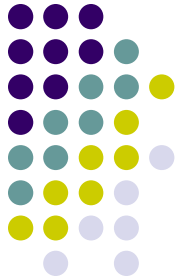
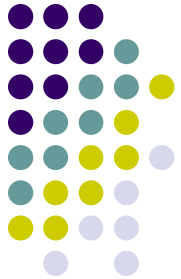
第七章 多线程程序设计



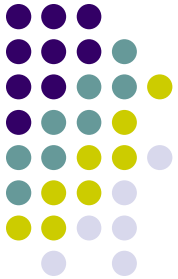
- §7.1多线程编程基础
- §7.2线程的生命周期
- §7.3线程的优先级

§7.1多线程编程基础



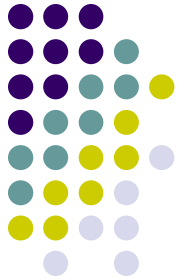
- 本节内容
  - **线程的概念**
  - **Thread类**
  - **Runnable接口**
  - **线程间的数据共享**
  - **多线程的同步控制**
  - **线程之间的通信**
  - **后台线程**

§7.1.1 线程的概念



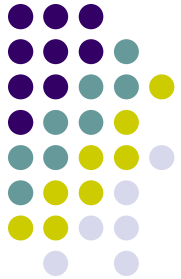
- 进程和线程的区别
- 进程
  - **一个独立程序的每一次运行称为一个进程，例如**
    - **用字处理软件编辑文稿时，同时打开mp3播放程序听音乐，这两个独立的程序在同时运行，称为两个进程**
  - **设置一个进程要占用相当一部分处理器时间和内存资源**
  - **大多数操作系统不允许进程访问其他进程的内存空间，进程间的通信很不方便，编程模型比较复杂。**

§7.1.1 线程的概念



- 线程
  - **一个程序中多段代码同时并发执行，称为多线程**
  - **通过多线程，一个进程表面上看同时可以执行一个以上的任务——并发**
  - **创建线程比创建进程开销要小得多，线程之间的协作和数据交换也比较容易**
  - **Java是第一个支持内置线程操作的主流编程语言**
  - **多数程序设计语言支持多线程要借助于操作系统“原语(primitives)”**

§7.1.2 Thread类



- Thread类
  - 在Java程序中创建多线程的方法之一是继承Thread类
  - 封装了Java程序中一个线程对象需要拥有的属性和方法
  - 从Thread类派生一个子类，并创建这个子类的对象，就可以产生一个新的线程。
  - 这个子类应该重写Thread类的run方法，在run方法中写入需要在新线程中执行的语句段。这个子类的对象需要调用start方法来启动，新线程将自动进入run方法。原线程将同时继续往下执行
  - Thread类直接继承了Object类，并实现了Runnable接口。它位于java.lang包中，因而程序开头不用import任何包就可直接使用。

§7.1.2 Thread类

- 在新线程中完成计算某个整数的阶乘

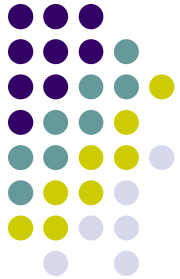
```
class FactorialThread extends Thread {
   private int num;
   public FactorialThread( int num ) { this.num=num; }
   public void run() {
        int i=num;
        int result=1;
        System.out.println("new thread started" );
        while(i>0) {
                result=result*i;
                i=i-1;
          }
       System.out.println("The factorial of "+num+" is "+result);
       System.out.println("new thread ends");
       }
}
```

```
public class MultiTreadEx1 {
  public static void main( String [] args ) {
        System.out.println("main thread starts");
        FactorialThread  thread = new FactorialThread(10);
        thread.start();
        System.out.println("main thread ends " );
  }
}
```

- 运行结果

```
main thread starts
main thread ends
new thread started
The factorial of 10 is 3628800
new thread ends
```

§7.1.2 Thread类



- 结果说明
  - **main线程已经执行完后，新线程才执行完**
  - **main函数调用thread.start()方法启动新线程后并不等待其run方法返回就继续运行，thread.run函数在一边独自运行，不影响原来的main函数的运行**
- 源程序修改
  - **如果启动新线程后希望主线程多持续一会再结束，可在start语句后加上让当前线程（这里当然是main）休息1毫秒的语句：**

```
try {  Thread.sleep(1); }
catch(Exception e){};
```

```
public class MultiTreadEx1 {
  public static void main( String [] args ) {
        System.out.println("main thread starts");
        FactorialThread  thread = new FactorialThread(10);
        thread.start();
        try {  Thread.sleep(1); }
        catch(Exception e){};
        System.out.println("main thread ends " );
  }
}
```

- 修改后运行结果

  main thread starts

  new thread stared

  The factorial of 10 is 3628800

  new thread ends

  main thread ends

- 运行结果说明
  - **新线程结束后main线程才结束**

7.1.2 Thread类(续)

——常用API函数

多线程编程基础

名称	说明
public Thread()	构造一个新的线程对象，默认名为Thread-n，n是从0开始递增的整数
public Thread(Runnable target)	构造一个新的线程对象，以一个实现Runnable接口的类的对象为参数。默认名为Thread-n，n是从0开始递增的整数
public Thread(String name)	构造一个新的线程对象，并同时指定线程名
public static Thread currentThread()	返回当前正在运行的线程对象
public static void yield()	使当前线程对象暂停，允许别的线程开始运行
public static void sleep(long millis)	使当前线程暂停运行指定毫秒数，但此线程并不失去已获得的锁旗标。

7.1.2 Thread类(续)

——常用API函数

public void start()	启动线程，JVM将调用此线程的run方法，结果是将同时运行两个线程，当前线程和执行run方法的线程
public void run()	Thread的子类应该重写此方法，内容应为该线程应执行的任务。
public final void stop()	停止线程运行，释放该线程占用的对象锁旗标。
public void interrupt()	打断此线程
public final void join()	在当前线程中加入调用join方法的线程A，直到线程A死亡才能继续执行当前线程
public final void join(long millis)	在当前线程中加入调用join方法的线程A，直到到达参数指定毫秒数或线程A死亡才能继续执行当前线程

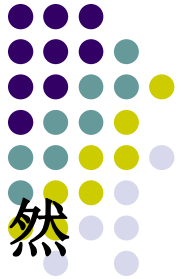
7.1.2 Thread类(续)

——常用API函数

多线程编程基础

public final void setPriority(int newPriority)	设置线程优先级
public final void setDaemon(Boolean on)	设置是否为后台线程，如果当前运行线程均为后台线程则JVM停止运行。这个方法必须在start()方法前使用
public final void checkAccess()	判断当前线程是否有权力修改调用此方法的线程
public void setName(String name)	更该本线程的名称为指定参数
public final boolean isAlive()	测试线程是否处于活动状态，如果线程被启动并且没有死亡则返回true

§7.1.2 Thread类

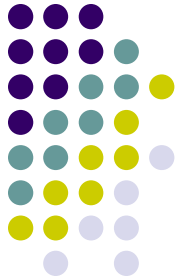


- 创建3个新线程，每个线程睡眠一段时间（0～6秒），然后结束

```
public class MultiTreadEx2 {
   public static void main( String [] args ) {
     //创建并命名每个线程
      TestThread thread1 = new TestThread( "thread1" );
      TestThread thread2 = new TestThread( "thread2" );
      TestThread thread3 = new TestThread( "thread3" );
      System.out.println( "Starting threads" );
      thread1.start(); // 启动线程1
      thread2.start(); // 启动线程2
      thread3.start(); // 启动线程3
      System.out.println( "Threads started, main ends\n" );
   }
}
```

```
class TestThread extends Thread {
   private int sleepTime;
   public TestThread( String name )
   {
      super( name );
      sleepTime = ( int ) ( Math.random() * 6000 );
   }
   public void run() {
      try {
         System.out.println(
                                    getName() + " going to sleep for " + sleepTime );
        Thread.sleep( sleepTime ); //线程休眠
      }
      catch ( InterruptedException exception )
      {;}
      System.out.println( getName() + " finished"
   }
}
```

§7.1.2 Thread类



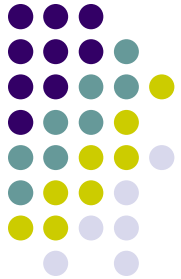
- 运行结果

Starting threads
Threads started, main ends

thread1 going to sleep for 3519
thread2 going to sleep for 1689
thread3 going to sleep for 5565
thread2 finished
thread1 finished
thread3 finished

- 说明
  - **由于线程3休眠时间最长，所以最后结束，线程2休眠时间最短，所以最先结束**
  - **每次运行，都会产生不同的随机休眠时间，所以结果都不相同**

§7.1.3 Runnable接口



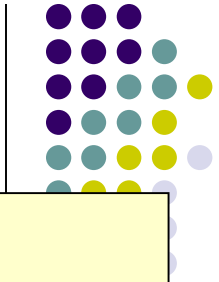
- Runnable接口
  - Java多线程机制的一个重要部分，实际上它只有一个run()方法
  - Thread类实现了Runnable接口，相对于Thread类，它更适合于多个线程处理同一资源
  - 实现Runnable接口的类的对象可以用来创建线程，这时start方法启动此线程就会在此线程上运行run()方法
  - 在编写复杂程序时相关的类可能已经继承了某个基类，而Java不支持多继承，在这种情况下，便需要通过实现Runnable接口来生成多线程

§7.1.3 Runnable接口

- 使用Runnable接口实现例MultiTreadEx1功能

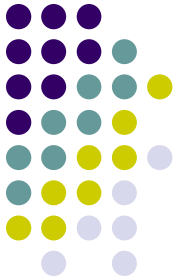
```
class FactorialThread implements Runnable {
   private int num;
   public FactorialThread( int num ) {
      this.num=num;
   }
   public void run()  {
       int i=num;
       int result=1;
       while(i>0) {
            result=result*i;
            i=i-1;
       }
       System.out.println("The factorial of "+num+" is "+result);
       System.out.println("new thread ends");
    }
}
```

§7.1.3 Runnable接口



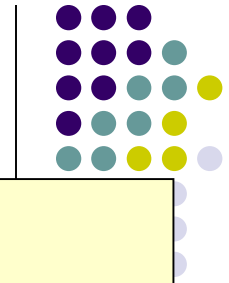
```
public class MultiTreadEx3 {
   public static void main( String [] args ) {
      System.out.println("main thread starts");
      FactorialThread  t = new FactorialThread(10);
      new Thread(t).start();
      System.out.println("new thread started,main thread ends " );
   }
}
```

§7.1.3 Runnable接口



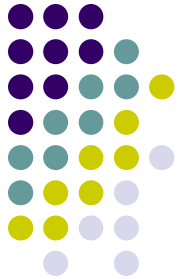
- 使用Runnable接口实现例MultiTreadEx2功能

```
public class MultiTreadEx4 {
    public static void main( String [] args ) {
        TestThread thread1 = new TestThread();
        TestThread thread2 = new TestThread();
        TestThread thread3 = new TestThread();
        System.out.println( "Starting threads" );

        new Thread(thread1,"Thread1").start();
        new Thread(thread2,"Thread2").start();
        new Thread(thread3,"Thread3").start();

        System.out.println( "Threads started, main ends\n" );
   }
}
```



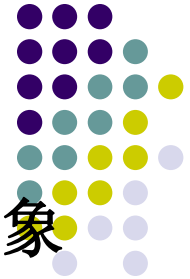
```
class TestThread implements Runnable {
   private int sleepTime;
   public TestThread() {
      sleepTime = ( int ) ( Math.random() * 6000 );
   }
   public void run() {
      try {
         System.out.println(
               Thread.currentThread().getName() + " going to sleep for "
                                                + sleepTime );
         Thread.sleep( sleepTime );
      }
      catch ( InterruptedException exception )
      {;}
      System.out.println( Thread.currentThread().getName()+ "finished" );
   }
}
```

§7.1.4 线程间的数据共享



- 代码共享
  - **多个线程的执行代码来自同一个类的run方法时，即称它们共享相同的代码**
- 数据共享
  - **当共享访问相同的对象时，即它们共享相同的数据**
  - **使用Runnable接口可以轻松实现多个线程共享相同数据，只要用同一个实现了Runnable接口的实例作为参数创建多个线程就可以了**

§7.1.4 线程间的数据共享

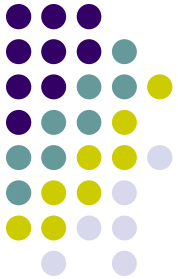


- 修改例MultiTreadEx4，只用一个Runnable类型的对象为参数创建3个新线程。

```
public class MultiTreadEx5{
    public static void main( String [] args )
    {
        TestThread threadobj = new TestThread();
        System.out.println( "Starting threads" );

        new Thread(threadobj,"Thread1").start();
        new Thread(threadobj,"Thread2").start();
        new Thread(threadobj,"Thread3").start();

        System.out.println( "Threads started, main ends\n" );
  }
}
```

```
class TestThread implements Runnable
{
        private int sleepTime;
        public TestThread()
        {
              sleepTime = ( int ) ( Math.random() * 6000 );
         }
         public void run()
         {
               try {
                     System.out.println(
                     Thread.currentThread().getName() + " going to sleep for "
                                                          +sleepTime );
                     Thread.sleep( sleepTime );
                }
               catch ( InterruptedException exception ) {};
              System.out.println( Thread.currentThread().getName() +
                                          "finished" );
          }
}
```

§7.1.4 线程间的数据共享

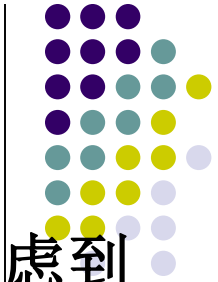


- 运行结果

```
Starting threads
Thread1 going to sleep for 966
Thread2 going to sleep for 966
Threads started, main ends
Thread3 going to sleep for 966
Thread1 finished
Thread2 finished
Thread3 finished
```

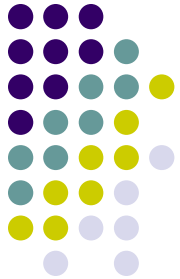
- 说明
  - **因为是用一个Runnable类型对象创建的3个新线程，这三个线程就共享了这个对象的私有成员sleepTime，在本次运行中，三个线程都休眠了966毫秒**

§7.1.4 线程间的数据共享



- 独立的同时运行的线程有时需要共享一些数据并且考虑到彼此的状态和动作
  - **例如生产/消费问题：生产线程产生数据流，然后这些数据流再被消费线程消费**
  - **假设一个Java应用程序，其中有一个线程负责往文件写数据，另一个线程从同一个文件中往外读数据，因为涉及到同一个资源，这里是同一个文件，这两个线程必须保证某种方式的同步**

§7.1.4 线程间的数据共享

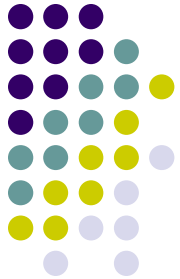


- 用三个线程模拟三个售票口，总共出售200张票
  - **用3个线程模仿3个售票口的售票行为**
  - **这3个线程应该共享200张票的数据**

```
public class MultiTreadEx6{
        public static void main(String[] args)
        {
                SellTickets t=new SellTickets();
                 new Thread(t).start();
                 new Thread(t).start();
                 new Thread(t).start();
         }
}
```

```
class SellTickets implements Runnable
{
        private int tickets=200;
        public void run()
        {
                while(tickets>0)
                {
                        System.out.println( Thread.currentThread().getName() +
                                                        " is selling ticket "+tickets--);
                }
        }
}
```

§7.1.4 线程间的数据共享



- 运行结果选最后几行如下

Thread-2 is selling ticket 6

Thread-1 is selling ticket 5

Thread-0 is selling ticket 4

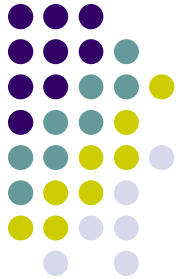
Thread-2 is selling ticket 3

Thread-1 is selling ticket 2

Thread-0 is selling ticket 1

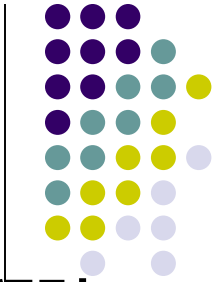
- 说明
  - **在这个例子中，创建了3个线程，每个线程调用的是同一个SellTickets对象中的run()方法，访问的是同一个对象中的变量（tickets）**
  - **如果是通过创建Thread类的子类来模拟售票过程，再创建3个新线程，则每个线程都会有各自的方法和变量，虽然方法是相同的，但变量却是各有200张票，因而结果将会是各卖出200张票，和原意就不符了**

§7.1.5 多线程的同步控制



- 有时线程之间彼此不独立、需要同步
  - **线程间的互斥**
    - **同时运行的几个线程需要共享一个（些）数据**
    - **一个线程对共享的数据进行操作时，不允许其他线程打断它，否则会破坏数据的完整性。即被多个线程共享的数据，在某一时刻只允许一个线程对其进行操作**
  - **“生产者/消费者” 问题**
    - **生产者产生数据，消费者消费数据，具体来说，假设有一个Java应用程序，其中有一个线程负责往数据区写数据，另一个线程从同一数据区中读数据，两个线程可以并行执行（类似于流水线上的两道工序）**
    - **如果数据区已满，，生产者要等消费者取走一些数据后才能再放；而当数据区没有数据时，消费者要等生产者放入一些数据后再取**

§7.1.5 多线程的同步控制



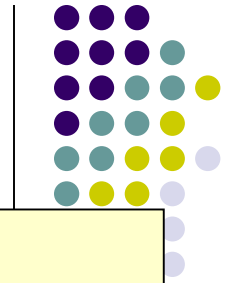
- 用两个线程模拟存票、售票过程
  - **假定开始售票处并没有票，一个线程往里存票，另外一个线程则往出卖票**
  - **我们新建一个票类对象，让存票和售票线程都访问它。**
  - **本例采用两个线程共享同一个数据对象来实现对同一份数据的操作**

```
public class MultiTreadEx7{
    public static void main(String[] args)
    {
        Tickets t=new Tickets(10);
        new Consumer(t).start();
        new Producer(t).start();
    }
}
```

§7.1.5 多线程的同步控制

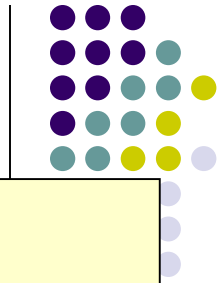
```
class Tickets {
        int number=0;          //票号
        int size;              //总票数
        boolean available=false;  //表示目前是否有票可售
        public Tickets(int size)  //构造函数，传入总票数参数
       {
             this.size=size;
       }
}
```

§7.1.5 多线程的同步控制



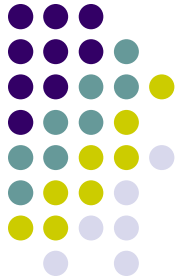
```
class Producer extends Thread
{
        Tickets t=null;
        public Producer(Tickets t)
        {
             this.t=t;
        }
        public void run()
        {
             while( t.number < t.size)
              {
                    System.out.println("Producer puts ticket " +(++t.number));
                    t.available=true;
               }
         }
}
```

§7.1.5 多线程的同步控制



```
class Consumer extends Thread  //售票线程
{
        Tickets t=null;
         int i=0;
         public Consumer(Tickets t)
         {
                this.t=t;
          }
          public void run()
          {
              while(i<t.size)
              {
                   if(t.available==true && i<=t.number)
                          System.out.println("Consumer buys ticket "+(++i));
                   if(i==t.number)
                         t.available=false;
              }
         }
}
```

§7.1.5 多线程的同步控制

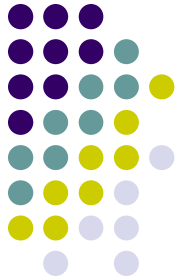


- 运行结果

Producer puts ticket 1
Producer puts ticket 2
Producer puts ticket 3
Producer puts ticket 4
Producer puts ticket 5
Producer puts ticket 6
Producer puts ticket 7
Producer puts ticket 8
Consumer buys ticket 1
Consumer buys ticket 2
Consumer buys ticket 3
Consumer buys ticket 4
Consumer buys ticket 5
Consumer buys ticket 6
Consumer buys ticket 7
Consumer buys ticket 8
Producer puts ticket 9
Producer puts ticket 10
Consumer buys ticket 9
Consumer buys ticket 10.

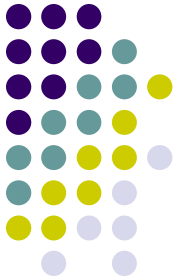
- 通过让两个线程操纵同一个票类对象，实现了数据共享的目的

§7.2线程的生命周期



- 线程的生命周期
  - **线程从产生到消亡的过程**
  - **一个线程在任何时刻都处于某种线程状态（thread state）**

7.2.1 线程的几种基本状态



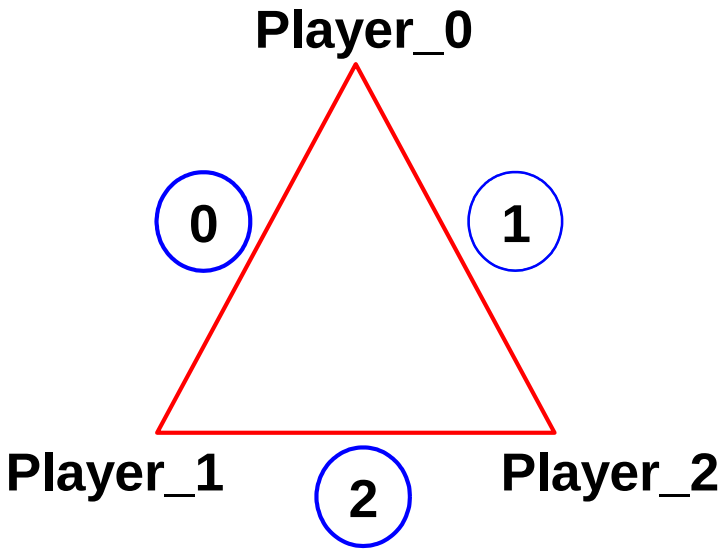
- 诞生状态
  - **线程刚刚被创建**
- 就绪状态
  - **线程的 start 方法已被执行**
  - **线程已准备好运行**
- 运行状态
  - **处理机分配给了线程，线程正在运行**
- 阻塞状态（Blocked）
  - **在线程发出输入/输出请求且必须等待其返回**
  - **遇到用synchronized标记的方法而未获得其监视器暂时不能进入执行时**
- 休眠状态（Sleeping）
  - **执行sleep方法而进入休眠**
- 死亡状态
  - **线程已完成或退出**

7.2.2 死锁问题

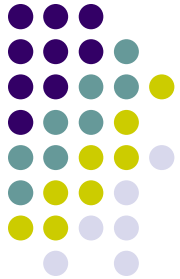
- 死锁
  - **线程在运行过程中，其中某个步骤往往需要满足一些条件才能继续进行下去，如果这个条件不能满足，线程将在这个步骤上出现阻塞**
  - **线程A可能会陷于对线程B的等待，而线程B同样陷于对线程C的等待，依次类推，整个等待链最后又可能回到线程A。如此一来便陷入一个彼此等待的轮回中，任何线程都动弹不得，此即所谓死锁（deadlock）**
  - **对于死锁问题，关键不在于出现问题后调试，而是在于预防**

7.2.2 死锁问题

- 设想一个游戏，规则为3个人站在三角形的三个顶点的位置上，三个边上放着三个球，如图所示。每个人都必须先拿到自己左手边的球，才能再拿到右手边的球，两手都有球之后，才能够把两个球都放下



7.2.3 控制线程的生命



- 结束线程的生命
  - **用stop方法可以结束线程的生命**
    - **但如果一个线程正在操作共享数据段，操作过程没有完成就用stop结束的话，将会导致数据的不完整，因此并不提倡使用此方法**
  - **通常，可通过控制run方法中循环条件的方式来结束一个线程**

7.2.3 控制线程的生命

- 线程不断显示递增整数，按下回车键则停止执行

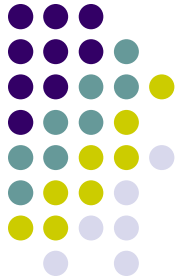
```
import java.io.*;
public class MutliThreadEx12{
        public static void main(String[] args) throws IOException
        {
            TestThread t=new TestThread();
            t.start();
            new BufferedReader(new InputStreamReader(System.in)).readLine();
            //等待键盘输入
            t.stopme();  //调用stopme方法结束t线程
        }
}
```

7.2.3 控制线程的生命

```
class TestThread extends Thread
{
      private boolean flag=true;
      public void stopme()
     {   //在此方法中控制循环条件
          flag=false;
      }
      public void run()
      {
          int i=0;
          while(flag)
         {
             System.out.println(i++);  //如果flag为真则一直显示递增整数
          }
       }
}
```

- 运行效果为按下回车键后则停止显示

7.3 线程的优先级



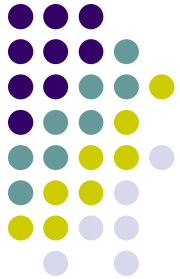
- 线程调度
  - **在单CPU的系统中，多个线程需要共享CPU，在任何时间点上实际只能有一个线程在运行**
  - **控制多个线程在同一个CPU上以某种顺序运行称为线程调度**
  - **Java虚拟机支持一种非常简单的、确定的调度算法，叫做固定优先级算法。这个算法基于线程的优先级对其进行调度**

7.3 线程的优先级

- 线程的优先级
  - 每个Java线程都有一个优先级，其范围都在1和10之间。默认情况下，每个线程的优先级都设置为5
  - 在线程A运行过程中创建的新的线程对象B，初始状态具有和线程A相同的优先级
  - 如果A是个后台线程，则B也是个后台线程
  - 可在线程创建之后的任何时候，通过

    setPriority(int priority)方法改变其原来的优先级

7.3 线程的优先级



- 基于线程优先级的线程调度
  - **具有较高优先级的线程比优先级较低的线程优先执行**
  - **对具有相同优先级的线程，Java的处理是随机的**
  - **底层操作系统支持的优先级可能要少于10个，这样会造成一些混乱。因此，只能将优先级作为一种很粗略的工具使用。最后的控制可以通过明智地使用yield()函数来完成**
  - **我们只能基于效率的考虑来使用线程优先级，而不能依靠线程优先级来保证算法的正确性**

7.3 线程的优先级

- 假设某线程正在运行，则只有出现以下情况之一，才会使其暂停运行
  - **一个具有更高优先级的线程变为就绪状态（Ready）；**
  - **由于输入/输出（或其他一些原因）、调用sleep、wait、yield方法使其发生阻塞；**
  - **对于支持时间分片的系统，时间片的时间期满**

7.3 线程的优先级

- 创建两个具有不同优先级的线程，都从1递增到400000，每增加50000显示一次

```
public class Ex8_13{
    public static void main(String[] args) {
        TestThread[] runners = new TestThread[2];
        for (int i = 0; i < 2; i++)
            runners[i] = new TestThread(i);
        runners[0].setPriority(2);   //设置第一个线程优先级为2
        runners[1].setPriority(3);   //设置第二个线程优先级为3
        for (int i = 0; i < 2; i++)
            runners[i].start();
    }
}
```

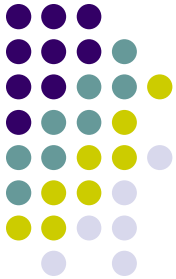
7.3 线程的优先级

```
class TestThread extends Thread{
        private int tick = 1;
        private int num;
        public TestThread(int i) {  this.num=i; }
        public void run()
        {
           while (tick < 400000) {
             tick++;
             if ((tick % 50000) == 0) {  //每隔5000进行显示
                 System.out.println("Thread #" + num + ", tick = " + tick);
              yield();  //放弃执行权
             }
          }
      }
}
```

7.3 线程的优先级

- 运行结果

```
Thread #1, tick = 50000
Thread #1, tick = 100000
Thread #1, tick = 150000
Thread #1, tick = 200000
Thread #1, tick = 250000
Thread #1, tick = 300000
Thread #1, tick = 350000
Thread #1, tick = 400000
Thread #0, tick = 50000
Thread #0, tick = 100000
Thread #0, tick = 150000
Thread #0, tick = 200000
Thread #0, tick = 250000
Thread #0, tick = 300000
Thread #0, tick = 350000
Thread #0, tick = 400000
```



- 结果说明
  - 具有较高优先级的线程1一直运行到结束，具有较低优先级的线程0才开始运行
  - 虽然具有较高优先级的线程1调用了yield方法放弃CPU资源，允许线程0进行争夺，但马上又被线程1抢夺了回去，所以有没有yield方法都没什么区别

7.3 线程的优先级

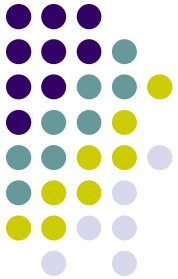
- 如果在yield方法后增加一行sleep语句，让线程1暂时放弃一下在CPU上的运行，哪怕是1毫秒，则线程0也可以有机会被调度。修改后的run方法如下

```
public void run() {
    while (tick < 400000) {
        tick++;
        if ((tick % 50000) == 0) {
            System.out.println("Thread #" + num + ", tick = " + tick);
            yield();
            try{ sleep(1);}catch(Exception e){};
        }
    }
}
```

7.3 线程的优先级

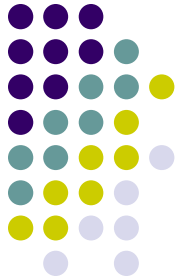
- 运行结果

```
Thread #1, tick = 50000
Thread #1, tick = 100000
Thread #1, tick = 150000
Thread #1, tick = 200000
Thread #0, tick = 50000
Thread #1, tick = 250000
Thread #1, tick = 300000
Thread #0, tick = 100000
Thread #1, tick = 350000
Thread #1, tick = 400000
Thread #0, tick = 150000
Thread #0, tick = 200000
Thread #0, tick = 250000
Thread #0, tick = 300000
Thread #0, tick = 350000
Thread #0, tick = 400000
```



- 结果说明
  - 具有较低优先权的线程0在线程1没有执行完毕前也获得了一部分执行，但线程1还是优先完成了执行
  - Java虚拟机本身并不支持某个线程抢夺另一个正在执行的具有同等优先级线程的执行权
  - 通常，我们在一个线程内部插入yield()语句，这个方法会使正在运行的线程暂时放弃执行，这是具有同样优先级的线程就有机会获得调度开始运行，但较低优先级的线程仍将被忽略不参加调度

本章小结



- 本章内容
  - **线程的基础知识**
  - **线程的生命周期**
  - **线程的优先级**
- 本章要求
  - **了解线程的概念**
  - **学会如何通过Thread类和Runnable接口创建线程，如何实现多线程的资源共享和通信，及如何控制线程的生命**
  - **掌握线程同步的方法**
  - **理解线程优先级的概念，以及基于优先级的线程调度**