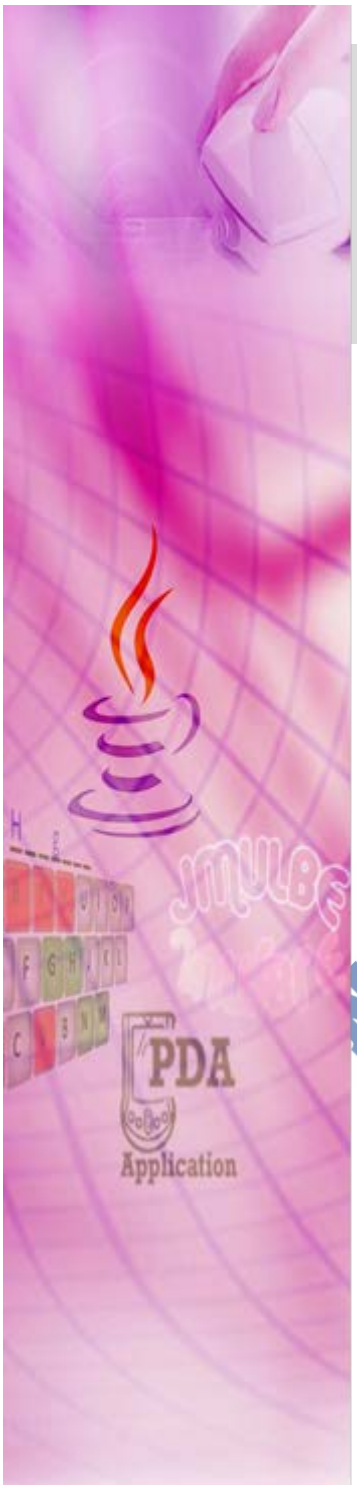


第 5 章

使用线程

大多数计算机游戏使用图形和声音。图形、分数和音频效果是同时呈现的。想象一个计算机游戏：您首先看到屏幕变幻，然后看到分数已经更新，最后听到声音效果。为了我们能够欣赏和享受游戏，所有这些元素都需要同时得到处理。换句话说，程序应该能够高效地同时执行三个任务。在 Java 中，您可使用*线程*实现以上功能。游戏可以分为三个子单元，可使用不同线程并行地执行它们。

本章讨论如何在 Java 中创建多线程应用程序。此外，还讨论线程优先级。

目标

在本章中，您将学习：

- 在 Java 中使用线程
- 创建线程


NIIT
training
china
.com

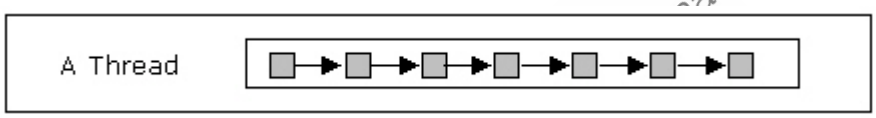
在 Java 中使用线程

线程可定义为程序内的单序列控制流。它是为定义唯一控制流而执行的指令序列。例如，中央处理器 (CPU) 同时执行各种进程，例如编写和打印文档、安装软件以及显示日期和时间。所有这些进程都由单独的线程处理。应用程序包含一个或多个可同时执行的进程。然后，每个进程分解为称为任务的小块。每个线程被分配了一个要独立执行的独特任务。例如，文本编辑器是使您能够同时执行两个任务（例如写入文档并且打印文档）的应用程序。

然后，可在应用程序中实现多线程以实现高性能并且利用 CPU 处理能力。为了实现多线程，Java 提供 `Thread` 类。要使用线程，首先需要了解其生命周期。

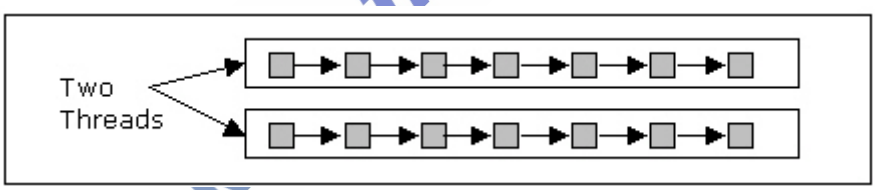
多线程的基本概念

单线程应用程序一次仅可以执行一项任务。下图显示单线程应用程序。



单线程应用程序

单线程应用程序必须等到一项任务完成后才能开始另一项任务。因此，应用程序的效率受到影响。另一方面，可在需要使应用程序能够同时执行多个任务时应用多线程。多线程有助于同时执行多个任务，这样就节省了时间并改进了应用程序的执行速度和效率。每个应用程序至少有一个线程，需要时可以创建更多线程。下图显示多线程应用程序。



多线程应用程序

微处理器将内存分配给应用程序执行的进程，每个进程占用其自己的地址空间或内存。但是，进程中的所有线程都占用相同的地址空间。

多线程的优点和缺点

当今大多数计算机使用支持多线程的软件和硬件（例如操作系统和多个处理器）。因此，要实现效率并确保优化的性能，大多数应用程序使用多线程。然而，要了解使用多线程的优点与缺点。多线程的各种优点有：

- **改进性能**：通过同时执行计算和输入/输出 (I/O) 操作改进了处理器的性能。
- **最小化系统资源使用**：通过使用共享相同的地址空间和属于同一个进程的线程，最小化系统资源的使用。

- **简化程序结构**：简化复杂应用程序（例如多媒体应用程序）的结构。可以为每项活动编写子程序以使复杂程序易于设计和编码。

多线程的各种缺点有：

- **竞争情况**：当两个或多个线程同时访问同一变量且至少一个线程尝试向该变量写入值时，称为竞争情况。当两个线程间未实现同步时，将引发此情况。例如，在文字处理器程序中，有两个线程 ThreadA 和 ThreadB。ThreadA 想要读取文件，同时 ThreadB 想要写入文件。如果 ThreadA 在 ThreadB 执行写操作前读取值，ThreadA 将无法从文件接收更新后的值。此类情况就称为竞争情况。
- **死锁情况**：当两个线程互相等待对方完成其操作然后才执行各自的操作时，将在计算机系统中发生*死锁*情况。结果，这两个线程被锁定，程序失败。例如，考虑两个线程 ThreadA 和 ThreadB。ThreadA 正等待 ThreadB 释放锁，而 ThreadB 也正在等 ThreadA 释放该锁以完成其事务。此情况为死锁情况。
- **锁饥饿**：当线程的执行由于其优先级低而被推迟时，将发生*锁饥饿*。JRE 基于它们的优先级执行线程，因为 CPU 一次只能执行一个线程。较高优先级的线程将在较低优先级的线程前执行，因此有可能较低优先级的线程总是被推迟执行。

Thread 类

`java.lang.Thread` 类用于构造和访问多线程应用程序中的单个线程。您可以使用 `Thread` 类或 `Runnable` 接口创建多线程应用程序。

`Thread` 类包含可用于处理线程（例如设置和检查线程的属性，导致线程等待以及中断或销毁）的各种方法。您可以通过扩展 `Thread` 类使类在单独的线程中运行。`Thread` 类中定义的一些方法如下：

- `int getPriority()`：返回线程的优先级。
- `boolean isAlive()`：确定线程是否正在运行。
- `static void sleep(long milliseconds)`：使线程执行暂停 `milliseconds` 值指定的一段时间。
- `String getName()`：返回线程的名称。
- `void start()`：通过调用 `run()` 方法启动线程执行。
- `static Thread currentThread()`：返回当前正在执行的线程对象的引用。
- `boolean isAlive()`：用于检查线程的存在性。如果线程正在运行，那么此方法返回 `true`，如果线程为新线程或已终止，那么该方法返回 `false`。
- `public final void join()`：允许线程等待直到调用该方法的线程终止为止。
- `void interrupt()`：用于使线程执行中断。
- `static boolean interrupted()`：用于确定当前线程是否被其他线程中断。

可使用以下代码了解 `Thread` 类中方法的用法：

```
class MainThreadDemo
{
public static void main(String args[])
```

```
{
Thread t= Thread.currentThread();
System.out.println(" The current thread:" + t);
t.setName("MainThread");
System.out.println(" The current thread after name change :"
+ t);
System.out.println(" The current Thread is going to sleep for
10 seconds");
try
{
   t.sleep(10000);
}
catch(InterruptedException e)
{
System.out.println("Main thread interrupted");
}
   System.out.println(" After 10 seconds..........the current
Thread is exiting now.");
}
}
```

执行上述代码后，会显示以下输出：

```
The current thread:Thread[main,5,main]
The current thread after name change :Thread[MainThread,5,main]
The current Thread is going to sleep for 10 seconds
After 10 seconds..........the current Thread is exiting now.
```

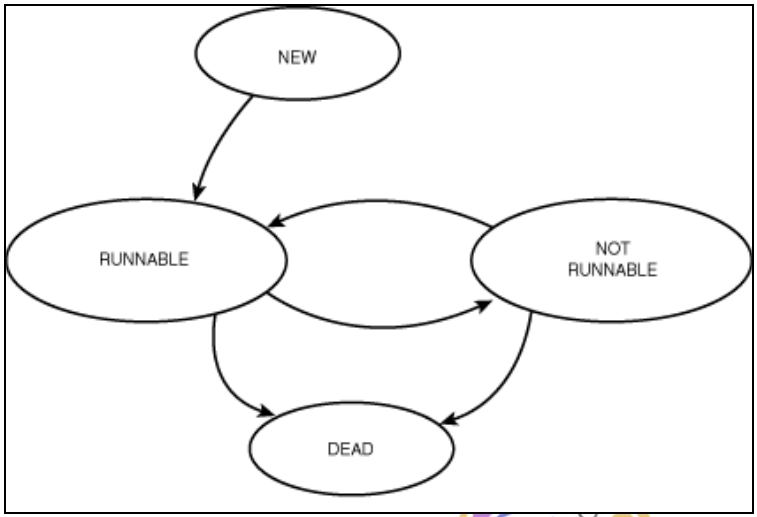
在上述代码中，要执行的第一个线程是主线程。它是在执行 Java 程序时自动创建的。在主程序中，通过调用 `currentThread()` 方法获取了当前线程的引用，并且其引用存储在引用变量 `t` 中。调用了 `setName()` 方法来设置线程的名称，并显示了关于该线程的信息。然后，使用 `sleep()` 方法使主线程休眠 10 秒，10 秒钟之后，该线程终止。只要其他线程中断了正在休眠的线程，那么 `sleep()` 方法就会抛出 `InterruptedException` 异常。

线程的生命周期

在线程的生命周期中的各种状态有：

- 新生
- 可运行
- 不可运行
- 终止或死亡

- 下图显示了线程的生命周期。



线程的生命周期

“新生”线程状态

创建了 `Thread` 类的实例后，线程进入新线程状态。以下代码段显示如何实例化 `Thread` 类：

```
Thread newThread = new Thread(this, "threadName");
```

在上述代码段中，创建了新线程。这个新线程是 `Thread` 类的空对象，没有给它分配系统资源（例如内存）。必须调用 `start()` 方法来启动线程。以下代码段显示了如何启动线程：

```
newThread.start();
```

“可运行”线程状态

当线程的 `start()` 方法被调用时，线程进入可运行状态。`start()` 方法向线程分配系统资源、调度线程并将控制传递到其 `run()` 方法。

单个处理器不能一次执行多个线程。因此，处理器维护线程队列。当线程启动时，它排队等候处理器时间，并等待自己得到执行。因此，线程的状态称为可运行但未在运行。

“不可运行”线程状态

在以下情况中，线程处于“不可运行”状态：

- **休眠**：通过调用 `sleep()` 方法，线程进入休眠模式。正在休眠的线程在指定的休眠时间过去后继续执行。
- **等待**：通过调用 `Object` 类的 `wait()` 方法，可让线程等待指定条件得到满足。可通过调用 `Object` 类的 `notify()` 或 `notifyAll()` 方法向线程发出条件状态通知。
- **正被其他线程阻止**：如果线程被 I/O 操作阻止，它进入“不可运行”状态。

“死亡”线程状态

一旦 `run()` 方法中的语句都已执行，线程就进入死亡状态。向线程对象分配 `null` 值也会将该线程的状态更改为死亡。`Thread` 类的 `isAlive()` 方法用于确定是否线程是否保活。不能重新启动死亡的线程。此外，可通过 `stop()` 方法停止线程执行来杀死线程。

创建线程

思考乱字游戏的场景：您需要在游戏中实现计时器。计时器将用于提供 60 秒的持续时间，在这段时间内玩家必须找出与乱字对应的正确单词。并且，当玩家在玩游戏时将在游戏屏幕上显示计时器。当计时器为 0 时将向玩家显示“Time Up!!!”消息，并且游戏将停止。

为了实现以上功能，您可创建线程。可以通过实例化 `Thread` 类的对象来实现。可使用以下方法在 Java 应用程序中创建线程：

- 通过扩展 `Thread` 类
- 通过实现 `Runnable` 接口

然后，您还可使用以上方法在应用程序中创建多个线程，并设置其优先级以确定它们在 JRE 中的执行顺序。

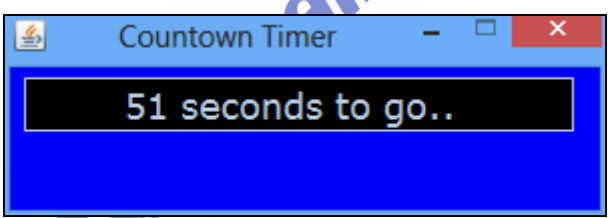
通过扩展 Thread 类创建线程

您可以通过扩展 `Thread` 类创建线程。需要在重写的 `run()` 方法中指定要由线程执行的任务。可以使用以下代码通过扩展 `Thread` 类来创建线程实现计时器：

```
import java.awt.Color;
import java.awt.Font;
import javax.swing.*;

public class CountdownTimer extends Thread {
JTextField tf;
JLabel l;
JFrame fr;
public void run()
{
buildGUI();
}

void display() {

for (int i = 60; i >= 0; i--)
{
try {
Thread.sleep(1000);
String s = Integer.toString(i);

tf.setText("        "+ s + " seconds to go..");
} catch (Exception e) {
System.out.println(e);
}

}
JOptionPane.showMessageDialog(fr, "Time Up !!!!");
tf.setText("");
tf.setEnabled(false);
}
```

```
public void buildGUI() {
fr = new JFrame("Countown Timer");
JPanel p = new JPanel();
l = new JLabel("");
tf = new JTextField(15);
tf.setEnabled(false);
Font f = new Font("Verdana", 0, 18);
tf.setFont(f);
tf.setBackground(Color.BLACK);
p.setBackground(Color.blue);
fr.add(p);
p.add(tf);
p.add(l);
fr.setVisible(true);
fr.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
fr.setSize(300, 100);
fr.setResizable(false);
display();
}

public static void main(String args[]) {
CountdownTimer obj = new CountdownTimer();
obj.start();
}
}
```

在执行上述代码之后，将显示以下输出：



输出

注释

上图中的时间可能不同。

在上述代码中，`CountdownTimer` 类扩展 `Thread` 类。`CountdownTimer` 类的新实例是在 `main()` 方法中创建的。`start()` 方法通过调用提供计时器实现的 `run()` 方法开始线程执行。

通过实现 Runnable 接口创建线程

思考以下场景：您需要在 Java 应用程序中实现线程，并且还要扩展另一个类以使用其功能（例如 JFrame）。由于 Java 不支持多继承，因此它提供 Runnable 接口来解决此问题。Runnable 接口仅包括 run() 方法，启动线程时执行此方法。

当 Java 程序需要从除 Thread 类之外的类继承时，您必须实现 Runnable 接口以实现线程。run() 方法的签名如下：

```
public void run()
```

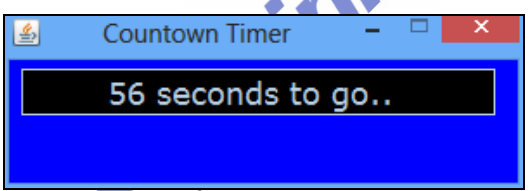
run() 方法包含新线程将执行的代码。调用 run() 方法时，另一个线程开始与程序中的主线程并行执行。

实现 Runnable 接口的类用于创建 Thread 类的实例。新线程对象通过调用 start() 方法开始执行。可以使用以下代码通过实现 Runnable 接口来创建线程实现计时器功能：

```
import java.awt.Color;
import java.awt.Font;
import javax.swing.*;

public class CountdownTimer implements Runnable {
JTextField tf;
JLabel l;
JFrame fr;
public void run()
{
buildGUI();
}

void display() {

for (int i = 60; i >= 0; i--)
{
try {
Thread.sleep(1000);
String s = Integer.toString(i);

tf.setText("        "+ s + " seconds to go..");
} catch (Exception e) {
System.out.println(e);
}

}
JOptionPane.showMessageDialog(fr, "Time Up !!!!");
tf.setText("");
tf.setEnabled(false);
}

public void buildGUI() {
fr = new JFrame("Countown Timer");
```

```
JPanel p = new JPanel();
l = new JLabel("");
tf = new JTextField(15);
tf.setEnabled(false);
Font f = new Font("Verdana", 0, 18);
tf.setFont(f);
tf.setBackground(Color.BLACK);
p.setBackground(Color.blue);
fr.add(p);
p.add(tf);
p.add(l);
fr.setVisible(true);
fr.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
fr.setSize(300, 100);
fr.setResizable(false);
display();
}

public static void main(String args[]) {
CountdownTimer obj = new CountdownTimer();
Thread CountdownThread = new Thread(obj);          //Initialize thread
CountdownThread.start();
}
}
```

执行上述代码后，会显示以下输出：



输出

注释

上图中的时间可能不同。

在上述代码中，`CountdownTimer` 类实现 `Runnable` 接口。在 `main()` 方法中，创建了 `CountdownTimer` 类的对象，该对象作为参数传递给 `Thread` 类的构造函数。以上对象表示可运行任务。`start()` 方法通过调用提供计时器实现的 `run()` 方法开始线程执行。

创建多线程

假设您需要开发赛车游戏，其中需要在框架之间同时移动多个对象（例如障碍物）。为此，您需要创建多个线程，其中每个线程表示一个障碍物。您可以通过实现 Runnable 接口或扩展 Thread 类来在程序中创建多个线程。

对于以上场景，可使用以下代码通过 Thread 类创建多个线程：

```
import java.awt.Color;
import java.util.Random;
import javax.swing.*;
public class Race extends Thread{

String ThreadName;
JLabel l;
JPanel l1,l2,l3;
JFrame fr;

public Race()
{
buildGUI();
}
public Race(String s)
{
super(s);
}

public void run()
{

if(Thread.currentThread().getName().equals("ObstacleA"))
{
runObstacleA();
}
if(Thread.currentThread().getName().equals("ObstacleB"))
{
runObstacleB();
}
if(Thread.currentThread().getName().equals("ObstacleC"))
{
runObstacleC();
}

}

public void runObstacleA()
{
Random ran = new Random();
int s = ran.nextInt(1000);
for(int i=-10;i<400;i++)
{
```

```
l1.setBounds(i, s, 20, 20);
try {
Thread.sleep(5);

}
catch(Exception e)
{
System.out.println(e);
}
}
runObstacleC();
}


public void runObstacleB()
{
Random ran = new Random();
int r = ran.nextInt(180);

for(int i=-10;i<400;i++)
{
l2.setBounds(i, r, 20, 20);
try {
Thread.sleep(11);

}
catch(Exception e)
{
System.out.println(e);
}
}
runObstacleA();
}

public void runObstacleC()
{
Random ran = new Random();
int m = ran.nextInt(10);

for(int i=-10;i<400;i++)
{
l3.setBounds(i, m, 20, 20);
try {
Thread.sleep(10);

}
catch(Exception e)
{
System.out.println(e);
}
}
runObstacleB();
}
public void buildGUI()
```

```
{
fr = new JFrame("Moving objects");
fr.setVisible(true);
fr.setSize(400,200);
fr.setLayout(null);

l = new JLabel("");
l.setBounds(10,10,400,20);
fr.add(l);

l1 = new JPanel();
l1.setSize(20, 20);
l1.setBackground(Color.red);
l1.setBounds(10,40,20,20);
fr.add(l1);
l2 = new JPanel();
l2.setSize(20, 20);
l2.setBackground(Color.blue);
l2.setBounds(10,80,20,20);
fr.add(l2);

l3 = new JPanel();
l3.setSize(20, 20);
l3.setBackground(Color.black);
l3.setBounds(10,120,20,20);
fr.add(l3);

}

public static void main(String args[])
{
                Race obj = new Race();
                Thread Obstacle1 = new Thread(obj);
                Thread Obstacle2 = new Thread(obj);
                Thread Obstacle3 = new Thread(obj);

                Obstacle1.setName("ObstacleA");
                Obstacle2.setName("ObstacleB");
                Obstacle3.setName("ObstacleC");
                Obstacle1.start();
                Obstacle2.start();
                Obstacle3.start();
}
}
```

在上述代码中，创建了三个线程引用 `Obstacle1`、`Obstacle2` 和 `Obstacle3`，并且 `Race` 类的 `obj` 对象作为参数传递。然后 `Obstacle1`、`Obstacle2` 和 `Obstacle3` 被分别赋予名称 `ObstacleA`、`ObstacleB` 和 `ObstacleC`。这些障碍物将出现在随机位置。

也可以使用以下代码通过使用 `Runnable` 接口来创建多个线程：

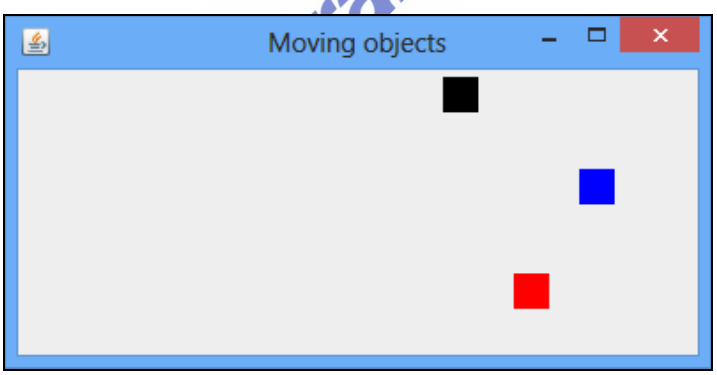
```
import java.awt.Color;
import java.util.Random;
import javax.swing.*;
```

```
public class Race implements Runnable{

String ThreadName;
JLabel l;
JPanel l1,l2,l3;
JFrame fr;

public Race()
{
buildGUI();
}

public void run()
{
if(Thread.currentThread().getName().equals("ObstacleA"))
{
runObstacleA();
}
if(Thread.currentThread().getName().equals("ObstacleB"))
{
runObstacleB();
}
if(Thread.currentThread().getName().equals("ObstacleC"))
{
runObstacleC();
}

}

public void runObstacleA()
{
Random ran = new Random();
int s = ran.nextInt(1000);
for(int i=-10;i<400;i++)
{

l1.setBounds(i, s, 20, 20);
try {
Thread.sleep(5);

}
catch(Exception e)
{
System.out.println(e);
}
}
runObstacleC();
}

public void runObstacleB()
{
Random ran = new Random();
int r = ran.nextInt(180);

for(int i=-10;i<400;i++)
```

```
{
l2.setBounds(i, r, 20, 20);
try {
Thread.sleep(11);

}
catch(Exception e)
{
System.out.println(e);
}
}
runObstacleA();
}

public void runObstacleC()
{
Random ran = new Random();
int m = ran.nextInt(10);

for(int i=-10;i<400;i++)
{
l3.setBounds(i, m, 20, 20);
try {
Thread.sleep(10);

}
catch(Exception e)
{
System.out.println(e);
}
}
runObstacleB();
}

public void buildGUI()
{
fr = new JFrame("Moving objects");
fr.setVisible(true);
fr.setSize(400,200);
fr.setLayout(null);

l = new JLabel("");
l.setBounds(10,10,400,20);
fr.add(l);

l1 = new JPanel();
l1.setSize(20, 20);
l1.setBackground(Color.red);
l1.setBounds(10,40,20,20);
fr.add(l1);

l2 = new JPanel();
l2.setSize(20, 20);
l2.setBackground(Color.blue);
l2.setBounds(10,80,20,20);
```

```
fr.add(l2);

l3 = new JPanel();
l3.setSize(20, 20);
l3.setBackground(Color.black);
l3.setBounds(10,120,20,20);
fr.add(l3);

}

public static void main(String args[])
{
Race obj = new Race();
    Thread Obstacle1 = new Thread(obj);
    Thread Obstacle2 = new Thread(obj);
           Thread Obstacle3 = new Thread(obj);

Obstacle1.setName("ObstacleA");
Obstacle2.setName("ObstacleB");
Obstacle3.setName("ObstacleC");
Obstacle1.start();
Obstacle2.start();
Obstacle3.start();
}
}
```

在上述代码中，三个线程 `ObstacleA`、`ObstacleB` 和 `ObstacleC` 共享 CPU 内存。这些障碍物将出现在框架上的随机位置并且跨框架移动。执行上述代码后，会显示以下输出：



输出

有时在多线程程序中，可能需要确保主线程在所有其他线程都已执行之后才执行完成。为此，需要知道某个线程是否已完成其执行，否则可通过 `main()` 方法加入某个线程的执行。这可通过 `isAlive()` 和 `join()` 方法来确定。

使用 `isAlive()` 方法

`isAlive()` 方法用于检查线程的存在性。您可以使用 `isAlive()` 方法查找线程的状态。`isAlive()` 方法的签名是：

```
boolean isAlive()
```

在上述签名中，如果线程正在运行，那么 isAlive() 方法返回 true；如果线程为新线程或已终止，那么该方法返回 false。如果 isAlive() 方法返回 true，那么线程可以处于“可运行”或“不可运行”状态。以下代码使用 isAlive() 方法：

```
class NewThreadClass implements Runnable
{
    Thread t;
    NewThreadClass()
    {
           t = new Thread(this,"ChildThread" );
           System.out.println("Thread created:" + t);
           t.start();
    }
    public void run()
    {
           try
           {
                  for(int i=1;i<=5;i++)
                  {
                         System.out.println(t + "loop :"+ i);
                         Thread.sleep(100);
                  }
           }
           catch( InterruptedException obj)
           {
                  System.out.println("Thread :"+ t + "interrupted");
           }
    }
}
class IsAliveDemo
{
     public static void main(String args[])
     {
    NewThreadClass obj = new NewThreadClass();
           System.out.println(obj.t  + "is alive ?:" +
obj.t.isAlive());
    try
    {
           for(int i=1;i<=5;i++)
           {
                  System.out.println("Main Thread loop:"+ i);
           Thread.sleep(200);
           }
     }
    catch(InterruptedException e)
    {
           System.out.println("Main thread is interrupted");}
           System.out.println(obj.t  + "is alive ?:" +
obj.t.isAlive());
           System.out.println("Main Thread is exiting");
    }
}
```

执行上述代码后，会显示以下输出：

```
Thread created:Thread[ChildThread,5,main]
Thread[ChildThread,5,main]is alive ?: true
Main Thread loop:1
Thread[ChildThread,5,main]loop :1
Thread[ChildThread,5,main]loop :2
Main Thread loop:2
Thread[ChildThread,5,main]loop :3
Thread[ChildThread,5,main]loop :4
Thread[ChildThread,5,main]loop :5
Main Thread loop:3
Main Thread loop:4
Main Thread loop:5
Thread[ChildThread,5,]is alive ?: false
Main Thread is exiting
```

在上述代码中，对线程调用了 `isAlive()` 方法以检查该线程是处于正在运行状态还是死亡状态。

使用 `join()` 方法

`join()` 方法允许线程等待直到调用该方法的线程终止为止。此外，`join()` 方法使您能够指定您等待指定线程终止所需的最长时间。`join()` 方法的签名是：

```
public final void join() throws InterruptedException
```

在以上签名中，如果另一个线程中断，`join()` 方法将抛出 `InterruptedException` 异常。一个线程的执行可由另一个线程使用 `Thread` 类的 `interrupt()` 方法中断。要确定当前线程是否已由另一个线程中断，`Thread` 类提供 `interrupted()` 方法。

然后，当指定线程死亡时，`join()` 方法将控制返回到调用方法。以下代码使用 `join()` 方法：

```
class ChildThread implements Runnable
{
Thread t;
ChildThread()
{
    t = new Thread(this,"ChildThread" );
    System.out.println("Thread created:" + t);
    t.start();
}
public void run()
{
    try
    {
                for(int i=1;i<=5;i++)
          {
                System.out.println(t + "loop :"+ i);
                Thread.sleep(500);
          }
    }
    catch( InterruptedException obj)
    {
```

```
            System.out.println("Thread :"+ t + "interrupted");
        }
    }
    }
    public class JoinDemo
    {
    public static void main(String args[])
    {
            ChildThread obj = new ChildThread();
            System.out.println(obj.t  + "is alive ?:" +
    obj.t.isAlive());
            try
            {
        System.out.println("Main thread waiting for child thread to finish");
                    obj.t.join();
              }
              catch(InterruptedException e)
              {
                System.out.println("Main thread is interrupted");
    }
                System.out.println(obj.t  + "is alive ?:" +
                obj.t.isAlive());
                System.out.println("Main Thread is exiting");
        }
    }
```

执行上述代码后，会显示以下输出：

```
Thread created:Thread[ChildThread,5,main]
Thread[ChildThread,5,main]is alive ?: true
Main thread waiting for child thread to finish
Thread[ChildThread,5,main]loop :1
Thread[ChildThread,5,main]loop :2
Thread[ChildThread,5,main]loop :3
Thread[ChildThread,5,main]loop :4
Thread[ChildThread,5,main]loop :5
Thread[ChildThread,5,]is alive ?: false
```

在上述代码中，对子线程调用了 `join()` 方法，主线程等待子线程完成执行。

了解线程优先级

JRE 基于线程的优先级执行线程。由于 CPU 一个时间只能执行一个线程，执行就绪的各个线程将被调度到一个将由处理器执行的队列中。JRE 通过使用固定优先级调度来调度线程。每个线程都有一个优先级，该优先级影响线程在处理器的线程队列中的位置。有较高优先级的线程将被调度为在较低优先级的线程之前执行。

定义线程优先级

*线程优先级*是介于 1 至 10 之间的整数，其指定一个线程相对于其他线程的优先级的优先级。在单个 CPU 上以指定顺序执行多个线程称为调度。

如果处理器遇到优先级更高的其他线程，那么当前线程将被推后，并执行具有较高优先级的线程。如果较高优先级的线程停止或变为不可运行，那么下一个较低优先级的线程开始执行。如果线程正在等待 I/O 操作，那么将该线程在队列中将被其他线程推后。当对其他较高优先级线程调用 `sleep()` 方法的时间结束时，线程在队列中也可能被推后。

设置线程优先级

一旦创建了线程，即使用 `Thread` 类的 `setPriority()` 方法设置其优先级。`setPriority()` 方法的签名是：

```
public final void setPriority(int newPriority)
```

在上述签名中，`newPriority` 参数指定线程的新优先级设置。优先级级别应处于 `MIN_PRIORITY` 和 `MAX_PRIORITY` 这两个常量的范围内。`MIN_PRIORITY` 和 `MAX_PRIORITY` 常量的值分别是 1 和 10。如果优先级水平小于 `MIN_PRIORITY` 和大于 `MAX_PRIORITY`，那么 `setPriority()` 方法将抛出 `IllegalArgumentException` 异常。

通过在 `setPriority()` 方法中指定 `NORM_PRIORITY` 常量，可用将线程设置为默认优先级。`MAX_PRIORITY` 是线程可具有的最高优先级。`MIN_PRIORITY` 是线程可具有的最低优先级，`NORM_PRIORITY` 是为线程设置的默认优先级。

您可以使用以下代码设置各种线程的优先级：

```
class ChildThread implements Runnable
{
Thread t;
ChildThread(int p)
{
t = new Thread(this,"ChildThread" );
t.setPriority(p);
System.out.println("Thread created:" + t);
}

public void run()
{
    try
   {
```

```
            for(int i=1;i<=5;i++)
            {
                  System.out.println(t + "loop :"+ i);
                        Thread.sleep(500);
    }
    }
    catch( InterruptedException obj)
    {
            System.out.println("Thread :"+ t + "interrupted");}
    }
}
class PriorityDemo
{
public static void main(String args[])
{
    ChildThread obj1 = new ChildThread(Thread.NORM_PRIORITY - 2);
    ChildThread obj2 = new ChildThread(Thread.NORM_PRIORITY + 2);
    ChildThread obj3 = new ChildThread(Thread.NORM_PRIORITY + 3);

    //Starting the threads with different priority
    obj1.t.start();
    obj2.t.start();
    obj3.t.start();
    try
    {
      System.out.println("Main thread waiting for child thread to
finish");
obj1.t.join();
obj2.t.join();
     obj3.t.join();
}

    catch(InterruptedException e)
    {
            System.out.println("Main thread is interrupted");}
            System.out.println(obj1.t  + "is alive ?:" +
obj1.t.isAlive());
            System.out.println(obj2.t  + "is alive ?:" +
obj2.t.isAlive());
            System.out.println(obj3.t  + "is alive ?:" +
obj3.t.isAlive());
            System.out.println("Main Thread is exiting");
     }
}
```

执行上述代码后，会显示以下输出：

```
Thread created:Thread[ChildThread,3,main]
Thread created:Thread[ChildThread,7,main]
Thread created:Thread[ChildThread,8,main]
Main thread waiting for child thread to finish
Thread[ChildThread,3,main]loop :1
Thread[ChildThread,7,main]loop :1
Thread[ChildThread,8,main]loop :1
Thread[ChildThread,7,main]loop :2
Thread[ChildThread,8,main]loop :2
Thread[ChildThread,3,main]loop :2
Thread[ChildThread,7,main]loop :3
Thread[ChildThread,8,main]loop :3
Thread[ChildThread,3,main]loop :3
Thread[ChildThread,7,main]loop :4
Thread[ChildThread,8,main]loop :4
Thread[ChildThread,3,main]loop :4
Thread[ChildThread,7,main]loop :5
Thread[ChildThread,8,main]loop :5
Thread[ChildThread,3,main]loop :5
Thread[ChildThread,3,]is alive ?: false
Thread[ChildThread,7,]is alive ?: false
Thread[ChildThread,8,]is alive ?: false
Main Thread is exiting
```

活动 5.1：在 Java 中实现线程

练习问题

1. 指出用于设置线程优先级的最小值。

   a. 5
   b. 10
   c. 1
   d. 15

2. 指出用于检查线程存在性的方法。

   a. `isAlive()`
   b. `sleep()`
   c. `syncrhonize()`
   d. `alive()`

3. 指出返回线程名称的方法。

   a. `name()`
   b. `getName()`
   c. `getThreadName()`
   d. `currentThreadName()`

4. 指出用于使线程执行暂停一段时间的方法。

   a. `isAlive()`
   b. `sleep()`
   c. `notify()`
   d. `stop()`

5. 思考以下语句：

   语句 A:不能重新启动死亡的线程。

   语句 B：当线程被 I/O 操作阻止时，它进入“不可运行”状态。

   根据以上语句，以下哪个选项是正确选项？

   a. 两个语句均错误。
   b. 两个语句均正确。
   c. 语句 A 正确，语句 B 错误。
   d. 语句 B 正确，语句 A 错误。

小结

在本章中，您学习了：

- 线程定义为程序的执行路径。它是一个指令序列，执行它可定义唯一的控制流。
- 创建两个或多个线程的程序称为多线程程序。
- 多线程的各种优点有：
  - 改进性能
  - 最小化系统资源使用
  - 程序结构简化
- 多线程的各种缺点有：
  - 竞争情况
  - 死锁
  - 锁饥饿
- `java.lang.Thread` 类用于构造和访问多线程应用程序中的单个线程。
- 在线程的生命周期中的各种状态有：
  - 新生
  - 可运行
  - 不可运行
  - 终止或死亡
- 您可以通过以下方法创建线程：
  - 通过扩展 `Thread` 类
  - 通过实现 `Runnable` 接口
- 线程优先级是介于 1 至 10 之间的整数，其指定一个线程相对于其他线程的优先级的优先级。
- 您可以在创建了线程优先级后，使用 `Thread` 类中声明的 `setPriority()` 方法对其进行设置。