.

```
// Приостановка и возобновление исполнения
// потока современным способом
class NewThread implements Runnable {
  String name; // имя потока исполнения
  Thread t;
  boolean suspendFlag;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("Новый поток: " + t);
    suspendFlag = false;
    t.start(); // запустить поток исполнения
  }

  // Точка входа в поток исполнения
  public void run() {
    try {
      for(int i = 15; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(200);
        synchronized(this) {
          while(suspendFlag) {
            wait();
          }
        }
      }
    } catch (InterruptedException e) {
      System.out.println(name + " прерван.");
    }

    System.out.println(name + " завершен.");
  }

  synchronized void mysuspend() {
    suspendFlag = true;
  }

  synchronized void myresume() {
    suspendFlag = false;
    notify();
  }
}

class SuspendResume {
  public static void main(String args[]) {
```

```
    NewThread ob1 = new NewThread("Один");
    NewThread ob2 = new NewThread("Два");

    try {
      Thread.sleep(1000);
      ob1.mysuspend();
      System.out.println("Приостановка потока Один");
      Thread.sleep(1000);
      ob1.myresume();
      System.out.println("Возобновление потока Один");
      ob2.mysuspend();
      System.out.println("Приостановка потока Два");
      Thread.sleep(1000);
      ob2.myresume();
      System.out.println("Возобновление потока Два");
    } catch (InterruptedException e) {
      System.out.println("Главный поток прерван");
    }

    // ожидать завершения потоков исполнения
    try {
      System.out.println("Ожидание завершения потоков.");
      ob1.t.join();
      ob2.t.join();
    } catch (InterruptedException e) {
      System.out.println("Главный поток прерван");
    }

    System.out.println("Главный поток завершен");
  }
}
```

Если запустить эту программу на выполнение, то можно увидеть, как исполнение потоков приостанавливается и возобновляется. Далее в этой книге будут представлены другие примеры, в которых применяется современный механизм управления исполнением потоков. И хотя этот механизм не так прост, как прежний, его следует все же придерживаться, чтобы избежать ошибок во время выполнения. Именно такой механизм *должен* применяться во всяком новом коде.

Получение состояния потока исполнения

Как упоминалось ранее в этой главе, поток исполнения может находиться в нескольких состояниях. Чтобы получить текущее состояние потока исполнения, достаточно вызвать метод `getState()`, определенный в классе `Thread`, следующим образом:

```
Thread.State getState()
```

Этот метод возвращает значение типа `Thread.State`, обозначающее состояние потока исполнения на момент вызова. Перечисление `State` определено в классе `Thread`. (Перечисление представляет собой список именованных констант и подробно обсуждается в главе 12.) Значения, которые может возвратить метод `getState()`, перечислены в табл. 11.2.

Таблица 11.2. Значения, возвращаемые методом getState()

Значение	Состояние
BLOCKED	Поток приостановил выполнение, поскольку ожидает получения блокировки
NEW	Поток еще не начал выполнение
RUNNABLE	Поток в настоящее время выполняется или начнет выполняться, когда получит доступ к ЦП
TERMINATED	Поток завершил выполнение
TIMED_WAITING	Поток приостановил выполнение на определенный промежуток времени, например после вызова метода **sleep()**. Поток переходит в это состояние и при вызове метода **wait()** или **join()**
WAITING	Поток приостановил выполнение, поскольку он ожидает некоторого действия, например вызова версии метода **wait()** или **join()** без заданного времени ожидания

На рис. 11.1 схематически показана взаимосвязь различных состояний потока исполнения.

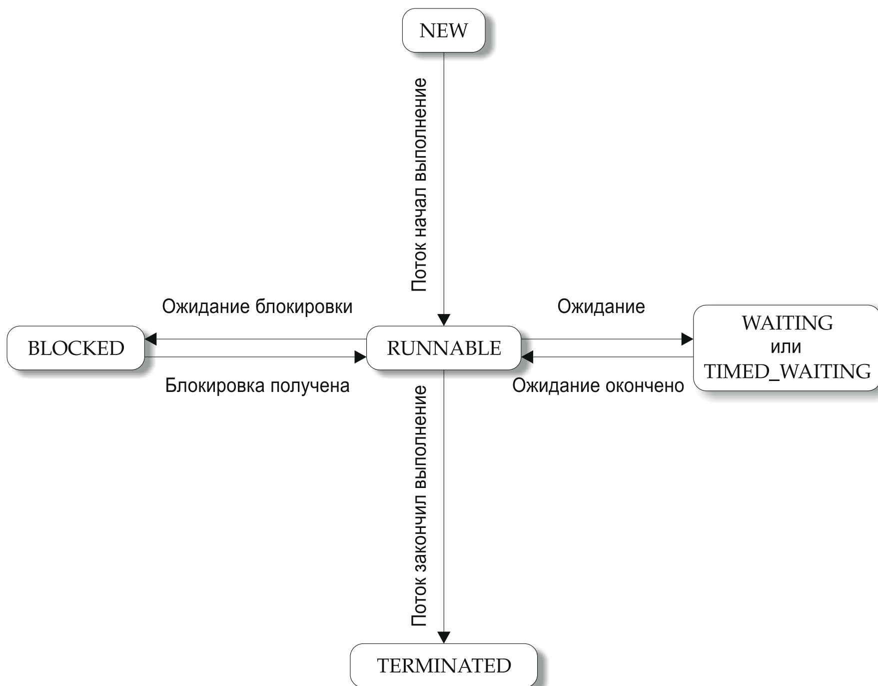

Рис. 11.1. Состояния потока исполнения

Имея в своем распоряжении экземпляр класса `Thread`, можно вызвать метод `getState()`, чтобы получить состояние потока исполнения. Например, в следующем фрагменте кода определяется, находится ли поток исполнения `thrd` в состоянии `RUNNABLE` во время вызова метода `getState()`:

```
Thread.State ts = thrd.getState();
if(ts == Thread.State.RUNNABLE) // ...
```

Следует, однако, иметь в виду, что состояние потока исполнения может измениться после вызова метода `getState()`. Поэтому в зависимости от обстоятельств состояние, полученное при вызове метода `getState()`, мгновение спустя может уже не отражать фактическое состояние потока исполнения. По этой и другим причинам метод `getState()` не предназначен для синхронизации потоков исполнения. Он служит прежде всего для отладки или профилирования характеристик потока во время выполнения.

Одновременное создание и запуск потоков исполнения фабричными методами

Иногда отделять создание потока исполнения от его запуска нежелательно. Иными словами, порой оказывается удобнее создать и сразу же запустить поток на исполнение. Этого можно, например, добиться с помощью статического фабричного метода. *Фабричным* называется такой метод, который возвращает объект своего класса. Как правило, фабричные методы объявляются статическими в своем классе. Они применяются в самых разных целях, например для установки объекта в определенное состояние перед его применением, а иногда и для повторного использования объекта. Что же касается одновременного создания и запуска потоков исполнения, то в фабричном методе сначала создается поток исполнения, затем вызывается метод `start()` для этого потока и, наконец, возвращается ссылка на него. Подобным способом можно создавать и сразу же запускать поток на исполнение в течение одного вызова метода, упрощая тем самым прикладной код.

Если снова обратиться к примеру рассматривавшейся ранее программы `ThreadDemo`, то в класс `NewThread` можно ввести приведенный ниже фабричный метод, позволяющий создавать и сразу же запускать поток на исполнение.

```
// Фабричный метод, создающий и сразу же запускающий
// поток на исполнение
public static NewThread createAndStart() {
  NewThread myThrd = new NewThread();
  myThrd.t.start();
  return myThrd;
}
```

Теперь с помощью метода `createAndStart()` следующие строки кода:

```
NewThread nt = new NewThread(); // создать новый поток
nt.t.start(); // запустить поток на исполнение
```

можно заменить приведенной ниже строкой кода. В итоге поток будет создаваться и сразу же запускаться на выполнение.

```
NewThread nt = NewThread.createAndStart();
```

В тех случаях, когда не требуется хранить ссылку на исполняющийся поток, его можно создать и запустить на исполнение в одной строке кода, не применяя

фабричный метод. Если еще раз обратиться к примеру рассматривавшейся ранее программы ThreadDemo, то в нее можно ввести следующую строку кода, где создается и сразу же запускается на исполнение новый поток типа NewThread:

```
new NewThread().t.start();
```

Но в реальных приложениях обычно требуется хранить ссылку на исполняющийся поток. Поэтому создавать и сразу же запускать его на исполнение лучше с помощью фабричного метода.

Применение многопоточности

Чтобы эффективно пользоваться многопоточными средствами в Java, необходимо научиться мыслить категориями параллельного, а не последовательного выполнения операций. Так, если в программе имеются две подсистемы, которые могут выполняться одновременно, их следует оформить в виде отдельных потоков исполнения. Благоразумно применяя многопоточность, можно научиться писать довольно эффективные программы. Но не следует забывать, что, создав слишком много потоков исполнения, можно снизить производительность программы в целом, вместо того чтобы повысить ее. Следует также иметь в виду, что переключение контекста с одного потока на другой требует определенных издержек. Если создать очень много потоков исполнения, то на переключение контекста будет затрачено больше времени ЦП, чем на выполнение самой программы! И последнее замечание: для создания прикладной программы, предназначенной для интенсивных вычислений и допускающей автоматическое масштабирование с целью задействовать имеющиеся процессоры в многоядерной системе, рекомендуется воспользоваться каркасом Fork/Join Framework, описанным в главе 28.