**一，sleep，wait，yield，join的区别**

①Sleep( )——Thread提供的静态方法

当前线程进入阻塞状态，**释放CPU资源**，但是**不释放锁**，一段时间后自动苏醒，沿着sleep( )继续向下执行

②wait( )——Objec提供的普通方法

**释放CPU资源**，**释放锁**

如果线程A调用了某个对象的wait( )，那么线程A会进入阻塞状态，进入到该对象的等待池中，直到有另一个线程调用了这个对象的notify( )或notifyAll( )才会将线程A唤醒

唤醒后的A，会不断尝试获取该对象的锁

——获取成功，线程A就沿着wait( )继续向下执行

——获取失败，线程A就滞留在wait( )处，不断尝试获取对象锁，直到获取为止，然后沿着wait( )继续向下执行

③yield( )——Thread提供的静态方法

**释放CPU资源**，当前线程进入就绪状态，后续仍会与其他线程争夺CPU资源

**yield( )不会释放锁**

**也就是说，如果线程A拿到某个对象锁后，在对象内部执行yield( ),会释放掉CPU资源，**

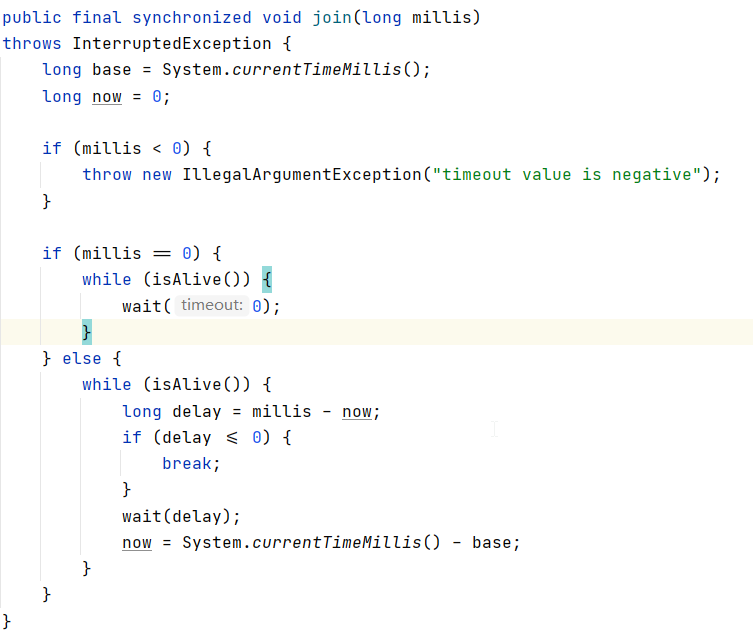
**进入就绪状态，但是不会释放掉对象锁，那么其他线程得不到对象锁，也就无法抢占CPU**

**资源，那么接下来仍然是线程A占据CPU资源**

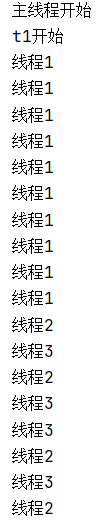
④join——Thread提供的普通方法

join的底层代码调用的是wait( )

我们来看看join的源码



我们举个最常见的例子，主线程(main)创建了三个子线程对象，t1，t2，t3

然后主线程开启子线程t1，然后主线程调用了t1的join，从源码中可以看出来，主线程调

用了t1的join，实际上是调用了t1对象的wait()，那么主线程就会进入t1对象的等待池中，

直到t1执行完毕才会唤醒主线程，那么主线程的代码就滞留在t1.join( )，直到t1执行完毕，

主线程才会被唤醒，才会继续向下执行，开启子线程t1和t2。

所以我们看到的输出结果是t1独自输出，t2，t3轮流输出

所以join()跟wait()是一样的，线程A在执行过程中，调用了线程对象B的join( )方法，那么

线程A就会**释放掉CPU资源**，**释放掉锁**，进入线程对象B的等待池中，直到线程对象B

执行完毕才会被唤醒，并尝试争夺锁，抢到锁后**，从join()的位置继续向下执行**

**二，wait()和notify()详解**

**1）基本概念**

①wait()，notify()，notifyAll()由Object提供

②等待池：每个对象都有一个等待池

③wait()和notify()必须放在同步方法or同步代码块中

**2）方法解析**

**①wait( )**

如果一个线程A调用了某个对象B的wait( )方法

（常见情况是，某个线程在某个对象的方法内部调用了wait( ) ）

那么该线程会释放掉CPU资源，释放掉对象锁，进入到该对象的等待池中

**②notify( )**

线程A进入对象B的等待池后，会一直沉睡，直到有另一个线程调用了这个对象B的notify( )，

（常见情况是，某个线程在某个对象的方法内部调用了notify( ) ）

对象B的等待池中的一个线程才会被唤醒，被唤醒的线程会尝试获取该对象的锁

——获取成功，从wait()继续向下执行

——获取失败，滞留在wait()处，不断尝试获取，直到获取成功，从wait()继续向下执行

**③notifyAll( )**

同上，不过这里唤醒的是等待池中的所有线程

**三，生产者和消费者模型**

核心原理：

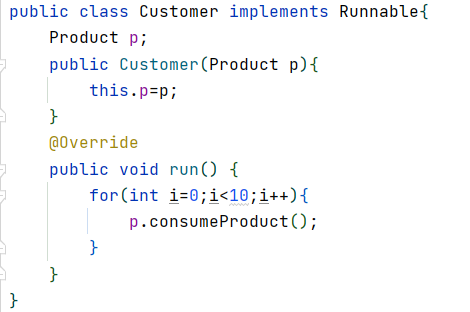
①生产者和消费者共享资源Product对象

②如果生产者调用Product对象的productProduct( )方法时，产品还没被消费者消费掉，那么生产者进入Product对象的等待池中中（wait( ) ）

③如果消费者调用Product对象的consumeProduct()方法时，产品如果还没有被生产，那么消费者进入Product对象的等待池中中（wait( ) ）

④生产者成功生产一个产品后，会调用Product对象的notify( )，从等待池中唤醒消费者

⑤消费者成功消费一个产品后，会调用Product对象的notify( )，从等待池中唤醒生产者





**四，Lock提供多个等待池**

**注意，一个对象只有一个等待池**

上述的生产者-消费者模型中，我们可以看到，我们只用了一个生产者和一个消费者，

生产者和消费者使用同一个对象的等待池

生产者执行完时，会唤醒等待池里的消费者

消费者执行完时，会唤醒等待池里的生产者，那么这就产生了一个问题，

当有多个生产者和多个消费者时，一个生产者执行完时，有可能唤醒等待池里的生

产者，而不是消费者

所以，我们需要有多个等待池，一个生产者用，一个消费者用

**一个Lock可以创建多个Condition，一个Condition就是一个等待池**

**所以一个Lock可以拥有多个等待池**

**Object中提供的wait(),notify(),notifyAll()与synchronized捆绑使用**

**Condition与Lock捆绑使用，也就是说如果你想使用Condition提供的方法，那么必须在**

**lock.lock()和lock.unlock()之间才可以使用**

**Condition提供的方法**

**①await()**

对应Object的wait()，如果线程调用了某个Condition对象的await()，那么该线程会释放

CPU资源和Lock锁，进入到该Condition的等待池中

**②signal()**

对应Object的notify()，如果线程调用了某个Condition对象的notify()，那么该Condition

对象的等待池中的一个线程会被唤醒

被唤醒的线程不断尝试取得Lock锁，取得后，沿着await()向下执行

**③signalAll()**

**采用Lock的代码如下，只修改了Product，同时，Test中有2个生产者，2个消费者**

