需求：使用多线程模拟出售电影票过程，并使用同步锁解决卖票中出现的问题

**package** xhu.edu.net.Thread;

/\*使用继承Thread类实现

\*

\*/

**public** **class** SellTickets **extends** Thread {

**private** **static** **int** *tickets* = 100;

@Override

**public** **void** run() {

**try** {

**while** (*tickets* > 0) {

*sell*();

Thread.*sleep*(500);

}

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**private** **synchronized** **static** **void** sell() **throws** InterruptedException {

**if** (*tickets* > 0) {

System.***out***.println(Thread.*currentThread*().getName() + "正在售出第" + (*tickets*--) + "张票");

}

}

}

/\*

\* sleep建议不要放同步方法里面，否则休息类似不休息,不能显著的体现多线程抢占式的现象

\*

\*

\*/

**public** **class** Test1 {

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

Thread t1 = **new** SellTickets();

Thread t2 = **new** SellTickets();

t1.setName("线程1");

t2.setName("线程2");

System.***out***.println(Thread.*currentThread*().getName());

System.***out***.println(Thread.*currentThread*().getPriority());

t1.start();

t2.start();

System.***out***.println("main ppp");

}

}

/\*需求：通过使用继承Thread类实现买票过程模拟

\*

\*

\* 问题及分析：

\* !!!!!静态即共享

\* 问题1：主线程为什么直接执行线程对象的run方法会以main线程进行串行？

\* 答：因为在主线程中创建了两个子线程对象并给他们命名，

\* 但是直接调用新线程对象的run方法，并不会在主线程中激活子线程，

\* 所以主线程会将两个对象的run()代码加载进运行内存并进行顺序执行一次，

\* 所以获取到的线程名也是main。

\*

\* 问题2：start（）与run（）之间的关系？

\* 答：start被调用时此线程开始执行; Java虚拟机会调用此线程的run方法，所以run方法也不是什么神奇的方法一样可以

\* 通过对象名进行调用。

\*

\*

\* 问题3：为什么会出现后面的票先卖出，和同一张票被卖出两次的现象,不应该是票按照大小顺序由两个线程无规律交替卖出吗？

\* 答：首先，明白场景：是在主线程中开启两个子线程，并各自调用各自没有加锁的run方法（也就说明除了赋值，读写数据的基本操作具有原子性），

\* 程序最终的运行是转化为计算机上的

\* 指令进行运行，但是这些指令真的是按照我们写的顺序进行执行的吗？然而并不是，计算机会根据代码书写情况对代码进行调整

\* 或者是说优化，只要不影响最终的结果，指令的执行顺序会发生该变，比如：i=10,print(i--),j=10,

\* 那么这个指令执行的顺序极大可能是i=10,j=10,print(i),i--。

\* 这样我们就能分析下System.out.println(Thread.currentThread().getName() + "正在售出第" + (tickets--) + "张票");

\* 这条语句会因为输出需要用到一些变量值，那么会先将这些值读入自己这个线程的内存，当需要tickets的值时，

\* 会从共享内存中读取它的值，赋给该线程中的tickets,然后与将要输出的语句连接为字符串，输出

\* 再将该线程中的tickets执行减一操作再赋给tickets，最后将tickets写入共享内存。

\*

\* （将System.out.println(Thread.currentThread().getName() + "正在售出第" + (tickets--) + "张票")分解

\* 首先tickets--非原子性操作，会被分解

\* 其次tickets=tickets-1;应该是放在控制台输出之前执行因为会涉及到字符串的连接，

\* 在tickets写回内存后，所占用的时间片结束，其他线程抢占到了cpu，并且在相应的时间片内将数据输出到了控制台，

\* 当先前的线程再次抢到cpu，依然输出原来的数据，所以出现编号大的票后卖出。

\*

\* 两个线程卖出同一张票也是同样的原因，当一个线程读取到tickets，将该值连接进入了字符串，

\* 并且进行了减一，但是还没有些回内存中，这个时候另一个线程也读取到了tickets并且执行了同样的过程，

\* 这时候写没写回内存，输出没输出都不重要了，因为他们两个输出的字符串中tickets值已经被固定写死了，

\* 即便后者写回了内存通知了其他线程要更改他们私有内存中的tickets的值也没有用了。

\*

\* ！！！！！这就是使用了多线程但是没有将相应的语句进行同步操作，而导致逻辑错误。所以synchronized的作用极其重要

\* ）

\*

\*

\*

\*

\*

\*

\*

\* 问题4：为什么使用锁之后还会出现卖出第0张票？

\* 答；因为即使是锁方法，但是在进入锁之前仍然会有概率多个线程同时到达锁前的一瞬间（这里就是同时进入循环中），

\* 所以前一个线程执行完后，后面的线程仍然会继续执行。所以为了保证数据的安全性，我们应该需要再加非0判断

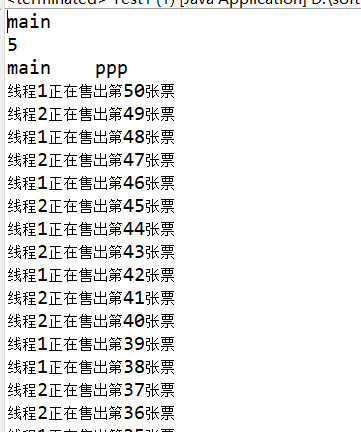
\*

\* 经过验证这里主线程的优先级与子线程的优先级一致，

\* 且当主线程的代码执行完成后，子线程才开始执行

\*/

运行截图：



虽然使用锁后保证了操作的原子性，但是效率明显感觉下降所以锁不能滥用。

总结：

为什么需要多线程？

1.服务程序，当有多个客户请求服务时，每个客户是独立进行。

2.模拟现实场景，比如卖票。

什么是线程安全？

当多个线程使用同一个资源时，往往会使得读写数据不一致，数据的不一致必然会导致程序的错误，所以需要线程安全。

线程出现安全问题的条件：

1.多线程。

2.多个线程之间共享统一个变量。

3.每个线程有着多部操作。（Java中只有读写变量的数据和给变量赋值的操作是原子性的）

Synchronized的作用与原理？

Synchrnized使得代码块保证了原子性（在数据库中叫做事务），其实现的原理是需要一个对象作为锁，任何对象都可以做锁，后来有专门的Lock类作为锁。因为每个对象上会有一个监听器，而这个监听器的使用权只能让一个线程获取，所以，synchronized将对象作为锁，只有获取到该对象的监听器的使用权才能进行对其下的代码块进行操作。

同步方法使用的锁对象：非静态方法使用的this对象作为锁，静态方法使用的这个对象类的Class（字节码）对象。

线程的状态？

1.新建状态：新建一个线程对象时。

2.就绪状态：使用了start（）方法，就进入了就绪状态（一切的资源已经准备好，只差cpu的执行权）

3.阻塞状态：因为某种原因导致线程无法继续执行，即使有cpu的执行权也无法进行推进。

4.运行状态：就绪的线程获取到cpu的执行权开始执行代码的状态，只能从就绪态进入运行状态。

5.死亡状态：线程正常结束或者异常退出，生命周期结束。