1. Многопоточное программирование

В отличие от большинства других машинных языков, Java обеспечивает встроенную поддержку для *многопоточного программирования.* Многопоточная программа содержит две и более частей, которые могут выполняться одновременно, конкурируя друг с другом. Каждая часть такой программы называется *потоком,* а каждый поток определяет отдельный путь выполнения (в последовательности операторов программы). Таким образом, многопоточность — это специализированная форма многозадачности.

Практически все современные операционные системы поддерживают многозадачность. Существуют две формы многозадачности. Одна основана на процессах, а другая — на потоках. *Процесс* — это, по-существу, выполняющаяся программа. Таким образом, основанная на процессах многозадачность — это свойство, которое позволяет компьютеру выполнять несколько программ одновременно, например, выполнять компилятор Java одновременно с использованием текстового редактора. В многозадачности, основанной на процессах, самой мелкой единицей диспетчеризации планировщика является *программа.*

В многозадачной среде, основанной на потоках, самой мелкой единицей диспетчеризации является *поток.* Это означает, что отдельная программа может исполнять несколько *задач* одновременно. Например, текстовый редактор может форматировать текст одновременно с печатью документа, поскольку эти два действия выполняются двумя отдельными потоками. Таким образом, многозадачность, основанная на процессах, имеет дело со всей картиной, а поточная многозадачность обрабатывает детали.

Многозадачные потоки требуют меньших накладных расходов по сравнению с многозадачными процессами. Процессы — это тяжеловесные задачи, которым требуются отдельные адресные пространства. Связи между процессами ограничены и стоят недешево. Переключение контекста от одного процесса к другому также весьма дорогостоящая задача. С другой стороны, потоки достаточно легковесны. Они совместно используют одно и то же адресное пространство и кооперативно оперируют с одним и тем же тяжеловесным процессом, межпоточные связи недороги, а переключение контекста от одного потока к другому имеет низкую стоимость. Хотя Java-программы и используют многозадачное окружение, основанное на процессах, многозадачность такого рода не

находится под управлением языка. Однако многопоточной многозадачностью Java все-таки управляет.

Многопоточность дает возможность писать очень эффективные программы, которые максимально используют CPU, потому что время его простоя можно свести к минимуму. Это особенно важно для интерактивной сетевой среды, в которой работает Java, потому что время простоя является общим. Например, скорость передачи данных по сети намного меньше, чем скорость, с которой компьютер может их обрабатывать. Даже локальные ресурсы файловой системы читаются и записываются намного медленнее, чем они могут быть обработаны CPU. И, конечно, ввод пользователя намного медленнее, чем работа компьютера. В однопоточной среде программа должна ждать окончания каждой своей задачи, прежде чем она сможет перейти к следующей (даже при том, что большую часть времени CPU простаивает). Многопоточность позволяет получить доступ к этому времени простоя и лучше его использовать.

1.1. Поточная модель Java

Исполнительная система Java во многом зависит от потоков, и все библиотеки классов разработаны с учетом многопоточности. Java использует потоки для обеспечения асинхронности во всей среде. Это помогает уменьшить неэффективность, предотвращая затраты циклов CPU.

Однопоточные системы используют подход, называемый *циклом событий с опросом* (event loop with polling). В этой модели, единственный поток управления выполняется в бесконечном цикле, опрашивая единственную очередь событий, чтобы решить, что делать дальше. Как только этот механизм опроса возвращает, скажем, сигнал готовности для чтения сетевого файла, цикл событий передает управление соответствующему обработчику событий. До возврата из этого обработчика в системе ничего больше случиться не может. Это потеря времени CPU может также привести к доминированию одной части программы над системой и задержке обработки других событий. В однопоточной среде, когда поток блокируется (т. е. приостанавливает выполнение), потому что он ожидает некоторый ресурс, останавливается выполнение всей программы.

Выгода от многопоточности Java заключается в том, что устраняется механизм "главный цикл/опрос". Один поток может делать паузу без остановки других частей программы. Например, время простоя, образующееся, когда поток читает данные из сети или ждет ввод пользователя, может использоваться в другом месте. Когда потоки

блокируются в Java-программе, паузу "держит" только один поток — тот, который блокирован. Остальные продолжают выполняться.

Потоки существуют в нескольких состояниях. Поток может быть в состоянии *выполнения.* Может находиться в состоянии *готовности к выполнению,* как только он получит время CPU. Выполняющийся поток может быть *приостановлен,* что временно притормаживает его действие. Затем приостановленный поток может быть *продолжен* (возобновлен) с того места, где он был остановлен. Поток может быть *блокирован* в ожидании ресурса. В любой момент выполнение потока может быть завершено, что немедленно останавливает его выполнение. После завершения поток не может быть продолжен.

1.2. Приоритеты потоков

Java назначает каждому потоку приоритет, который определяет порядок обработки этого потока относительно других потоков. Приоритеты потоков — это целые числа, которые определяют относительный приоритет одного потока над другим. Приоритет потока используется для того, чтобы решить, когда переключаться от одного выполняющегося потока к следующему. Это называется *переключением контекста.* Правила, которые определяют, когда переключение контекста имеет место:

□ Поток может *добровольно отказаться от управления.* Это делается явно, переходом в режим ожидания или блокированием на ожидающем вводе/выводе. В этом сценарии просматриваются все потоки, и CPU передается самому высокоприоритетному потоку, который готов к выполнению.

□ Поток может быть *приостановлен более приоритетным потоком.* В этом случае занимающий процессор низкоприоритетный поток временно останавливается (независимо от того, что он делает) потоком с более высоким приоритетом. Иначе говоря, как только поток с более высоким приоритетом хочет выполниться, он сразу же это делает. Данный механизм называется *упреждающей многозадачностью* (preemptive multitasking).

В случаях, где два потока с одинаковым приоритетом конкурируют за циклы CPU, ситуация немного сложнее. Для операционных систем типа Windows 98, потоки равного приоритета квантуются (во времени) автоматически, циклическим способом. Для других типов операционных систем, типа Solaris 2.x, потоки равного приоритета должны добровольно передавать управление другим потокам. Если они этого не делают, другие потоки не будут выполняться.

1.3. Синхронизация

Поскольку многопоточность обеспечивает *асинхронное* поведение программ, должен существовать способ добиться *синхронности,* когда в этом возникает необходимость. Например, если нужно, чтобы два потока взаимодействовали и совместно использовали сложную структуру данных типа связного списка, нужно каким-то образом гарантировать отсутствие между ними конфликтов. Следует удержать один поток от записи данных, пока другой поток находится в процессе их чтения. Для этой цели Java эксплуатирует старую, но изящную модель синхронизации процессов — монитор. *Монитор* — это механизм управления связью между процессами, первоначально определенный Хором (Hoare, C.A.R). Можно представлять монитор, как очень маленький блок, который содержит только один поток. Как только поток входит в монитор, все другие потоки должны ждать, пока данный не выйдет из монитора. Таким образом, монитор можно использовать для защиты совместно используемого (разделяемого) ресурса от управления несколькими потоками одновременно.

Большинство многопоточных систем создает мониторы как объекты, которые программа должна явно получить и использовать. Java обеспечивает более простое решение. В Java-системе нет класса с именем Monitor. Вместо этого, каждый объект имеет свой собственный неявный монитор, который вводится автоматически при вызове одного из методов объекта. Как только поток оказывается внутри синхронизированного метода, никакой другой поток не может вызывать иной синхронизированный метод того же объекта. Это дает возможность писать очень ясный и краткий многопоточный код, т. к. поддержка синхронизации встроена в язык.

1.4. Передача сообщений

После того как программа разделена на отдельные потоки, нужно определить, как они будут взаимодействовать друг с другом. При программировании на большинстве других языков требуется, в зависимости от операционной системы, установить связь между потоками. Это, конечно, увеличивает издержки (как в программировании, так и в расходовании ресурсов). В противоположность этому, Java обеспечивает ясный, дешевый путь для взаимного общения двух (или нескольких) потоков через вызовы предопределенных методов, которыми обладают все объекты. Система передачи сообщений Java позволяет потоку войти в *синхронизированный* метод на объекте и затем ждать там, пока некоторый другой поток явно не уведомит его о выходе.

Класс Thread и интерфейс Runnable

Многопоточная система Java построена на классе Thread, его методах и связанном с ним интерфейсе Runnable. Thread инкапсулирует поток выполнения. Чтобы создать новый поток, программа должна будет или расширять класс Thread или реализовать интерфейс Runnable.

Класс Thread определяет несколько методов, которые помогают управлять потоками. Табл. 11.1 содержит описание некоторых методов.

Таблица 11.1. *Некоторые методы класса* ***Thread***

Метод	Значение
getName()	Получить имя потока
getPriority()	Получить приоритет потока
isAlive()	Определить, выполняется ли еще поток
join()	Ждать завершения потока
run()	Указать точку входа в поток
sleep()	Приостановить поток на определенный период времени
start()	Запустить поток с помощью вызова его метода run()

1.5. Главный поток

При запуске Java-программы начинает выполняться один поток называемый *главным потоком* программы, потому что он выполняется в начале программы. Главный поток важен по двум причинам:

□ Это поток, из которого будут порождены все другие "дочерние" потоки.

□ Это должен быть последний поток, в котором заканчивается выполнение. Когда главный поток останавливается, программа завершается.

Хотя главный поток создается автоматически после запуска программы, он может управляться через Thread-объект. Для организации управления нужно получить ссылку на него, вызывая метод currentThread(), который является public static членом класса Thread. Вот его общая форма:

```
static Thread currentThread()
```

Этот метод возвращает ссылку на поток, в котором он вызывается. Через ссылку на главный поток можно управлять им точно так же, как любым другим потоком. Начнем со следующего примера:

Программа 58. Управление главным потоком

```
// Файл CurrentThreadDemo.java
// Управление главным потоком.
class CurrentThreadDemo {
   public static void main(String args[]) {
      Thread t = Thread.currentThread(); // Ссылка на главный поток
      System.out.println("Текущий поток:  " + t);
// Изменить имя потока
      t.setName("My Thread");
      System.out.println("После изменения имени:  " + t);
      try {
         for(int n = 5; n > 0; n--) {
            System.out.println(n);
            Thread.sleep(1000); // Засыпаем на 1000 миллисекунд или 1 сек
         }
      }
      catch (InterruptedException e) {
         System.out.println("Главный поток завершен");
      }
   }
}
```

В этой программе ссылка к текущему потоку (главному потоку в этом случае) получена с помощью вызова currentThread() и сохранена в локальной переменной t. Затем, программа отображает информацию о потоке, вызывает метод setName(), чтобы поменять внутреннее имя потока, и вновь выводит информацию о потоке. Далее, запускается обратный (от 5) цикл for, приостанавливающий поток на одну секунду на каждом шаге. Пауза выполняется методом sleep(). Аргумент sleep() определяет время задержки в миллисекундах. Обратите внимание на обрамляющий этот цикл блок try/catch. Метод sleep() класса Thread мог бы выбросить исключение типа InterruptedException если бы некоторый другой поток хотел прервать это ожидание. Данный пример только печатает сообщение, если он получает прерывание. В реальной программе нужно было бы обработать его по-другому. Вывод, сгенерированный этой программой:

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

Обратите внимание на выводы, использующие t в качестве аргумента println(). Они отображают (по порядку): имя потока, его приоритет и имя его группы. По умолчанию имя главного потока — main. Его приоритет равен 5, что тоже является значением по

умолчанию. Последний идентификатор main — имя группы потоков, которой принадлежит данный поток. *Группа потоков* — структура данных, контролирующая состояние совокупности потоков в целом. После изменения имени потока переменная t снова выводится на экран. На сей раз отображается новое имя потока.

Рассмотрим подробнее методы класса Thread, которые используются в программе. Метод sleep() заставляет поток, из которого он вызывается, приостановить выполнение на указанное (в аргументе вызова) число миллисекунд. Его общая форма имеет вид:

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

Число миллисекунд интервала приостановки определяется в параметре milliseconds. Кроме того, данный метод может выбросить исключение типа InterruptedException.

Метод sleep() имеет вторую форму, которая позволяет определять период приостановки в долях миллисекунд и наносекунд:

```
static void sleep(long milliseconds, int nanoseconds)
                  throws InterruptedException
```

Эта вторая форма полезна только в средах, где допускаются короткие периоды времени, измеряемые в наносекундах.

Как показывает предыдущая программа, используя метод setName(), можно *установить* (записать из программы) новое имя потока. Можно также *получить* (прочитать) существующее имя потока, вызывая метод getName() (однако обратите внимание, что эта процедура не показана в программе). Оба метода являются членами класса Thread и объявляются в таких формах:

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

где threadName определяет имя потока. (final запрещает переопределение данных методов в производых классах)

1.6. Создание потока

Для создания *потока* создается *объект* типа Thread. В Java это можно выполнить двумя способами:

- реализовать интерфейс Runnable;
- наследовать класс Thread, определив его подкласс.

В следующих двух разделах рассматривается каждый из указанных способов.

Реализация интерфейса Runnable

Самый простой способ создания потока заключается в определении класса, который реализует интерфейс Runnabie. В Runnabie определен некоторый абстрактный (без тела) модуль выполняемого кода. Создавать поток можно на любом объекте, который реализует интерфейс Runnable. Для реализации Runnable в классе нужно определить только один метод с именем run(). Форма его объявления:

```
public void run()
```

Внутри run() нужно определить код, образующий новый поток. run() может вызывать другие методы, использовать другие классы и объявлять переменные точно так же, как это делает главный (main) поток. Единственное различие состоит в том, что run() устанавливает в данной программе точку входа для запуска другого, конкурирующего потока выполнения. Этот поток завершит свое выполнение при возврате из run().

После создания класса, который реализует Runnable, нужно создать объект типа Thread внутри этого класса. В классе Thread определено несколько конструкторов. Мы будем использовать конструктор следующей формы:

```
Thread (Runnable threadOb, String threadName)
```

Здесь threadOb — объект класса, реализующего интерфейс Runnable. Он определяет, где начнется выполнение нового потока. Имя нового потока определяет параметр threadName.

Созданный поток запускается методом start(), который объявлен в Thread. В действительности start() вызывает run(). Формат метода start():

```
void start()
```

Рассмотрим пример, который создает новый поток и начинает его выполнение:

Программа 59. Создание второго потока

```
// Файл ThreadDemo.java
// Создание второго потока.
class NewThread implements Runnable {
   Thread t;          // Ссылка на поток
   NewThread() {         // Конструктор
// Создать новый, второй поток.
      t = new Thread(this, "Demo Thread");     // (1) Создание объекта класса
Thread
      System.out.println("Дочерний поток:  " + t);
      t.start();         // (2) Стартовать поток, вызывается run()
   }
// Это точка входа во второй поток.
```

```
   public void run() {                        // (3) Переопределение run()
      try {
         for(int i = 5; i > 0; i--) {
            System.out.println("Дочерний поток:  " + i);
            Thread.sleep(500);
         }
      }
      catch (InterruptedException e) {
         System.out.println(
                     "Прерывание дочернего потока.");
      }
      System.out.println("Завершение дочернего потока.");
   }
}
class ThreadDemo {
   public static void main(String args[])   {
      new NewThread();             // (4) 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("Прерывание главного потока.");
      }
      System.out.println("Завершение главного потока.");
   }
}
```

В конструкторе класса NewThread, новый Thread объект создается инструкцией:

```
t = new Thread(this,  "Demo Thread");
```

Передача this указывает, чтобы новый поток на this-объекте вызвал метод run(). Затем вызывается метод start(), который начинает выполнение потока в методе run(). Это приводит к запуску цикла for дочернего потока. После вызова start() конструктор NewThread возвращается к main(). Когда главный поток возобновляет выполнение, он входит в свой for-цикл. Оба потока выполняются, используя CPU совместно, до конца своих циклов. Вывод этой программы следующий:

```
Дочерний поток: Thread[DemoThread,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
Завершение дочернего потока.
```

```
Главный поток: 2
Главный поток: 1
Завершение главного потока.
```

Как уже говорилось, в многопоточной программе главный поток должен заканчивать выполнение последним. Если он заканчивается прежде, чем завершится дочерний поток, то исполнительная система Java может "зависнуть". Предыдущая программа гарантирует, что главный поток заканчивается последним, потому что он бездействует 1000 миллисекунд между итерациями цикла, а дочерний поток — только 500 миллисекунд. Это заставляет дочерний поток завершиться раньше главного. Далее будет описан лучший способ предоставления гарантии более позднего завершения главного потока.

Расширение Thread

Для создания потока создается новый класс, который расширяет `Thread`, а затем — экземпляр этого класса. Расширяющий класс должен переопределять метод `run()`, который является точкой входа для нового потока. Он должен также вызывать `start()`, чтобы начать выполнение нового потока. Ниже приведена предыдущая программа, переписанная так, чтобы расширить `Thread`:

Программа 60. Создание потока расширением Thread

```
// Файл ExtendThread.java
// Создает второй поток расширением класса Thread
class NewThread extends Thread {
   NewThread() {
// Создать новый, второй поток
      super("Demo Thread");// (1) Вызов конструктора суперкласса Thread
      System.out.println("Дочерний поток: " + this);
      start();                // (2) Стартовать поток
   }
   // Это точка входа для второго потока.
   public void run() {            // (3)
      try {
         for(int i = 5; i > 0; i--) {
            System.out.println("Дочерний поток:  " + i);
            Thread.sleep(500);
         }
      }
      catch (InterruptedException e) {
         System.out.println("Прерывание дочернего потока.");
      }
      System.out.println("Завершение дочернего потока.");
   }
}
class ExtendThread {
   public static void main(String args[]) {
```

```
        new NewThread();          // (4) 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("Прерывание главного потока.");
        }
        System.out.println("Завершение главного потока.");
     }
}
```

Эта программа генерирует тот же вывод, что и предшествующая версия. Нетрудно заметить, что дочерний поток создается с помощью объекта NewThread, который получен из Thread.

Обратите внимание на обращение к методу super() внутри NewThread. Оно вызывает следующую форму конструктора Thread:

```
public Thread(String threadName)
```

где threadName определяет имя потока.

Выбор подхода

Java имеет два способа создания дочерних потоков. Класс Thread определяет несколько методов, которые могут быть переопределены производным классом. Из них только один должен быть переопределен обязательно — метод run(). Это, конечно, тот же самый метод, который требовался для реализации интерфейса Runnable. Классы должны расширяться только тогда, когда они каким-то образом улучшаются или изменяются, поэтому если не будет переопределяться никакой другой метода класса Thread, то, вероятно, лучше всего просто реализовать интерфейс Runnable напрямую. Далее мы будем создавать потоки, используя *классы,* которые реализуют интерфейс Runnable.

1.7. Создание нескольких потоков

До сих пор мы использовали только два потока — главный и один дочерний. Однако программа может порождать столько потоков, сколько необходимо. Например, следующая программа создает три дочерних потока:

Программа 61. Создание нескольких потоков

```
// Файл MultiThreadDemo.java
// Создание множественных потоков.
class NewThread implements Runnable {
```

```
   String name;             // Имя потока
   Thread t;
   NewThread(String threadname) {
      name = threadname;
      t = new Thread(this, name);
      System.out.println("Новый поток:  " + t);
      t.start();               // Старт потока
   }
   // Это точка входа для потока.
   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("Первый");                  // Старт потока
      new NewThread("Второй");
      new NewThread("Третий");
      try {
      // Ждать завершения других потоков
         Thread.sleep(10000);
      }
      catch (InterruptedException e) {
         System.out.println("Прерывание главного потока.");
      }
      System.out.println("Завершение главного потока.");
   }
}
```

Вывод этой программы:

```
Новый поток:  Thread[Первый,5,main]
Первый:  5
Новый поток:  Thread[Второй,5,main]
Новый поток:  Thread[Третий,5,main]
Второй:  5
Третий:  5
Первый:  4
Третий:  4
Второй:  4
Первый:  3
Второй:  3
Третий:  3
Первый:  2
Третий:  2
Второй:  2
```

```
Первый:  1
Третий:  1
Второй:  1
Первый завершен.
Третий завершен.
Второй завершен.
```

Как видите, после запуска все три дочерних потока совместно используют CPU. Обратите внимание на вызов sleep(10000) в main(). Он заставляет главный поток бездействовать в течение десяти секунд и гарантирует, что тот закончится последним.

1.8. Использование методов isAlive() и join()

Итак, главный поток должен быть последним завершающимся потоком. В предшествующих примерах это выполнялось вызовом метода sleep() в main() с достаточно длинной задержкой, чтобы гарантировать, что все дочерние потоки закончатся до завершения главного потока. Однако такое решение едва ли удовлетворительно. Кроме того, возникает вопрос: как один поток может знать, закончился ли другой? Класс Thread обеспечивает средства, с помощью которых можно ответить на него.

Существуют два способа определения, закончился ли поток. Один из них позволяет вызывать метод isAlive() на потоке. Этот метод определен в Thread и его общая форма выглядит так:

```
final boolean isAlive()
```

Метод isAlive() возвращает true, если поток, на котором он вызывается еще выполняется. В противном случае возвращается false.

В то время как isAlive() полезен только иногда, чаще для ожидания завершения потока вызывается метод join() следующего формата:

```
final void join() throws InterruptedException
```

Этот метод ждет завершения потока, на котором он вызван. Его имя происходит из концепции перевода потока в состояние ожидания, пока указанный поток не присоединит его. Дополнительные формы join() позволяют определять максимальное время ожидания завершения указанного потока.

Ниже показана улучшенная версия предшествующего примера, использующая join() и гарантирующая, что главный поток останавливается последним. Она также демонстрирует метод isAlive().

Программа 62. Использование join() для создания потоков

```
// Файл DemoJoin.java
```

```
// Использование join() для ожидания окончания потока.
class NewThread implements Runnable {
   String name;          // Имя потока
   Thread t;
   NewThread(String threadname) {
      name = threadname;
      t = new Thread(this, name);
      System.out.println("Новый поток:  " + t);
      t.start();            // Старт потока
   }
// Это точка входа потока.
   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("Первый");
      NewThread ob2 = new NewThread("Второй");
      NewThread ob3 = new NewThread("Третий");
      System.out.println("Первый поток выполняется:  " + ob1.t.isAlive());
      System.out.println("Второй поток выполняется:  " + ob2.t.isAlive());
      System.out.println("Третий поток выполняется:  " + ob3.t.isAlive());
// Ждать завершения потоков
      try {
         System.out.println("Ждите завершения потоков.");
         ob1.t.join();
         System.out.println("Главный дождался окончания (join) первого");
         ob2.t.join();
         System.out.println("Главный дождался окончания (join) второго");
         ob3.t.join();
         System.out.println("Главный дождался окончания (join) третьего");
      }
      catch (InterruptedException e) {
         System.out.println("Прерывание главного потока");
      }
      System.out.println("Первый поток выполняется: " + ob1.t.isAlive());
      System.out.println("Второй поток выполняется: " + ob2.t.isAlive());
      System.out.println("Третий поток выполняется: " + ob3.t.isAlive());
      System.out.println("Завершение главного потока.");
   }
}
```

Пример вывода этой программы:

```
Новый поток:  Thread[Первый,5,main]
Новый поток:  Thread[Второй,5,main]
```

```
Новый поток:  Thread[Третий,5,main]
Первый: 5
Второй: 5
Первый поток выполняется:  true
Второй поток выполняется:  true
Третий: 5
Третий поток выполняется:  true
Ждите завершения потоков.
Третий: 4
Второй: 4
Первый: 4
Первый: 3
Второй: 3
Третий: 3
Третий: 2
Второй: 2
Первый: 2
Первый: 1
Третий: 1
Второй: 1
Первый завершен.
Третий завершен.
Главный дождался окончания (join) первого
Второй завершен.
Главный дождался окончания (join) второго
Главный дождался окончания (join) третьего
Первый поток выполняется: false
Второй поток выполняется: false
Третий поток выполняется: false
Завершение главного потока.
```

Нетрудно видеть, что после возврата из join() потоки прекращают выполняться.

1.9. Приоритеты потоков

Планировщик потоков использует их приоритеты для принятия решений о том, когда нужно разрешать выполнение тому или иному потоку. Теоретически высокоприоритетные потоки получают больше времени CPU, чем низкоприоритетные. На практике, однако, количество времени CPU, которое поток получает, часто зависит от нескольких факторов помимо его приоритета. (Например, относительная доступность времени CPU может зависеть от того, как операционная система реализует многозадачный режим.) Высокоприоритетный поток может также упреждать низкоприоритетный (т. е. перехватывать у него управление процессором). Скажем, когда низкоприоритетный поток выполняется, а высокоприоритетный поток возобновляется (от ожидания на вводе/выводе, к примеру), высокоприоритетный поток будет упреждать низкоприоритетный.

Теоретически, потоки равного приоритета должны получить равный доступ к CPU. Но следует иметь в виду, что Java спроектирован для работы в широком диапазоне сред. Некоторые из этих сред реализуют многозадачный режим существенно иначе, чем другие. Для безопасности потоки, которые совместно используют один и тот же приоритет, должны время от времени уступать друг другу управление. Это гарантирует, что все потоки имеют шанс выполниться под неприоритетной операционной системой. Практически, даже в неприоритетных средах, большинство потоков все еще получают шанс выполняться, потому что большинство из них неизбежно сталкивается с некоторыми блокирующими ситуациями, типа ожидания ввода/вывода. Когда это случается, блокированный поток приостанавливается, а другие могут продолжаться. Но, если нужно плавное многопоточное выполнение, лучше не полагаться на подобную ситуацию. К тому же, некоторые типы задач интенсивно используют CPU. Такие потоки доминируют над CPU и, чтобы иные потоки тоже могли выполняться, необходимо, чтобы эти первые время от времени уступали управление.

Для установки приоритета потока есть метод setPriority(), который является членом класса Thread. Вот его общая форма:

```
final void setPriority(int level)
```

где level определяет новый приоритет для вызывающего потока. Значение параметра level должно быть в пределах диапазона MIN_PRIORITY и MAX_PRIORITY. В настоящее время эти значения равны 1 и 10, соответственно. Чтобы вернуть потоку приоритет, заданный по умолчанию, определите NORM_PRIORITY, который в настоящее время равен 5. Эти приоритеты определены в Thread как final-переменные.

Текущее значение приоритета можно получить, методом getPriority() класса Thread:

```
final int getPriority()
```

Реализации Java могут иметь радикально различное поведение, когда они переходят к планированию. Версии Windows 95/98/NT работают примерно так, как вы и ожидаете. Однако другие версии могут работать совершенно иначе. Большинство несоответствий возникает при работе с потоками, которые полагаются на приоритетное поведение, вместо совместного отказа от времени CPU. Самый безопасный способ получить предсказуемое многоплатформное поведение в Java состоит в том, чтобы использовать потоки, добровольно уступающие управление CPU.

Следующий пример демонстрирует два потока с различными приоритетами, которые не выполняются на приоритетной платформе

таким же образом, как они выполняются на неприоритетной платформе. Один поток устанавливает приоритет на два уровня выше нормального, как определено в Thread.NORM_PRIORITY, а другой — на два уровня ниже. Потоки запускаются и допускаются к выполнению на десять секунд. Каждый поток осуществляет цикл, отсчитывающий несколько итераций. После десяти секунд главный поток останавливает оба потока. Затем на экране отображается количество итераций цикла, сделанных каждым потоком.

Программа 63. Работа с приоритетами потоков

```
// Файл HiLoPri.java

// Демонстрирует приоритеты потоков.
class clicker implements Runnable {
   long click = 0;
   Thread t;
   private volatile boolean running = true;
   public clicker(int p) {
      t = new Thread(this);
      t.setPriority(p);
   }
   public void run() {
      while (running) {
         click++;
      }
   }
   public void stop() {
      running = false;
   }
   public void start() {
      t.start() ;
   }
}
class HiLoPri {
   public static void main(String args[]) {
      Thread.currentThread().setPriority (Thread.MAX_PRIORITY);
      clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
      clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
      lo.start();
      hi.start();
      try {
         Thread.sleep(10000);
      }
      catch (InterruptedException e) {
         System.out.println("Прерывание главного потока.");
      }
      lo.stop();
      hi.stop();
// Ждать завершения дочерних потоков.
      try {
         hi.t.join();
```

```
            lo.t.join();
        }
        catch (InterruptedException e) {
            System.out.println("InterruptedException caught");
        }
        System.out.println("Поток с низким приоритетом:  " + lo.click);
        System.out.println("Поток с высоким приоритетом:  " + hi.click);
    }
}
```

Ниже показан вывод этой программы, выполнявшейся в среде Windows XP. Он указывает, что потоки делали контекстное переключение даже без добровольной уступки CPU или блокировки для ввода/вывода. Высокоприоритетный поток получил приблизительно 99 процентов времени CPU.

```
Поток с низким приоритетом:          6380438
Поток с высоким приоритетом:  1150520475
```

Конечно, точный вывод этой программы зависит от быстродействия CPU и числа других задач, работающих в системе. Когда эта же программа будет выполняться под неприоритетной системой, результаты окажутся другими.

Еще одно замечание относительно предыдущей программы. Обратите внимание, что переменной running (в ее объявлении) предшествует ключевое слово volatile. Использование volatile гарантирует, что значение running просматривается на каждой итерации следующего цикла:

```
while (running) {
   click++;
}
```

Без использования volatile Java имеет возможность оптимизировать цикл таким способом, что значение running будет содержаться в регистре CPU и не обязательно повторно просматриваться в каждой итерации. Использование volatile предотвращает эту оптимизацию, сообщая Java, что running может изменяться способами, напрямую не очевидными для данного кода.

1.10. Синхронизация

Когда несколько потоков нуждаются в доступе к разделяемому ресурсу, им необходим некоторый способ гарантии того, что ресурс будет использоваться одновременно только одним потоком. Процесс, с помощью которого это достигается, называется ***синхронизацией.*** Java обеспечивает поддержку синхронизации на уровне языка.

Ключом к синхронизации является концепция монитора (также называемая *семафором). Монитор* — это объект, который используется

для взаимоисключающей блокировки (mutually exclusive lock), или *mutex*. Только один поток может иметь *собственный* монитор в заданный момент. Когда поток получает блокировку, говорят, что он *вошел* в монитор. Все другие потоки, пытающиеся вводить блокированный монитор, будут приостановлены, пока первый не вышел из монитора. Говорят, что другие потоки ***ожидают*** монитор. При желании поток, владеющий монитором, может повторно вводить тот же самый монитор.

Если вы работали с синхронизацией в других языках, таких как C или C++, то знаете, как сложно ее использовать. Это происходит потому, что большинство языков сами по себе не поддерживают синхронизацию. Вместо этого, чтобы синхронизировать потоки, ваши программы должны применять примитивы операционной системы. К счастью, из-за того, что Java осуществляет синхронизацию через элементы языка, большинство сложностей, связанных с синхронизацией, было устранено.

Синхронизировать код можно двумя способами. Оба используют ключевое слово synchronized и рассмотрены ниже.

Использование синхронизированных методов

Синхронизация в Java проста потому, что каждый объект имеет свой собственный неявный связанный с ним монитор. Чтобы ввести монитор объекта, просто вызывают метод, который был модифицирован ключевым словом synchronized. Пока поток находится внутри синхронизированного метода, все другие потоки, пытающиеся вызвать его (или любой другой синхронизированный метод) на том же самом экземпляре, должны ждать. Чтобы выйти из монитора и оставить управление объектом следующему ожидающему потоку, владелец монитора просто возвращается из синхронизированного метода.

Чтобы понять потребность в синхронизации, начнем с простого примера, который ее не использует, хотя должен. Следующая программа имеет три простых класса. Первый, Callme, содержит одиночный метод с именем call(). Метод call() имеет параметр типа string с именем msg. Этот метод пытается печатать строку msg внутри квадратных скобок. Интересно обратить внимание на то, что после того, как call() печатает открывающую скобку и строку msg, он вызывает Thread.sleep(1000), приостанавливающий текущий поток на одну секунду.

Конструктор следующего класса (с именем Caller) имеет в качестве параметров ссылки на экземпляры класса Callme и string, которые сохраняются в экземплярных переменных target и msg, соответственно. Кроме того, конструктор создает новый поток, вызывающий метод run()

его объекта. Поток стартует немедленно. Метод run() класса Caller вызывает метод call() на экземпляре target класса Callme, передавая ему (в качестве аргумента) строку msg. Затем, запускается класс synch, создавая одиночный Callme-экземпляр и три caller-экземпляра (каждый — с уникальной строкой сообщения). Каждому Caller-объекту (ob1, ob2 и ob3) передается один и тот же экземпляр Callme (target).

Программа 64. Несинхронизированная программа

```
// Файл Synch.java
// Эта программа не синхронизирована.
class Callme {           // Обычный класс
   void call(String msg) {
      System.out.print("[" + msg);
      try {
         Thread.sleep(1000);
      }
      catch(InterruptedException e) {
         System.out.println("Прерывание");
      }
      System.out.println("]");
   }
}
class Caller implements Runnable {    // Класс потоковый
   String msg;
   Callme target;                  // Ссылка на объект класса Callme
   Thread t;
   public Caller(Callme targ, String s) {
      target = targ;
      msg = s;
      t = new Thread(this);
      t.start();
   }
   public void run() {
      target.call(msg);     // Выводит [msg и засыпает на 1000 млсек
                            // и выводит ]
   }
}
class Synch {
   public static void main(String args[]) {
      Callme target = new Callme();
      Caller ob1 = new Caller(target,  "Привет");
      Caller ob2 = new Caller(target,  "Синхронизированный");
      Caller ob3 = new Caller(target,  "Мир");
      // Ждать завершения потоков
      try {
         ob1.t.join();
         ob2.t.join();
         ob3.t.join();
      }
      catch(InterruptedException e) {
         System.out.println ("Прерывание") ;
      }
```

```
   }
}
```

Вывод этой программы:

```
[Привет[Синхронизированный[Мир]
]
]
```

Правильный вывод программы должен иметь вид:

```
[Привет]
[Синхронизированный]
[Мир]
```

Вызывая sleep(), метод call() разрешает переключать выполнение на другой поток. Это приводит к выводу перепутанных строк сообщений. В данной программе нет возможности предохранить три потока от вызова одного и того же метода на одном и том же объекте в одно и то же время. Это известно как ***состояние состязаний*** (гонок), потому что три потока состязаются друг с другом, чтобы завершить метод. Представленный пример использовал метод sleep(), чтобы сделать эффекты повторяемыми и очевидными. В большинстве ситуаций состояние состязания более тонко и менее предсказуемо, потому что невозможно точно знать, когда произойдет переключение контекста. Это может заставить программу некоторое время выполняться правильно, а затем — неправильно.

Чтобы стабилизировать предшествующую программу, нужно *сериализовать* (преобразовать в последовательную форму) доступ к методу call(). То есть требуется организовать доступ к нему только одного потока одновременно. Для этого нужно просто в начало определения метода call() вставить ключевое слово synchronized, например, как в следующем фрагменте:

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

…

}
```

Это предохраняет потоки от вызова call(), пока его использует другой поток. После добавления synchronized вывод программы становится таким:

```
[Привет]
[Мир]
[Синхронизированный]
```

Всякий раз, когда имеется метод (или группа методов), который управляет внутренним состоянием объекта в многопоточной ситуации, нужно использовать ключевое слово synchronized для предотвращения

состязаний. Как только поток вызывает синхронизированный метод на некотором экземпляре, никакой другой поток не может вызвать никакой другой синхронизированный метод на том же самом экземпляре. Однако несинхронизированные методы на этом экземпляре остаются вызываемыми.

Оператор synchronized

Хотя определения синхронизированных методов внутри классов — это простые и эффективные средства достижения синхронизации, они не будут работать во всех случаях. Чтобы понять, почему так происходит, рассмотрим следующую ситуацию. Пусть нужно синхронизировать доступ к объектам класса, который не был разработан для многопоточного доступа. То есть класс не использует синхронизированные методы. Кроме того, этот класс был создан третьим лицом, и нет доступа к исходному коду. Таким образом, нет возможности добавлять спецификатор synchronized к соответствующим методам в классе. Для синхронизации доступа к объекту этого класса нужно поместить вызовы методов, определенных этим классом внутрь синхронизированного блока. Вот общая форма оператора synchronized:

```
synchronized(object)   {
// операторы для синхронизации
}
```

где object — ссылка на объект, который нужно синхронизировать. Если нужно синхронизировать одиночный оператор, то фигурные скобки можно опустить. Блок гарантирует, что вызов метода, который является членом объекта object, происходит только после того, как текущий поток успешно ввел монитор объекта.

Вот альтернативная версия предыдущего примера, использующая синхронизированный блок в методе run():

Программа 65. Синхронизированный блок

```
// Файл Synch1.java
// Эта программа использует синхронизированный блок,
class Callme {
   void call(String msg)  {
      System.out.print("[" + msg);
      try {
         Thread.sleep(1000);
      }
      catch (InterruptedException e) {
         System.out.println("Прерывание") ;
      }
      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 ();
   }
// синхронизировать обращения к 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, "Привет");
      Caller ob2 = new Caller(target,  "Синхронизированный");
      Caller ob3 = new Caller(target,  "Мир");
// ждать завершения потоков
      try {
         ob1.t.join();
         ob2.t.join ();
         ob3.t.join();
      }
      catch(InterruptedException e) {
         System.out.println ("Прерывание") ;
      }
   }
}
```

Здесь метод call() не модифицируется ключевым словом synchronized. Вместо этого внутри метода run() класса caller используется оператор synchronized. Он выполняет такой же правильный вывод, как в предыдущем примере, потому что каждый поток ожидает завершения предшествующего потока перед своим продолжением.

1.11. Межпоточные связи

Предыдущие примеры безоговорочно блокировали другие потоки от асинхронного доступа к некоторым методам. Использование неявных мониторов в объектах Java — довольно мощное средство, но можно достичь более тонкого уровня управления через *связь между процессами.*

Как обсуждалось ранее, многопоточность заменяет программирование цикла событий делением задач на дискретные и логические модули. Потоки также обеспечивают и второе

преимущество — они отменяют опрос. Опрос обычно реализуется циклом, который используется для повторяющейся проверки некоторого условия. Как только условие становится истинным, предпринимается соответствующее действие. На этом теряется время CPU. Например, рассмотрим классическую проблему организации очереди, где один поток производит некоторые данные, а другой — их потребляет. Чтобы сделать задачу более интересной, предположим, что, прежде чем генерировать данные, производитель должен ждать, пока потребитель не закончит свою работу. В системе опроса, потребитель тратил бы впустую много циклов CPU на ожидание конца работы производителя. Как только производитель закончил свою работу, он вынужден начать опрос, затрачивая много циклов CPU на ожидание конца работы потребителя. Ясно, что такая ситуация нежелательна.

Чтобы устранить опросы, Java содержит изящный механизм межпроцессовой связи через методы wait(), notify() и notifyAll(). Они реализованы как final-методы в классе object, поэтому доступны всем классам. Все три метода можно вызывать только внутри синхронизированного метода. Правила для использования этих методов:

□ wait() сообщает вызывающему потоку, что нужно уступить монитор и переходить в режим ожидания ("спячки"), пока некоторый другой поток не введет тот же монитор и не вызовет notify();

□ notify() "пробуждает" первый поток, который вызвал wait() на том же самом объекте;

□ notifyAll() пробуждает все потоки, которые вызывали wait(), на том же самом объекте. Первым будет выполняться самый приоритетный поток.

Эти методы объявляются в классе object в следующей форме:

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

Существуют дополнительные формы wait(), которые позволяют указать период времени ожидания.

Следующий пример программы неправильно реализует простую форму задачи производитель/потребитель. Программа состоит из четырех классов: Q — очередь, которую вы пробуете синхронизировать; Producer — поточный объект, который производит элементы ввода в очередь; consumer — поточный объект, который потребляет элементы ввода очереди; и PC — крошечный класс, который создает одиночные классы Q, Producer и Consumer.

Программа 66. Некорректная связь производителя и потребителя

```
// Файл PC.java
// Некорректная реализация производителя и потребителя.
class Q {                    // Очередь
   int n;
   synchronized int get() { // Получить
      System.out.println("Received:  " + n);   // Получить n
      try{
         Thread.sleep(200);
      }
      catch(InterruptedException e) {
         System.out.println("Сон get() прерван");
      }
      return n;
   }
   synchronized void put(int nn) {// Положить
      n = nn;
      System.out.println("Given:  " + n);   // Дано n
      try{
         Thread.sleep(200);
      }
      catch(InterruptedException e) {
         System.out.println("Сон get() прерван");
      }
   }
   Q()          // Конструктор
   { n = 0;}
}
class Producer implements Runnable {  // Производитель
   Q q;               // Частью этого класса является очередь q
   Producer(Q qq) {
      this.q = qq;
      new Thread(this, "Producer").start();
   }
   public void run() {
      int i = 0;
      while(true) {
         q.put(i++);
         if(i == 6)
            break;       // Выход из цикла
      }
   }
}
class Consumer implements Runnable {  // Потребитель
   Q q;
   Consumer (Q qq) {
      this.q = qq;
      new Thread(this, "Consumer").start();
   }
   public void run() {
      while(true) {
         q.get () ;
```

```
            if(5 == q.n)
               break;
         }
      }
   }
   class PC {
      public static void main(String args[]) {
         Q q = new Q();        // Создаем очередь
         new Producer(q);      // Передаем эту призводителю
         new Consumer(q);      // Эту очередь передаем потребителю
         System.out.println("To interrupt, press Control-C.");
      }
   }
```

Хотя методы put() и get() класса Q синхронизированы, ничто не останавливает производителя от переполнения потребителя, так же, как ничто не будет останавливать потребителя от потребления одного и того же значения очереди дважды. Таким образом, получается следующий ошибочный вывод (точный вывод будет меняться в зависимости от быстродействия процессора и загруженности задачами):

```
To interrupt, press Control-C.
Given:  0
Received:  0
Given:  1
Given:  2
Received:  2
Received:  2
Received:  2
Received:  2
Received:  2
Received:  2
Received:  2
Received:  2
Received:  2
Given:  3
Received:  3
Given:  4
Given:  5
Received:  5
```

Заметьте, что после выдачи производителем элемента (2) потребитель запускается и получает ту же 2 девять раз (это видно в последовательных строчках вывода). Значение 4 не потреблено потребителем.

Для получения корректной версии этой программы необходимо использовать методы wait() и notify(), чтобы сигнализировать в обоих направлениях, как показано ниже:

Программа 67. Корректная связь производителя и потребителя

```
// Файл PCFixed.java
// Корректная реализация поставщика и потребителя.
class Q {
   int n;
   boolean valueSet = false;// false - значение не установлено
   synchronized int get() { // Взять
      if(!valueSet)      // Если не установлено значение в очереди,
         try {
            wait();  // надо ждать
         }
         catch(InterruptedException e) {
            System.out.println("InterruptedException выброшено");
         }
      System.out.println("Получено: " + n);
      valueSet = false;     // Значение получено,
      notify();             // можно устанавливать новое, пробуждае put()
      return n;
   }
   synchronized void put(int nn) { // Положить
      if(valueSet)          // Если значение уже есть в очереди,
         try {
            wait () ;       // надо ждать
         }
         catch(InterruptedException e) {
            System.out.println("InterruptedException выброшено");
         }
      this.n = nn;          // Установка значения в очередь
      valueSet = true;
      System.out.println("Отдано:  " + n);
      notify ();               // Можно его брать
   }
}
class Producer implements Runnable {
   Q q;
   Producer(Q qq) {      // Производитель связывается с очередью
      this.q = qq;
      new Thread(this, "Producer").start();
   }
   public void run() {
      int i = 0;
      while(true) {
         if(i > 7)
            break;
         q.put(i++);
      }
   }
}
class Consumer implements Runnable {
   Q q;
   Consumer(Q qq) {
      this.q = qq;
      new Thread(this, "Consumer").start();
   }
   public void run() {
      while(true) {
```

```
            q.get ();
            if(q.n == 7)
               break;
         }
      }
}
class PCFixed {
   public static void main(String args[]) {
      Q q = new Q();
      new Producer(q);   // Производитель и потребитель
      new Consumer(q);   // используют ОДИН объект q
      System.out.println("Для прерывания нажмите Control-C.");
   }
}
```

Внутри get() вызывается wait(), что приостанавливает его выполнение, пока Producer не уведомит его, что некоторые данные готовы, вызвав notify(). Тогда выполнение внутри get() возобновляется и get() вызывает notify(), сообщая объекту Producer, что можно дальше размещать данные в очереди. Внутри put() метод wait() приостанавливает выполнение, пока Consumer не удалит элемент из очереди. Когда выполнение возобновляется, в очередь помещается следующий элемент данных и вызывается notify(). Это извещает Consumer, что он модет теперь удалить элемент.

Вывод этой программы показывает ее чисто синхронное поведение:

```
Отдано 1
Получено 1
Отдано 2
Получено 2
Отдано 3
Получено 3
Отдано 4
Получено 4
Отдано 5
Получено 5
```

1.12. Блокировка

Специальный тип ошибки, которую нужно избегать и которая специально относится к многозадачности, это — (взаимная) *блокировка.* Она происходит, когда два потока имеют циклическую зависимость от пары синхронизированных объектов. Например, предположим, что один поток вводит монитор в объект x, а другой поток вводит монитор в объект y. Если поток в x пробует вызвать любой синхронизированный метод объекта y, это приведет к блокировке, как и ожидается. Однако если поток в y, в свою очередь, пробует вызвать любой синхронизированный метод объекта x, то он будет всегда ждать, т. к. для получения доступа к x, он был бы должен снять свою собственную

блокировку с y, чтобы первый поток мог завершиться. Взаимоблокировка — трудная ошибка для отладки по двум причинам:

□ Вообще говоря, она происходит очень редко, когда интервалы временного квантования двух потоков находятся в определенном соотношении.

□ Она может включать больше двух потоков и синхронизированных объектов. (То есть блокировка может происходить через более замысловатые последовательность событий, чем только что описано.)

Для полного понимания блокировки полезно увидеть ее в действии. С дующий пример создает два класса (A и B) с методами foo() и bar(), соответственно, которые делают краткую паузу перед попыткой вызвать методы другого класса. Главный класс (с именем Deadlock) создает экземпляры типа A и B, и затем стартует второй поток для установки состояния блокировки. Оба метода используют sleep() для вызова состояния блокировки.

Программа 68. Блокировка

```
// Файл Deadlock.java
// Пример блокировки.
class A {
   synchronized void foo(B b) {
      String name = Thread.currentThread().getName();
      System.out.println(name + " вошел в A.foo");
      try {
         Thread.sleep(1000);
      }
      catch(Exception e) {
         System.out.println("A прерван");
      }
      System.out.println(name + " пытается вызвать B.last()");
      b.last();
   }
   synchronized void last() {
      System.out.println("Внутри A.last") ;
   }
}
class B {
   synchronized void bar(A a) {
      String name = Thread.currentThread().getName();
      System.out.println(name + " вошел в B.bar");
      try {
         Thread.sleep(1000);
      }
      catch(Exception e) {
         System.out.println("B прерван");
      }
      System.out.println(name + " пытается вызвать A.last()");
      a.last();
   }
```

```
   synchronized void last() {
      System.out.println("Внутри B.last");
   }
}
class Deadlock implements Runnable {
   A a = new A();     // a, b – два объекта
   B b = new B();
   Deadlock() {
      Thread.currentThread().setName("MainThread");
      Thread t = new Thread (this, "RacingThread");
      t.start();
      a.foo(b);             // Получить блокировку на a в этом потоке
      System.out.println("Возврат в главный поток");
   }
   public void run() {
      b.bar(a);          // Получить блокировку на b в другом потоке
      System.out.println("Возврат в другой поток");
   }
   public static void main(String args[]) {
      new Deadlock();
   }
}
```

Когда вы выполните эту программу, то увидите следующий вывод:

```
MainThread вошел в A.foo
RacingThread вошел в B.bar
MainThread пытается вызвать B.last()
RacingThread пытается вызвать A.last()
```

Поскольку программа была блокирована, вы должны нажать клавиши <Ctrl>+<C>, чтобы закончить программу. Вы можете увидеть полный кэш-дамп потока и монитора, если нажмете клавиши <Ctrl>+<Break> на PC (или <Ctrl>+<\> на Solaris). Вы увидите, что RacingThread имеет монитор на b, в то время как он ожидает монитор на a. В то же самое время, MainThread имеет a и ожидает получения b. Эта программа никогда не будет завершена. Как показывает данный пример, если ваша многопоточная программа иногда блокируется, то блокировка — это одно из первых состояний, которое вы должны проверить.

1.13. Приостановка, возобновление и остановка потоков

Приостановка выполнения потока иногда полезна. Например, отдельные потоки могут использоваться, чтобы отображать время дня. Если пользователь не хочет видеть отображения часов, то их поток может быть приостановлен. В любом случае приостановка потока — простое дело. После приостановки перезапуск потока также не сложен.

Механизмы приостановки, остановки и возобновления потоков различны для Java 2 и более ранних версий Java. Хотя для всего нового кода следует использовать подход Java 2, необходимо все-таки понять, как подобные операции выполнялись в более ранних средах Java. Например, вам требуется модифицировать или поддерживать программы старых версий. Вы также должны понимать, почему были сделаны изменения для Java 2. По этим причинам, следующий раздел описывает первоначальный способ управления выполнением потока, а следующий — новый подход, используемый в Java 2.

Приостановка, возобновление и остановка потоков в Java 1.1 и более ранних версиях

До Java 2 для приостановки и перезапуска выполнения потока программа использовала методы suspend() и resume(), которые определены в классе Thread. Они имеют такую форму:

```
final void suspend ()
final void resume ()
```

Представленная далее программа демонстрирует эти методы:

Программа 69. Приостановка и возобновление потоков с использование suspend() и resume()

```
// Файл SuspendResume.java
// Использование suspend() и resume().
class NewThread implements Runnable {
   String name;             // Имя потока
   Thread t;
   NewThread(String threadname) {
      name = threadname;
      t = new Thread(this, name);
      System.out.println("Новый поток: " + t);
      t.start();             // Запуск (старт)  потока
   }
// Это точка входа для потока.
   public void run() {
      try {
         for (int i = 15; i > 0; i--) {
            System.out.println(name + ": " + i);
            Thread.sleep(200);
         }
      }
      catch (InterruptedException e) {
         System.out.println(name + " прерван.");
      }
      System.out.println(name + " завершен.");
   }
}
class SuspendResume {
```

```
   public static void main(String args[]) {
      NewThread ob1 = new NewThread("Первый");
      NewThread ob2 = new NewThread("Второй");
      try {
         Thread.sleep(1000);   // Спит главный поток
         ob1.t.suspend();
         System.out.println("Приостановка первого потока");
         Thread.sleep(1000);
         Ob1.t.resume();
         System.out.println("Перезапуск Первого потока");
         ob2.t.suspend();
         System.out.println("Приостановка Второго потока");
         Thread.sleep(1000);
         ob2.t.resume();
         System.out.println("Перезапуск Второго потока");
      }
      catch (InterruptedException e) {
         System.out.println("Прерывание главного потока");
      }
// Ждать завершения потока
      try {
         System.out.println("Ждите завершения потоков.");
         ob1.t.join();
         ob2.t.join();
      }
      catch (InterruptedException e) {
         System.out.println("Прерывание главного потока.");
      }
      System.out.println("Завершение главного потока.");
   }
}
```

Пример вывода этой программы:

```
Новый поток: Thread[One/5,main] Первый: 15
Новый поток: Thread[Two,5,main]
Второй: 15
Первый: 14
Второй: 14
Первый: 13
Второй: 13
Первый: 12
Второй: 12
Первый: 11
Второй: 11
Приостановка Первого потока
Второй: 10
Второй:  9
Второй: 8
Второй: 7
Второй:  6
Перезапуск Первого потока
Приостановка Второго потока
Первый: 10
Первый:  9
```

```
Первый: 8
Первый:  7
Первый:  6
Перезапуск Второго потока
Ждите завершения потоков.
Второй: 5
Первый: 5
Второй: 4
Первый: 4
Второй: 3
Первый: 3
Второй: 2
Первый: 2
Второй: 1
Первый: 1
Второй завершен.
Первый завершен.
Завершение главного потока.
```

Класс Thread также определяет метод с именем stop(), который останавливает поток. Его сигнатура имеет следующий вид:

```
void stop()
```

Если поток был остановлен, то его нельзя перезапускать с помощью метода

```
resume ().
```

Приостановка, возобновление и остановка потока в Java 2

Хотя применение методов suspend(), resume() и stop(), определенных в классе Thread, кажется совершенно разумным и удобным подходом к управлению выполнением потоков, они не должны использоваться в новых Java-программах. И вот почему. Метод suspend() класса Thread исключен из Java 2. Это было сделано, потому что suspend() может иногда вызывать серьезные системные отказы. Предположим, что поток получил блокировки на критических структурах данных. Если этот поток приостанавливается в такой точке, то указанные блокировки не отменяются. Так что другие потоки, ожидающие эти ресурсы, могут быть заблокированы.

Метод resume() также исключен. Он не вызывает проблемы, но не может использоваться без своего напарника — метода suspend().

Метод stop() класса Thread также исключен из Java 2. Это было сделано, потому что данный метод может иногда вызывать серьезные системные отказы. Предположим, что поток пишет в критически важную структуру данных и завершил только часть ее изменений. Если

он останавливается в такой точке, то структура данных может оказаться в разрушенном состоянии.

Поскольку в Java 2 запрещено использовать методы suspend(), resume() или stop() для управления потоком, можно подумать, что нет никакого способа для приостановки, перезапуска или завершения потока. Но, к счастью, это не верно. Вместо этого, поток должен быть спроектирован так, чтобы метод run() периодически проверял, должен ли этот поток приостанавливать, возобновлять или останавливать свое собственное выполнение. Это, как правило, выполняется применением флажковой переменной, которая указывает состояние выполнения потока. Пока флажок установлен на "выполнение", метод run() должен продолжать позволять потоку выполняться. Если эта переменная установлена на "приостановить", поток должен сделать паузу. Если она установлена на "стоп", поток должен завершиться. Конечно, существует много способов записи такого кода, но центральная тема будет одна и та же для всех программ.

Следующий пример иллюстрирует, как методы wait() и notify(), унаследованные от object, можно использовать для управления выполнением потока. Данный пример подобен программе в предыдущем разделе. Однако вызовы исключенных методов были удалены. Рассмотрим работу этой программы.

Класс NewThread содержит boolean-переменную экземпляра с именем suspendFlag, которая используется для управления выполнением потока. Она инициализирована (с помощью конструктора) значением false. Метод run() содержит синхронизированный блочный оператор, который проверяет suspendFlag. Если эта переменная — true, то вызывается метод wait(), чтобы приостановить выполнение потока. Метод mysuspend() устанавливает в suspendFlag значение true. Метод myresume() устанавливает в suspendFlag значение false и вызывает метод notify(), чтобы пробудить поток. Наконец, метод main() был модифицирован для вызова методов mysuspend() и myresume().

Программа 70. Приостановка и возобновление потоков для Java 2

```
// Файл SuspendResume.java
// Приостанавливает и возобновляет поток для Java 2
class NewThread implements Runnable {
   String name;             // Имя потока
   Thread t;
   boolean suspendFlag;
   NewThread(String threadname) {
      name = threadname;
      t = new Thread(this, name);
      System.out.println("Новый поток:  " + t);
      suspendFlag = false;
```

```
      t.start();               // Старт потока
   }
// Это точка входа для потока.
   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.");
   }
   void mysuspend() {
      suspendFlag = true;
   }
   synchronized void myresume() {
      suspendFlag = false;
      notify();
   }
}
class SuspendResume {
   public static void main(String args[]) {
      NewThread ob1 = new NewThread("Первый");
      NewThread ob2 = new NewThread("Второй");
      try {
         Thread.sleep(1000);
         ob1.mysuspend();
         System.out.println("Приостановка Первого потока");
         Thread.sleep(1000);
         ob1.myresume();
         System.out.println("Перезапуск Первого потока");
         ob2.mysuspend();
         System.out.println("Приостановка Второго потока");
         Thread.sleep(1000);
         ob2.myresume();
         System.out.println("Перезапуск Второго потока");
      }
      catch (InterruptedException e) {
         System.out.println("Прерывание главного потока");
      }
// ждать завершения потока
      try {
         System.out.println("Ждите завершения потоков.");
         ob1.t.join();
         ob2.t.join();
      }
      catch (InterruptedException e) {
```

```
            System.out.println("Прерывание главного потока.");
        }
        System.out.println("Завершение главного потока.");
    }
}
```

Вывод этой программы идентичен показанному в предыдущем разделе. Позже вы увидите больше примеров, которые используют новый (принадлежащий языку Java 2) механизм управления потоками. Хотя этот механизм не столь же "чист" как предыдущий способ, однако, данная методика способна гарантировать отсутствие ошибок времени выполнения. В этом заключается подход, который ***должен*** использоваться для всех новых программ.

1.14. Использование многопоточности

Ключом к эффективному использованию многопоточности, поддерживаемой Java, является скорее параллельное (одновременное), чем последовательное мышление. Например, когда есть две подсистемы внутри программы, которые могут выполняться одновременно, можно организуйте их индивидуальными потоками. С использованием многопоточности можно создавать очень эффективные программы. Однако, если создано слишком много потоков, то, напротив, ухудшится эффективность программы, так как с переключением контекста связаны некоторые издержки и на изменения контекстов будет потрачено больше времени CPU, чем на выполнение программы.