Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования «Чувашский государственный университет имени И.Н. Ульянова»

Многопоточное программирование

Лекция 16

Назарова Ольга Васильевна

Многопоточность — это специализированная форма многозадачности.

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.

Поточная модель Java. Класс Thread и интерфейс Runnable

Потоки существуют в нескольких состояниях: готовности, блокирования, остановки, возобновления.

Многпоточная 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. 16.1 содержит описание некоторых методов.

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()

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()

Пример управления 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");
      }
   }
}
```

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.

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. После создания класса, который реализует Runnable, нужно создать объект типа Thread внутри этого класса. Для этого можно использовать конструктор:

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()

```
// Фаил 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");
```

Использование методов isAlive() и join()

Главный поток должен быть последним завершающимся потоком. В предшествующих примерах это выполнялось вызовом метода sleep() в main() с длительной задержкой, чтобы все остальные потоки успели завершиться.

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.

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.Только один поток может иметь собственный монитор в заданный момент. Все другие потоки, пытающиеся вводить блокированный монитор, будут приостановлены, пока первый не вышел из монитора.

Синхронизировать код можно в Java двумя способами.

Использование синхронизированных методов

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, надо выйти из синхронизированного метода.

```
// 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]
```

Оператор synchronized

Хотя определения синхронизированных методов внутри классов — это простые и эффективные средства достижения синхронизации, они не будут работать во всех случаях. Нужно поместить вызовы методов в синхронизированный блок.

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

где object — ссылка на объект, который нужно синхронизировать. Блок гарантирует, что вызов метода, который является членом объекта object, происходит только после того, как текущий поток успешно ввел монитор объекта.