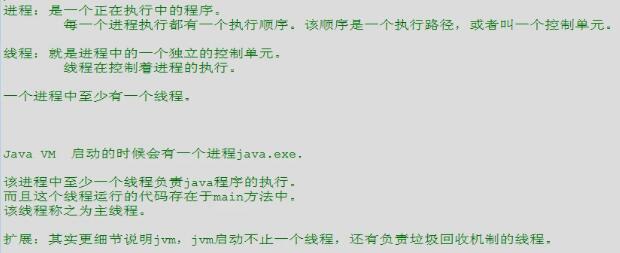
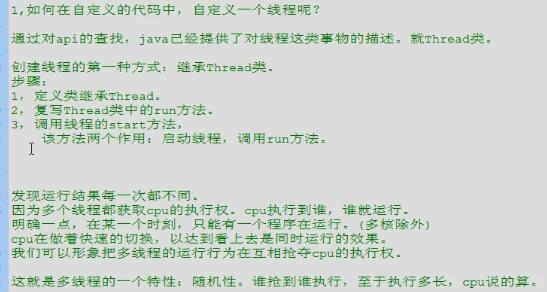
**P130 多线程（概述）**



**P131 多线程（创建线程-继承Thread类）**

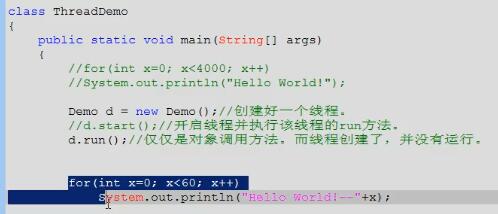


**P132 多线程（创建线程-run和start特点）**

创建线程的步骤：定义类继承Thread 🡪 复写Thread类中的run方法

目的就是将自定义的代码储存在run方法，让线程运行。

其实，程序运行的入口main函数也是一个线程，只不过是主线程。主线程就把自定义的代码储存在main方法中，让主线程运行。所以，要想新建一个线程运行一段程序，该新线程就得有个空间储存新线程的代码，该空间就是run。为什么要新建空间呢？很简单，如果跟主线程一样也放在main方法中，那不就成了主线程中的代码，由主线程来完成运行了吗。



线程创建好了之后要由start函数来开启，如果选择调用run方法，就变成了主线程创立了一个对象，并调用了对象中的run方法，这一系列操作就将由主线程完成，那么就变成了主线程先运行run方法，等run方法全部运行完成以后，再接下来执行下面的循环。

**P133 多线程（线程练习）**

//

**P134 多线程（线程运行状态）**

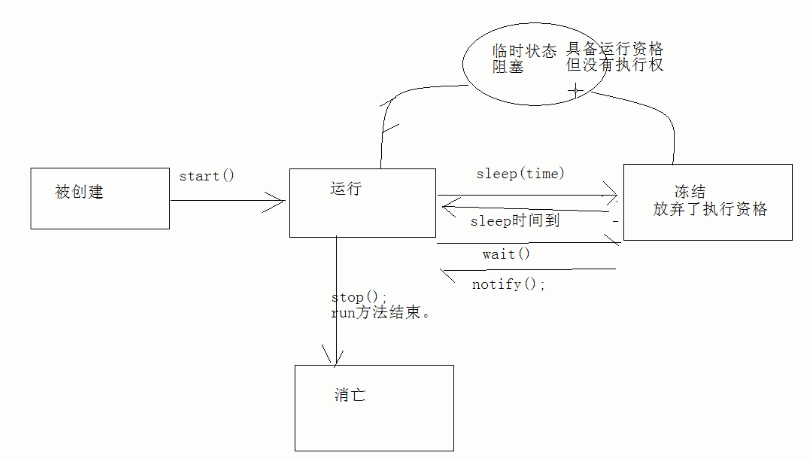
从冻结状态恢复后，该线程可能直接进入运行，也可能先进入临时状态。

运行状态：既有运行资格，又有执行权

临时状态：有运行资格，但没有执行权

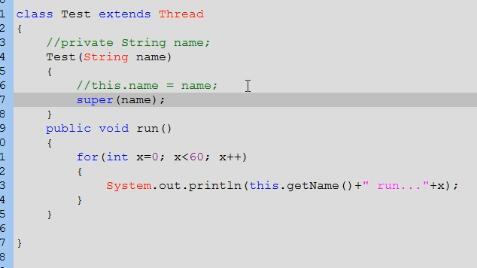
冻结状态：放弃了执行资格

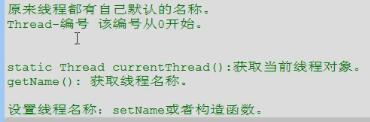
--wait和sleep的区别：sleep是冻结你预先设定好的时间，然后自动恢复；wait是无限制的冻结下去直至你手动唤醒。



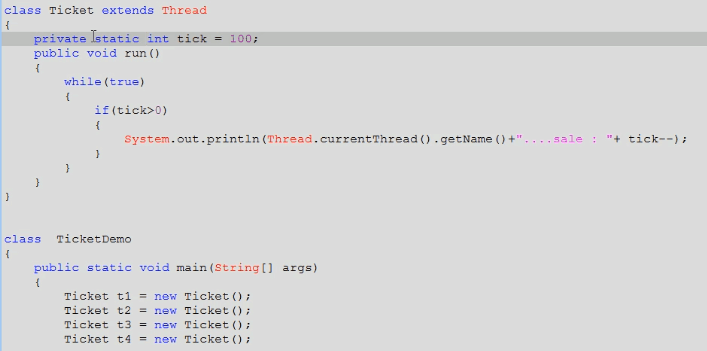
**P135 多线程（获取线程对象以及名称）**

想自定义线程名称时。绿色部份的方法传递不进去，要么用父类的构造函数，由父类定义好的一系列功能传递进去（推荐），要么就this.setName(name)传进去。

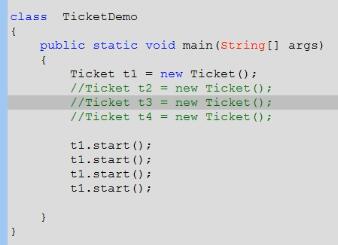




**P136 多线程（售票的例子）**



如果不想把tick定义为静态，能否创建4个相同的线程？



不行，程序会报错：线程状态出问题了

因为通过第一个start，你已经让线程从创建状态进入了运行状态，后面的start就没意义了

**P137 多线程（创建线程-实现Runable接口）**

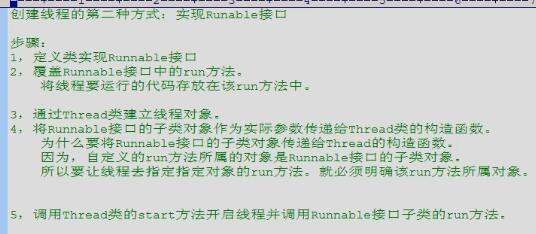
前面的例子，我们都是使用创建线程的第一种方法

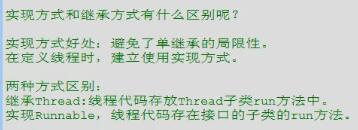




另一个方法就是使用Runable接口

Runable接口应该由那些打算通过某一线程执行其实例的类来实现。类必须定义一个称为run的无参数方法。

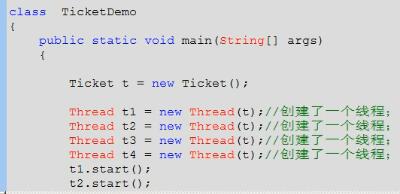




而且，实现方式的话，变量不需要static也能全局共享。如下例，因为你只建立了一个Ticket对象。

再次强调下接口子类的定义

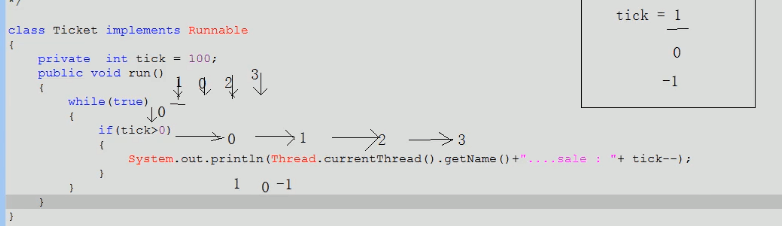




该类该怎么定义怎么定义，该怎么继承怎么继承。但该类设计完成后，加上Runable接口，并额外加一个run方法，把想要让线程运行的程序放进去即可。

Thread类的构造函数接受Runable类型的接口。这也是多态的体现。

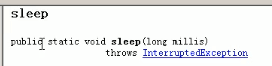
**P138 多线程（多线程的安全问题）**



举一个特殊例子来说明安全问题

比如要卖100张tick，当tick=1时，线程0进入if判断并通过，此时cpu切换到线程1；线程1进入if判断并通过，此时cpu切换到线程2；线程2进入if判断并通过，此时cpu切换到线程3。此时切换回线程0，输出tick=1并且tick值减1；此时切换回线程1，输出tick=0并且tick值减1；此时切换回线程2，输出tick=-1并且tick值减1；此时切换回线程3，输出tick=-2并且tick值减1。

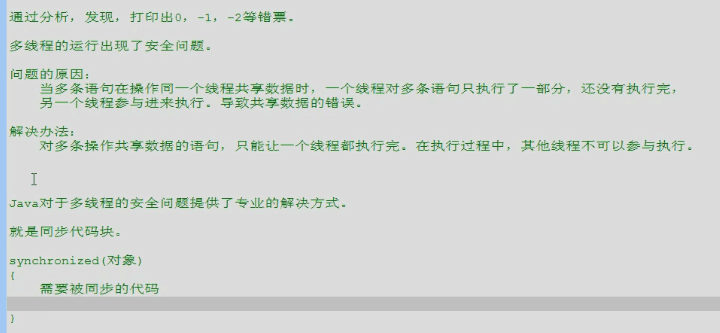
我们调用sleep方法来实现这个安全问题



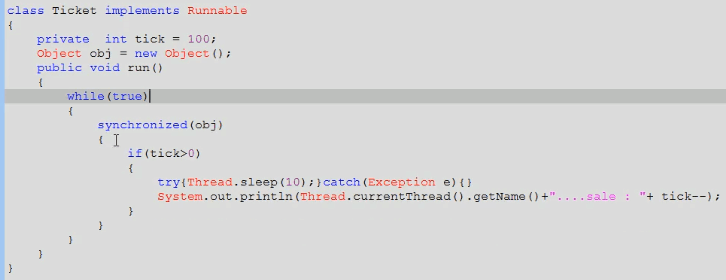
可以看到，调用sleep方法会抛出一个异常。

一般异常的解决有两种：继续抛出（因为我们定义的类实际上是Runable接口的子类，接口并没有处理异常的方法，所以该解决方法不可行）或者就地处理

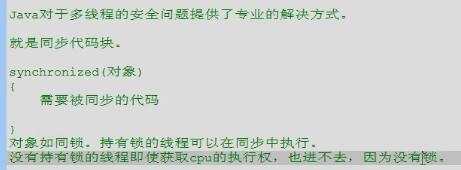




解决方法

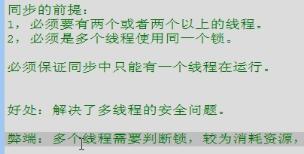


**P139 多线程（多线程同步代码块）**

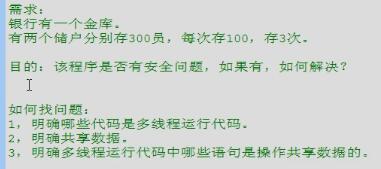


这么理解：这个对象相当于0/1标识符.当线程0读到synchronize(对象)时，先判断对象的状态，当对象为1时，线程0进入代码下一行，同时将对象置为0.这时如果cpu切换到其他线程，其他线程看到对象值为0，只能在外面等着。过了一会线程0重新那会执行权，执行完所有的同步代码后离开，并且将对象值重新置为1。

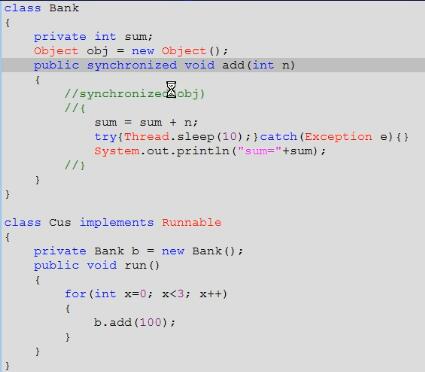
原理相当于很多人上同一个公共厕所。



**P140 多线程（多线程-同步函数）**



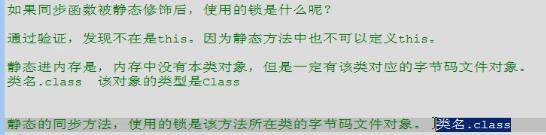
问题：有可能一个客户存完100后，打印之前切换到其他线程，其他线程也存入100.这时打印就会是200，200而不是100，200



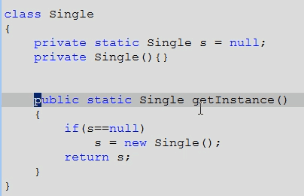
**P141 多线程（多线程-同步函数的锁是this）**

同步函数的锁是this

**P142 多线程（多线程-静态同步函数的锁是Class对象）**

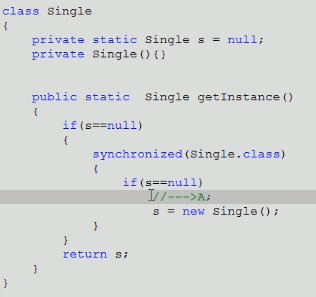


**P143 多线程（多线程-单例设计模式-懒汉式）**



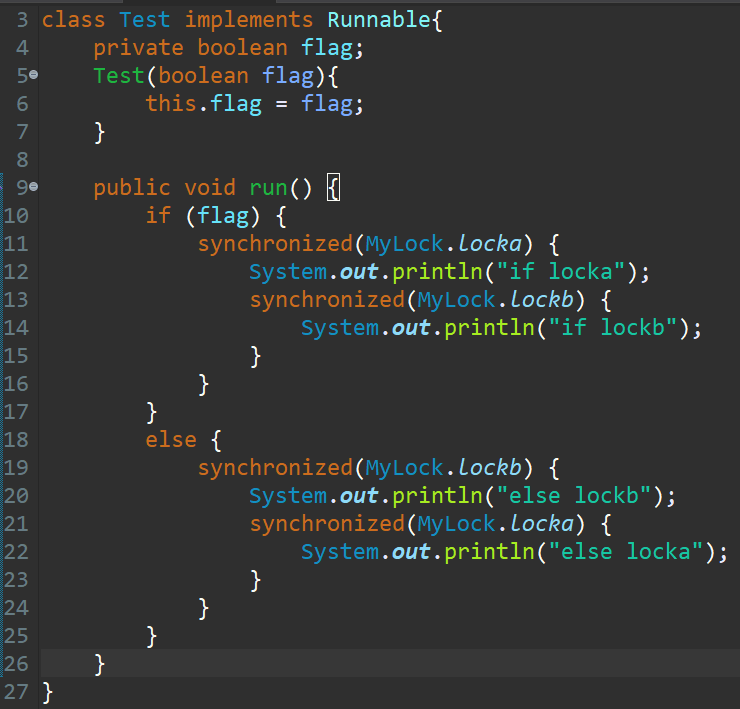
有多条语句在操作共享数据s，就可能new多个对象，造成安全隐患

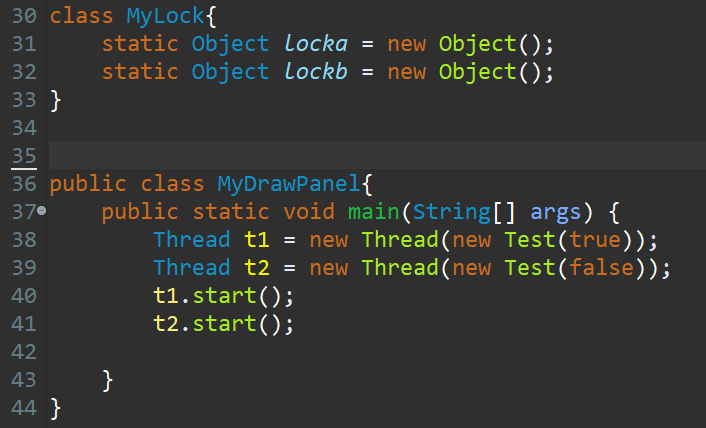
改进：



**P144 多线程（多线程-死锁）**

死锁的例子



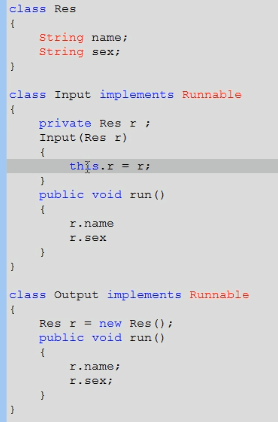


**P145 多线程（线程间通信-示例代码）**

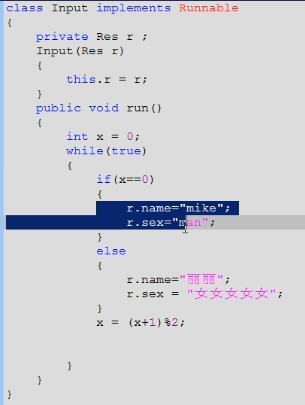


需求：

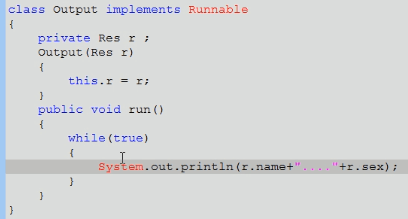
有一堆人物信息，包括姓名和性别，一个线程在存，一个线程在取。

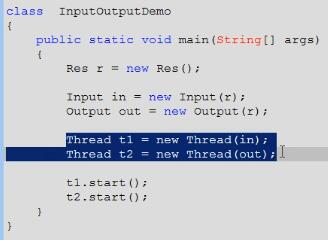


存入



取出

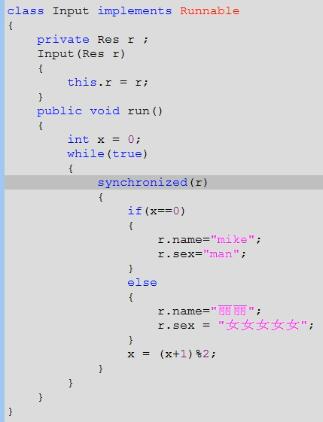


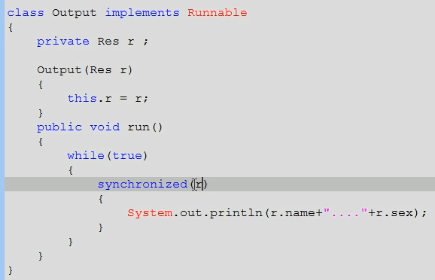


这时会出现一个问题：存入和取出不同步，即取出的姓名和性别不一定对应。

**P146 多线程（线程间通信-解决安全问题）**

在上一 P中问题的解决





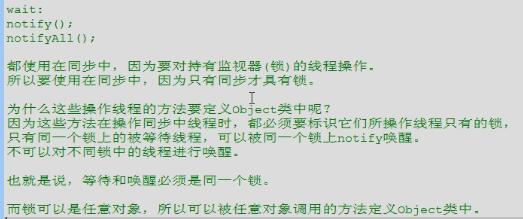
这样，就给Input和Output上了同一把锁r。其实只要是同一把锁就可以，现成的同一把锁还有：Input.class;Output.class;Res.class;InputOutputDemo.class;

**P147 多线程（线程间通信-等待唤醒机制）**

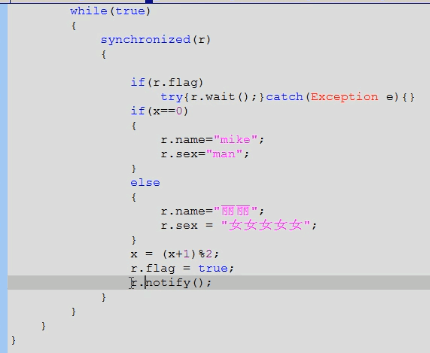
虽然上一P通过同步解决了姓名和性别不匹配的问题，但是上面程序还存在着一个问题：轮到Input执行时，疯狂往里存，丽丽mike不停互相覆盖，但是没往外取过；轮到Output执行时，疯狂对着同一个姓名性别重复输出。

那么，能不能设计成这样：Input每存入一个，Output就读取一个？

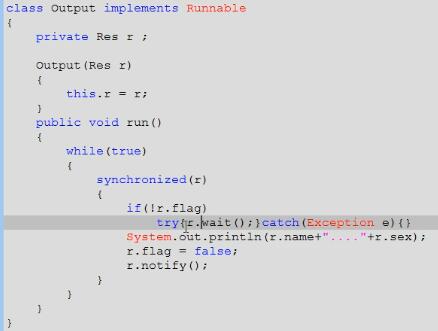
解决方法：等待唤醒机制：



Input



Output



怎么理解一个线程在唤醒另一个线程后，还能执行该线程使其进入冻结？

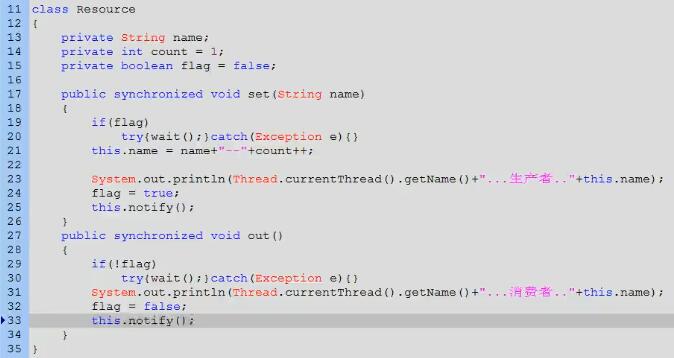
并不是一定要让一个程序连续执行两次才能唤起另一个线程并冻结该线程。应该这么想：冻结唤醒机制目的就是为了防止一个线程运行多次而另一个线程没运行过。这样一来，比如Output运行一次，唤醒Input，此时切换到Input线程，就算Output还未被冻结，目的也达到了：Output只执行了一次就切换到了另一线程。但是当Output运行一次，唤醒Input，此时仍然在Output线程，那么Output线程就会被冻结，强制使目的也达成了。

**P148 多线程（线程间通信-代码优化）**

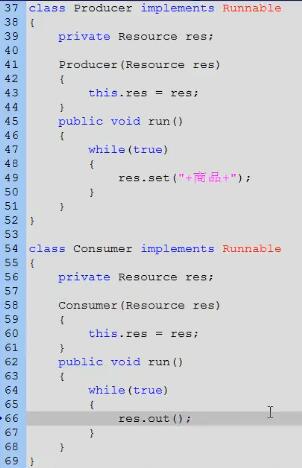
//

**P149 多线程（线程间通信-生产者消费者）**

复杂化上面的例子：有两个线程负责生产，有两个线程负责消费



这样set和out只能执行一个，在一个没有完全执行完之前，另一个无法执行。



当运行以上两段程序时，会出现一个安全问题：生产者生产了两个，消费者只消费了一个；或者是生产者生产了一个，消费者消费了两个。为什么会出现这种现象呢？

比如有4个线程，t1，t2生产；t3，t4消费（以下例子并不是程序就是这么走，而是由于cpu分配资源的随机性，有出现以下情况的可能性。而安全问题一出现就是很严重的问题，不能因为发生概率小而忽视）

首先t1抢到资源，生产了一个并将标志置为真，然后t1继续占用资源，进入20行，t1冻结。此时t2抢到资源，进入20行，t2冻结。此时t3抢到资源，消费了一个并将标志置为假，同时唤醒了t1（冻结的线程储存在线程池中，先冻结的先被唤醒），然后t3继续占用资源，进入30行，t3冻结。此时t4抢到资源，进入30行，t4冻结。//以上从输出来看，生产了一个，消费了一个，还算正常。接下来问题来了。  
此时t1占用资源，从20行醒来进入21行，生产了一个，同时唤醒了t2。然后t2抢到资源，从20行醒来进入21行，又生产了一个。//此时我们看到，生产者连续生产了两个，出现了安全问题。

所以以上安全问题出现的原因之一：线程被唤醒后直接进行了下面阶段的操作，而没有重新对flag的布尔值进行判断。

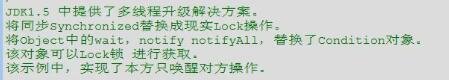
解决办法：将if判断设置为while判断，那么每一个被唤醒的线程，在获得执行权的时候都要重新对当前标志进行判断。但是这样一来又出现了一个新问题：容易所以的线程都进入冻结，程序卡死在那。

解决方法：将单个唤醒notify替换为全部唤醒notifyAll

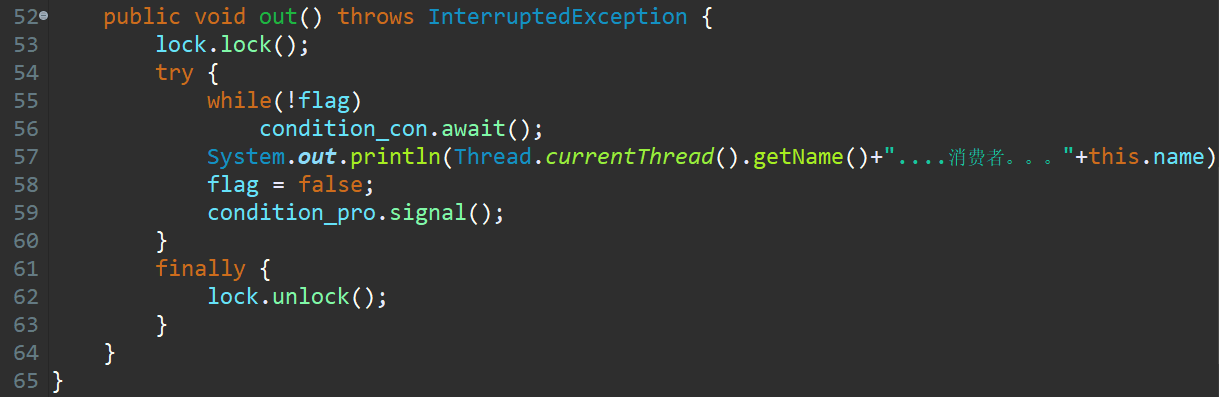


这就是一般多线程的标准格式，while判断，notifyAll唤醒全部

**P150 多线程（线程间通信-生产者消费者JDK5.0升级版）**







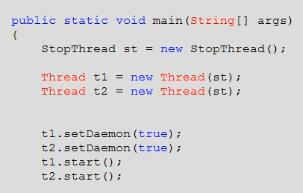
好处：能够指定冻结和唤醒的线程。

**P151 多线程（停止线程）**

既然notify和interrupt都能解除冻结的线程，那么有什么区别呢？

-interrupt在解除冻结的同时，还能抛出一个异常，那么我们就可以在捕获该异常的同时，将run里面的循环体结束循环。

**P152 多线程（守护线程）**



守护线程要在start之前就定义好。

守护线程有啥用？

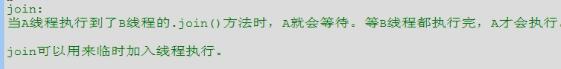
当非守护线程运行完后，守护线程也自动结束

**P153 多线程（join方法）**

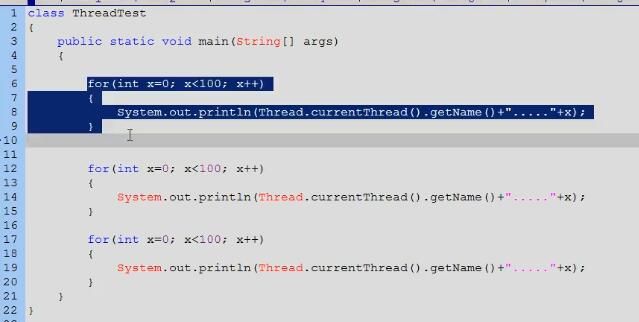


第23行的join方法

主线程碰到join就会冻结，一直到join的线程执行完毕才恢复

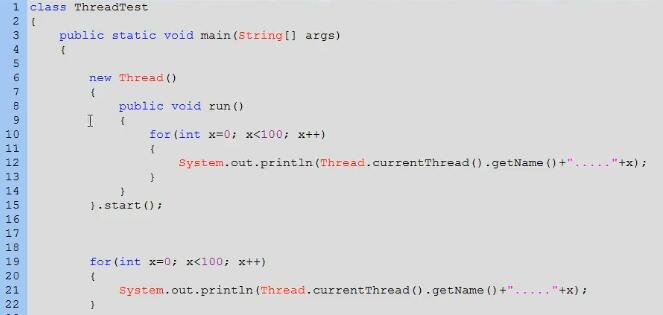


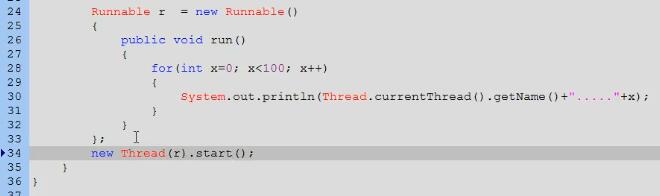
**P154 多线程（优先级&yield方法）**



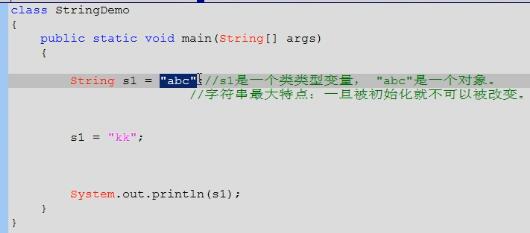
此时，3个循环体只能依次执行

改进：定义匿名内部类，就地建立线程的两种方法

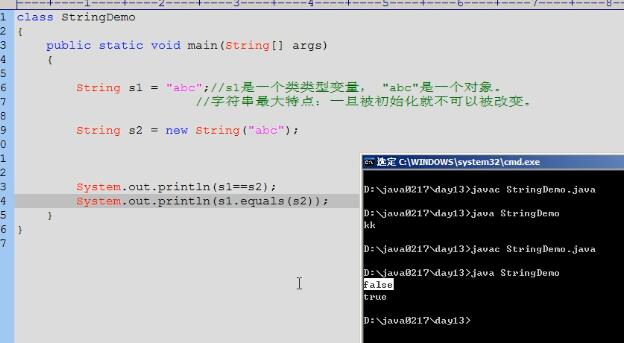




**P155 String（概述）**



虽然最后输出的结果是”kk”,但是“abc”没有改变过，变的是s1指向的对象。这里相当于new了一个对象“kk”并让s1指向它。



-s1==s2；比较的是两个地址值

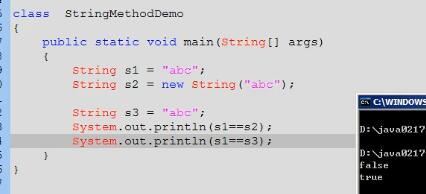
-equals讲道理比较的也是地址值，但是在String中对equals方法进行了复写，比较的是两者所指向的对象。

问：s1和s2有什么区别？

-s1有一个对象，s2有两个对象：new了一个，然后String()中又有一个对象。

了解即可：字符串在内存当中其实有一个常量池，存放的是”a”、”b”这些字符，池子里面是个数组，对字符进行组装。

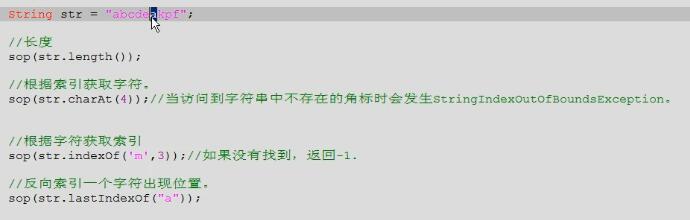
**P156 String（常见功能-获取和判断）**



为什么第二个是true？

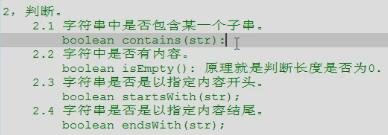
-s1在内存中创建了字符串对象”abc”，字符串是一种特殊的对象，存在于常量池中（常量池也在方法区中，除此之外还有方法代码，静态区）。所以s3在进行初始化时发现“abc“已经存在，就不会独立开辟空间了。

常见的字符串获取操作



indexOf()可以传入字符、字符串、ascii码

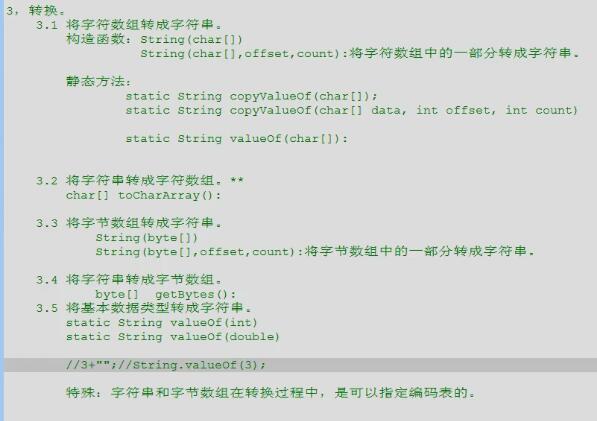
常见的字符串判断操作



以2.1为例

首先字符串是一个对象，有contains方法，所以直接 对象.contains()进行调用，向方法中传入一个String类型的子串，该方法就会返回一个boolean类型的值。

**P157 String（常见功能-转换）**



**P158 String（常见功能-切割和替换）**



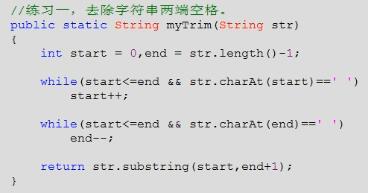
即s == s1

**P159 String（常见功能-比较和去除空格）**

//

**P160 String（字符串练习1）**

不借助trim方法（跟python很像，挺简单的）



**P161 String（字符串练习2）**

//

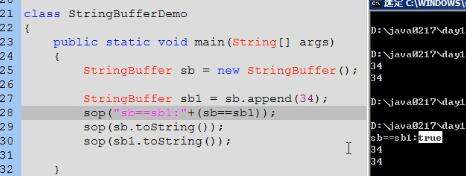
**P162 String（字符串练习3）**

//

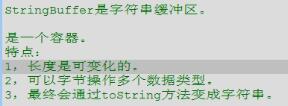
**P163 String（字符串练习4）**

//

**P164 StringBuffer（常见功能-添加）**



说明sb和sb1指向了同一个对象



**P165 StringBuffer（常见功能-删除和修改）**

//

**P166 StringBuilder**

单线程建议使用StringBuilder（线程不同步）

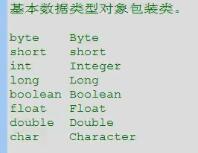
多线程建议使用StringBuffer（线程同步）

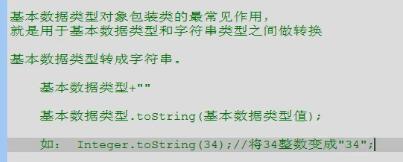
Java升级不外乎3个因素：

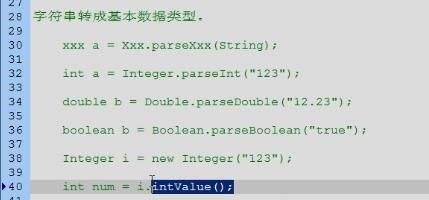
1. 提高效率
2. 简化书写
3. 提高安全性

StringBuilder的出现是原因1.

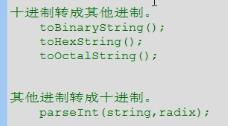
**P167 基本数据类型对象包装类**



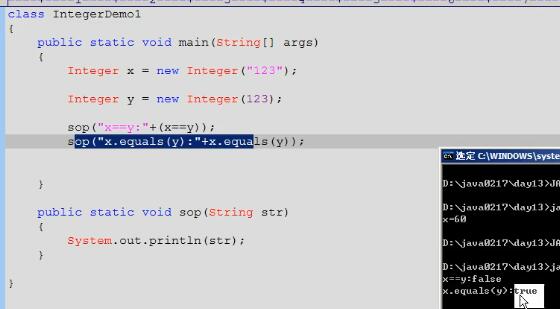




38和40行的效果等同于32行



**P168 基本数据类型对象包装类新特性**

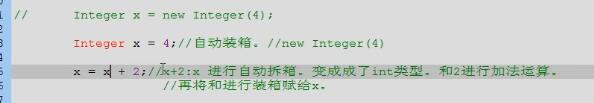


不要被什么int，byte之类的迷惑，把它当成普通的对象处理

“x==y”：xnew了一个对象，y又new了一个对象，两个对象的地址值当然不同。

“x.equals(y)”：Integer类重写了equals方法，比较的是对象的值，相等。

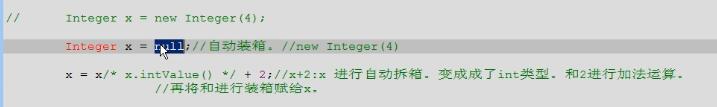
JDK1.5版本以后出现的第一个新特性



下面这行看上去跟 int x =4； 很相似，其实是等价于上面这行：new了一个Integer类型的变量x，又new了一个Integer对象并传入参数4，并将地址赋给x。是为了简化书写的优化

类类型变量必然指向对象

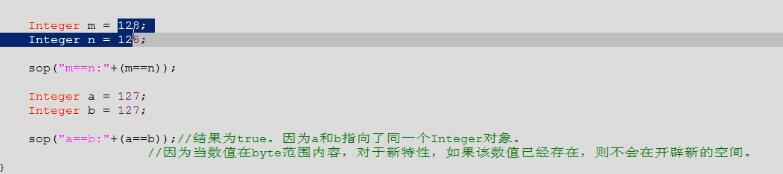
但是在使用自动拆箱和装箱时有个问题



-x毕竟是个类类型变量，可以为空。此时就会产生空指针异常。

所以在使用该方法直接运算时，注意要判断是否为空

JDK1.5版本以后出现的第二个新特性



第一个输出false，很好理解，因为m和n都是类类型变量，指向了不同的对象。

第二个输出true。这是新特性。