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.

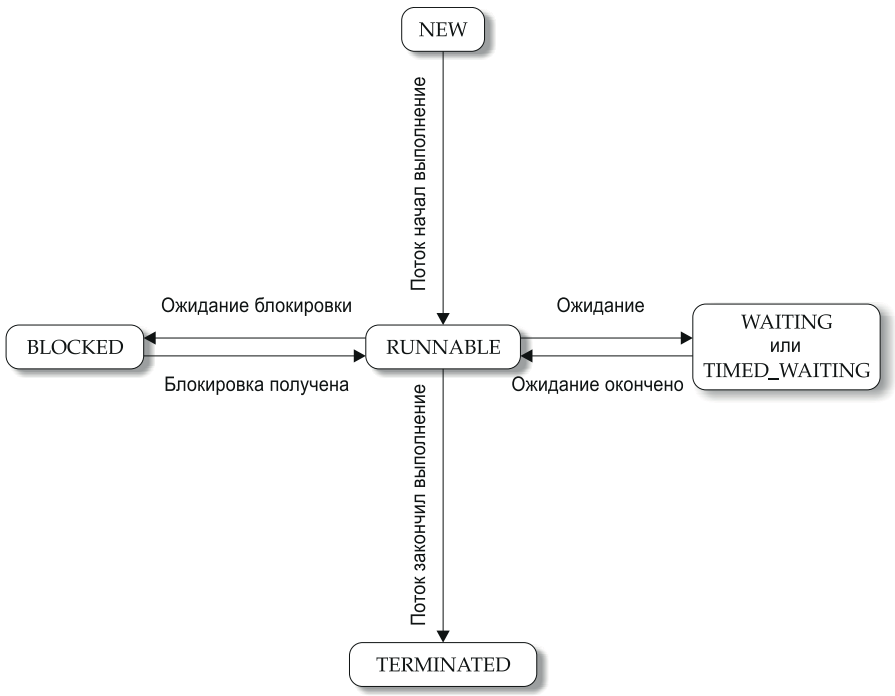

Рис. 11.1. Состояния потока исполнения

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.