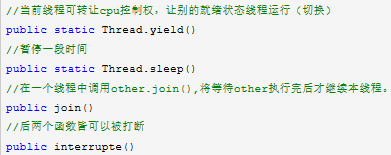
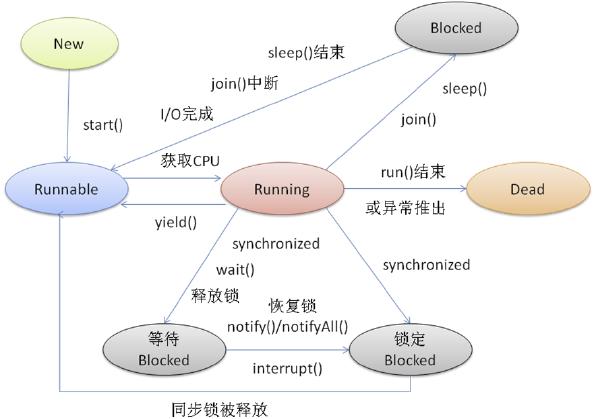
## 线程的状态



线程在Running的过程中可能会遇到阻塞(Blocked)情况

1. 调用join()和sleep()方法，sleep()时间结束或被打断，join()中断,IO完成都会回到Runnable状态，等待JVM的调度。
2. 调用wait()，使该线程处于等待池(wait blocked pool),直到notify()/notifyAll()，线程被唤醒被放到锁定池(lock blocked pool )，释放同步锁使线程回到可运行状态（Runnable）
3. 对Running状态的线程加同步锁(Synchronized)使其进入(lock blocked pool ),同步锁被释放进入可运行状态(Runnable)。

此外，在runnable状态的线程是处于被调度的线程，此时的调度顺序是不一定的。Thread类中的yield方法可以让一个running状态的线程转入runnable。

**关于中断**：它并不像stop方法那样会中断一个正在运行的线程。线程会不时地检测中断标识位，以判断线程是否应该被中断（中断标识值是否为true）。只会影响到wait状态、sleep状态和join状态。被打断的线程会抛出InterruptedException。  
Thread.interrupted()检查当前线程是否发生中断，返回boolean  
synchronized在获锁的过程中是不能被中断的。

中断是一个状态！interrupt()方法只是将这个状态置为true而已。所以说正常运行的程序不去检测状态，就不会终止，而wait等阻塞方法会去检查并抛出异常。如果在正常运行的程序中添加while(!Thread.interrupted()) ，则同样可以在中断后离开代码体

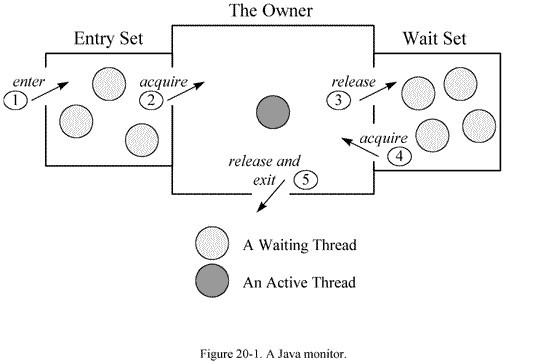
## 对象同步synchronized

, wait, notify 是任何对象都具有的同步工具。

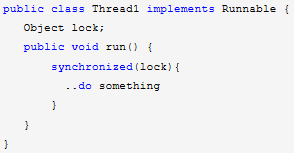
他们是应用于同步问题的人工线程调度工具。讲其本质，首先就要明确monitor的概念，Java中的每个对象都有一个监视器，来监测并发代码的重入。在非多线程编码时该监视器不发挥作用，反之如果在synchronized 范围内，监视器发挥作用。

wait/notify必须存在于synchronized块中。并且，这三个关键字针对的是同一个监视器（某对象的监视器）。这意味着wait之后，其他线程可以进入同步块执行。

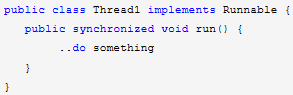
当某代码并不持有监视器的使用权时（如图中5的状态，即脱离同步块）去wait或notify，会抛出java.lang.IllegalMonitorStateException。也包括在synchronized块中去调用另一个对象的wait/notify，因为不同对象的监视器不同，同样会抛出此异常。



在多线程环境下，synchronized块中的方法获取了lock实例的monitor，如果实例相同，那么只有一个线程能执行该块内容



直接用于方法： 相当于上面代码中用lock来锁定的效果，实际获取的是Thread1类的monitor。更进一步，如果修饰的是static方法，则锁定该类所有实例

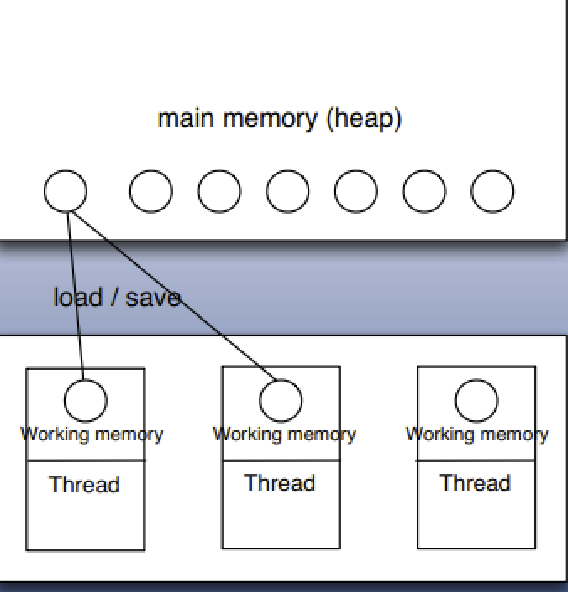


synchronized, wait, notify结合:典型场景生产者消费者问题

## volatile

多线程的内存模型：main memory（主存）、working memory（线程栈），在处理数据时，线程会把值从主存load到本地栈，完成操作后再save回去(volatile关键词的作用：每次针对该变量的操作都激发一次load and save)。



针对多线程使用的变量如果不是volatile或者final修饰的，很有可能产生不可预知的结果（另一个线程修改了这个值，但是之后在某线程看到的是修改之前的值）。其实道理上讲同一实例的同一属性本身只有一个副本。但是多线程是会缓存值的，本质上，volatile就是不去缓存，直接取值。在线程安全的情况下加volatile会牺牲性能。

## 基本线程类

基本线程类指的是Thread类，Runnable接口，Callable接口

### Thread 类

实现了Runnable接口，启动一个线程的方法：

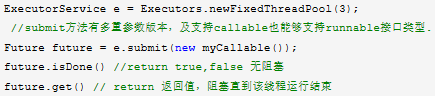


### Runnable

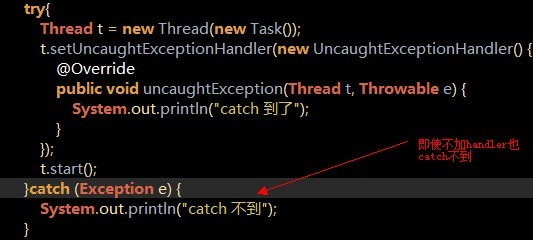
与Thread类似

### Callable

future模式：并发模式的一种，可以有两种形式，即无阻塞和阻塞，分别是isDone和get。其中Future对象用来存放该线程的返回值以及状态



## 如何获取线程中的异常



不能用try,catch来获取线程中的异常

## 多线程包:java.util.concurrent

### ThreadLocal

当使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。常用于用户登录控制，如记录session信息。

实现：每个Thread都持有一个TreadLocalMap类型的变量（该类是一个轻量级的Map，功能与map一样，区别是桶里放的是entry而不是entry的链表。功能还是一个map。）以本身为key，以目标为value。  
主要方法是get()和set(T a)，set之后在map里维护一个threadLocal -> a，get时将a返回。ThreadLocal是一个特殊的容器。

### 原子类（AtomicInteger、AtomicBoolean……）

如果使用atomic wrapper class如atomicInteger，或者使用自己保证原子的操作，则等同于synchronized

### Lock类

**ReentrantLock**　  
可重入的意义在于持有锁的线程可以继续持有，并且要释放对等的次数后才真正释放该锁。  
1.先new一个实例

static ReentrantLock r=new ReentrantLock();

2.加锁

r.lock()或r.lockInterruptibly();

后者可被打断。当a线程lock后，b线程阻塞，此时如果是lockInterruptibly，那么在调用b.interrupt()之后，b线程退出阻塞，并放弃对资源的争抢，进入catch块。（如果使用后者，必须throw interruptable exception 或catch）

3.释放锁

r.unlock()

必须做！要放在finally里面。以防止异常跳出了正常流程，导致灾难。这里补充一个小知识点，finally是可以信任的：经过测试，哪怕是发生了OutofMemoryError，finally块中的语句执行也能够得到保证。

**ReentrantReadWriteLock**

可重入读写锁（读写锁的一个实现）

　ReentrantReadWriteLock lock = new ReentrantReadWriteLock()

　　ReadLock r = lock.readLock();

　　WriteLock w = lock.writeLock();

两者都有lock,unlock方法。写写，写读互斥；读读不互斥。可以实现并发读的高效线程安全代码

### 容器类

**BlockingQueue**  
阻塞队列。该类是java.util.concurrent包下的重要类，通过对Queue的学习可以得知，这个queue是单向队列，可以在队列头添加元素和在队尾删除或取出元素。类似于一个管　　道，特别适用于先进先出策略的一些应用场景。普通的queue接口主要实现有PriorityQueue（优先队列）

除了传统的queue功能（表格左边的两列）之外，还提供了阻塞接口put和take，带超时功能的阻塞接口offer和poll。put会在队列满的时候阻塞，直到有空间时被唤醒；take在队　列空的时候阻塞，直到有东西拿的时候才被唤醒。用于生产者-消费者模型尤其好用，堪称神器。

常见的阻塞队列有：

ArrayListBlockingQueue

LinkedListBlockingQueue

DelayQueue

SynchronousQueue

**ConcurrentHashMap**  
高效的线程安全哈希map。请对比hashTable , concurrentHashMap, HashMap

### 管理类

**ThreadPoolExecutor**

ExecutorService内部是通过ThreadPoolExecutor实现的，掌握该类有助于理解线程池的管理，本质上，他们都是ThreadPoolExecutor类的各种实现版本

