Язык программирования Java

Мультимедийный курс

автор: Васильев А.Н.

www.vasilev.kiev.ua

Киев 2017

Многопоточное программирование

- Знакомство с потоками
- Способы создания дочерних потоков
- Явная реализация интерфейса Runnable
- Создание потока с использованием анонимного класса
- Создание потока с использованием лямбда-выражения
- Наследование класса Thread
- Работа с потоками
- Синхронизация потоков

Знакомство с потоками

В Java есть встроенная поддержка для многопоточного программирования. При многопоточном программировании несколько блоков программного кода выполняются одновременно. Такие одновременной выполняемые блоки программного кода называются потоками

С одним потоком мы неявно уже имели дело - это главный поток программы, который фактически и есть сама программа. Из главного потока можно запускать другие потоки, которые называют дочерними

Для запуска дочернего потока нам необходимо:

- определить программный код, который должен выполняться в рамках запускаемого (дочернего) потока
- запустить программный код в режиме потока

Знакомство с потоками

Для определения программного кода (для выполнения в дочернем потоке), необходимо создать объект класса, реализующего интерфейс `Runnable`:

- В интерфейсе `Runnable` объявлен метод `run()`
- При создании объекта на основе класса, реализующего интерфейс `Runnable`, определяется код метода `run()`
- Код метода `run()` выполняется при выполнении потока

- Для запуска дочернего потока необходимо создать объект класса `Thread` и вызвать из этого объекта метод `start()`
- Необходимо связать объект, в методе `run()` которого реализован программный код потока, с объектом класса `Thread`, из которого вызывается метод `start()` для запуска потока
- Такое "связывание" происходит при создании объекта класса `Thread`: аргументом конструктору класса `Thread` передается объект с методом `run()`

Знакомство с потоками

Класс `Thread` реализует интерфейс `Runnable`, но только с пустым телом для метода `run()`: в классе `Thread` метод `run()` определен так, что в нем нет команд

Если создать подкласс путем наследования класса `Thread`, то в этом подклассе можно переопределить метод `run()` (и соответственно, класс реализует интерфейс `Runnable`), у этого класса будет метод `start()`, который можно использовать для запуска дочернего потока

Поэтому один из путей создания дочернего потока базируется на создании подкласса класса `Thread`

Явная реализация интерфейса Runnable

Реализация интерфейса Runnable - 1

```
// Класс реализует интерфейс Runnable:
class MyThread implements Runnable{
   // Описание метода run() (программный код для потока):
   public void run(){
      for(int i=1;i<=5;i++){
         // Отображение сообщения:
         System.out.println("Дочерний поток:\t"+i);
         try{
            // Задержка в выполнении потока:
            Thread.sleep(1200);
         }
         // Обработка исключения:
         catch(InterruptedException e){
            System.out.println("Прерывание дочернего потока");
         }
      }
   }
}
// Продолжение на следующем слайде!!!
```

Явная реализация интерфейса Runnable

Реализация интерфейса Runnable - 2

```
// Главный класс:
class CreatingThreadDemo{
   // Главный поток:
   public static void main(String[] args){
      System.out.println("Начинается выполнение программы");
      // Создание объекта класса Thread для запуска
      // дочернего потока на выполнение:
      Thread t=new Thread(new MyThread());
      System.out.println("Запускается дочерний поток");
      // Запуск дочернего потока:
      t.start();
      for(int k=0;k<=5;k++){
         // Отображение сообщения:
         System.out.println("Главный поток:\t"+(char)('A'+k));
         try{
            // Задержка в выполнении главного потока:
            Thread.sleep(2000);
         } // Обработка исключения:
         catch(InterruptedException e){
            System.out.println("Прерывание главного потока");
         }
      }
      System.out.println("Выполнение программы завершено");
   }
}
```

Явная реализация интерфейса Runnable

Детали

До выполнения `catch`-блока дело обычно не доходит. Также следует учесть, что в принципе обработку исключения в методе `main()` можно было не выполнять, но тогда в описании метода `main()` пришлось бы указывать инструкцию `throws` с указанием класса `InterruptedException` исключения, которое может быть выброшено методом. Что касается метода `run()`, то в интерфейсе `Runnable` он объявлен как такой, что не выбрасывает исключение класса `InterruptedException`, поэтому и описан он должен быть соответствующим образом.

☑ На заметку

Метод `join()` может выбрасывать неконтролируемое исключение класса `InterruptedException`.

Явная реализация интерфейса Runnable

Результат

```
Начинается выполнение программы
Запускается дочерний поток
Главный поток:   A
Дочерний поток:  1
Дочерний поток:  2
Главный поток:   B
Дочерний поток:  3
Дочерний поток:  4
Главный поток:   C
Дочерний поток:  5
Главный поток:   D
Главный поток:   E
Главный поток:   F
Выполнение программы завершено
```

Создание потока с использованием анонимного класса

Создание потока с использованием анонимного класса

```
class NewThreadDemo{
   public static void main(String[] args) throws InterruptedException{
      System.out.println("Начинается выполнение программы");
      // Аргументом конструктору передается анонимный объект анонимного класса:
      Thread t=new Thread(new Runnable(){
         public void run(){
            for(int i=1;i<=5;i++){
               System.out.println("Дочерний поток:\t"+i);
               try{
                  Thread.sleep(4500);
               }
               catch(InterruptedException e){
                  System.out.println("Прерывание дочернего потока");
      }}}
      });
      System.out.println("Запускается дочерний поток");
      t.start();
      for(int k=0;k<=5;k++){
         System.out.println("Главный поток:\t"+(char)('A'+k));
         Thread.sleep(2000);
      } // Проверка дочернего потока:
      if(t.isAlive()){
         t.join(); // Ожидание завершения дочернего потока
      }
      System.out.println("Выполнение программы завершено");
}}
```

Создание потока с использованием анонимного класса

Детали

Поскольку метод `join()` может выбрасывать контролируемое исключение класса `InterruptedException`, то инструкцию вызова метода `join()` следует размещать либо в `try`-блоке с предусмотренной возможностью перехвата и обработки исключения, либо метод, в котором вызывается метод `join()`, в описании должен содержать `throws`-инструкцию с названием класса `InterruptedException`. Используя такую инструкцию в описании метода `main()` мы решаем сразу две проблемы: можно не использовать обработку ошибки класса `InterruptedException` ни при вызове метода `sleep()`, ни при вызове метода `join()`.

Создание потока с использованием анонимного класса

Результат

```
Начинается выполнение программы
Запускается дочерний поток
Главный поток:   A
Дочерний поток:  1
Главный поток:   B
Главный поток:   C
Дочерний поток:  2
Главный поток:   D
Главный поток:   E
Дочерний поток:  3
Главный поток:   F
Дочерний поток:  4
Дочерний поток:  5
Выполнение программы завершено
```

Создание потока с использованием лямбда-выражения

Создание потока с использованием лямбда-выражения

```
class LambdaInThreadDemo{
   public static void main(String[] args) throws InterruptedException{
      System.out.println("Начинается выполнение программы");
      // Интерфейсной переменной значением присваивается лямбда-выражение:
      Runnable r=()->{
         for(int i=1;i<=5;i++){
            System.out.println("Дочерний поток:\t"+i);
            try{
               Thread.sleep(4500);
            }
            catch(InterruptedException e){
               System.out.println("Прерывание дочернего потока");
            }
         }
      }; // Интерфейсная переменная - аргумент конструктора класса Thread:
      Thread t=new Thread(r);
      System.out.println("Запускается дочерний поток");
      t.start();
      for(int k=0;k<=5;k++){
         System.out.println("Главный поток:\t"+(char)('A'+k));
         Thread.sleep(2000);
      }
      if(t.isAlive()){
         t.join();
      }
      System.out.println("Выполнение программы завершено"); }}
```

Создание потока с использованием лямбда-выражения

☑ **На заметку**

Мы воспользовались тем обстоятельством, что интерфейс `Runnable` является функциональным. В интерфейсе `Runnable` имеется единственный абстрактный метод `run()`. Поэтому интерфейсной переменной типа `Runnable` можно присвоить значением лямбда-выражение, определяющее код метода, не имеющего аргументов и не возвращающего результат.

Результат

```
Начинается выполнение программы
Запускается дочерний поток
Главный поток:   A
Дочерний поток:  1
Главный поток:   B
Главный поток:   C
Дочерний поток:  2
Главный поток:   D
Главный поток:   E
Дочерний поток:  3
Главный поток:   F
Дочерний поток:  4
Дочерний поток:  5
Выполнение программы завершено
```

Наследование класса Thread

Наследование класса Thread

```
class MyThread extends Thread{        // Подкласс класса Thread
   public void run(){                 // Переопределение метода run()
      for(int i=1;i<=5;i++){
         System.out.println("Дочерний поток:\t"+i);
         try{
            Thread.sleep(4500);
         }
         catch(InterruptedException e){
            System.out.println("Прерывание дочернего потока");
}}}} // Главный класс:
class ExtendingThreadDemo{
   public static void main(String[] args) throws InterruptedException{
      System.out.println("Начинается выполнение программы");
      MyThread t=new MyThread();     // Создание объекта класса MyThread
      System.out.println("Запускается дочерний поток");
      t.start();                     // Запуск дочернего потока
      for(int k=0;k<=5;k++){
         System.out.println("Главный поток:\t"+(char)('A'+k));
         Thread.sleep(2000);
      }
      if(t.isAlive()){
         t.join();
      }
      System.out.println("Выполнение программы завершено");
   }
}
```

Результат - как и в предыдущем примере

Главный поток

Главный поток

```
class MainThreadDemo{
   public static void main(String[] args){
      // Объектная переменная для записи ссылки на поток:
      Thread t;
      // Получение ссылки на объект главного потока:
      t=Thread.currentThread();
      // Отображение информации о потоке:
      System.out.println(t);
      // Изменение имени потока:
      t.setName("Главный поток");
      // Задается приоритетность потока:
      t.setPriority(7);
      // Отображение информации о потоке:
      System.out.println(t);
   }
}
```

Результат

```
Thread[main,5,main]
Thread[Главный поток,7,main]
```

Создание нескольких потоков

Создание нескольких потоков - 1

```
import java.util.Random;   // Импорт класса Random
// Подкласс MyThread создается наследованием суперкласса Thread:
class MyThread extends Thread{
   private int count;                  // Количество сообщений
   MyThread(String name,int count){ // Конструктор
      super(name);                     // Вызов конструктора суперкласса
      this.count=count;                // Значение целочисленного поля
      start();                         // Запуск потока на выполнение
   } // Переопределение метода run():
   public void run(){ // Отображение сообщения о запуске потока:
      System.out.println("Выполняется поток "+getName());
      // Создание объекта класса Random для генерирования случайных чисел:
      Random rnd=new Random();
      // Оператор цикла, в котором выводятся сообщения:
      for(int k=1;k<=count;k++){
         System.out.println("Сообщение от потока
"+getName()+":\t"+getName().charAt(0)+k);
         try{ // Задержка в выполнении потока:
            Thread.sleep(1000+rnd.nextInt(2001));
         }
         catch(InterruptedException e){
            System.out.println("Прерывание потока "+getName());
         }
      } // Сообщение о завершении выполнения потока:
      System.out.println("Поток "+getName()+" завершен");
}} // Продолжение на следующем слайде!!!
```

Создание нескольких потоков

Создание нескольких потоков - 2

```
// Главный класс:
class MultiThreadDemo{
   public static void main(String[] args) throws InterruptedException{
      System.out.println("Начинает выполняться главный поток");
      // Создание объектов - запуск потоков:
      MyThread A=new MyThread("ALPHA",5);
      MyThread B=new MyThread("BRAVO",3);
      MyThread C=new MyThread("CHARLIE",7);
      // Сообщения главного потока:
      for(int k=1;k<=4;k++){
         System.out.println("Сообщение от главного потока:\t"+k);
         // Задержка в выполнении потока:
         Thread.sleep(2000);
      } // Ожидание завершения дочерних потоков:
      if(A.isAlive()){
         A.join();
      }
      if(B.isAlive()){
         B.join();
      }
      if(C.isAlive()){
         C.join();
      } // Сообщение о завершении главного потока:
      System.out.println("Главный поток завершен");
   }
}
```

Создание нескольких потоков

Результат

```
Начинает выполняться главный поток
Выполняется поток ALPHA
Сообщение от главного потока:  1
Выполняется поток BRAVO
Выполняется поток CHARLIE
Сообщение от потока BRAVO:     B1
Сообщение от потока CHARLIE:   C1
Сообщение от потока ALPHA:     A1
Сообщение от потока BRAVO:     B2
Сообщение от потока ALPHA:     A2
Сообщение от главного потока:  2
Сообщение от потока ALPHA:     A3
Сообщение от потока CHARLIE:   C2
Сообщение от главного потока:  3
Сообщение от потока BRAVO:     B3
Поток BRAVO завершен
Сообщение от потока CHARLIE:   C3
Сообщение от потока ALPHA:     A4
Сообщение от главного потока:  4
Сообщение от потока ALPHA:     A5
Сообщение от потока CHARLIE:   C4
Поток ALPHA завершен
Сообщение от потока CHARLIE:   C5
Сообщение от потока CHARLIE:   C6
Сообщение от потока CHARLIE:   C7
Поток CHARLIE завершен
Главный поток завершен
```

На заметку

Детали

Конструктор класса `Thread` имеет несколько версий. В частности, можно вызывать конструктор класса `Thread`:

- без аргументов - создается объект потока с параметрами по умолчанию;
- с текстовым аргументом - такой аргумент задает название потока;
- с аргументом, являющимся объектом класса, реализующим интерфейс `Runnable` - такой аргумент содержит метод `run()`, код которого выполняется в потоке;
- можно передать первым аргументом объект с методом `run()`, а вторым - текстовое значение, определяющее название потока.

Существуют и другие версии конструктора класса `Thread`.

Создание демон-потока

Есть специальный тип потоков, которые называются демон-потоками

Обычно такие потоки выполняются в фоновом режиме и играют некоторую "вспомогательную" роль по "обслуживанию" других потоков

Главная особенность демон-потока состоит в том, что при завершении главного потока демон-поток завершается автоматически

Программа: создание демон-потока

Мы рассмотрим задачу о вычислении суммы натуральных чисел

Программу организуем с привлечением дочернего демон-потока: при запуске программы запускается демон-поток, в котором собственно происходит вычисление суммы натуральных чисел

Каждое новое значение для суммы отображается в консольном окне. Одновременно из главного потока отображается диалоговое окно, в котором пользователь должен нажать одну из двух кнопок (Yes или No)

Если пользователь нажимает кнопку No, то окно закрывается, но процесс вычисления суммы продолжается, и окно с двумя кнопками появится снова. Все это продолжается до тех пор, пока пользователь не нажмет в окне кнопку Yes

Программа: создание демон-потока

Создание демон-потока - 1

```
// Статический импорт:
import static javax.swing.JOptionPane.*;
class DaemonThreadDemo{
   public static void main(String[] args) throws InterruptedException{
      // Создание объекта для дочернего потока. Первый
      // аргумент конструктора является лямбда-выражением,
      // второй аргумент - название потока:
      Thread t=new Thread(()->{
         // Индексная переменная и
         // переменная для записи суммы чисел:
         int k=1,s=0;
         // Бесконечный цикл для вычисления суммы:
         while(true){
            // Отображение сообщения:
            System.out.println(Thread.currentThread().getName()+": "+s);
            try{
               // Задержка в выполнении потока:
               Thread.sleep(1000);
            } // Обработка исключения:
            catch(InterruptedException e){}
            // Прибавление слагаемого к сумме:
            s+=k;
            // Увеличение значения индексной переменной:
            k++;
         }
      },"Вычисление суммы"); // Продолжение на следующем слайде!!!
```

Программа: создание демон-потока

Создание демон-потока - 2

```
        // Статус демон-потока:
        t.setDaemon(true);
        // Переменная для записи результат
        // выбора пользователя (нажатая кнопка):
        int res;
        // Запуск потока на выполнение:
        t.start();
        // Отображение диалогового окна:
        do{
            // Задержка в выполнении потока:
            Thread.sleep(3000);
            // Отображение окна и запоминание
            // выбора пользователя:
            res=showConfirmDialog(null,
                // Текст в окне:
                "Закончить вычисление суммы?",
                // Название окна:
                "Сумма чисел",
                // Кнопки в окне:
                YES_NO_OPTION);
        }while(res!=YES_OPTION);
    }
}
```

Программа: создание демон-потока

☑ **На заметку**

Могут использоваться такие константы для определения кнопок, отображаемых в диалоговом окне: `DEFAULT_OPTION` (по умолчанию отображается кнопка **OK**), `YES_NO_OPTION` (отображаются кнопки **Yes** и **No**), `YES_NO_CANCEL_OPTION` (отображаются кнопки **Yes**, **No** и **Cancel**) и `OK_CANCEL_OPTION` (отображаются кнопки **OK** и **Cancel**).

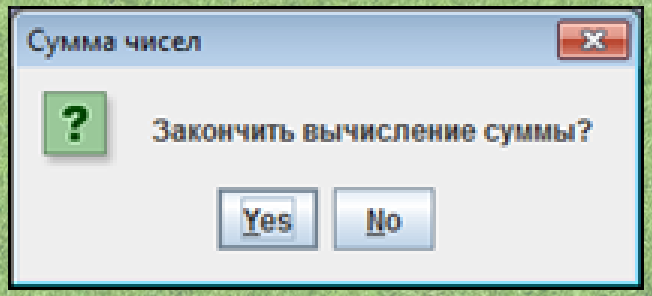
☑ **На заметку**

Константы, определяющие результат метода `showConfirmDialog()`, такие: `YES_OPTION` (нажата кнопка **Yes**), `NO_OPTION` (нажата кнопка **No**), `CANCEL_OPTION` (нажата кнопка **Cancel**), `OK_OPTION` (нажата кнопка **OK**) и `CLOSED_OPTION` (окно закрыто щелчком на системной пиктограмме).

Программа: создание демон-потока

Создание демон-потока - результат

```
Вычисление суммы: 0
Вычисление суммы: 1
Вычисление суммы: 3
Вычисление суммы: 6
Вычисление суммы: 10
Вычисление суммы: 15
Вычисление суммы: 21
Вычисление суммы: 28
```



Синхронизация потоков

Нередко случается так, что несколько потоков в процессе выполнения обращаются к общему ресурсу. Здесь может возникнуть проблема, связанная с тем, что несколько потоков пытаются одновременно выполнить некоторые операции с данным общим ресурсом

Решение проблемы может быть связано с синхронизацией потоков при обращении к ресурсу

Идея синхронизации простая: общий ресурс при обращении к нему одного потока блокируется для всех других потоков

Если речь идет о синхронизации потоков при обращении к некоторому объекту, то используется такой шаблон:

```
synchronized(объект){
    // код синхронизации
}
```

Если нужно синхронизировать метод (такой метод будет доступен в каждый момент только для одного потока), то в сигнатуре метода указывается ключевое слово `synchronized`

Синхронизация потоков

Синхронизация потоков - 1

```
class MyNumber{                          // Класс с целочисленным полем
   int number;
} // Класс для создания потоков:
class MyThread extends Thread{
   private MyNumber obj;                 // Ссылка на объект с целочисленным полем
   private int time;                     // Интервал приостановки в выполнении потока
   private int count;                    // Количество циклов
   private boolean state;                // Поле логического типа
   MyThread(String name,MyNumber obj,int time,int count,boolean state){
      super(name);                       // Вызов конструктора суперкласса
      this.obj=obj;                      // Ссылка на объект
      this.time=time;                    // Интервал задержки
      this.count=count;                  // Количество циклов
      this.state=state;                  // Значение поля логического типа
      start();                           // Запуск потока
   } // Метод для отображения "линии" из символов:
   private void line(){
      char s;                            // Локальная символьная переменная
      if(state) s='-';                   // Значение переменной
      else s='*';
      for(int k=1;k<=35;k++){            // Отображение последовательности символов
         System.out.print(s);
      }
      System.out.println("");
   }
   // Продолжение на следующем слайде!!!
```

Синхронизация потоков

Синхронизация потоков - 2

```
   // Переопределение метода run():
   public void run(){
      for(int k=1;k<=count;k++){      // Оператор цикла
         synchronized(obj){           // Блок синхронизации
            line();                   // Отображается "линия"
            // Отображение сообщения:
            System.out.println("Поток "+getName()+": исходное значение
"+obj.number);
            try{                      // Контролируемый код
               Thread.sleep(time);    // Задержка в выполнении потока
            } // Обработка исключения:
            catch(InterruptedException e){
               System.out.println(e);
            } // Изменение значения поля объекта:
            if(state) obj.number++;
            else obj.number--;
            // Отображение сообщения:
            System.out.println("Поток "+getName()+": новое значение
"+obj.number);
            line();                   // Отображение "линии"
         } // Завершение блока синхронизации
      }
   }
}
// Продолжение на следующем слайде!!!
```

Синхронизация потоков

Синхронизация потоков - 3

```
// Главный класс:
class SynchronizedThreadsDemo{
   // Главный метод:
   public static void main(String[] args){
      // Целочисленные переменные:
      int n=100,count=5,time=1000,dt=200;
      // Создание объекта с целочисленным полем:
      MyNumber obj=new MyNumber();
      // Присваивание полю объекта значения:
      obj.number=n;
      // Создание первого потока:
      MyThread Alpha=new MyThread("Alpha",obj,time+dt,count,true);
      // Создание второго потока:
      MyThread Bravo=new MyThread("Bravo",obj,time-dt,count,false);
      // Контролируемый код:
      try{
         // Ожидание завершения потоков:
         if(Alpha.isAlive()) Alpha.join();
         if(Bravo.isAlive()) Bravo.join();
      } // Обработка исключения:
      catch(InterruptedException e){
         System.out.println(e);
      }
   }
}
```

Результат

```
-----------------------------------
Поток Alpha: исходное значение 100
Поток Alpha: новое значение 101
-----------------------------------
***********************************
Поток Bravo: исходное значение 101
Поток Bravo: новое значение 100
***********************************
***********************************
Поток Bravo: исходное значение 100
Поток Bravo: новое значение 99
***********************************
***********************************
Поток Bravo: исходное значение 99
Поток Bravo: новое значение 98
***********************************
***********************************
Поток Bravo: исходное значение 98
Поток Bravo: новое значение 97
***********************************
-----------------------------------
Поток Alpha: исходное значение 97
Поток Alpha: новое значение 98
-----------------------------------
-----------------------------------
Поток Alpha: исходное значение 98
Поток Alpha: новое значение 99
-----------------------------------
-----------------------------------
Поток Alpha: исходное значение 99
Поток Alpha: новое значение 100
-----------------------------------
-----------------------------------
Поток Alpha: исходное значение 100
Поток Alpha: новое значение 101
-----------------------------------
***********************************
Поток Bravo: исходное значение 101
Поток Bravo: новое значение 100
***********************************
```

Домашнее задание

- Проанализировать программный код примеров, представленных в презентации
- Набрать код и проверить его функциональность

Продолжение следует…

Методы для работы с потоками - 1

Метод	Описание
activeCount()	Методом возвращается количество активных потоков в группе
checkAccess()	Метод позволяет выполнить проверку на предмет того, может ли указанный поток быть изменен выполняемым потоком
currentThread()	Метод возвращает результатом ссылку на объект текущего потока (потока, в котором вызывается метод)
enumerate()	Метод предназначен для копирования в массив ссылок на все активные потоки в данной группе потоков (и ее подгруппах)
getId()	Методом в качестве результата возвращается идентификатор потока - целое число, идентифицирующее поток
getName()	Метод в качестве результата возвращает имя потока
getPriority()	Метод возвращает результатом приоритет потока (целое число в диапазоне значений от 1 до 10)
getState()	Метод возвращает объект, определяющий статус потока
getThreadGroup()	Результатом метода является объект, определяющий группу, к которой принадлежит поток

Методы для работы с потоками - 2

Метод	Описание
holdsLock()	Метод в качестве результата возвращается логическое значение, определяющее, удерживает ли метод монитор (специальный объект, используемый для блокировки некоторого ресурса с целью недопущения обращения к нему сразу нескольких методов)
interrupt()	Метод предназначен для выполнения прерывания потока
interrupted()	Метод используется для проверки того, был ли поток прерван. При этом меняется статус потока (в отношении прерывания)
isAlive()	Метод для определения того, активен ли поток
isDaemon()	Проверка потока на предмет того, является ли он демон-потоком (такие потоки обычно выполняются в фоновом режиме и автоматически завершаются при завершении главного потока)
isInterrupted()	Метод используется для проверки того, был ли поток прерван. При этом статус потока (в отношении прерывания) не меняется
join()	Вызов метода приводит к тому, что поток, в котором вызывается метод, ожидает завершения потока, из объекта которого вызывается метод

Методы для работы с потоками - 3

Метод	Описание
notify()	При вызове метода один из потоков, находящихся в режиме ожидания (для получения доступа к ресурсу), переводится в режим выполнения
notifyAll()	При вызове метода все потоки, находящиеся в режиме ожидания (для получения доступа к ресурсу), переводятся в режим выполнения
run()	Метод с кодом, выполняемым в рамках потока. Определяет точку входа в поток
setDaemon()	Метод позволяет определить поток как демон-поток (особый вид потоков, которые автоматически завершаются при завершении выполнения главного потока)
setName()	Метод позволяет задать название для потока
setPriority()	Метод предназначен для присваивания значения приоритету потока

Методы для работы с потоками - 4

Метод	Описание
sleep()	Метод позволяет приостановить выполнение потока. Аргументом методу передается целое число, определяющее время (в миллисекундах) приостановки в выполнении потока
start()	Метод вызывается при запуске потока на выполнение
wait()	Метод предназначен для перевода потока в режим ожидания (при получении доступа к ресурсу)
yield()	Методом посылается сигнал диспетчеру потоков, что текущий поток готов уступить использование процессора в пользу других потоков