Потоки

Для некоторых задач бывает удобно (и даже необходимо) организовать параллельное выполнение нескольких частей программы, чтобы создать у пользователя впечатление одновременного выполнения этих частей, или — если на компьютере установлено несколько процессоров — чтобы они действительно выполнялись одновременно.

Каждая из этих самостоятельных подзадач называется потоком (*thread*). Программа пишется так, словно каждый поток запускается сам по себе и использует процессор в монопольном режиме. На самом деле существует некоторый системный механизм, который обеспечивает совместное использованием процессора, но в основном думать об этом вам не придется. Модель потоков (и ее поддержка в языке *Java*) является программным механизмом, упрощающим одновременное выполнение нескольких операций в одной и той же программе.

Потоки

Как только вы создадите новый поток, создастся новый стэк, в который будут помещаться записи о вызовах методов, совершенных из этого нового потока.
JVM от Sun, начиная с версии 1.2 мапит Java-потоки на нативные потоки операционной системы один-к-одному, однако у JVM свой планировщик потоков, который не зависит от планировщика операционной системы под которой работает JVM. Одно следует знать точно: когда дело доходит до потоков, то здесь нельзя давать ни каких гарантий. Нельзя точно определить как будут выполняться параллельные потоки. Ответственность за это полностью лежит на планировщике JVM.
В Java есть два вида потоков: потоки-демоны (daemon threads) и пользовательские потоки (user threads). Здесь мы будем рассматривать в основном user threads. Разница между этими двумя типами потоков в том, что JVM завершает выполнение программы когда все пользовательские потоки завершат свое выполнение. Как только завершил свое выполнение последний пользовательский поток, JVM остановится не зависимо от того, в каком состоянии находятся потоки-демоны.

Потоки

Программный поток представляет некоторую задачу или операцию, поэтому нам понадобятся средства для описания этой задачи. Их предоставляет интерфейс ***Runnable***. Чтобы определить задачу, реализуйте ***Runnable*** и напишите метод ***run()***, содержащий код выполнения нужных действий.

```
class LiftOff implements Runnable {
 protected int countDown = 10; // Значение по умолчанию
 private static int taskCount = 0;
 private final int id = taskCount++;
 public LiftOff() {}
 public LiftOff(int countDown) {
   this.countDown = countDown;
 }
 public String status() {
   return "#" + id + "(" +
     (countDown > 0 ? countDown : "Liftoff!") + "), ";
 }
 public void run() {
   while(countDown-- > 0) {
     System.out.print(status());
     Thread.yield();
   }
 }}
```

Потоки

Метод ***run()*** обычно содержит некоторый цикл, который продолжает выполняться до тех пор, пока не будет достигнуто некоторое завершающее условие. Следовательно, вы должны задать условие выхода из цикла (например, просто вернуть управление командой ***return***). Часто ***run()*** выполняется в виде бесконечного цикла, а это означает, что при отсутствии завершающего условия выполнение будет продолжаться бесконечно

```
public class MainThread {
  public static void main(String[] args) {
    LiftOff launch = new LiftOff();
    launch.run();
  }
}
```

Потоки

Определить и инстанциировать поток можно двумя способами:
Расширить класс java.lang.Thread
Реализовать интерфейс Runnable
Предпочтительным способом является реализация интерфейса Runnable. Расширить Thread – это просто, но этот способ не является хорошей практикой ООП.

Расширение java.lang.Thread

```
public class MyThread extends Thread {

    @Override
    public void run() {
        System.out.println(“MyThread");
    }
}
```

Перегруженные run() игнорируются классом Thread. Класс Thread ожидает запуска именно run() без аргументов.

java.lang.Runnable

```
class MyRunnable implements Runnable {

    @Override
    public void run() {
        System.out.println(“MyRunnable");
    }
}
```

Каждый поток начинается с создания экземпляра Thread. Независимо от того, расширяли вы Thread или реализовывали Runnable, для выполнения работы, нужно иметь экземпляр Thread.

```
MyThread t = new MyThread();
```

Если вы реализовывали Runnable, вам все равно придется создать экземпляр класса Thread.

```
MyRunnable r = new MyRunnable();
Thread t = new Thread(r);
```

Потоки

Конструктору ***Thread*** передается только объект ***Runnable***. Метод ***start()*** выполняет необходимую инициализацию потока, после чего вызов метода ***run()*** интерфейса ***Runnable*** запускает задачу на выполнение в новом потоке. Т.е. для того, чтобы запустить поток на исполнение надо вызвать метод start()

```
t.start()
```

После этого происходит следующее:

Стартует новый поток выполнения (с новым стэком вызовов).

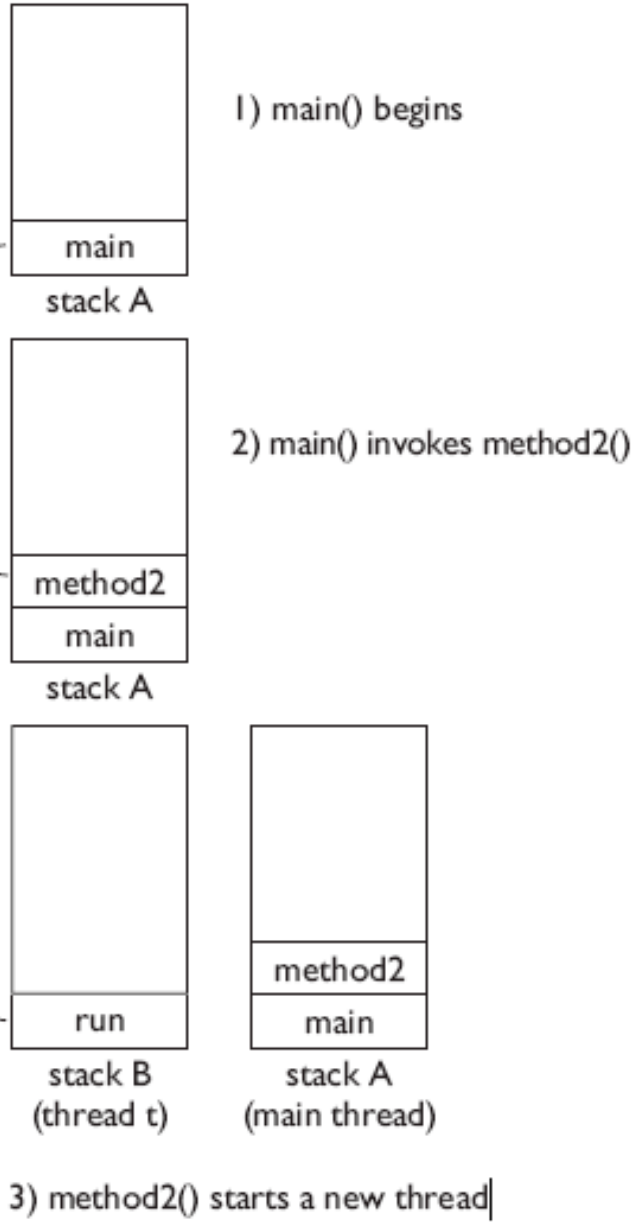
Поток переходит из состояния new (новый) в состояние работоспособный (runnable).

Когда поток получает шанс выполниться, он вызывает метод run().

Еще раз заметьте, что для старта потока, нужно вызывать метод start() экземпляра класса Thread. Если вы вызовете метод run() из вашего класса, который реализует интерфейс Runnable или даже если вы вызовете run из класса, расширяющего Thread, то ничего страшного не произойдет. Не возникнет ни каких исключительных ситуаций относящихся к потокам. Метод просто выполнится, но новый поток не создастся! Метод выполнится в том же потоке, из которого был запущен!

Потоки

```
public static void main(String [] args) {

   //  running

   //  some code

   //  in main()

   method2();

   //  running

   //  more code
}

static void method2() {
   Runnable r = new MyRunnable();

   Thread t = new Thread(r);

   t.start();

   //  do more stuff

}
```



Потоки

```
public class Starter {
    public static void main(String[] args) {
        NameRunnable nr = new NameRunnable();
        Thread one = new Thread(nr);
        Thread two = new Thread(nr);
        Thread three = new Thread(nr);
        one.setName("Первый");
        two.setName("Второй");
        three.setName("Третий");
        one.start();
        two.start();
        three.start(); }}

class NameRunnable implements Runnable {
    public void run() {
        for (int x = 1; x <= 3; x++) {
            System.out.println("Запущен "
                    + Thread.currentThread().getName()
                    + ", x равен " + x);
        }}}
```

Потоки

В каждом отдельном потоке, порядок выполнения предсказуем, т.е., например, наши циклы всегда будут инкрементировать x и выводить в консоль сообщение, но порядок выполнения потоков не предсказуем. Велика вероятность того, что такой маленький и простой цикл всегда будет успевать выполняться до переключения на другой поток, но если увеличить длину цикла хотя бы до 400, можно увидеть, что потоки прерываются.

Так же не гарантируется что если поток1 начал выполняться первым, то он и закончит выполняться первым. Он может закончить свою работу и самым последним. Мы не контролируем планировщик потоков, поэтому не можем предсказать порядок выполнения, а соответственно, и порядок завершения потоков. Однако, существует способ сказать потоку чтобы не запускался пока какой-нибудь другой поток не закончился.

Когда метод run() потока завершился, поток перестает быть потоком исполнения, стек вызовов удаляется, а поток считается мертвым (dead). Если поток был запущен, то он никогда не может быть запущен повторно! Если у вас есть ссылка на поток, и вы вызовете его метод start() повторно, то получите IllegalThreadStateException. Запустить поток можно только из состояния new (новый), а состояние new он имеет только перед первым запуском. Если попытаться запустить поток из состояния runnable или dead, то получим IllegalThreadStateException.

Состояния потоков

Новый (new). После создания экземпляра потока, он находится в состоянии Новый до тех пор, пока не вызван метод start(). В этом состоянии поток не считается живым.

Работоспособный (runnable). Поток переходит в состояние Работоспособный, когда вызывается метод start(). Поток может перейти в это состояние также из состояния Работающий или из состояния Блокирован. Когда поток находится в этом состоянии, он считается живым.

Работающий (running). Поток переходит из состояния Работоспособный в состояние Работающий, когда Планировщик потоков выбирает его из runnable pool как работающий в данный момент.

Ожидающий (waiting)/Заблокированный (blocked)/Спящий(sleeping). Эти состояния характеризуют поток как не готовый к работе.

поток уже не runnable, но он может вернуться в это состояние. Поток может быть заблокирован – это может означать, например, что он ждет освобождения каких-нибудь ресурсов, и разблокируется когда эти ресурсы освободятся. Поток может спать потому, что в его методе run() встретился метод sleep(). Просыпается он тогда, когда время его сна истекло (таймер). Поток может находиться в состоянии ожидания если в его методе run() встретится метод wait().

Мёртвый (dead). Поток считается мёртвым, когда его метод run() полностью выполнен.

sleep()

Метод sleep() – это статический метод класса Thread. Метод sleep() вызывает засыпание текущего потока на заданное время (в миллисекундах).

Во время выполнения метода sleep() система перестает выделять потоку процессорное время, распределяя его между другими потоками. Метод sleep() может выполняться либо заданное кол-во времени (миллисекунды или наносекунды) либо до тех пор пока он не будет остановлен прерыванием (в этом случае он сгенерирует исключение InterruptedException).

Самое главное, что метод sleep может усыпить только текущий поток, т.е. даже если есть ссылка на другой объект-поток, то его усыпить из другого потока не получится.

```
class NameRunnable implements Runnable {
    public void run() {
        for (int x = 1; x <= 4; x++) {
            System.out.println("Запущен "
                    + Thread.currentThread().getName()
                    + ", x равен " + x);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {}
    }
 }
```

yelld()

Статический метод Thread.yield() заставляет процессор переключиться на обработку других потоков системы. Метод может быть полезным, например, когда поток ожидает наступления какого-либо события и необходимо чтобы проверка его наступления происходила как можно чаще. В этом случае можно поместить проверку события и метод Thread.yield() в цикл:

```
//Ожидание поступления сообщения
while(!msgQueue.hasMessages())//Пока в очереди нет
сообщений
{
    Thread.yield();//Передать управление другим потокам
}
```

Тут не предоставляется ни каких гарантий что один поток уступит другому. Метод yield() не пытается отправить поток в состояние waiting/sleeping/blocking, он пытается перевести его из состояния running в runnable и вызов этого метода может не возыметь ни какого эффекта.

join()

В Java предусмотрен механизм, позволяющий одному потоку ждать завершения выполнения другого. Для этого используется метод join(). Например, чтобы главный поток подождал завершения побочного потока myThready, необходимо выполнить инструкцию myThready.join() в главном потоке. Как только поток myThready завершится, метод join() вернет управление, и главный поток сможет продолжить выполнение.

```
Thinker brain = new Thinker(); //Thinker - потомок класса Thread.
brain.start();               //Начать "обдумывание".
do
{   //mThinkIndicator - анимированная картинка.
    mThinkIndicator.refresh();

    try{  //Подождать окончания мысли четверть секунды.
        brain.join(250);
    }catch(InterruptedException e){}
}
while(brain.isAlive());
```

Синхронизация

```
class Account{
    int summ;
}
class AccountDoer implements Runnable{
    Account workAccount;
    public AccountDoer(Account ac){
        this.workAccount=ac;
    }
    @Override
    public void run() {
        doSomeWithAccount(100);
    }
    private void doSomeWithAccount(int k){
        if (workAccount.summ>k){
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {}
            workAccount.summ=workAccount.summ-k;
        }
    }}
```

Синхронизация

```
public class SyncTest {
    Account myAccount = new Account();
    public static void main(String[] str)
                          throws InterruptedException{
        SyncTest st = new SyncTest();
        st.myAccount.summ=150;
        AccountDoer d1 = new AccountDoer(st.myAccount);
        AccountDoer d2 = new AccountDoer(st.myAccount);
        Thread t1 = new Thread(d1);
        Thread t2 = new Thread(d2);
        t1.start();
        t2.start();
        Thread.sleep(5000);
        System.out.println(st.myAccount.summ);

    }
}
```

Синхронизация

```
class Account{
    int summ;
   public void decSumm(int i){
        if (summ>i){
            try {
                Thread.sleep(100);
            } catch (InterruptedException ex) {}
            summ=summ-i;
        }}}
class AccountDoer implements Runnable{
    Account workAccount;
    public AccountDoer(Account ac){
        this.workAccount=ac;
    }
    @Override
    public void run() {
        doSomeWithAccount(100);
    }
    private void doSomeWithAccount(int k){
        workAccount.decSumm(k);}}
```

Синхронизация

```
class Account{
    int summ;
  synchronized public void decSumm(int i){
        if (summ>i){
            try {
                Thread.sleep(100);
            } catch (InterruptedException ex) {
            }
            summ=summ-i;
        }
    }
}
```

Синхронизация

```
class AccountDoer implements Runnable{
    Account workAccount;
    public AccountDoer(Account ac){
        this.workAccount=ac;
    }
    @Override
    public void run() {
        doSomeWithAccount(100);
    }
    private void doSomeWithAccount(int k){
        synchronized(workAccount){
        if (workAccount.summ>k){
            try {
                Thread.sleep(100);
            } catch (InterruptedException ex) {}
            workAccount.summ=workAccount.summ-k;
        }
        }
    }}
```

Синхронизация

Каждый объект в Java имеет встроенную блокировку, которая вступает в игру только в том случае, если объект имеет синхронизированные методы. Когда мы входим в синхронизированный нестатический метод, мы автоматически получаем блокировку, связанную с текущим экземпляром класса (текущий – это тот, код которого выполняется в данный момент).

Поскольку существует только одна блокировка на объект, то если один поток захватил блокировку, ни один другой поток не может ее захватить, пока первый не отпустит ее. Это означает, что ни какой другой поток не может войти в синхронизированный код пока блокировка не будет освобождена. Запомните следующие ключевые моменты блокировки и синхронизации:

- Только методы (или блоки) могут быть synchronized.
- Каждый объект имеет только одну блокировку.
- Не все методы в классе нуждаются в синхронизации. Класс может иметь как синхронизированные методы, так и не синхронизированные.

Синхронизация

• Если два потока собираются выполнить синхронизированный метод в классе, и оба потока используют один и тот же экземпляр класса для вызова метода, то только один поток сможет выполнить метод. Другой поток будет ждать пока первый не закончит выполнение метода. Иными словами, как только поток получает блокировку на объект, ни один другой поток не может войти ни в один из синхронизированных методов класса (для этого объекта).

• Если класс имеет и синхронизированные и не синхронизированные методы, то несколько потоков могут получить доступ к не синхронизированным методам класса. Если ваш метод не имеет доступа к данным, которые вы хотите защитить, то вам не нужно объявлять этот метод как синхронизированный. В некоторых случаях синхронизация может привести к взаимной блокировке, поэтому не стоит ей злоупотреблять.

• Если поток уходит спать, он удерживает все свои блокировки, а не освобождает их.

• Поток может захватить несколько блокировок. Например, поток может войти в синхронизированный метод, таким образом захватив блокировку, а затем сразу же вызвать синхронизированный метод другого объекта, получив также и его блокировку.

• Можно синхронизировать не весь метод, а только блок кода.

Потоко-безопасные классы

Когда класс тщательно синхронизирован, говорят что он является “потоко-безопасным” (“thread-safe”). Многие классы в Java API являются потоко-безопасными. Например, StringBuffer и StringBuilder практически идентичные классы, однако все методы в StringBuffer синхронизированы, а в StringBuilder – нет, поэтому использование StringBuffer в многопоточной среде безопасно, а StringBuilder – нет. Тем не менее, не всегда можно полагаться на то, что класс обеспечивает необходимую защиту. Вам все еще необходимо тщательно продумать как использовать эти классы.

Существуют потокобезопасные обретки для стандартных коллекций. Их можно получить вызвав один из статических методов класса Collection, например

```
List names =
        Collections.synchronizedList(new LinkedList());
```

```
class ThreadSafeStack<T>{
    private static class Elem<T>{
        T value;
        Elem<T> next;
        Elem(T val){value=val;next=null;}
        Elem(T val,Elem<T> next){
            value=val;
            this.next=next;
        }
    }
    private Elem<T> body;
    synchronized public boolean isEmpty(){
        return body==null;
    }
    synchronized public void push(T value){
        Elem<T> temp = new Elem<>(value,body);
        body=temp;}
    synchronized public T pop(){
           T temp = body.value;
           body=body.next;
           return temp;
      } }
```

Потоко-безопасные классы

```
public static void main(String[] args) {
  final ThreadSafeStack<Integer> st =
                                    new ThreadSafeStack<>();
  st.push(10);
  class StackPoper extends Thread {
         public void run() {
             if (!st.isEmpty()){
                 try {
                     Thread.sleep(100);
                 } catch (InterruptedException ex) {}
                 st.pop();
             }
         }
     }

   Thread t1 = new StackPoper();
   Thread t2 = new StackPoper();
   t1.start();
   t2.start();
 }
```

Потоко-безопасные классы

```
synchronized public T pop(){
       if (body!=null){
          T temp = body.value;
          body=body.next;
          return temp;
       }
      else
          return null;
   }
```

wait(), notify() и notifyAll()

Методы wait() и notify() являются методами класса Object. Каждый объект имеет блокировку. Таким же образом, каждый объект может иметь список потоков, которые ждут от него сигнала (уведомления). Поток попадает в этот список ожидающих при помощи вызова wait() целевого объекта. С этого момента он не выполняет ни каких инструкций пока не будет вызван метод notify(). Если один объект ждут несколько потоков, то после вызова notify(), планировщиком потоков будет выбран один поток (какой именно не известно), который приступит к исполнению. Если ожидающих потоков нет, то ничего и не произойдет.

```
public class ConnectionPool {
 private List<Connection> connections = createConnections();
 private List<Connection> createConnections() {
    List<Connection> conns = new ArrayList<Connection>(5);
     for (int i = 0; i < 5; i++) { ... add a Connection to conns }
     return conns; }
}
```

wait(), notify() и notifyAll()

```
public Connection getConnection()
                    throws InterruptedException {
  synchronized (connections) {
    while (connections.isEmpty()) {
       //sleep(100);
       connections.wait();
    }
return connections.remove(0); }
}
```

wait(), notify() и notifyAll()

```
public void returnConnection(Connection conn) {
synchronized (connections) {
  connections.add(conn);
  connections.notify();
 }
}
```

```
public void returnConnection(Connection conn) {
synchronized (connections) {
connections.add(conn);
connections.notify();
// bad: woken thread can't start until we
// come out of synchronized block!
updateStatistics(conn); }
}
```

wait(), notify() и notifyAll()

единственное место, где допустимо вызывать метод ***wait()***, — это синхронизированный метод или блок (метод ***sleep()*** можно вызывать в любом месте, так как он не манипулирует блокировкой). Если вызвать метод ***wait()*** или ***notify()*** в обычном методе, программа скомпилируется, однако при ее выполнении возникнет исключение ***IllegalMonitorStateException*** Это сообщение означает, что поток, востребовавший методы ***wait(), notify()*** или ***notifyAll()***, должен быть «хозяином» блокируемого объекта (владеть объектом блокировки) перед вызовом любого из данных методов.

Присоединение к нескольким потокам

```
public class TestJoin {
    public static void main(String[] arg)
                                throws InterruptedException{
        Thread[] pool = new Thread[10];
        for(int i=0;i<10; i++){
            pool[i] = new Thread(new MyThread(i));
            pool[i].start();
            //pool[i].join();

        }
        for (int i=0;i<10; i++){
            pool[i].join();
        }
        System.out.println("Thats all!!!");
   }
}
```

Прерывание потока

Java Memory Model JMM

Модель памяти Java (англ. *Java Memory Model*, *JMM*) описывает поведение потоков в среде исполнения Java. Модель памяти — часть семантики языка Java, и описывает, на что может и на что не должен рассчитывать программист, разрабатывающий ПО не для конкретной Java-машины, а для Java в целом.

Две основные вещи, которые вводят энтропию в многопоточный код - это reordering и visibility.

Видимость (visibility)

Один поток может в какой-то момент временно сохранить значение некоторых полей не в основную память, а в регистры или локальный кэш процессора, таким образом второй поток, выполняемый на другом процессоре, читая из основной памяти, может не увидеть последних изменений поля. И наоборот, если поток на протяжении какого-то времени работает с регистрами и локальными кэшами, читая данные оттуда, он может сразу не увидеть изменений, сделанных другим потоком в основную память.

Java Memory Model JMM

Reordering

Для увеличения производительности процессор/компилятор могут переставлять местами некоторые инструкции/операции. Вернее, с точки зрения потока, наблюдающего за выполнением операций в другом потоке, операции могут быть выполнены не в том порядке, в котором они идут в исходном коде. Так же эффект реордеринга может наблюдаться, когда один поток кладет результаты первой операции в регистр или локальный кэш, а результат второй операции кладет непосредственно в основную память. Тогда второй поток, обращаясь к основной памяти может сначала увидеть результат второй операции, и только потом первой, когда все регистры или кэши синхронизируются с основной памятью.

Еще одна причина reordering, может заключаться в том, что процессор может решить поменять порядок выполнения операций, если, например, сочтет что такая последовательность выполнится быстрее.

```
//first thread
result = calc();
reslitReady = true;
```

```
//second thread
if (reslitReady){
    takeDesision(result);
}
```

Java Memory Model JMM

На машинах x86 довольно сложно показать ломающий reordering. Это можно показать на каком-нибудь ARM'е или PowerPC.

Happens-before

Пусть есть поток **X** и поток **Y** (не обязательно отличающийся от потока **X**). И пусть есть операции **A** (выполняющаяся в потоке **X**) и **B** (выполняющаяся в потоке **Y**).

В таком случае, **A happens-before B** означает, что *все изменения, выполненные потоком **X** до момента операции **A** и изменения, которые повлекла эта операция, видны потоку **Y** в момент выполнения операции **B** и после выполнения этой операции.*

Happens-before

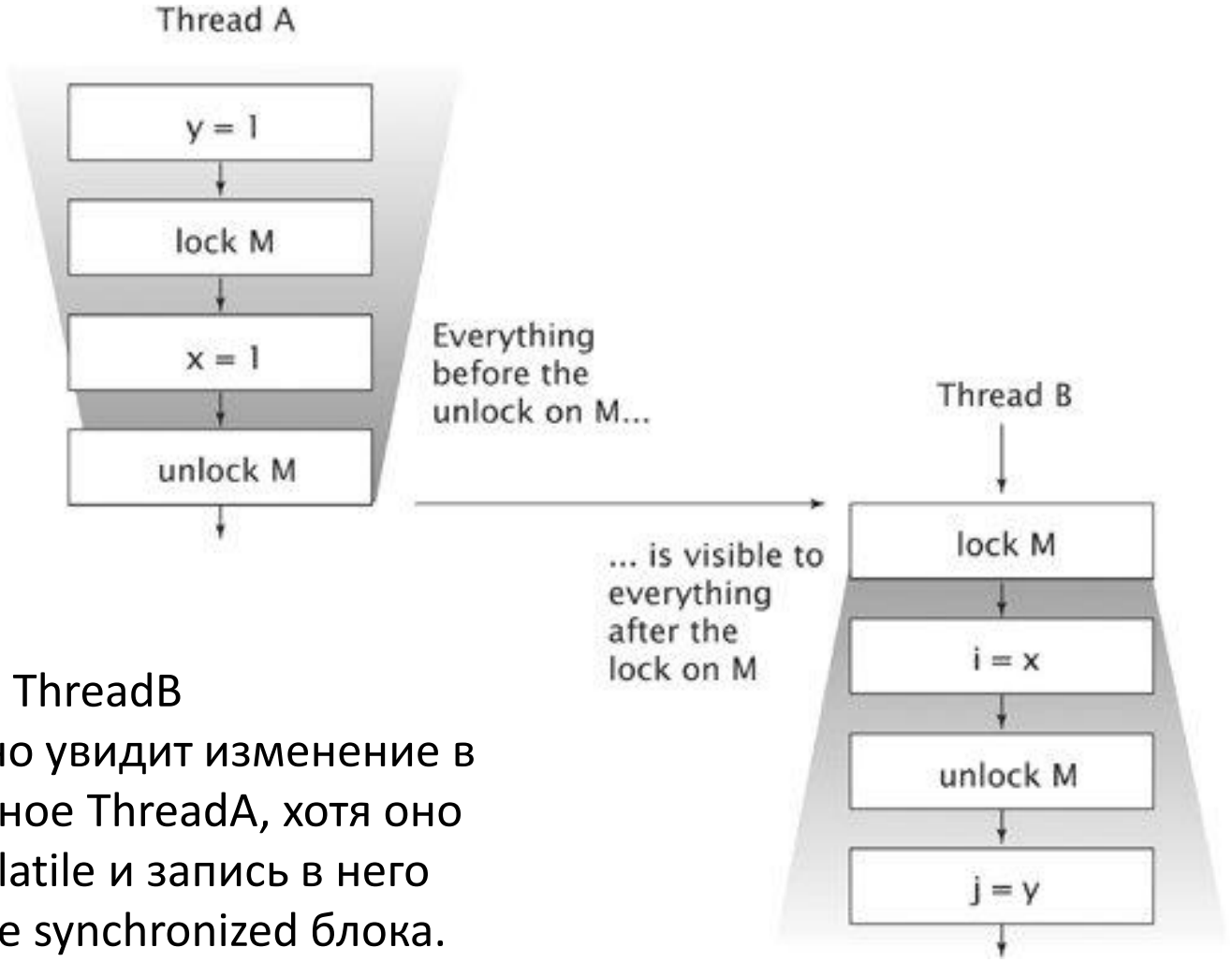
Список операций связанных отношением happens-before:

- В рамках одного поток любая операция *happens-before* любой операцией следующей за ней в исходном коде
- Освобождение лока (unlock) *happens-before* захват того же лока (lock)
- Выход из synhronized блока/метода *happens-before* вход в synhronized блок/метод на том же мониторе
- Запись volatile поля *happens-before* чтение того же самого volatile поля
- Завершение метода run экземпляра класса Thread *happens-before* выход из метода join() или возвращение false методом isAlive() экземпляром того же треда
- Вызов метода start() экземпляра класса Thread *happens-before* начало метода run() экземпляра того же треда
- Завершение конструктора *happens-before* начало метода finalize() этого класса
- Вызов метода interrupt() на потоке *happens-before* когда поток обнаружил, что данный метод был вызван либо путем выбрасывания исключения InterruptedException, либо с помощью методов isInterrupted() или interrupted()

Связь happens-before транзитивна, т.е. если X happens-before Y, а Y happens-before Z, то X happens-before Z.

Happens-before

В отношении happens-before есть очень большой дополнительный бонус: данное отношение дает не только видимость volatile полей или результатов операций защищенных монитором или локом, но и видимость вообще всего-всего, что делалось до события hapens-before.



Так на рисунке ThreadB гарантированно увидит изменение в поле *y*, сделанное ThreadA, хотя оно не является volatile и запись в него происходит вне synchronized блока.

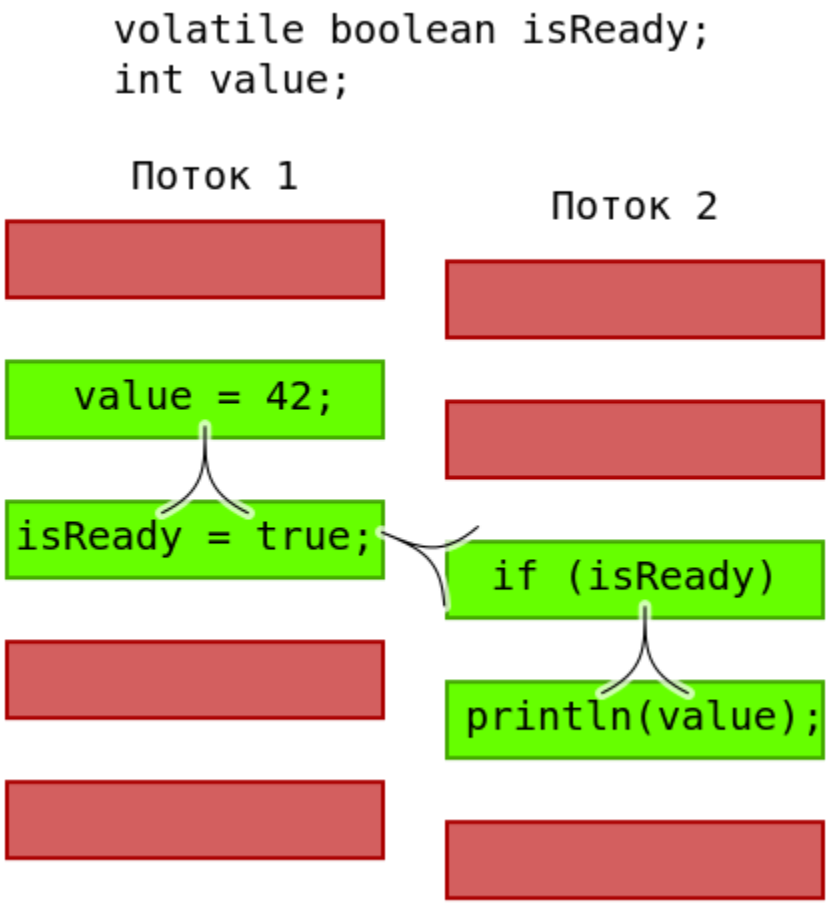
volatile переменные

Ключевое слово volatile применяется только к переменным и имеет следующие эффекты в многопоточном программировании:

1. переменная всегда считывается из основной памяти, и никогда не кэшируется в память потока, а значит всегда доступна любому потоку;
2. при запросах на чтение и запись от нескольких потоков, системой гарантируется выполнение вначале запросов на запись;
3. гарантируется атомарность операций чтения/записи, правда это актуально для переменных только типа long и double, для остальных типов эти действия и так атомарны. Для всех прочих операций, как ++, синхронизация делается внешним образом,

либо используются атомарные типы как AtomicInteger из пакета java.util.concurrent.atomic;

4. в результате предыдущих пунктов, потоки не блокируются в ожидании освобождения монитора;



```
class AtomicityTest implements Runnable {
  private int i = 0;
  public int getValue() { return i; }
  private synchronized void evenIncrement() { i++; i++; }
  public void run() {
    while(true)
      evenIncrement();
  }
  public static void main(String[] args) {
   ExecutorService exec = Executors.newCachedThreadPool();
   AtomicityTest at = new AtomicityTest();
   exec.execute(at);
   while(true) {
      int val = at.getValue();
      if(val % 2 != 0) {
        System.out.println(val);
        System.exit(0);
      }
    }
  }}
```

Атомарные классы

```
public class MyObject
   {
      private int i;
      public void op1() { i++; }
      public void op2() { i--; }
}
```

```
public class MyObject {
  private int i;
  public synchronized void op1() { i++; }
  public synchronized void op2() { i--; }
}
```

```
public class MyObject {
  private int i;
  public synchronized void op1() { i++; }
  public synchronized void op2() { i--; }
  public void op3() { i = 0; }
}
```

Пусть на ресурсе R определен набор связанных операций {O1, O2 ... OX). Мы говорим, что операция On ресурса R является атомарной, если в процессе ее работы никакая другая операция из множества {O1; ON} не может изменить состояние этого объекта так, что это повлияет на результат операции On.

Сравнение и замена (Compare and swap - CAS)

Операция CAS включает 3 объекта-операнда: адрес ячейки памяти (V), ожидаемое старое значение (A) и новое значение (B). Процессор атомарно обновляет адрес ячейки, если новое значение совпадает со старым, иначе изменений не зафиксируется. В любом случае, будет выведена величина, которая предшествовала времени запроса. Некоторые варианты метода CAS просто сообщают, успешно ли прошла операция, вместо того, чтобы отобразить само текущее значение. Фактически, CAS только сообщает: "Наверное, адрес V равняется A; если так оно и есть, поместите туда же B, в противном случае не делайте этого, но обязательно скажите мне, какая величина - текущая."

Сравнение и замена (Compare and swap - CAS)

Самым естественным методом использования CAS для синхронизации будет чтение значения A с адреса V, проделать многошаговое вычисление для получения нового значения B, и затем воспользоваться методом CAS для замены значения параметра V с прежнего, A, на новое, B. CAS выполнит задание, если V за это время не менялось.

Задания таких методов, как CAS, позволяют алгоритму выполнить последовательность операций "чтение-модификация-запись" без опасения, что в это же время другой поток изменит переменную, потому что если бы это произошло, CAS обнаружил бы это и прервал бы выполнение своего задания, а алгоритм задал бы повтор операции.

```
public class SynchronizedCounter {
  private int value;
  public synchronized int getValue() { return value; }
  public synchronized int increment() { return ++value; }
  public synchronized int decrement() { return --value; }
}
```

```
public class SimulatedCAS {
   private int value;
   public synchronized int getValue() { return value; }
   public synchronized int compareAndSwap(int
                        expectedValue, int newValue) {
     int oldValue = value;
     if (value == expectedValue) value = newValue;
     return oldValue;
} }
```

```
public class CasCounter {
    private SimulatedCAS value;
    public int getValue() { return value.getValue(); }
    public int increment() {
       int oldValue = value.getValue();
      while (value.compareAndSwap(oldValue, oldValue+1)
                                              != oldValue)
          oldValue = value.getValue();
      return oldValue + 1; }
}
```

Безблокировочные алгоритмы и алгоритмы, не требующие ожидания (wait-free)

```
public class ConcurrentStack<E> {
    AtomicReference<Node<E>> head = new AtomicReference();
    public void push(E item) {
        Node<E> newHead = new Node(item); Node<E> oldHead;
        do {
            oldHead = head.get();
            newHead.next = oldHead;
        } while (!head.compareAndSet(oldHead, newHead));}
    public E pop() {
        Node<E> oldHead;Node<E> newHead;
        do {
            oldHead = head.get();
            if (oldHead == null)
                return null;
            newHead = oldHead.next;
        } while (!head.compareAndSet(oldHead,newHead));
        return oldHead.item;
    }
    static class Node<E> {
        final E item;
        Node<E> next;
        public Node(E item) { this.item = item; }}}
```

Исполнители

Исполнители (***executors***), появившиеся в библиотеке ***java.util.concurrent*** в *Java SE5*, упрощают многозадачное программирование за счет автоматизации управления объектами ***Thread***. Они создают дополнительную логическую прослойку между клиентом и выполнением задачи; задача выполняется не напрямую клиентом, а промежуточным объектом. Исполнители позволяют управлять выполнением асинхронных задач без явного управления жизненным циклом потоков. Именно такой способ запуска задач рекомендуется использовать в *Java SE5/6*.

Вместо явного создания объектов ***Thread*** мы можем воспользоваться исполнителем.

```
class MyThread implements Runnable{
    private int i;
    public MyThread(int i){
       this.i=i;
    }
    @Override
    public void run() {
        while (!Thread.interrupted()){
            System.out.println(i);
        }
    }}
```

Исполнители

```
public class TestExecutors {
    public static void main(String[] str)
                                throws InterruptedException{
     ExecutorService sv = Executors.newCachedThreadPool();
     for (int i=0;i<3;i++){
            sv.execute(new MyThread(i));
     }
     Thread.sleep(1000);
     sv.shutdownNow();
    }
}
```

Объект ***ExecutorService*** умеет создавать необходимый контекст для выполнения объектов ***Runnable***. Класс ***CachedThreadPool*** создает один поток для каждой задачи. Обратите внимание: объект ***ExecutorService*** создается статическим методом класса ***Executors***, определяющим разновидность исполнителя:

Исполнители

В потоковом пуле фиксированного размера (***FixedThreadPool***) используется ограниченный набор потоков для выполнения переданных задач

```
class MyThread implements Runnable{
    private int i;
    public MyThread(int i){this.i=i;}
    @Override
    public void run() {
        for (int j=0;j<1000;j++){
            System.out.println(i);
            Thread.yield();
        }
    }}
public class TestExecutors {
    public static void main(String[] str)
                                throws InterruptedException{
     ExecutorService sv = Executors.newFixedThreadPool(2);
        for (int i=0;i<4;i++){
            sv.execute(new MyThread(i));
        }
       sv.shutdown();         }}
```

Возврат значений из задач

Интерфейс ***Runnable*** представляет отдельную задачу, которая выполняет некоторую работу, но не возвращает значения. Если вы хотите, чтобы задача возвращала значение, реализуйте интерфейс ***Callable*** вместо интерфейса ***Runnable***. Параметризованный интерфейс ***Callable***, появившийся в *Java SE5*, имеет параметр типа, представляющий возвращаемое значение метода ***call()*** (вместо ***run()***), а для его вызова должен использоваться метод ***ExecutorService.submit()***.

```
class TaskWithResult implements Callable<Integer> {
  private int id;
  public TaskWithResult(int id) {
    this.id = id;
  }
  public Integer call() {
    for (int i=0;i<100000;i++){
      Thread.yield();
    }
    return id;
  }
}
```

Возврат значений из задач

```
public class TestCallable{
  public static void main(String[] args) {
    ExecutorService exec =
Executors.newCachedThreadPool();
    ArrayList<Future<Integer>> results =new ArrayList<>();
    for(int i = 0; i < 10; i++)
      results.add(exec.submit(new TaskWithResult(i)));
    int doneCount=0;
    Future<Integer> fs;
    while (doneCount<10){
        for(int i=0;i<results.size();i++){
          fs=results.get(i);
            try {
            if (fs.isDone()){
                System.out.println(fs.get());
                results.remove(fs);doneCount++;}}
              catch(InterruptedException e) {}
              catch(ExecutionException e) {}
    }}
  exec.shutdown();}}
```

Возврат значений из задач

Метод ***submit()*** создает объект ***Future***, параметризованный по типу результата, возвращаемому ***Callable***. Вы можете обратиться к ***Future*** с запросом ***isDone()***, чтобы узнать, завершена ли операция. После завершения задачи и появления результата производится его выборка методом ***get()***. Если ***get()*** вызывается без предварительной проверки ***isDone()***, вызов блокируется до появления результата. Также можно вызвать ***get()*** с интервалом тайм-аута.

Перегруженный метод ***Executors.callable()*** получает ***Runnable*** и выдает ***Callable.*** ***ExecutorService*** содержит методы для выполнения коллекций объектов ***Callable***.

Потоки-демоны

Демоном называется поток, предоставляющий некоторый сервис, работая в фоновом режиме во время выполнения программы, но при этом не является ее неотъемлемой частью. Таким образом, когда все потоки не-демоны заканчивают свою деятельность, программа завершается. И наоборот, если существуют работающие потоки не-демоны, программа продолжает выполнение.

Демоны завершаются «внезапно», при завершении последнего не-демона. Таким образом, сразу же при выходе из ***main()*** *JVM* немедленно прерывает работу всех демонов, не соблюдая никакие формальности. Невозможность корректного завершения демонов ограничивает возможности их применения.

Объекты Lock

Библиотека *Java SE5* ***java.utiLconcurrent*** также содержит явный механизм управления мьютексами, определенный в ***java.util.concurrent.locks***. Объект ***Lock*** можно явно создать в программе, установить или снять блокировку; правда, полученный код будет менее элегантным, чем при использовании встроенной формы. С другой стороны, он обладает большей гибкостью при решении некоторых типов задач.

```
import java.util.concurrent.locks.*;
public class MutexEvenGenerator  {
  private int currentEvenValue = 0;
  private Lock lock = new ReentrantLock();
  public int next() {
    lock.lock();
    try {
      ++currentEvenValue;
      ++currentEvenValue;
      return currentEvenValue;
    } finally {
      lock.unlock();
    }}
```

Объекты Lock

При использовании объектов ***Lock*** следует применять идиому, показанную в примере: сразу же за вызовом ***lock()*** необходимо разместить конструкцию ***try-finally***, при этом в секцию ***finally*** включается вызов ***unlock()*** — только так можно гарантировать снятие блокировки.

Хотя ***try-finally*** требует большего объема кода, чем ключевое слово ***synchronized***, явное использование объектов ***Lock*** обладает своими преимуществами. При работе с объектами ***Lock*** можно сделать все необходимое в секции ***finally***.

В общем случае использование ***synchronized*** уменьшает объем кода, а также радикально снижает вероятность ошибки со стороны программиста, поэтому явные операции с объектами ***Lock*** обычно выполняются только при решении особых задач. Например, с ключевым словом ***synchronized*** нельзя попытаться получить блокировку с неудачным исходом или попытаться получить блокировку в течение некоторого промежутка времени с последующим отказом — в подобных случаях приходится использовать библиотеку ***concurrent***:

Объекты Lock

```
boolean captured = false;
    try {
      captured = lock.tryLock(2, TimeUnit.SECONDS);
    } catch(InterruptedException e) {
      throw new RuntimeException(e);
    }
    try {
      System.out.println("tryLock(2, TimeUnit.SECONDS): "+
        captured);
    } finally {
      if(captured)
        lock.unlock();
    }
```

CountDownLatch

```
class TaskPortion implements Runnable {
  private static int counter = 0;
  private final int id = counter++;
  private static Random rand = new Random(47);
  private final CountDownLatch latch;
  TaskPortion(CountDownLatch latch) {
    this.latch = latch;
  }
  public void run() {
    try {
      doWork();latch.countDown();
    } catch(InterruptedException ex) {}
  }
 public void doWork() throws InterruptedException {
    TimeUnit.MILLISECONDS.sleep(rand.nextInt(2000));
    System.out.println(this + "completed");
  }
  public String toString() {
    return String.format("%1$-3d ", id);}}
```

CountDownLatch

```
class WaitingTask implements Runnable {
  private static int counter = 0;
  private final int id = counter++;
  private final CountDownLatch latch;
  WaitingTask(CountDownLatch latch) {
    this.latch = latch;
  }
  public void run() {
    try {
      latch.await();
     System.out.println("Latch barrier passed " + this);
    } catch(InterruptedException ex) {
      System.out.println(this + " interrupted");
    }
  }
  public String toString() {
    return String.format("WaitingTask %1$-3d ", id);
  }
}
```

CountDownLatch

```
 class CountDownLatchDemo {
  static final int SIZE = 100;
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    CountDownLatch latch = new CountDownLatch(SIZE);
    for(int i = 0; i < 10; i++)
      exec.execute(new WaitingTask(latch));
    for(int i = 0; i < SIZE; i++)
      exec.execute(new TaskPortion(latch));
    System.out.println("Launched all tasks");
    exec.shutdown();
  }
}
```

CountDownLatch

Класс ***CyclicBarrier*** используется при создании группы параллельно выполняемых задач, завершения которых необходимо дождаться до перехода к следующей фазе. Все параллельные задачи «приостанавливаются» у барьера, чтобы сделать возможным их согласованное продвижение вперед. Класс очень похож на ***CountDownLatch***, за одним важным исключением: ***CountDownLatch*** является «одноразовым», а ***CyclicBarrier*** может использоваться снова и снова.

```
class ServiceMan {
  private CyclicBarrier queue;
  private List<String> inQueue;
  public ServiceMan(int hardWorking) {
    inQueue = new ArrayList<String>();
    queue = new CyclicBarrier(hardWorking, new Runnable(){
       @Override
        public void run() {
          System.out.println("Filling " + inQueue);
          inQueue.clear();}});}
    public void recharge(String name) {
        try{  inQueue.add(name); queue.await();  }
        catch (InterruptedException e) {}
        catch (BrokenBarrierException e) {}} }
```

CountDownLatch

```
class Printer implements Runnable {
    private String name;
    private Random rand;
    private ServiceMan serviceMan;
    public Printer(ServiceMan serviceMan, String name) {
        this.name = name;
        this.serviceMan = serviceMan;
        this.rand = new Random();
    }
    public void run() {
        try {
            while (true) {
                TimeUnit.SECONDS.sleep(rand.nextInt(10));
                System.out.println(name + " is empty");
                serviceMan.recharge(name);
                System.out.println(name + " is ready");
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }}}
```

```
public class PrinterRecharger {
    public static void main(String args[]) {
        ServiceMan serviceMan = new ServiceMan(3);
        for (int i = 0; i < 15; i++) {
  new Thread(
    new Printer(serviceMan, "Printer" + (i + 1))).start();
        }
    }
}
```

http://www.rsdn.ru/forum/java/3622844.flat