# 10-11. Thread'ы и средства синхронизации thread'ов

## Класс Thread

В Java функциональность отдельного потока заключается в классе Thread. И чтобы создать новый поток, нам надо создать объект этого класса. Но все потоки не создаются сами по себе. Когда запускается программа, начинает работать главный поток этой программы. От этого главного потока порождаются все остальные дочерние потоки.

С помощью статического метода `Thread.currentThread()` мы можем получить текущий поток выполнения:

```
public static void main(String[] args) {
    Thread t = Thread.currentThread(); // получаем главный поток
    System.out.println(t.getName()); // main
}
```

По умолчанию именем главного потока будет *main*.

Для управления потоком класс `Thread` предоставляет еще ряд методов. Наиболее используемые из них:

```
* `getName()`: возвращает имя потока
* `setName(String name)`: устанавливает имя потока
* `getPriority()`: возвращает приоритет потока
* `setPriority(int proirity)`: устанавливает приоритет потока. Приоритет
является одним из ключевых факторов для выбора системой потока из кучи
потоков для выполнения. В этот метод в качестве параметра передается числовое
значение приоритета - от 1 до 10. По умолчанию главному потоку выставляется
средний приоритет - 5.
* `isAlive()`: возвращает true, если поток активен
* `isInterrupted()`: возвращает true, если поток был прерван
* `join()`: ожидает завершение потока
* `run()`: определяет точку входа в поток
* `sleep()`: приостанавливает поток на заданное количество миллисекунд
* `start()`: запускает поток, вызывая его метод `run()`
```

Мы можем вывести всю информацию о потоке:

```
public static void main(String[] args) {
    Thread t = Thread.currentThread(); // получаем главный поток
    System.out.println(t); // Thread[main,5,main]
}
```

Первое *main* будет представлять имя потока (что можно получить через `t.getName()`), второе значение 5 предоставляет приоритет потока (также можно получить через `t.getPriority()`), и последнее main представляет имя группы потоков, к которому относится текущий - по умолчанию также main (также можно получить через `t.getThreadGroup().getName()`).

## Недостатки при использовании потоков

Далее мы рассмотрим, как создавать и использовать потоки. Это довольно легко. Однако при создании многопоточного приложения нам следует учитывать ряд обстоятельств, которые негативно могут сказаться на работе приложения.

На некоторых платформах запуск новых потоков может замедлить работу приложения. Что может иметь большое значение, если нам критичная производительность приложения.

Для каждого потока создается свой собственный стек в памяти, куда помещаются все локальные переменные и ряд других данных, связанных с выполнением потока. Соответственно, чем больше потоков создается, тем больше памяти используется. При этом надо помнить, в любой системе размеры используемой памяти ограничены. Кроме того, во многих системах может быть ограничение на количество потоков. Но даже если такого ограничения нет, то в любом случае имеется естественное ограничение в виде максимальной скорости процессора.

# Создание и завершение потоков

Для создания нового потока мы можем создать новый класс, либо наследуя его от класса `Thread`, либо реализуя в классе интерфейс `Runnable`.

Создадим свой класс на основе `Thread`:

```
public class JThread extends Thread {

    JThread(String name){
        super(name);
    }

    public void run(){
        System.out.printf("Поток %s начал работу... \n",
Thread.currentThread().getName());

        try{
            Thread.sleep(500);
        }
        catch(InterruptedException e){
            System.out.println("Поток прерван");
        }

        System.out.printf("Поток %s завершил работу... \n",
Thread.currentThread().getName());
    }
}
```

Класс потока называется `JThread`. Предполагается, что в конструктор класса передается имя потока, которое затем передается в конструктор базового класса. И также здесь переопределяется метод `run()`, код которого собственно и будет представлять весь тот код, который выполняется в потоке.

Теперь применим этот класс в главном классе программы:

```
public static void main(String[] args) {
    System.out.println("Главный поток начал работу...");
    new JThread("JThread").start();
    System.out.println("Главный поток завершил работу...");
}
```

Вывод в консоль:

```
Главный поток начал работу...
Главный поток завершил работу...
Поток JThread начал работу...
Поток JThread завершил работу...
```

Здесь в методе `main` в конструктор `JThread` передается произвольное название потока, и затем вызывается метод `start()`. По сути этот метод как раз и вызывает переопределенный метод `run()` класса `JThread`.

Обратите внимание, что главный поток завершает работу раньше, чем порожденный им дочерний поток `JThread`.

Аналогично созданию одного потока мы можем запускать сразу несколько потоков:

```
public static void main(String[] args) {
    System.out.println("Главный поток начал работу...");

    for(int i = 1; i < 6; i++)
        new JThread("JThread " + i).start();

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

Вывод в консоль:

```
Главный поток начал работу...
Главный поток завершил работу...
Поток JThread 2 начал работу...
Поток JThread 5 начал работу...
Поток JThread 4 начал работу...
Поток JThread 1 начал работу...
Поток JThread 3 начал работу...
Поток JThread 1 завершил работу...
Поток JThread 2 завершил работу...
Поток JThread 5 завершил работу...
Поток JThread 4 завершил работу...
Поток JThread 3 завершил работу...
```

При запуске потоков в примерах выше главный поток завершался до дочернего потока. Как правило, более распространенной ситуацией является случай, когда главный поток завершается самым последним. Для этого надо применить метод `join()`:

```
public static void main(String[] args) {
    System.out.println("Главный поток начал работу...");

    JThread t = new JThread("JThread ");
    t.start();

    try {
        t.join();
    }
    catch(InterruptedException e) {
        System.out.printf("Поток %s прерван", t.getName());
    }

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

Метод `join()` заставляет вызвавший поток (в данном случае главный поток) ожидать завершения вызываемого потока, для которого и применяется метод `join()` (в данном случае поток `JThread`).

Вывод в консоль:

```
Главный поток начал работу...
Поток JThread  начал работу...
Поток JThread  завершил работу...
Главный поток завершил работу...
```

Если в программе используется несколько дочерних потоков, и надо, чтобы главный поток завершался после дочерних, то для каждого дочернего потока надо вызвать метод `join()`.

# Реализация интерфейса Runnable

Другой способ определения потока представляет реализация интерфейса `Runnable`. Этот интерфейс имеет один метод `run()`:

```
interface Runnable {
    void run();
}
```

В методе `run()` собственно определяется весь тот код, который выполняется при запуске потока.

После определения объекта `Runnable` он передается в один из конструкторов класса `Thread`:

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

Для реализации интерфейса определим следующий класс `MyThread`:

```
public class MyThread implements Runnable {
    MyThread() {}

    public void run() {
        System.out.printf("Поток %s начал работу... \n",
Thread.currentThread().getName());

        try {
            Thread.sleep(500);
        }
        catch(InterruptedException e){
            System.out.println("Поток прерван");
        }

        System.out.printf("Поток %s завершил работу... \n",
Thread.currentThread().getName());
    }
}
```

Реализация интерфейса `Runnable` во многом аналогична переопределению класса `Thread`. Также в методе `run()` определяется простейший код, который усыпляет поток на 500 миллисекунд.

Теперь используем этот класс в главном классе программы:

```
public static void main(String[] args) {
    System.out.println("Главный поток начал работу...");

    new Thread(new MyThread(), "MyThread").start();

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

В методе `main` вызывается конструктор `Thread`, в который передается объект `MyThread`. И чтобы запустить поток, вызывается метод `start()`. В итоге консоль выведет что-то наподобие следующего:

```
Главный поток начал работу...
Главный поток завершил работу...
Поток MyThread начал работу...
Поток MyThread завершил работу...
```

## Завершение потока

Особо следует остановиться на механизме завершения потока. Все примеры выше представляли поток как последовательный набор операций. После выполнения последней операции завершался и поток. Однако нередко имеет место и другая организация потока в виде бесконечного цикла. Например, поток сервера в бесконечном цикле прослушивает определенный порт на предмет получения данных. И в этом случае мы также должны предусмотреть механизм завершения потока. Как правило, это делается с помощью опроса логической переменной. И если она равна, например, `false`, то поток завершает бесконечный цикл и заканчивает свое выполнение.

Определим следующий класс потока:

```
public class MyThread implements Runnable {
    private boolean isActive;

    void disable() {
        isActive = false;
    }

    MyThread() {
        isActive = true;
    }

    public void run() {
        System.out.printf("Поток %s начал работу... \n",
Thread.currentThread().getName());

        int counter = 1; // счетчик циклов
        while(isActive) {
            System.out.println("Цикл " + counter++);

            try {
                Thread.sleep(500);
            }
            catch(InterruptedException e) {
                System.out.println("Поток прерван");
            }
        }

    System.out.printf("Поток %s завершил работу... \n",
Thread.currentThread().getName());
    }
}
```

Переменная `isActive` указывает на активность потока. С помощью метода `disable()` мы можем сбросить состояние этой переменной.

Теперь используем этот класс:

```
public static void main(String[] args) {
    System.out.println("Главный поток начал работу...");

    MyThread myThread = new MyThread();
    new Thread(myThread, "MyThread").start();

    try {
        Thread.sleep(1100);

        myThread.disable();

        Thread.sleep(1000);
    }
    catch(InterruptedException e) {
        System.out.println("Поток прерван");
    }

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

Итак, вначале запускается дочерний поток: new `Thread(myThread,"MyThread").start()`. Затем на 1100 миллисекунд останавливаем главный поток и потом вызываем метод `myThread.disable()`, который переключает в потоке флаг `isActive`. И дочерний поток завершается.

## Синхронизация потоков. Оператор synchronized

При работе потоки нередко обращаются к каким-то общим ресурсам, которые определены вне потока, например, обращение к какому-то файлу. Если одновременно несколько потоков обратятся к общему ресурсу, то результаты выполнения программы могут быть неожиданными и даже непредсказуемыми. Например, определим следующий код:

```
public class ThreadsApp {
    public static void main(String[] args) {
        CommonResource commonResource = new CommonResource();

        for (int i = 1; i < 6; i++) {
            Thread t = new Thread(new CountThread(commonResource));
            t.setName("Поток " + i);
            t.start();
        }
    }
}

class CommonResource {
    int x = 0;
}

class CountThread implements Runnable {
    CommonResource res;
    CountThread(CommonResource res) {
        this.res = res;
    }

    public void run() {
        res.x = 1;

        for (int i = 1; i < 5; i++) {
            System.out.printf("%s %d \n", Thread.currentThread().getName(),
res.x);

            res.x++;
            try {
                Thread.sleep(100);
            }
            catch(InterruptedException e){}
        }
    }
}
```

Здесь определен класс `CommonResource`, который представляет общий ресурс и в котором определено одно целочисленное поле `x`.

Этот ресурс используется классом потока `CountThread`. Этот класс просто увеличивает в цикле значение x на единицу. Причем при входе в поток значение `x = 1`:

```
res.x = 1;
```

То есть в итоге мы ожидаем, что после выполнения цикла `res.x` будет равно 4.

В главном классе программы запускается пять потоков. То есть мы ожидаем, что каждый поток будет увеличивать `res.x` с 1 до 4 и так пять раз. Но если мы посмотрим на результат работы программы, то он будет иным:

```
Поток 1 1
Поток 2 1
Поток 3 1
Поток 5 1
Поток 4 1
Поток 5 6
Поток 2 6
Поток 1 6
Поток 3 6
Поток 4 6
Поток 4 11
Поток 2 11
Поток 5 11
Поток 3 11
Поток 1 11
Поток 4 16
Поток 1 16
Поток 3 16
Поток 5 16
Поток 2 16
```

То есть пока один поток не окончил работу с полем `res.x`, с ним начинает работать другой поток.

Чтобы избежать подобной ситуации, надо синхронизировать потоки. Одним из способов синхронизации является использование ключевого слова `synchronized`. Этот оператор предваряет блок кода или метод, который подлежит синхронизации. Для его применения изменим класс `CountThread`:

```
class CountThread implements Runnable {
    CommonResource res;
    CountThread(CommonResource res) {
        this.res = res;
    }

    public void run() {
        synchronized(res) {
            res.x = 1;

            for (int i = 1; i < 5; i++) {
                System.out.printf("%s %d \n",
Thread.currentThread().getName(), res.x);
                res.x++;

                try {
                    Thread.sleep(100);
                }
                catch(InterruptedException e){}
            }
        }
    }
}
```

При создании синхронизированного блока кода после оператора `synchronized` идет объект-заглушка: `synchronized(res)`. Причем в качестве объекта может использоваться только объект какого-нибудь класса, но не примитивного типа.

Каждый объект в Java имеет ассоциированный с ним **монитор**. **Монитор** представляет своего рода инструмент для управления доступа к объекту. Когда выполнение кода доходит до оператора `synchronized`, **монитор** объекта `res` блокируется, и на время его блокировки монопольный доступ к блоку кода имеет только один поток, который и произвел блокировку. После окончания работы блока кода, **монитор** объекта `res` освобождается и становится доступным для других потоков.

После освобождения **монитора** его захватывает другой поток, а все остальные потоки продолжают ожидать его освобождения.

При применении оператора `synchronized` к методу пока этот метод не завершит выполнение, монопольный доступ имеет только один поток - первый, который начал его выполнение. Для применения `synchronized` к методу, изменим классы программы:

```
public class ThreadsApp {
    public static void main(String[] args) {
        CommonResource commonResource = new CommonResource();

        for (int i = 1; i < 6; i++){
            Thread t = new Thread(new CountThread(commonResource));

            t.setName("Поток "+ i);
            t.start();
        }
    }
}

class CommonResource{
    int x;

    synchronized void increment() {
        x = 1;
        for (int i = 1; i < 5; i++) {
            System.out.printf("%s %d \n", Thread.currentThread().getName(),
x);

            x++;
            try {
                Thread.sleep(100);
            }
            catch(InterruptedException e){}
        }
    }
}

class CountThread implements Runnable {
    CommonResource res;
    CountThread(CommonResource res) {
        this.res = res;
    }

    public void run() {
        res.increment();
    }
}
```

Результат работы в данном случае будет аналогичен примеру выше с блоком `synchronized`. Здесь опять в дело вступает **монитор** объекта `CommonResource` - общего объекта для всех потоков. Поэтому синхронизированным объявляется не метод `run()` в классе `CountThread`, а метод `increment` класса `CommonResource`. Когда первый поток начинает выполнение метода `increment`, он захватывает **монитор** объекта `CommonResource`. А все потоки также продолжают ожидать его освобождения.

## Методы wait и notify

## Volatile