

Java SE程序设计 北京圣思园科技有限公司

主讲人 张龙

All Rights Reserved



多线程 (Multi-Thread)

线程概念

线程的生命周期

线程的实现

线程的优先级

多线程的同步

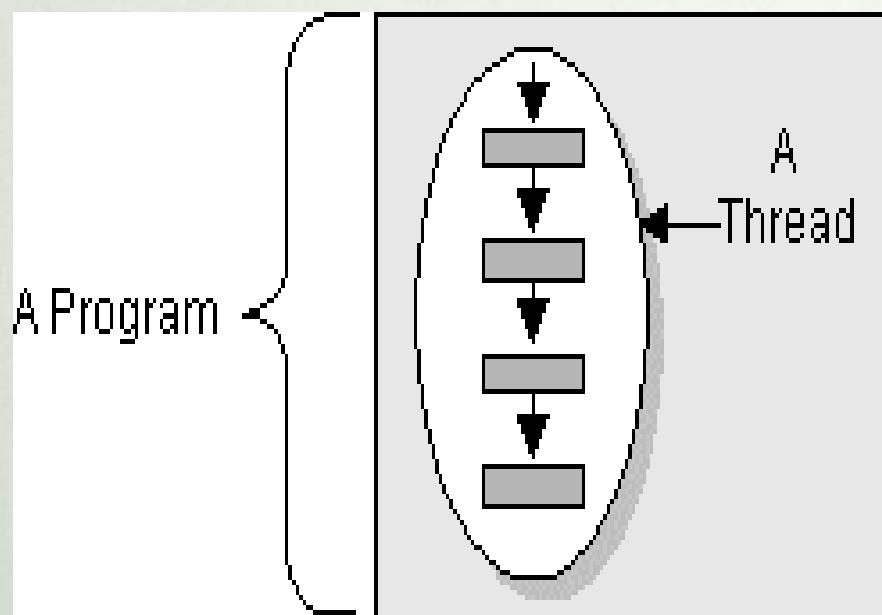
线程组



线程概念

1. 什么是线程：

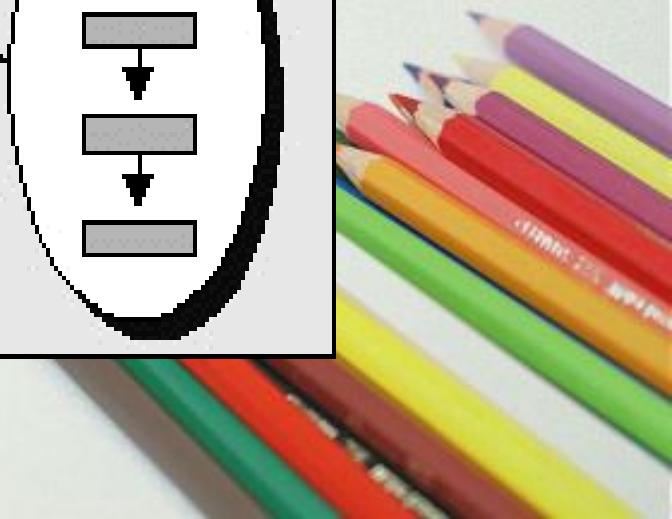
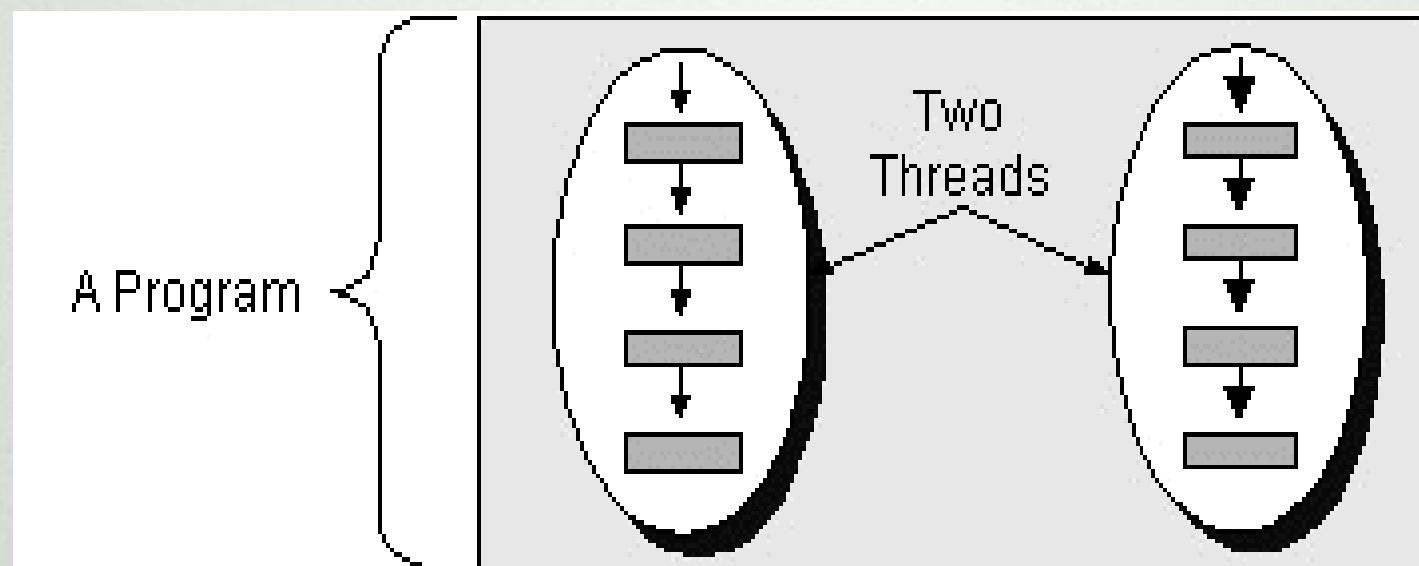
线程就是程序中单独顺序的流控制。线程本身不能运行，它只能用于程序中。



线程概念

2. 什么是多线程:

多线程则指的是在单个程序中可以同时运行多个不同的线程执行不同的任务.



线程概念

说明：

线程是程序内的顺序控制流，只能使用分配给程序的资源和环境。



多线程编程的目的

- 多线程编程的目的，就是"**最大限度地利用CPU资源**"，当某一线程的处理不需要占用CPU而只和I/O等资源打交道时，让需要占用CPU资源的其它线程有机会获得CPU资源。从根本上说，这就是多线程编程的最终目的。



进程



单线程

- 当程序启动运行时，就自动产生一个线程，主方法**main**就在这个**主线程**上运行



多线程

- 一个进程可以包含一个或多个线程
- 一个程序实现多个代码同时交替运行就需要产生多个线程
- CPU随机的抽出时间，让我们的程序一会做这件事情，一会做另外一件事情



多线程

- 同其他大多数编程语言不同，Java内置支持多线程编程（**multithreaded programming**）。多线程程序包含两条或两条以上并发运行的部分，把程序中每个这样的部分都叫作一个线程（**thread**）。每个线程都有独立的执行路径，因此多线程是多任务处理的一种特殊形式。
- 多任务处理被所有的现代操作系统所支持。然而，多任务处理有两种截然不同的类型：基于进程的和基于线程的。

多线程

1. 基于进程的多任务处理是更熟悉的形式。**进程 (process)** 本质上是一个执行的程序。因此基于进程的多任务处理的特点是允许你的计算机同时运行两个或更多的程序。举例来说，基于进程的多任务处理使你在运用文本编辑器的时候可以同时运行 Java 编译器。在基于进程的多任务处理中，程序是调度程序所分派的最小代码单位。
2. 而在基于线程 (**thread-based**) 的多任务处理环境中，**线程是最小的执行单位。这意味着一个程序可以同时执行两个或者多个任务的功能。** 例如，一个文本编辑器可以在打印的同时格式化文本。



线程与进程的区别

- 多个进程的内部数据和状态都是完全独立的，而多线程是共享一块内存空间和一组系统资源，有可能互相影响。
- 线程本身的数据通常只有寄存器数据，以及一个程序执行时使用的堆栈，所以线程的切换比进程切换的负担要小。



Process vs Thread

- 多线程程序比多进程程序需要更少的管理费用。进程是重量级的任务，需要分配给它们独立的地址空间。进程间通信是昂贵和受限的。进程间的转换也是很需要花费的。另一方面，线程是轻量级的选手。它们共享相同的地址空间并且共同分享同一个进程。线程间通信是便宜的，线程间的转换也是低成本的。



Process vs Thread

- 多线程可帮助你编写出**CPU**最大利用率的高效程序，使得空闲时间保持最低。这对**Java**运行的交互式的网络互连环境是至关重要的，因为空闲时间是公共的。例如，网络的数据传输速率远低于计算机处理能力，而本地文件系统资源的读写速度也远低于**CPU**的处理能力。当然，用户输入也比计算机慢很多。在传统的单线程环境中，程序必须等待每一个这样的任务完成以后才能执行下一步—尽管**CPU**有很多空闲时间。多线程使你能够获得并充分利用这些空闲时间。



Java线程模型

- Java多线程的优点就在于取消了主循环/轮询机制。一个线程可以暂停而不影响程序的其他部分。例如，当一个线程从网络读取数据或等待用户输入时产生的空闲时间可以被利用到其他地方。多线程允许活的循环在每一帧间隙中沉睡一秒而不暂停整个系统。



多线程的优势

线程的实现

在Java中通过**run**方法为线程指明要完成的任务，有两种技术来为线程提供**run**方法。

1. 继承**Thread**类并重写**run**方法。
2. 通过定义实现**Runnable**接口的类进而实现**run**方法。



线程的实现

1. 继承**Thread**类并重载**run**方法。
 - **Thread**类：是专门用来创建线程和对线程进行操作的类。**Thread**中定义了许多方法对线程进行操作。
 - **Thread**类在缺省情况下**run**方法什么都不做。可以通过继承**Thread**类并重写**Thread**类的**run**方法实现用户线程。



线程的实现

1. 继承Thread类并重写run方法。

总体结构如下：

```
public class MyThread extends Thread {  
    public void run() {  
        ... ...  
    }  
}
```

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

见例题：[TwoThreadsTest.java](#)



线程的实现

2. 实现**Runnable**接口的类实现**run**方法。

通过建立一个实现了**Runnable**接口的类，并以它作为线程的目标对象来创建一个线程。

Runnable接口：定义了一个抽象方法**run()**。定义如下：

```
public interface java.lang.Runnable{  
    public abstract void run();  
}
```



线程的实现

2. 实现Runnable接口的类实现run方法。

创建的总体框架如下：

- ```
class MyRunner implements Runnable {
 public void run() {
 ...
 } }
```
  - ```
MyRunner r = new MyRunner();
```
 - ```
Thread t = new Thread(ThreadGroup group,
 Runnable target,
 String
 name);
```
- 例如： 

```
Thread t = new Thread(r, "aa");
```

见例题：[ThreadTester.java](#)/[ThreadTest.java](#)



# 线程的实现

## 总结：

1. 两种方法均需执行线程的**start**方法为线程分配必须的系统资源、调度线程运行并执行线程的**run**方法。
2. 在具体应用中，采用哪种方法来构造线程体要视情况而定。通常，当一个线程已继承了另一个类时，就应该用第二种方法来构造，即实现**Runnable**接口。
3. 线程的消亡不能通过调用一个**stop()**命令。而是让**run()**方法自然结束。



# 停止线程 推荐的方式

```
public class MyThread implements Runnable
{ private boolean flag=true;
 public void run ()
 { while (flag)
 {...}
 }
 public void stopRunning()
 { flag=false; }
}

public class ControlThread
{ private Runnable r=new MyThread();
 private Thread t=new Thread(r);

 public void startThread()
 { t.start(); }

 publi void stopThread()
 { r.stopRunning(); }
}
```



# 线程的生命周期

线程的生命周期：一个线程从创建到消亡的过程。

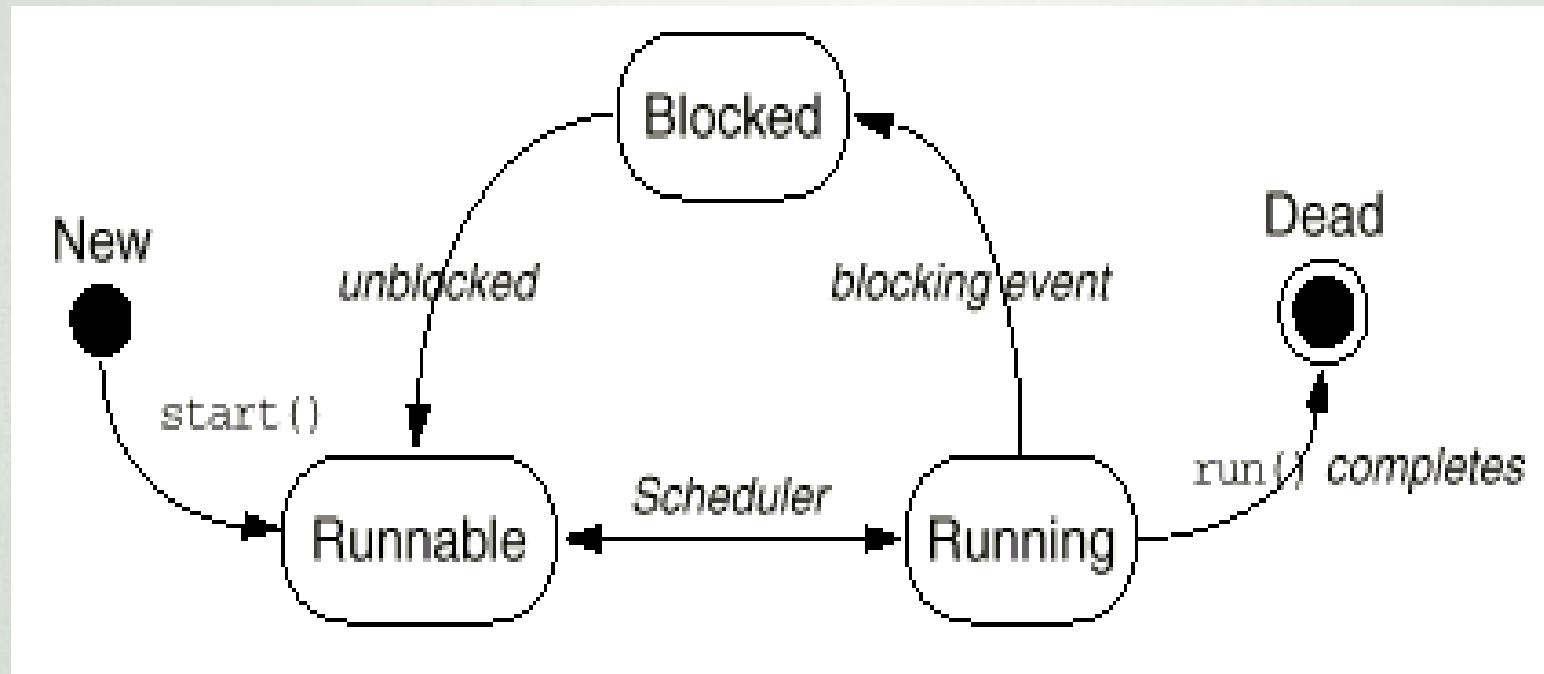
线程的生命周期可分为四个状态：

1. 创建状态
2. 可运行状态
3. 不可运行状态
4. 消亡状态



# 线程的生命周期

线程的状态转换图：



# 线程的生命周期

## 1. 创建状态

- 当用**new**操作符创建一个新的线程对象时，该线程处于创建状态。
- 处于创建状态的线程只是一个空的线程对象，系统不为它分配资源



# 线程的生命周期

## 2. 可运行状态

- 执行线程的**start()**方法将为线程分配必须的系统资源，安排其运行，并调用线程体—**run()**方法，这样就使得该线程处于可运行( **Runnable** )状态。
- 这一状态并不是运行中状态 ( **Running** )，因为线程也许实际上并未真正运行。



# 线程的生命周期

## 3. 不可运行状态

当发生下列事件时，处于运行状态的线程会转入到不可运行状态。

- 调用了**sleep ()** 方法；
- 线程调用**wait**方法等待特定条件的满足
- 线程输入/输出阻塞



# 线程的生命周期

返回可运行状态：

- 处于睡眠状态的线程在指定的时间过去后
- 如果线程在等待某一条件，另一个对象必须通过**notify()**或**notifyAll()**方法通知等待线程条件的改变
- 如果线程是因为输入/输出阻塞，等待输入/输出完成



# 线程的生命周期

## 4. 消亡状态

当线程的**run**方法执行结束后，该线程自然消亡。



# 线程的优先级

## 1. 线程的优先级及其设置

设置优先级是为了在多线程环境中便于系统对线程的调度，优先级高的线程将优先执行。

一个线程的优先级设置遵从以下原则：

- 线程创建时，子继承父的优先级
- 线程创建后，可通过调用**setPriority()**方法改变优先级。
- 线程的优先级是1-10之间的正整数。

1 - MIN\_PRIORITY,

10 – MAX\_PRIORITY

5- NORM\_PRIORITY



# 线程的优先级

## 2. 线程的调度策略

线程调度器选择优先级最高的线程运行。但是，如果发生以下情况，就会终止线程的运行。

- 线程体中调用了**yield()**方法，让出了对CPU的占用权
- 线程体中调用了**sleep()**方法，使线程进入睡眠状态
- 线程由于I/O操作而受阻塞
- 另一个更高优先级的线程出现。
- 在支持时间片的系统中，该线程的时间片用完。



# 线程的优先级

见例题：

RaceTest1.java

RaceTest2.java



# 线程的优先级

Thread的一些常用方法。

- 测试 threads:  
isAlive()
- Thread priority:  
t getPriority()  
t setPriority()
- threads 进入非执行状态  
Thread. sleep()  
Thread. yield()



# 多线程的同步

## 1. 为什么要引入同步机制

在多线程环境中，可能会有两个甚至更多的线程试图同时访问一个有限的资源。必须对这种潜在资源冲突进行预防。

解决方法：在线程使用一个资源时为其加锁即可。访问资源的第一个线程为其加上锁以后，其他线程便不能再使用那个资源，除非被解锁。



# 多线程的同步

## 2. 怎样实现同步

对于访问某个关键共享资源的所有方法，都必须  
把它们设为**synchronized**

例如：

```
synchronized void f() { /* ... */ }
synchronized void g() { /* ... */ }
```

如果想保护某些资源不被多个线程同时访问，可以强  
制通过**synchronized**方法访问那些资源。  
调用**synchronized**方法时，对象就会被锁定。



# 多线程的同步

```
public class MyStack {
 int idx = 0;
 char [] data = new char[6];

 public synchronized void push(char c) {
 data[idx] = c;
 idx++;
 }

 public synchronized char pop() {
 idx--;
 return data[idx];
 }
}
```



# 多线程的同步

## 2. 怎样实现同步

说明：

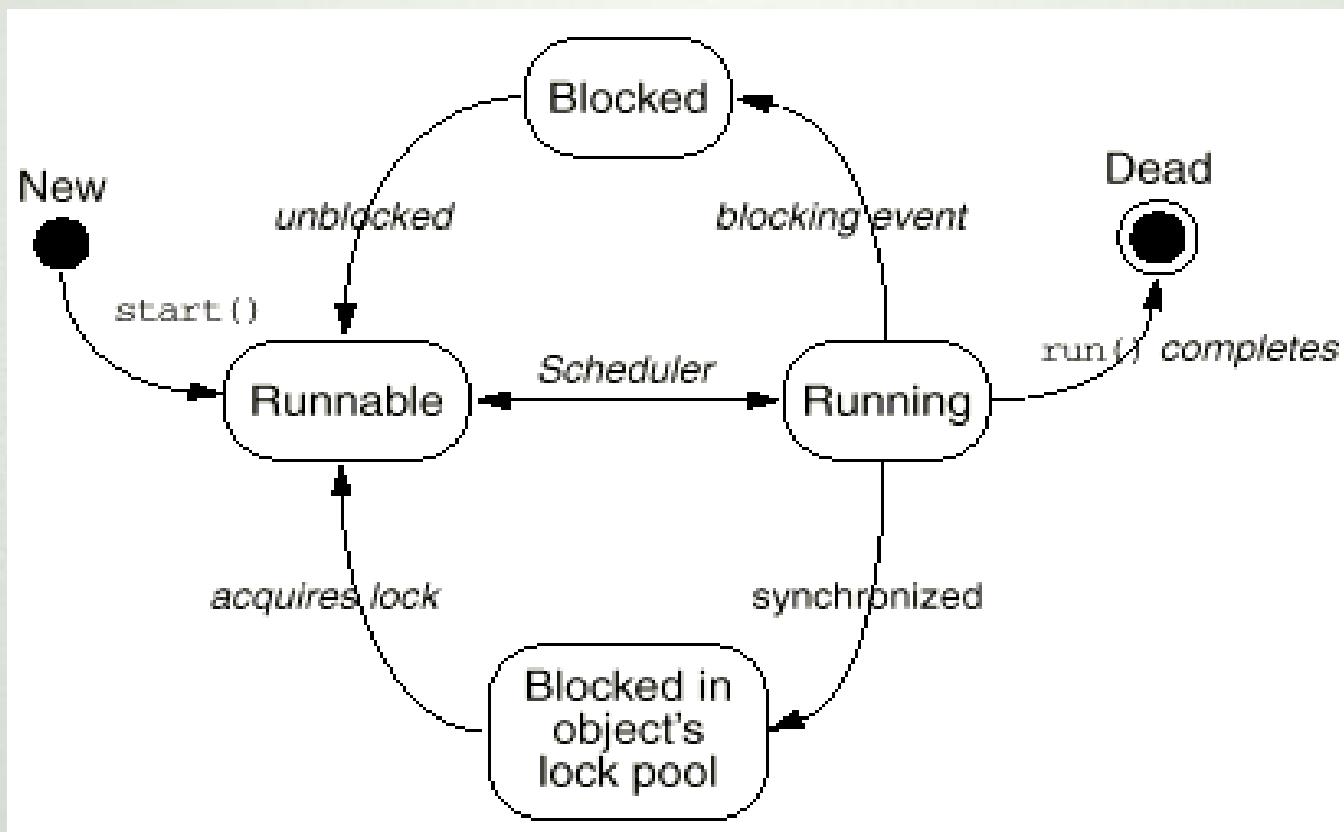
- 当**synchronized**方法执行完或发生异常时，会自动释放锁。
- 被**synchronized**保护的数据应该是私有（**private**）的。



# 多线程的同步

## 2. 怎样实现同步

同步的线程状态图：



# 多线程的同步

## 2. 怎样实现同步

线程间的相互作用：

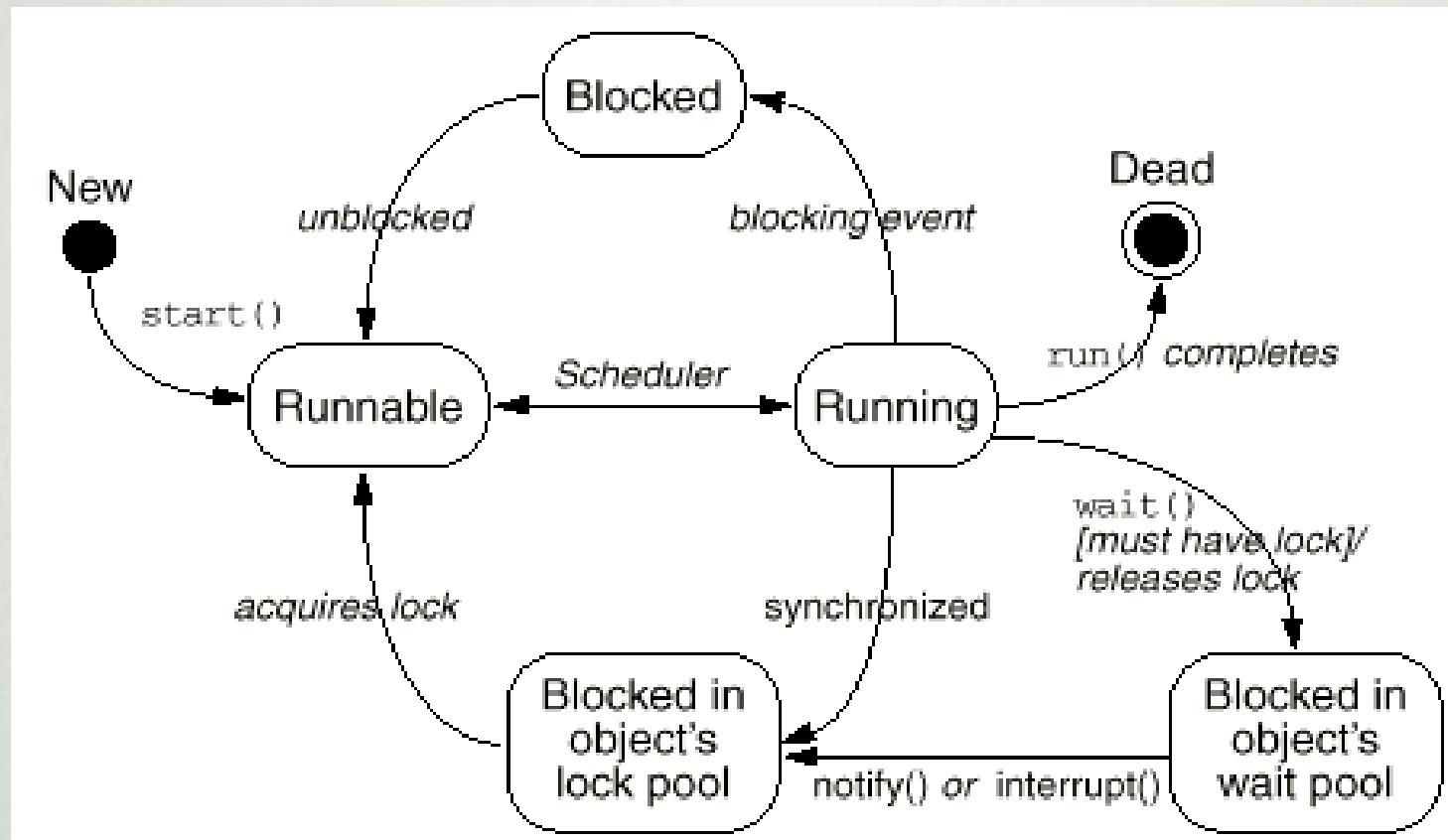
- wait and notify
- The pools:
  - Wait pool
  - Lock pool



# 多线程的同步

## 2. 怎样实现同步

具有**wait()**和**notify()**的线程状态图：



# 多线程的同步

## 2. 怎样实现同步

让我们看一个同步的综合例题：

生产者和消费者问题。

Producer.java

Consumer.java

CubbyHole.java

ProducerConsumerTest.java



# 线程组

线程组：

所有线程都隶属于一个线程组。那可以是一个默认线程组，亦可是一个创建线程时明确指定的组。

说明：

- 在创建之初，线程被限制到一个组里，而且不能改变到一个不同的组。
- 若创建多个线程而不指定一个组，它们就会与创建它的线程属于同一个组。

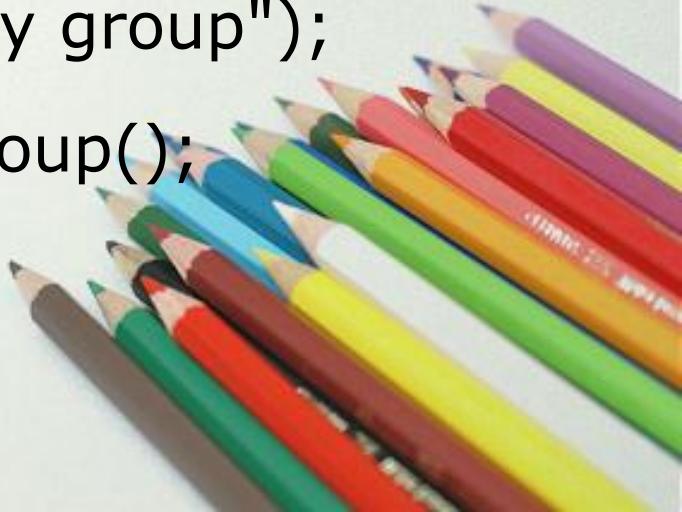


# 线程组

例子：

```
public Thread(ThreadGroup group, Runnable target)
public Thread(ThreadGroup group, String name)
public Thread(ThreadGroup group, Runnable target,
String name)
```

```
ThreadGroup myThreadGroup = new ThreadGroup(
 "My Group of Threads");
Thread myThread = new Thread(myThreadGroup,
 "a thread for my group");
theGroup = myThread.getThreadGroup();
```



# Timer和TimerTask

- Java 2的1.3版在java.util中增加了一个有趣又有用的功能部件：提供了提前安排将来某时间要执行任务的能力。支持这项功能的类是Timer和TimerTask。使用这些类可以创建一个工作于后台的线程，该线程等待一段指定的时间。当指定的时间到来时，与该线程相连的任务被执行。不同的选项允许安排一个任务重复执行，或安排一个任务在指定的时间运行。尽管永远都可能使用Thread类利用手工方法创建一个在指定的时间执行的任务，但是使用Timer和TimerTask却大大简化了这一过程。



# Timer和TimerTask

- Timer和TimerTask一起工作。Timer是一个用于安排一个将来执行的任务的类。被安排的任务必须是TimerTask的一个实例。因此，为了安排一个任务，首先应该创建一个TimerTask对象，然后使用Timer的一个实例安排执行它。



# Timer和TimerTask

- TimerTask实现了Runnable接口；因此它可以被用于创建一个执行线程。它的构造函数如下所示：
  - TimerTask( )
- TimerTask的run( )是一个抽象方法，这意味着它可以被覆盖。由Runnable接口定义的run( )方法包含了将被执行的程序代码。因此创建一个定时器任务的最简单的办法是扩展TimerTask和重写run( )



# Timer和TimerTask

- 一旦任务被创建，它将通过一个类型Timer的对象被安排执行。Timer的构造函数如下：
  - Timer( )
  - Timer(boolean DThread)
- 第一种形式创建一个以常规线程方式运行的Timer对象。第二种形式当DThread为true时，使用后台线程。只要剩下的程序继续运行，后台线程就会执行。



# Timer和TimerTask

- 一旦**Timer**被创建，将可以通过调用创建的**Timer**的**schedule( )**方法来安排任务。有几种**schedule( )**方法的形式，这些形式允许用各种办法来安排任务。



# Timer和TimerTask

- 参见程序

