Многопоточное программирование

Язык Java имеет встроенную поддержку многопоточного программирования.

Многопоточная программа содержит две или более частей, которые могут выполняться одновременно. Каждая часть такой программы называется потоком (**thread**), и каждый поток задает отдельный путь выполнения.

Многопоточность - это одна из форм многозадачности.

Процесс - выполняющаяся программа.
«*Многозадачность на процессах*» - средство, которое позволяет компьютеру одновременно выполнять две или более программ (программа представляет собой наименьший элемент кода, которым может управлять ОС).

В среде «*поточной многозадачности*» наименьшим элементом управляемого кода является поток. Одна программа может выполнять две или более задач одновременно.

В однопоточной системе используются подход «цикл событий с опросом», когда единственный поток управления выполняется в бесконечном цикле, опрашивая единственную очередь событий для принятия решения о том, что делать дальше.

Поэтому в однопоточной программе, когда поток приостанавливает выполнение по причине ожидания ресурса, выполнение всей программы приостанавливается.

В многопоточной системе механизм циклического опроса исключается. Один поток может быть приостановлен без остановки других частей (потоков) программы.

Многозадачность

Процессы (программы, которые выполняются параллельно, имеют собственное адресное пространство)

Потоки (*threads*) (части программы, которые выполняются параллельно, используют одно адресное пространство)

Состояния потока:

1. выполняется;
2. готов к выполнению;
3. приостановлен;
4. возобновлен;
5. прерван;
6. заблокирован в ожидании ресурса.

Каждому потоку можно назначить **приоритет**.

Приоритет – целое число.

Приоритет используется для принятия решения при переключении управления от одного потока к другому.

Правила передачи управления:

1. Поток может добровольно уступить управление (ресурсы передаются потоку с более высоким приоритетом);
2. Поток может быть прерван более высокоприоритетным потоком. Высокоприоритетный поток выполняется когда «захочет» - это вытесняющая многозадачность.

Если приоритеты потоков равны, то их выполнение зависит от реализации многопоточности в ОС.

Многопоточность обеспечивает асинхронное поведение программ. Если 2 или более потоков используют одну структуру данных, необходимо исключить конфликты между ними с помощью синхронизации.

Например, если требуется, чтобы два потока взаимодействовали и разделяли сложную структуру данных, тогда требуется предотвратить запись данных в одном потоке, когда другой занимается их чтением. Для этой цели в Java реализован монитор.

Монитор - это управляющий механизм. Каждый объект имеет свой собственный неявный монитор, вход в который осуществляется автоматически, когда вызывается синхронизированный метод объекта.

Когда поток находится внутри синхронизированного метода, ни один другой поток не может вызвать ни один синхронизированный метод этого объекта.

Для реализации многопоточности в JAVA используются класс **Thread** или интерфейс **Runnable**.

При запуске java-программы запускается **главный поток**.

Из главного потока порождаются все дочерние потоки. Главный поток должен быть последним потоком, завершающим выполнение программы.

Методы класса Thread:

1. ***static Thread currentThread()*** – возвращает текущий поток;
2. ***void setName(String name)*** – задать имя потока;
3. ***String getName()*** – получить имя потока;
4. ***int getPriority()*** – получить приоритет потока;
5. ***void setPriority()*** – установить приоритет потока;
6. ***void run()*** – указывает точку входа в поток;
7. ***static void sleep(long millsec)*** – приостанавливает поток на *millsec* миллисекунд;
8. ***void start()*** – запускает метод run() потока;
9. ***String to String()*** – возвращает имя, приоритет и группу потока;
10. ***boolean isAlive()*** – определяет, выполняется ли поток;
11. ***void join()*** – ждать завершения потока.

(метод stop() устаревший deprecated, не используется)

Пример работы с главным потоком:

```
class CurrThread{
  public static void main(String args[]){
    Thread t=Thread. currentThread( );          // получение текущего
                                                //потока в t
    System.out.println("Текущий поток: "+ t);      // метод toString()
    t.setName("My Thread");                    // устанавливает новое имя
    System.out.println("Новое имя: "+t);
    try{
      for(int n=5;n>0;n--) {
        System.out.println(n);
        Thread.sleep(1000);  }          // приостанавливаем текущ. поток
                  }
    catch(InterruptedException e){                  // исключение от
                                                   //Thread.sleep(1000)
                  System.out.println("Главный поток прерван");
                                                  }
            }
  }
```

2 способа создания потока

1. Реализовать интерфейс **Runnable:**

Можно создать поток на любом объекте класса, реализующего данный интерфейс. Для этого в классе необходимо переопределить метод **run()**. Внутри **run()** определяется код, который будет выполнятся создаваемым потоком. Поток завершится, когда **run()** вернет управление.

Конструктор:

Thread(Runnable объект_потока**, String** имя_потока**);**

объект_потока – это объект класса, реализующего **Runnable**

Для запуска потока необходимо вызвать метод **start()**, который вызывает **run()** для объекта данного класса.

Пример создания потока с помощью интерфейса **Runnable**

```
class NewThread implements Runnable{
    Thread t;                                   // объявляем потоковую переменную
    NewThread( ){
          t=new Thread(this, “Дочерний поток”);          // создаем поток
          System.out.println(“Дочерний поток создан”:+t);
          t.start();                            // запускаем поток (вызов метода run())
                    }
    public void run(){                  //действия, которые будет выполнять поток,
      try{                              // обычно в ∞цикле с возможностью выхода
         for(int i=5; i>0; i--){                  // или приостановки (sleep(…))
         System.out.println(“Дочерний поток:”+t);
         Thread.sleep(500);  }                   // приостанавливаем, чтобы главный
          }                                       // главный поток получил управление
     catch(InterruptedException e){
          System.out.println(“Дочерний поток прерван”);
                                          }
     System.out.println(“Дочерний поток завершен”);
                              }
  }     //объект такого класса содержит потоковую переменную
```

2. Расширить класс **Thread**:

Создаем класс, который наследуется от класса **Thread**. Создаем объект данного класса , запускаем методом **start()**, который вызывает **run()**. Метод **run()** необходимо переопределить в данном классе.

Пример:

```
class NewThread extends Thread{
  NewThread(){
    super("Дочерний поток"); // вызов констр. Thread(String name)
    System.out.println("Дочерний поток"+this);
    start(); // запуск потока может быть в другом месте
                    }
        public void run(){  //определяет действия для выполнения
                        …      // как и в примере с Runnable
                            }
        …
 } // объект этого класса является потоком
```

Класс для создания и запуска потоков

```
class ThreadDemo{
  public static void main(String args[]){
     new NewTread();              // создание дочернего потока
     try{
       for(int n=5;n>0;n--) {
       System.out.println(n);
       Thread.sleep(1000);  }    //приостанавливаем главный поток,
        }                        //чтобы дочерний мог получить управление
    catch(InterruptedException e){
        System.out.println("Главный поток прерван");
                                              }
    System.out.println("Главный поток завершен");
        }
  }
```

Главный поток завершится последним, т.к. он будет приостанавливаться на большее время, чем дочерний.

Класс Thread определяет несколько методов, которые могут быть переопределены в классах-наследниках. Из этих методов **run( )** должен быть переопределен обязательно.

При реализации интерфейса Runnable достаточно определить только метод **run()**.

Класс Thread следует расширять только в случаях, когда он должен быть усовершенствован или некоторым образом модифицирован. Поэтому если в программе не требуется переопределять другие методы класса Thread, то лучше реализовать интерфейс Runnable для создания многопоточности.

Для создания множества потоков переопределим конструктор
класса NewThread:

```
class NewThread implements Runnable{    // или extends Thread
  Thread t;           // не требуется для классаThread
  String name;

  NewThread(String name){
        this.name = name;
        t=new Thread(this, name);     // или  super(name);  start();
        t.start();
                        }
      }
```

Для создания потоков в требуемом месте программы:

```
new NewThread(“Один”);
new NewThread(“Два”);
new NewThread(“Три”);
```

Методы isAlive() и join()

1. boolean isAlive(): true – если поток, для которого вызван метод, еще выполняется, **false –** иначе

2. void join() throws InterruptedException - ждет завершения потока, для которого вызван.

```
        ……
    NewThread ob1 = new NewThread(“Один”);
        ……
    System.out.println(“Один запущен”+ob1.t.isAlive());
//System.out.println(“Один запущен”+ob1.isAllive()); для подкласса Thread
        ……
     try{
        ob1.t.join();
      //ob1.join(); для подкласса Thread
        }
     catch(InterruptedException e){
        System.out.println(“Прерывание главного потока”);
                                    }
```

Приоритеты потоков

Приоритеты потоков используются планировщиком потоков для принятия решений о том, когда каждому из потоков будет разрешено работать. Теоретически высокоприоритетные потоки получают больше времени процессора, чем низкоприоритетные. Практически - объем времени процессора, который получает поток, часто зависит от того, как ОС реализует многозадачность. Высокоприоритетный поток может также выгружать низкоприоритетный.

Теоретически потоки с равным приоритетом должны получать равный доступ к центральному процессору. В целях безопасности потоки, которые разделяют один и тот же приоритет, должны получать управление в равной степени (для того, чтобы все потоки получили возможность выполняться в среде ОС с не вытесняющей многозадачностью).

Приоритеты потоков

1.int getPriority() – получить приоритет потока;

2.void setPriority() – установить приоритет потока;

public static final int MIN_PRIORITY=1
public static final int NORM_PRIORITY=5
public static final int MAX_PRIORITY=10

По умолчанию всем потокам присваивается **NORM_PRIORITY.**

Thread.currentThread().setPriority(Thread.MAX_PRIORITY);

int PRIOR = Thread.currentThread().getPriority();

или

Thread t =Thread.currentThread().setPriority(Thread.NORM_PRIORITY - 2);

int PRIOR = t.getPriority();

Синхронизация – процесс, с помощью которого обеспечивается то, что ресурс будет использоваться одним потоком в один момент времени. Необходимо использовать, когда два или более потоков имеют доступ к одному ресурсу.

Монитор – объект, который используется для взаимного исключения потоков. Только один поток может владеть монитором в один момент времени. Когда поток входит в монитор на объекте, остальные потоки ждут , пока он не покинет монитор. В JAVA каждый объект имеет свой собственный неявный монитор, вход в который осуществляется автоматически при вызове синхронизированного метода объекта.

Ключевое слово **synchronized**

Чтобы войти в монитор объекта, необходимо вызвать метод с модификатором synchronized. Когда поток находится внутри синхронизированного метода, все другие потоки, которые пытаются его вызвать (или вызвать любые другие синхронизированные методы) на этом же объекте, должны ожидать. Несинхронизированные методы остаются доступными для вызова.

Оператор **synchronized**

synchronized(объект){
// операторы и методы, подлежащие синхронизированию
}

объект – ссылка на синхронизированный объект.

Используется, если класс не содержит синхронизированных методов или нет доступа к коду класса. Блок synchronized гарантирует, что вызов метода объекта произойдет только тогда, когда текущий поток успешно войдет в монитор объекта.

```
class CallMe{
 void call(String msg){
          System.out.println(“[”+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, “Поток”);
                t.start();
                }
public void run(){
        synchronized(target){
                        target.call(msg);
                                }
                }
} // можно вместо синхронизированного блока   обозначить
  // метод call() ключевым словом synchronized
```

```
class Synchr{
public static void main(String args[]){
  CallMe target = new CallMe();
  Caller ob1=new Caller(target,”Добро пожаловать”);
  Caller ob2=new Caller(target,”в синхронизированный”);
  Caller ob3=new Caller(target,”мир”);
  try{
        ob1.t.join();
        ob2.t.join();
        ob3.t.join();
      }
  catch(InterruptedException e){
       System.out.println(“Прерывание главного потока”); }
} }
```

Синхронизация используется

[Добро пожаловать]
[в синхронизированный]
[мир]

Синхронизация не используется

[Добро пожаловать
[в синхронизированный
[мир
]
]
]

Взаимная блокировка

- это ошибка, которая происходит, когда потоки имеют циклическую зависимость от пары синхронизированных объектов.

Пусть два объекта A и B имеют синхронизированные методы.

2 потока:

поток1 входит в монитор на объекте A,
поток2 входит в монитор на объекте B.

Поток1 вызывает синхронизированный метод из B внутри синхронизированного метода A,
поток2 вызывает синхронизированный метод из A внутри синхронизированного метода B.

Поток1 на A будет ждать, пока закончит работу поток2 на B, а поток 2 на B будет ждать, пока закончит работу поток1 на A.

```
class A{
        synchronized void As(B b){
                try{  Thread.sleep(500);  }
                catch(InterruptedException e){
                        System.out.println("Поток на А прерван");  }
                b.last();          }   //вызов синхронизированного метода для В

        synchronized void last(){
                System.out.println("Внутри last() класса А");    }
       }

class B{
        synchronized void Bs(A a){
                try{  Thread.sleep(500);  }
                catch(InterruptedException e){
                        System.out.println("Поток на В прерван");  }
                a.last();          }  //вызов синхронизированного метода для В

        synchronized void last(){
                System.out.println("Внутри last() класса В");    }
       }
```

```
class Lock implements Runnable{
        A a = new A();
        B b = new B();
        Lock(){
                Thread t = new Thread(this,”Новый поток”);
                t.start();
                a.As(b); // вызывет главный поток
                 }
        public void run(){
                b.Bs(a); // вызывает дочерний поток
                                }
        public static void main(String args[]){

                        new Lock();
                                                        }
 }
```

Межпотоковые коммуникации (класс Object)

1. **void wait() throws InterruptedException** – принуждает вызывающий поток отдать монитор и приостановить выполнение до тех пор, пока какой-нибудь другой поток войдет в тот же монитор и вызовет метод notify();

2. void notify() – возобновляет работу потока, который вызвал wait() на том же самом объекте;

3. void notifyAll() возобновляет работу всех потоков, ожидающих на объекте вызова (одному из потоков достается доступ).

Все методы могут быть вызваны только из **synchronized** контекста.

```
class Queue {  // очередь, которую нужно синхронизировать
    int n;

   synchronized int get(){
       System.out.println(“Получено: “+ n);
       return n; }

    synchronized void put (int n) {
      this.n = n;
     System.out.println("Отправлено: "+ n); }
}
```

```
class  Producer implements Runnable { // класс добавляет элементы в очередь
 Queue q;

 Producer (Queue q) {
      this.q = q;
      new Thread (this, “Поставщик”). start() ; }

public void run () {
  int i = 0;
  while (true) {   q.put (i++) ; }   }


class Consumer implements Runnable { // класс удаляет элементы из очереди
 Queue q;

 Consumer (Queue q) {
    this.q = q;
    new Thread(this, “Потребитель”).start(); }

 public void run() {
     while (true) { q.get(); }  }
```

```
class DemoQueue { // демонстрация работы с очередью

  public static void main(String args[]) {

        Queue q = new Queue() ; // очередь

         new Producer(q) ; // поток- поставщик элементов

         new Consumer(q) ; // поток – потребитель элементов
} }
```

Несмотря на то, что методы put() и get() в классе Queue синхронизированы, т.е. не могут быть вызваны одновременно двумя и более потоками, поставщик может поставлять элементы несколько раз подряд, не давая потребителю извлечь элемент из очереди, или потребитель может извлекать один и тот же элемент несколько раз, не давая поставщику возможность добавить элемент в очередь. Можно использовать методы wait() и notify() для управления потоками.

```
class Queue {  // очередь, которую нужно синхронизировать
   int n;      boolean valueSet = false;

  synchronized int get(){
      while(!valueSet)
          try {    wait();   } // останавливает поток, который вызвал wait()
          catch(InterruptedException e) { System.out.println(“Прерывание!);}
      System.out.println(“Получено: “+ n);
      valueSet = false;
      notify () ; // возобновляет поток, который был остановлен на этом объекте
      return n; }

   synchronized void put (int n) {
     while(valueSet)
          try {   wait(); } // // останавливает поток, который вызвал wait()
          catch(InterruptedException e) { System.out.println(“Прерывание!”) ; }
     this.n = n;
     valueSet = true;
     System.out.println("Отправлено: "+ n);
     notify () ; } // возобновляет поток, который был остановлен на этом объекте
}
```