



```

11     private void openDoor()
12     {
13         System.out.println("此方法为 private 方法");
14     }
15
16     public void openDoors()
17     {
18         openDoor();
19     }
20 }
21
22 //wood_Door 继承与 door
23 public class wood_Door extends door
24 {
25     private String wood_color;
26
27     //wood_Door 类的构造器
28     public wood_Door()
29     {
30         this.wood_color = "是本类里的成员变量";
31     }
32
33     public static void main(String[] args)
34     {
35         wood_Door wd = new wood_Door();
36         //访问的是父类的成员变量和方法
37         wd.openDoors();
38         //访问的是本类的成员变量
39         System.out.println(wd.wood_color);
40     }
41 }

```



图 8-3 private 修饰符

【运行结果】使用 `javac` 编译程序将产生一个和该程序对应的 `class` 程序，然后使用 `Java` 运行编译产生的 `class` 程序，运行结果如图 8-3 所示。

【代码解析】`door` 的成员变量 `color` 和方法 `openDoor` 声明为 `private` 类型，那么 `door` 的子类 `A2` 就不能访问到 `door` 的成员变量和方法，因为修饰符 `private` 的关系，没有继承到 `A2` 中。如果要让 `A2` 能够访问到 `door` 的 `private` 类型的成员变量和方法要在父类 `door` 里添加一个 `public` 类型的方法 `openDoors`。方法 `openDoors` 被描述为 `public` 类型，是能够被 `door` 的子类访问的。

### 8.2.3 default：声明成员变量为默认类型

如果不给成员变量添加任何修饰符，就表示这个成员变量被修饰为 `default` 类型。在一个同包里的类或子类是能够访问的，相当于 `public` 类型。但是在不同包里的类或子类没有继承该成员变量，是访问不到它的。

【范例 8-10】下面是使用默认修饰符的程序。

示例代码 8-10

```

01 //程序是放在这个包 a 下面
02 package a;
03
04 class door
05 {
06     String color; //为定义为 public 类型

```

```

07
08     //一个没有任何修饰的方法
09     void openDoor()
10     {
11         System.out.println("此方法没有修饰符,为 default 类型!");
12     }
13 }
14
15 //wood_Door 类继承与 door
16 public class wood_Door extends door
17 {
18     private String wood_color;
19
20     //wood_Door 类的构造器
21     public wood_Door()
22     {
23         this.wood_color = "本类成员变量被修饰为 default";
24     }
25
26     public static void main(String[] args)
27     {
28         wood_Door wd = new wood_Door();
29         //访问的是父类的成员变量和方法
30         wd.openDoor();           //这行代码不能编译通过
31         //访问的是本类的成员变量
32         System.out.println(wd.wood_color);
33     }
34 }

```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序,然后使用 Java 运行编译产生的 class 程序,运行结果如图 8-4 所示。

**【代码解析】**在本类里,成员变量或方法不添加任何修饰符的话就相当于 public 类型。如果子类在另一个包里,而成员变量或方法没有添加修饰符,子类是访问不到的。

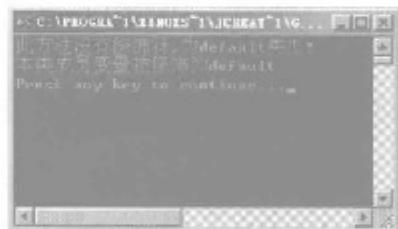


图 8-4 默认修饰符

#### 8.2.4 protected: 声明成员变量为保护类型

protected 表明被它修饰的成员变量为保护类型,在同一包里和 public 类型是一样的,也是能够访问到的。但是如果在不同包里的 protected 类型的成员变量就只能通过子类来访问,这个修饰符是区别于其他的修饰符的。

**【范例 8-11】**下面是使用 protected 修饰符的程序。

示例代码 8-11

```

01 //程序是放在这个包 a 下面
02 package a;
03
04 class door
05 {
06     protected String color; //为定义为 public 类型
07
08     //此方法为修饰为保护类型
09     protected void openDoor()

```



```
10     {
11         System.out.println("protected类型的成员变量,只能在不同包外的子类访问到!");
12     }
13 }
14
15 // wood_door 类继承与 door 类
16 public class wood_door extends door
17 {
18     public static void main(String[] args)
19     {
20         wood_door wd = new wood_door ();
21         //访问的是父类的成员变量和方法
22         wd.openDoor ();
23     }
24 }
```



图 8-5 protected 修饰符

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 8-5 所示。

**【代码解析】**door 的成员变量方法 openDoor 修饰为 protected 类型, 只能通过在不同包下的子类来访问。如果 wood\_Door 和 door 在同一包下, 那么和 public 类型是一样的了。

如果其他类要访问 door 的方法, 方法 openDoor 是访问不到的, 因为对其他类而言相当于 private 类型的。



### 8.3 成员变量的覆盖

正如前面所举爸爸和儿子的例子, 爸爸的眼睛是单眼皮, 儿子的是双眼皮, 不能说儿子没有继承爸爸的特性, 只能说明儿子的特性把爸爸的特性覆盖了。成员变量的覆盖是子类里有和父类里相同的成员变量或方法, 继承的关系, 子类的成员变量将会使用, 而父类的成员变量被保护起来。有时也因修饰符原因而变化, 下面用代码来说明。

**【范例 8-12】**看下面的成员变量覆盖的程序。

#### 示例代码 8-12

```
01 //fruit 被描述为一个水果类
02 class fruit
03 {
04     static String color = "黄色";           //修饰为一个静态的成员变量
05     String size = "大";
06
07     //一个静态方法返回类型为 color, color 的值是什么就返回什么
08     static String getFruitColor ()
09     {
10         return color;
11     }
12
13     //一个静态方法返回类型为 size, size 的值是什么就返回什么
14     String getFruitSize()
```

```

15      {
16          return size;
17      }
18  }
19
20 //apple 类被描述为一个苹果类, 它继承与 fruit
21 public class apple extends fruit
22 {
23     static String appleColor = "绿色";
24     String appleSize = "小";
25
26     //一个静态方法
27     static String getFruitColor()
28     {
29         return appleColor;
30     }
31
32     //一个 default 类型的方法
33     String getFruitSize()
34     {
35         return appleSize;
36     }
37
38     public static void main(String[] args)
39     {
40         fruit f = new apple();           //创建一个 fruit 类型的 apple 对象
41         System.out.println(f.getFruitColor()); //执行 getFruitColor 方法
42         System.out.println(f.getFruitSize()); //执行 getFruitSize 方法
43     }
44 }

```

**【运行结果】** 使用 `javac` 编译程序将产生一个和该程序对应的 `class` 程序, 然后使用 Java 运行编译产生的 `class` 程序, 运行结果如图 8-6 所示。

**【代码解析】** 对象引用 `f` 在编译期的类型为 `fruit`, 当在运行期时为 `apple`, `apple` 是 `fruit` 的子类, `apple` 的方法 `getFruitSize` 被继承, 而在 `fruit` 里有和 `apple` 同名的方法, 那么 `fruit` 的同名方法被覆盖起来, 在调用的时候所调用的是子类 `apple` 的方法, 所以运行的结果是“黄色”。

`apple` 的方法 `getFruitColor` 修饰为 `static` 类型, 表示这个 `getFruitColor` 方法为类方法, 在 `apple` 继承 `fruit` 的时候是没有被覆盖的, `b` 在编译期为 `fruit` 的类型, 读者应该试着理解 `static` 关键字在这里的作用, 如果不能理解请注意后面的具体讲述。



图 8-6 成员变量覆盖

## 8.4 对象引用

对象引用就好比一个人的名字, 是一个代号, 也是为了方便和容易记忆所用的。比如去商店里买水果, 进门就说我要买水果, 而售货员也不知道要买的是什么。在 Java 里可以定义一个类, 这个类里有很多的成员变量和方法, 再给这个类起一个名字, 这个名字就是这个对象的引用。

```
bike b = new bike();
```



代码说明：

- bike b 是创建 Like 类的一个对象应用，而这个 b 就相当于 bike 的名字。
- new bike () 相当于把 bike 这个类实例化了，真实存在于内存当中。

**【范例 8-13】**下面是一个使用对象引用的程序。

示例代码 8-13

```
01 // bike 类，描述的是一个自行车
02 public class bike
03 {
04     public String name;           //自行车的名称
05     public String color;          //自行车的颜色
06
07     public static void main(String args[])
08     {
09         bike b1 = new bike ();    //创建一个 bike 对象实例，对象引用的名字为 b1
10         bike b2 = new bike ();    //创建一个 bike 对象实例，对象引用的名字为 b2
11
12         bike b3 = new bike ();    //创建一个 bike 对象实例，对象引用的名字为 b3
13         bike b4 = b3;           //创建一个 bike 类的对象引用 b4，把 b3 的对象应用赋给 b4
14     }
15 }
```

**【代码解析】**b1 和 b2 分别创建了两个 bike 对象。对象的引用分别为 b1 和 b2，是 bike 类的实例化的引用名称。所谓实例化，是让 bike 对象在内存中创建起来供 b1 和 b2 这两个对象引用所使用，虽然 b1 和 b2 创建的都是 bike 类，但它们不是同一个对象，就是说 b1 和 b2 是两辆自行车，没有任何关系。

b3 和 b4 的关系就比较特殊了，b3 是创建了 bike 对象的对象引用，在内存中创建了这个 bike 类的实体，b4 只是创建了一个引用，引用的是 bike 这个类，但没有在内存中创建。写成 b4 = b3 就表示为把 b3 这个对象引用所指向的对象 bike 赋给 b4，让 b4 在内存中也指向这个 bike，在 b3 操作这个 bike 对象的时候，b4 也能体现出来，就好比这个自行车有两个名字。



## 8.5 方法的重写和重载

方法的重写和重载是体现继承特性的重要方面，理解了方法的重写和重载，可以为以后学习多态打下基础，本节重点学习重写和重载的用法和区别。

### 8.5.1 方法重写的特点

自行车和公路赛车都是靠外力移动的，这一点二者是相同的。公路赛车继承了这一特点，但公路赛车的移动速度就不相同了，移动的快慢是由它们各自移动特性不相同决定的，方法继承的特点和成员变量的覆盖很类似，但也有特殊情况，方法重写也可以叫方法的覆盖，关键字为 `override`。

**【范例 8-14】**下面用例子说明在日常生活中自行车和公路赛车的相同点和不同点。

示例代码 8-14

```
01 // bike 类所描述的是一个自行车
02 class bike
```

```

03  {
04      //自行车的移动速度
05      public void move()
06      {
07          System.out.println("速度慢的！");
08      }
09  }
10
11 //racing_cycle类所描述的是一个公路赛车，继承了bike类
12 public class racing_cycle extends bike
13 {
14     //公路赛车的移动速度
15     public void move()          //继承的关系 racing_cycle类重写了 move 方法
16     {
17         System.out.println("速度快的！");
18     }
19
20     public static void main(String args[])
21     {
22         racing_cycle rc = new racing_cycle(); //创建了一个公路赛车的对象，对象引用为 rc
23         rc.move();                         //执行了 move 方法
24     }
25 }

```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 8-7 所示。

【代码解析】racing\_cycle 类继承了 bike 类，有 bike 类的所有公共特性。racing\_cycle 类继承了 bike 类的方法 move，但是在 racing\_cycle 类里进行了改进，移动速度比 bike 类快了，很自然地把移动速度慢的特性重写了或者说覆盖了，因此结果为速度快的。如果把 racing\_cycle rc = new racing\_cycle(); 改变成 bike rc = new racing\_cycle(); 会变成什么结果呢，请读者考虑一下。

【范例 8-15】下面通过例子说明类方法的重写，请读者注意观察方法前面的修饰符 static，考虑在这里有什么用处。

#### 示例代码 8-15

```

01 // bike类所描述的是一个自行车
02 class bike
03 {
04     //自行车的移动速度
05     public static void move()          //注意此方法被修饰为了 static
06     {
07         System.out.println("速度慢的！");
08     }
09 }
10
11 //racing_cycle类所描述的是一个公路赛车，继承与 bike 类
12 public class racing_cycle extends bike
13 {
14     //公路赛车的移动速度
15     public static void move()          //注意此方法被修饰为了 static，并重写 move 方法
16     {

```



图 8-7 重写特点



```

17         System.out.println("速度很快的！");
18     }
19
20     public static void main(String args[])
21     {
22         bike rc = new racing_cycle(); //注意此处的用法，体会有什么好处
23         rc.move(); //执行相应的方法 move
24     }
25 }

```

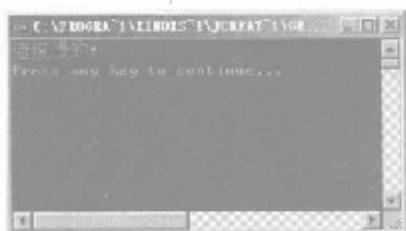


图 8-8 static 修饰符的使用

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 8-8 所示。

【代码解析】这里用了 static 关键字，先简单的介绍一下，static 表示为静态的，只能有一份，是通过类调用的。创建了一个对象引用 rc，这个对象引用在编译期时的状态为 bike 类型，但是到了运行期的状态变为 racing\_cycle 类型的了，而方法 move 被声明为 static 类型的，修饰为类方法。类方法是不能重写的，因为没有继承过来，所以运行的还是 bike 类里的 move 方法。

### 8.5.2 方法重载的特点

方法的重载就好比日常生活中人的名字，有大名也有小名，但这些名字都指的是这个人，不同点就是让这个人去做的事情可能不一样，如图 8-9 所示。

这和 Java 里的重载很相似，关键字为 overload。下面看一下重载的要求。

- 重载的方法名称相同，但方法的参数列表不相同。如参数个数和参数类型等。
- 重载的方法的返回值可以相同也可以不相同。

示例代码如下：

```

public String move();
public String move(String name);
public void move(String name, int spend);

```

代码说明：

- 虽然三种方法的名称是相同的，但这三种方法的参数列表，即个数和类型，是不相同的。
- 判断方法是否是重载，看参数列表是很重要的。

什么是传递基本类型，所谓基本类型就是用于数学计算的那些类型。当有两个名称一样的方法时，根据传递数值的类型来匹配哪两个方法的参数列表是相同的。

【范例 8-16】下面是描述重载特点的程序。

#### 示例代码 8-16

```

01 //math 类，描述的是一个数学类，用于数学计算
02 class math
03 {
04     //重载方法 add，参数列表为 int 型的
05     public int add(int i, int j)
06     {

```



图 8-9 重载

```

07         return i + j;
08     }
09
10     //重载方法 add, 参数列表为 float 型的
11     public float add(float i, float j)
12     {
13         return i + j;
14     }
15
16     public static void main(String args[])
17     {
18         math m = new math();           //创建了一个对象实例 math, 它的对象引用为 m
19         System.out.println(m.add(1.0f, 2.3f));    //执行重载方法, 根据添加参数的类型来执行那个方法
20         System.out.println(m.add(1, 2));
21     }
22 }

```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 8-10 所示。

【代码解析】在对象引用 m 执行方法 add 的时候添加的参数为 1.0f, 这个 f 表示这个参数的类型为一个浮点参数, 第二个参数同样也是浮点类型的, 所以方法 add 将调用的是参数列表为 float 浮点类型的方法。在对象引用 m 执行方法 add 的时候添加的参数为 1,

这个参数的类型为一个整型参数, 第二个参数同样也是整型类型的, 所以方法 add 将调用的是参数列表为 int 整型类型的方法。

请读者考虑, 如果把方法里的参数换成不同类型, Java 将怎样选择执行哪个方法呢?

接下来再看一下方法传递引用类型的重载, 对象引用就是在创建一个对象后, 用一个名称去表示这个对象的代号。就好比 math m = new math(); m 就是 math 对象实例的对象引用。

【范例 8-17】下面用代码说明方法传递引用类型的重载。

示例代码 8-17

```

01 //dec1 类主要是为了存放两个数值, 没有其他含义
02 class dec1
03 {
04     int m = 2;
05     int n = 3;
06 }
07
08 //dec2 类主要是为了存放两个数值, 没有其他含义
09 class dec2
10 {
11     float f = 2.6f;
12     float l = 3.1f;
13 }
14
15 //math 类, 描述的是一个数学类, 用于数学计算
16 class math
17 {
18     //重载方法 add, 参数列表为 dec1 类的对象引用

```

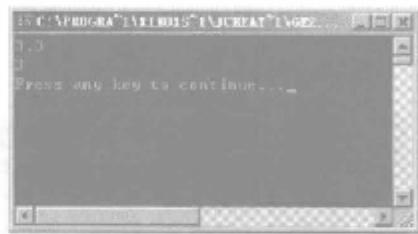


图 8-10 重载特点



## 21 天学通 Java

```
19     public int add(Dec1 d)
20     {
21         return d.m + d.n;
22     }
23
24     //重载方法 add, 参数列表为 dec1 类的对象引用和 dec2 类的对象引用
25     public float add(Dec1 d, Dec2 j)
26     {
27         return d.m + j.f;
28     }
29
30     public static void main(String args[])
31     {
32         Math m = new Math();           //创建了 Math 类的对象实例, 它的对象引用为 m
33         Dec1 d1 = new Dec1();         //创建了 Dec1 类的对象实例, 它的对象引用为 d1
34         Dec2 d2 = new Dec2();         //创建了 Dec2 类的对象实例, 它的对象引用为 d2
35         System.out.println(m.add(d1)); //执行重载方法, 根据 add 方法的要求, 要传递
36                                         //一个对象引用
37         System.out.println(m.add(d1, d2));
38     }
```



图 8-11 传递引用类型重载

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 8-11 所示。

【代码解析】和基本类型的参数一样, 参数列表和哪个方法的参数列表相同就执行哪个。如没有找到完全一样的那个方法的参数列表, 就选择能转换的那个方法, 也就是最兼容的那个。

传递引用类型有以下好处。

- 更能体现面向对象编程的特性, 对敏感数据有很好的保护性, 只需知道类的名称就可完成操作。
- 可以减少代码的复用, 使耦合度降低。

### 8.5.3 重写的返回类型

方法有它的返回类型, 而重写的方法也有返回类型, 并有一些相应的限制。方法被重写后, 返回的类型为基本类型时, 重写方法的返回要必须一样, 否则会出现错误。

【范例 8-18】下面是演示重写返回类型的代码。

示例代码 8-18

```
01 //Math 类所描述的是一个数学类
02 class Math
03 {
04     //一个方法 add, 返回的是两个 int 类型的数值相加的结果
05     public int add()
06     {
07         return 10 + 15;           //执行整型计算
08     }
09 }
10
11 //Son 类所描述的是用于浮点计算的
```

```

12  public class Son extends math
13  {
14      public float add()
15      {
16          return 6.32f + 15.6f;          //执行浮点计算
17      }
18
19  public static void main(String args[])
20  {
21      math m = new Son();          //创建一个 Son 类的对象实例, 对象引用的名称为 m
22      System.out.println(m.add()); //打印并显示结果
23  }
24 }

```

**【代码解析】**重写就是把父类的方法继承过来，并改写其方法体，让其适应新的需要，在上面的代码里方法 add 的返回参数是不相同，所以编译不能通过。把方法 add 的返回类型改成一样的类型就能通过编译，打印出结果来。

**【范例 8-19】**修改上面的代码，使程序能够正常运行。

示例代码 8-19

```

01 //math 类所描述的是一个数学类
02 class math
03 {
04     //一个方法 add, 返回的是两个 int 类型的数值相加的结果
05     public int add()
06     {
07         return 10 + 15;            //执行整型计算
08     }
09 }
10
11 //Son 类所描述的是用于浮点计算的
12 public class Son extends math
13 {
14     public int add()
15     {
16         return 6 + 15;            //执行浮点计算
17     }
18
19 public static void main(String args[])
20 {
21     math m = new Son();          //创建一个 Son 类的对象实例, 对象引用的名称为 m
22     System.out.println(m.add()); //打印并显示结果
23 }
24 }

```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 8-12 所示。

**【代码解析】**从运行结果中可以看出，使用的是子类中的方法。这里需要注意的是返回类型必须是一样的。

接下来看一下方法返回的是引用类型的内容。方法被重写后，方法返回的数据类型必须相同，否则就是父类的子类。

**【范例 8-20】**看下面方法重写的程序。

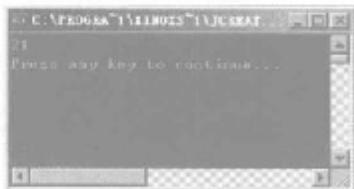


图 8-12 修改后



示例代码 8-20

```
01 //door 类所描述的是一个门
02 class door
03 {
04
05 }
06
07 //wood_Door 类所描述的是一个木门
08 class wood_Door extends door
09 {
10
11 }
12
13 class math
14 {
15     //父类 math 类的方法，返回的类型的 door 类的对象引用
16     public door getMes()
17     {
18         return new door();
19     }
20 }
21
22 //Son 类继承与 math 类
23 public class Son extends math
24 {
25     //wood_Door 方法重写了父类的 getMes 方法，返回的类型是 wood_Door 类的对象引用
26     public wood_Door getMes()
27     {
28         return new wood_Door();
29     }
30
31     public static void main(String args[])
32     {
33         math m = new Son();          //创建 Son 类的对象实例，Son 类的对象引用为 m
34         System.out.println(m.getMes()); //执行相应方法并打印出来。
35     }
36 }
```

**【代码解析】**方法返回引用类型和返回基本类型很相似，返回类型要相同。如果不相同返回，类型也必须是父类的派生类也就是说是子类。

#### 8.5.4 重写是基于继承的

重写和重载的产生是基于继承的，如果没有发生继承就不会产生重写和重载了。举一个例子来说，自行车通过外力可以移动，而公路赛车通过外力也可以移动，公路赛车继承了自行车的特性。也可以说成，公路赛车是自行车的一个升级版本。自行车相当于公路赛车的一个参照点吧，如图 8-13 所示。

图 8-13 自行车和公路赛车

自行车

公路赛车

【范例 8-21】下面举一个重写基于继承的例子。

示例代码 8-21

```
01 //把上个例子加以修改
02 //door 类所描述的是一个门
03 class door
04 {
05
06 }
07
```

```

08 //wood_Door 类所描述的是一个木门
09 class wood_Door extends door
10 {
11
12 }
13
14 class math
15 {
16     //getMes 方法被修饰为了 static public 类型, 请注意
17     static public door getMes()
18     {
19         return new door();
20     }
21 }
22
23 public class Son extends math
24 {
25     //getMes 方法被修饰为了 static public 类型, 请注意
26     static public wood_Door getMes()
27     {
28         return new wood_Door();
29     }
30
31     public static void main(String args[])
32     {
33         //创建了一个 Son 类的对象实例, 对象引用的名称 m
34         math m = new Son();
35         //打印并显示结果
36         System.out.println(m.getMes());
37     }
38 }

```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 8-14 所示。

**【代码解析】**Son 类继承与 math 类, 应该继承了 math 类的所有公共成员变量和方法, 在 Son 类里有相同的方法名称, 产生了重写。请大家注意父类和子类的 getMes 是被修饰为 static 类型的, static 关键字表示这个方法为静态方法, 具有唯一性, 是个类方法。因为 static 类型的方法不能被重写, 所以方法 m.getMes 调用的是父类的方法 getMes。



图 8-14 基于继承

### 8.5.5 静态方法是不能重写的

静态方法就是被修饰为了 static 类型的方法, 在类里声明具有唯一性, 不是通过类的实例化而存在的, 而是通过类的建立而存在, 可以理解为用关键字 new 创建对象了, 就是把这个对象实例化了。

**【范例 8-22】**下面通过代码演示来说明静态方法不能被重写。

#### 示例代码 8-22

```

01 //door 类所描述的是一个门
02 class door
03 {
04     //没有任何类体, 是个空类

```



```
05  }
06
07 class wood_Door extends door
08 {
09     //没有任何类体，是个空类
10 }
11
12 class math
13 {
14     //getMes 方法被描述为 static 类型的静态方法
15     static public door getMes()
16     {
17         return new door();
18     }
19 }
20
21 //Son 类继承了 math
22 public class Son extends math
23 {
24     //getMes 方法被描述为 static 类型的静态方法，注意此方法没有重写父类的方法，请考虑
25     static public wood_Door getMes()
26     {
27         return new wood_Door();
28     }
29
30     public static void main(String args[])
31     {
32         //创建了 Son 类的对象实例，对象引用的名称是
33         Son m = new Son();
34         //打印结果并显示出来
35         System.out.println(m.getMes());
36     }
37 }
```

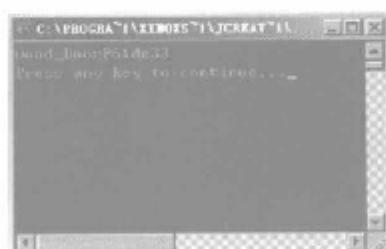


图 8-15 静态方法不能重写

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 8-15 所示。

**【代码解析】**经过前面的学习，读者知道了继承的特性，可以让子类继承父类的公共成员变量和方法，并经过重写可以适应新的需要。static 关键字可以修饰方法的访问权限，这里的方法 getMes 是一个静态方法，在这个类里只有一份，不能通过实例化这个类而创建第二份，是个类方法。因为方法 getMes 被修饰为了类方法，子类就不能通过继承来重写这个方法了。这段代码里只是隐藏了父类的方法。

**【范例 8-23】**下面再来看一个相关静态方法的程序。

### 示例代码 8-23

```
01 //door 类所描述的是一个门
02 class door
03 {
04     //没有任何类体，是个空类
05 }
06
07 class wood_Door extends door
```

```

08  {
09      //没有任何类体，是个空类
10  }
11
12 class math
13 {
14     //getMes 方法被描述为 static 类型的静态方法
15     static public door getMes()
16     {
17         return new door();
18     }
19 }
20
21 //Son 类继承了 math
22 public class Son extends math
23 {
24     //getMes 方法被描述为 static 类型的静态方法，注意此方法没有重写父类的方法，请考虑
25     static public wood_Door getMes()
26     {
27         return new wood_Door();
28     }
29
30     public static void main(String args[])
31     {
32         //创建了 Son 类的对象实例，对象引用的名称是 math 类型的 m
33         math m = new Son();
34         //打印结果并显示出来
35         System.out.println(m.getMes());
36     }
37 }

```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 8-16 所示。

【代码解析】这段代码和上面的代码很相似，只有一个不同点，但是代码的含义已经差了很多。上面一段代码编译期的对象引用的类型为 Son 类型，而现在这段代码在编译期为 math 类型。因为编译期为 math 类型，又因为方法 getMes 被修饰为 static 类型的。所以对象引用 m 在运行期被强制转型为 math 类型。

对本节的内容进行如下总结。

- 父类的静态方法可以被子类的静态方法覆盖。
- 父类的非静态方法不能被子类的静态方法覆盖。
- 父类的静态方法不能被子类的非静态方法覆盖。
- 覆盖是用于父类和子类之间。
- 重写是用在同一个类中，有相同的方法名，但参数不一样。

### 8.5.6 三者之间的关系

重写的关键字是 override，重载的关键字为 overload，重写、重载、覆盖都是基于继承的关系。当继承的关系发生了，想用父类的方法就用 super 关键字来引用，如果想用新的方法了就重写下，来完成新的功能需要。



图 8-16 变换类型



对覆盖总结如下：

- 覆盖的方法的参数列表必须要和被覆盖的方法的参数列表完全相同，才能达到覆盖的效果。
- 覆盖的方法的返回值必须和被覆盖的方法的返回值一致。
- 覆盖的方法所抛出的异常必须和被覆盖方法的所抛出的异常一致，或者是其子类。
- 被覆盖的方法不能为 `private`，否则在其子类中只是新定义了一个方法，并没有对其进行覆盖。

对重载总结如下：

- 使用重载时只能定义不同的参数列表。
- 不能通过重载的方法的返回类型、访问权限和抛出的异常来进行方法的重载。

对重写总结如下：

- 重写的方法存在于父类中，也存在于子类中。
- 重写的方法在运行期采用多态的形式。
- 重写的方法不能比被重写的方法有更高的访问限制。
- 重写的方法不能比被重写的方法有更多的异常。

### 8.5.7 重写 `toString` 方法

`toString` 方法是 Java 里 `Object` 类的方法，很多类都重写了该方法，该方法返回对象的状态信息。下面是这个方法的原型。

```
public String toString()
```

**【范例 8-24】**下面是一个重写了方法 `toString` 的例子。

示例代码 8-24

```
01 //racing_cycle 类所描述的是公路赛车
02 public class racing_cycle
03 {
04     //racing_cycle 类的构造器
05     public racing_cycle()
06     {
07     }
08 }
09
10 //公路赛车的移动速度
11 public void move()
12 {
13     System.out.println("速度快的！");
14 }
15
16 //下面的方法 toString 重写 Object 类的 toString 方法
17 public String toString()
18 {
19     return "此对象描述的是公路赛车";
20 }
21
22 public static void main(String args[])
23 {
24     //创建了 racing_cycle 类的对象实例，对象引用的名称为 rc
25     racing_cycle rc = new racing_cycle();
```

```

26         //打印对象引用 rc，并显示结果
27         System.out.println(rc);
28     }
29 }

```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 8-17 所示。

【代码解析】在平常的开发中，大多数时候都是用多态的形式调用方法 `toString`。无论什么类型的对象引用都可以重写方法 `toString`。

### 8.5.8 重写 `equals` 方法

方法 `equals` 也是 `Object` 类的方法，很多类也进行了重写，一般重写 `equals` 方法是为了比较两个对象的内容是否相等。下面是这个方法的原型。

```

public boolean equals (Object obj)
{
    return (this == obj);           //这里比较的是两个对象的引用
}

```

【范例 8-25】下面是一个重写了 `equals` 方法的范例。

示例代码 8-25

```

01 //racing_cycle 类所描述的是公路赛车
02 public class racing_cycle
03 {
04     public String color;           //颜色
05     public String speed;          //速度
06
07     public racing_cycle()        //无参的构造器
08     {
09     }
10
11
12     public racing_cycle(String color, String speed)    //带两个参数的构造器
13     {
14         this.color = color;
15         this.speed = speed;
16     }
17
18     public static void main(String args[])
19     {
20         racing_cycle rc1 = new racing_cycle("黄色", "快"); //用带两个参数的
21         //构造器创建两个对象 rc1,rc2
22         racing_cycle rc2 = new racing_cycle("绿色", "慢");
23
24         //比较对象引用 rc1 和对象引用 rc2 的值是否相等
25         if (rc1.equals(rc2))
26         {
27             System.out.println("两个对象的值相同");
28         }
29         else
30         {
31             System.out.println("两个对象的值不相同");
32         }

```

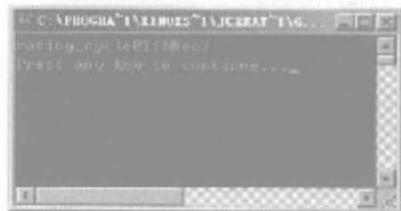


图 8-17 重写 `toString` 方法



```
33 }  
34 }
```

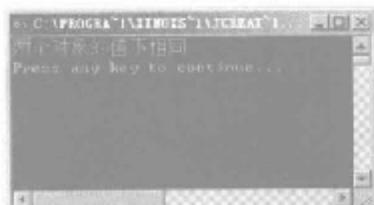


图 8-18 重写 equals 方法

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 8-18 所示。

【代码解析】对象引用 rc1 和对象引用 rc2 进行比较，对象引用就相当于内存的地址，地址不是重复，当然不是同一个引用了。如果用方法 equals 比较两个对象，不能比较内容相等时，就是没有重写该方法。

**【范例 8-26】**下面是一个重写 equals 方法的类的程序。

示例代码 8-26

```
01 //racing_cycle 类所描述的是公路赛车  
02 public class racing_cycle  
03 {  
04     public static void main(String args[])  
05     {  
06         Integer i = 2;    //Integer 为封装类，i 为对象引用  
07         Integer j = 2;  
08  
09         if(i.equals(j))  
10         {  
11             System.out.println("两个对象的值相同");  
12         }  
13         else  
14         {  
15             System.out.println("两个对象的值不相同");  
16         }  
17     }  
18 }
```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 8-19 所示。

【代码解析】在该程序中使用的是 Integer 类。在程序中定义了两个 Integer 类，并为它们都指定了引用。在程序中还使用 equals 方法来判断两个 Integer 类是否相同。



图 8-19 重写 equals 方法的类



## 8.6 final 与继承的关系

final 关键字有最终、不变的意思，可以修饰成员变量，也可以修饰方法和类，通过 final 关键字的修饰可以改变其特性。

- final 关键字修饰类时，说明其类不能有子类，也就是说该类不能被继承，该类的成员变量在这里将不起作用。
- final 关键字修饰方法时，说明该方法不能被重写，因为类都不能继承了，方法就更不能重写了。
- 类里可以含有 final 关键字修饰的方法。

- final 关键字修饰的成员变量的对象引用不能修改。
- final 关键字修饰的类里的方法默认被修饰为 final。

**【范例 8-27】**下面是说明 final 和继承的关系的一个程序。

示例代码 8-27

```

01  public class racing_cycle
02  {
03      //成员变量 i 被声明为 final、static 类型的，表示为常量，不能改变其值和引用
04      public final static int i = 11;
05
06      public static void main(String args[])
07      {
08          racing_cycle rc = new racing_cycle();
09          //这里将提示编译错误
10          i = i + 1;
11          //打印其结果
12          System.out.println(i);
13      }
14  }

```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行将发生异常。

**【代码解析】**该程序在运行时是会发生异常的。因为在前面将 i 变量定义为 final 类型，从而该值是不可以改变的。而在 main 方法中要将 i 变量增加 1，所以会出现异常的。



## 8.7 abstract 与继承的关系

abstract 关键字是表示抽象的意思。所谓抽象，就好比在日常生活中人们设计的图纸，而这个图纸就好比是一个抽象的房子似的，需要把房子盖起来实现这个图纸。在 Java 里，抽象类里最少要含有一个抽象方法，让它的子类去实现这个抽象方法，抽象类里也可以定义一些方法。下面用例子来演示。

**【范例 8-28】**下面是定义抽象类的程序。

示例代码 8-28

```

01  //bike 类所描述的是一个自行车，被声明为 abstract 抽象的
02  abstract class bike
03  {
04      public String name = "抽象类的成员变量";
05
06      public String getMessage()
07      {
08          //返回 bike 类里的成员变量 name 的值
09          return name;
10      }
11
12      //注意这个 public 类型的方法 getMes，被修饰为了 abstract
13      abstract public String getMes();
14  }
15
16  //racing_cycle 类所描述的是公路赛车，继承与 bike 类，bike 类是抽象了，并实现了相应的抽象
17  //方法
18  public class racing_cycle extends bike

```



```

18  {
19      //实现了父类 bike 的抽象方法 getMes
20      public String getMes()
21      {
22          return getMessage();
23      }
24
25      public static void main(String args[])
26      {
27          racing_cycle rc = new racing_cycle();
28          //打印并显示运行结果
29          System.out.println(rc.getMes());
30      }
31  }

```

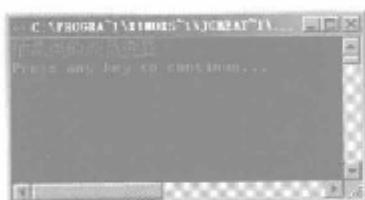


图 8-20 抽象类

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 8-20 所示。

### 【代码解析】

- 从此例子可以看出, 父类 bike 被修饰为 abstract, 是一个抽象类, 继承于 bike 的子类 racing\_cycle, 实现了父类的抽象方法。
- 如果是抽象类的子类就必须实现抽象类的抽象方法。对于接口就好比是一个功能的模型。
- 抽象类是不能实例化的, 只能去实现所有的抽象方法, 否则就只能还是一个抽象类。
- 抽象类与普通继承类的区别就是, 使用父类的方法、重写父类的方法或通过父类与子类相互引用来完成功能。
- 继承与抽象的相同点与不同点。
- 继承了抽象类就必须去实现该抽象类的方法。
- 如果抽象类的子类没有实现父类的抽象方法, 就必须为抽象类。
- 抽象类和继承一样, 都必须是单继承, 或是多层继承。



## 8.8 什么是多态

这里拿苹果和馒头来打个比方, 苹果属于水果的一种, 馒头属于面食的一种, 而苹果和馒头都可以属于食物。一种物品有两种状态表现, 这就是多态, 如图 8-21 所示。

所谓多态, 就好比一个物品在一个时期有两种状态一样。在 Java 里, 通过运用多态可以使对象有更大的灵活性, 这一点仅通过继承无法实现, 但是多态也是基于继承的。

【范例 8-29】下面是使用多态的程序。

### 示例代码 8-29

```

01 //fruit 类所描述的是水果
02 class fruit

```



图 8-21 多态

```

03  {
04  //fruit类的方法getMes
05  public void getMes()
06  {
07      System.out.println("父类");
08  }
09 }
10
11 //apple类所描述的是苹果,继承与fruit类
12 class apple extends fruit
13 {
14 //apple类的方法getMes
15 public void getMes()
16 {
17     System.out.println("apple子类");
18 }
19 }
20
21 //orange类所描述的是橘子,继承与fruit类
22 class orange extends fruit
23 {
24 //orange类的方法getMes()
25 public void getMes()
26 {
27     System.out.println("orange子类");
28 }
29 }
30
31 public class racing_cycle
32 {
33 //方法show的参数为一个父类型
34 public void show(fruit f)
35 {
36     f.getMes();
37 }
38
39 public static void main(String args[])
40 {
41     //创建相应的对象实例
42     racing_cycle rc = new racing_cycle();
43     fruit f = new fruit();
44     apple a = new apple();
45     orange o = new orange();
46
47 //把apple类的对象引用a传入racing_cycle类的方法show里
48     rc.show(a);
49     rc.show(o);
50 }
51 }

```

**【运行结果】**使用javac编译程序将产生一个和该程序对应的class程序,然后使用Java运行编译产生的class程序,运行结果如图8-22所示。

**【代码解析】**将子类的对象引用a传递给父类的对象引用f,然后调用相应的对象引用的重写方法。子类的对象引用是通过在运行期动态绑定才对应到相应的子类的方法上的。



图8-22 使用多态



对本节学习的内容总结如下：

- static 修饰的方法和 final 修饰的方法是在编译期绑定的；而其他的方法是在运行期绑定，动态地判断是什么类型。
- 多态是基于继承的，是类和接口相结合来实现的。



## 8.9 什么是枚举类

所谓枚举就好比日常生活中的星期一、星期二到星期天，是一组连续的数据。在 Java 里，枚举类就是一组连续的基本类型的数值。下面举例如何创建枚举类型。

```
public enum Grade
{
    A, B, C, D, E, F
};
```

**【范例 8-30】**下面是一个枚举类的完整例子。

示例代码 8-30

```
01 //声明为一个枚举类型，对象引用的民成为 colors
02 enum colors(green, yellow)
03
04 //racing_cycle 类所描述的是公路赛车
05 public class racing_cycle
06 {
07     public colors color;      //创建一个枚举类型的对象引用为 color
08     public String speed;      //公路赛车的速度
09
10    //racing_cycle 类的两个参数的构造器
11    public racing_cycle(colors color, String speed)
12    {
13        this.color = color;
14        this.speed = speed;
15    }
16
17    //重写了 object 类的 toString 方法
18    public String toString()
19    {
20        //创建一个 String 类型的临时成员变量 emp
21        String emp = "";
22
23        switch(color)
24        {
25            case green:
26                emp = "绿色";
27                break;
28            case yellow:
29                emp = "黄色";
30                break;
31        }
32
33        return emp + " : " + speed;
34    }
35
36    public static void main(String args[])
37    {
38        //创建 racing_cycle 类的对象实例，对象引用名称为 rc
39        racing_cycle rc = new racing_cycle(colors.green, "快的");
```

```

40         //打印，并显示重写后的toString方法
41         System.out.println(rc);
42     }
43 }

```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 8-23 所示。

【代码解析】枚举类型就好比一个类封装了一些成员变量，提高了开发的效率，避免了一些编译期的错误。当对象的取值在确定的情况下，用枚举会很方便。



图 8-23 枚举

## 8.10 什么是反射机制

在日常生活中，通过放大镜可以看清楚物体的内部结构。在 Java 中，反射机制就起到放大镜的效果，可以通过类名，加载这个类，显示出这个类的方法等信息。

【范例 8-31】下面是讲解反射机制的一个程序。

示例代码 8-31

```

01 //引入反射机制必备的 reflect 包
02 import java.lang.reflect.*;
03
04 public class racing_cycle
05 {
06     public static void main(String args[])
07     {
08         try
09         {
10             //在运行期动态加载 Java.util.Date 下面的类
11             Class c = Class.forName("java.util.Date");
12             //把类里的各个方法写入到数组m里
13             Method m[] = c.getDeclaredMethods();
14             //通过循环把数组里的方法名称显示出来
15             for (int i = 0; i < m.length; i++)
16             {
17                 //把名称打印出来。
18                 System.out.println(m[i].toString());
19             }
20         }
21         catch (Throwable e)
22         {
23             System.err.println(e);
24         }
25     }
26 }

```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果会出现很多内容，比较重要的内容如下所示。

```

public int java.util.Date.hashCode()
public int java.util.Date.compareTo(java.util.Date)
public volatile int java.util.Date.compareTo(java.lang.Object)
public java.lang.Object java.util.Date.clone()
public boolean java.util.Date.equals(java.lang.Object)
public java.lang.String java.util.Date.toString()
private void java.util.Date.writeObject(java.io.ObjectOutputStream) throws

```



```

java.io.IOException
    private void java.util.Date.readObject(java.io.ObjectInputStream) throws java.
    io.IOException,java.lang.ClassNotFoundException
        public static long java.util.Date.parse(java.lang.String)
        private static final sun.util.calendar.BaseCalendar$Date
    java.util.Date.normalize(sun.util.calendar.BaseCalendar$Date)
        private final sun.util.calendar.BaseCalendar$Date java.util.Date.normalize()
        public boolean java.util.Date.after(java.util.Date)
        public boolean java.util.Date.before(java.util.Date)
    .....

```

【代码解析】使用 `class.forName` 方法载入指定的类，通过方法 `getDeclaredMethods` 取得这个类的所有方法并把其存入一个数组里，通过循环打印并显示出来。`java.lang.reflect.*` 这个包下面的类是用来获得类的构造函数、方法等分析的能力。



## 8.11 什么是泛型

在日常生活中，橡皮泥通过外力可以改变其形状，其形状是不固定的。在 Java 中，通过泛型可以给开发带来方便，通过参数的指定，可以改变其类型。下面通过代码演示。

【范例 8-32】下面是一个没有使用泛型的例子。

示例代码 8-32

```

01 //引入集合 ArrayList 的类包
02 import java.util.ArrayList;
03
04 public class racing_cycle
05 {
06     //String 类型的成员变量 n
07     public String n = "100";
08
09     public static void main(String args[])
10     {
11         //创建 racing_cycle 类的对象实例，对象引用为 rc
12         racing_cycle rc = new racing_cycle();
13         //创建一个 ArrayList 的对象集合，对象引用为 al
14         ArrayList al = new ArrayList();
15         //向集合 al 里添加元素，也就说把 n 的值 100 添加进去
16         al.add(rc.n);
17         //在取出来转型赋给 s
18         String s = (String)al.get(0);
19         //打印并显示结果
20         System.out.println(s);
21     }
22 }

```

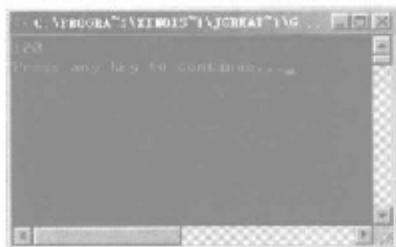


图 8-24 没有使用泛型

【运行结果】使用 `javac` 编译程序将产生一个和该程序对应的 `class` 程序，然后使用 Java 运行编译产生的 `class` 程序，运行结果如图 8-24 所示。

【代码解析】此段代码的主要点在 `String s = (String)al.get(0);` 这里。 `ArrayList` 集合是有序的集合，用方法 `al.add` 添加元素。因为 `al.get` 方法的返回类型为 `Object` 类型的，而要赋值的对象引用为 `String` 类型，

所以要转型。

**【范例 8-33】** 使用泛型来修改上面的代码。

示例代码 8-33

```

01 //引入集合 ArrayList 的类包
02 import java.util.ArrayList;
03
04 public class racing_cycle
05 {
06     //String 类型的成员变量 n
07     public String n = "100";
08
09     public static void main(String args[])
10     {
11         //创建 racing_cycle 类的对象实例, 对象引用为 rc
12         racing_cycle rc = new racing_cycle();
13         //创建一个 ArrayList 的对象集合, 对象引用为 al
14         ArrayList<String> al = new ArrayList<String>();
15         //向集合 al 里添加元素, 也就是说把 n 的值 100 添加进去
16         al.add(rc.n);
17         //在取出来转型赋给 s
18         String s = al.get(0);
19         //打印并显示结果
20         System.out.println(s);
21     }
22 }
```

**【运行结果】** 使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 8-25 所示。

**【代码解析】** 运行结果和上面代码的运行结果一样。 ArrayList 集合默认接收的元素类型为 Object 类型, 只是通过泛型对这个 ArrayList 集合里的元素的类型进行了设置。因为 ArrayList 集合默认的元素类型为 String 类型了, 在存入新元素和取出元素的时候当然不用转型了。

使用泛型给程序员的代码编写带来了好处, 也带来的缺点, 了解它的好处和缺点, 能给程序编写带来很多好处和便利。对泛型的好处总结如下:

- 使用泛型, 正如上面代码所示, 能使代码看起来灵活; 容易管理, 不容易产生错误。
- 使用泛型能使代码量减少, 能产生很多公共代码。
- 使用泛型在代码编译的时候能进行类型的检查并自动转换, 使代码的运行效率得到提高。
- 使用泛型在编译进行自动转换的时候出现了错误, 会进行错误提示。
- 使用泛型的时候参数只能是类的类型, 不能是简单类型。
- 使用泛型的时候参数可以有多个。
- 使用泛型的时候参数也能继承别的类型。



图 8-25 使用泛型



## 8.12 综合练习

### 1. 4 种权限修饰符的不同点有哪些？

【提示】从基本定义中进行分析。public 修饰符表明被它修饰的成员变量为公共类型，这样这个成员变量在任何包里都能访问，包括子类也能访问到。private 表明被它修饰的成员变量为私有类型，表示除了本类外任何类都不能访问到这个成员变量，具有很好的保护性。如果不给成员变量添加任何修饰符，就表示这个成员变量被修饰为 default 类型，在同一个包里的类或子类是能够访问的，就相当于 public 类型，但是在不同包里的类或子类没有继承该成员变量，是访问不到的。protected 表明被它修饰的成员变量为保护类型，在同一包里和 public 类型是一样的，也是能够访问到的，但是如果在不同包里的 protected 类型的成员变量就只能通过子类来访问，这个修饰符是区别于其他修饰符的。

### 2. 重写和重载的区别有哪些？

【提示】重写是基于继承的，重写是重写父类中的方法，从而在子类中出现一个和该方法相同名称的方法。重载的方法名称相同，但方法的参数列表不相同。如参数个数和参数类型等。重载的方法的返回值可以相同也可以不相同。



## 8.13 小结

通过本章的学习，读者可以了解继承的相关概念、用法和注意事项等。对修饰符所修饰的成员变量和方法应多加理解，会对以后的编码有帮助。本章学习重点是方法和成员变量的重写、重载、覆盖等概念。如果读者想了解更多的关于本章的内容，可以参考电子工业出版社出版的《深入浅出 JDK 6.0》一书进行学习。



## 8.14 习题

### 一、填空题

1. 类的继承是通过 Java 保留的关键字 \_\_\_\_\_ 来修饰的，通过 extends 的关键字表明前者具备后者的公共的 \_\_\_\_\_ 和 \_\_\_\_\_ 。
2. \_\_\_\_\_ 是修饰的当前成员变量的访问限制和状态的。
3. public 表明被它修饰的成员变量为 \_\_\_\_\_ 类型，那么这个成员变量在任何包里都能访问，包括子类也能访问到。
4. private 表明被它修饰的成员变量为 \_\_\_\_\_ 类型，表示除了本类外任何类都不能访问到这个成员变量，具有很好的保护性。
5. 成员变量的覆盖是子类里有和父类里相同的成员变量或方法，继承的关系，子类的成员变量将会 \_\_\_\_\_，而父类的成员变量将会 \_\_\_\_\_。
6. 方法的 \_\_\_\_\_ 和 \_\_\_\_\_ 是体现继承特性的重要方面。
7. 重载的方法名称相同，但方法的 \_\_\_\_\_ 不相同。

**二、选择题**

1. 选择下面程序的运行结果 ( )。

```

01  public class Hello extends OtherHello
02  {
03      public static void main(String args[])
04      {
05          OtherHello a=new Hello();
06          a.myVoid();
07      }
08      public void myVoid()
09      {
10          System.out.println("b");
11      }
12  }
13  class OtherHello
14  {
15      public void myVoid()
16      {
17          System.out.println("a");
18      }
19  }

```

- A. a      B. b      C. 编译错误      D. 没有任何输出

2. 选择下面程序的运行结果 ( )。

```

01  public class Hello extends OtherHello
02  {
03      public static void main(String args[])
04      {
05          Hello a=new Hello();
06          long l=a.myVoid(3);
07          System.out.println(l);
08      }
09      public long myVoid(long x)
10      {
11          return x*6;
12      }
13  }
14  class OtherHello
15  {
16      public int myVoid(int x)
17      {
18          return x*5;
19      }
20  }

```

- A. 15      B. 18  
C. 第 9 行发生编译错误      D. 第 16 行发生编译错误

**三、简答题**

1. 简述重写、重载和覆盖三者之间的关系。  
2. 简述 4 种访问修饰符的不同。

**四、编程题**

1. 编写一个既有重写又有重载的程序。  
2. 编写一个声明为 `protected` 修饰的成员变量，在不同的包中访问该成员变量。

# 第9章 接口

本章将介绍接口的各方面知识，主要有抽象类以及多态的特性等。通过学习这些知识可以让读者更深入地了解面向对象的思想，以及在平常编码中的一些注意事项。通过本章的学习，读者应该实现如下几个目标。

- 会定义接口和访问接口中的变量。
- 熟练掌握接口的使用。
- 了解接口和抽象类的区别。
- 了解接口的多态问题。
- 熟练掌握使用 instanceof 判断类型。



## 9.1 什么是接口

接口就是一个完成某些特定功能的类。在日常生活中，一个产品的说明书可以让消费者更多地了解产品的功能，以及使用中的注意事项，如图 9-1 所示。在 Java 中也是如此，接口是一个功能的集合。



图 9-1 说明书

### 9.1.1 接口的定义

首先来举一个例子，汽车的移动就好比一个接口，在以后生产的汽车中都遵循这个接口进行制造。而接口中只定义了汽车移动的形式，没有具体定义汽车是怎么进行移动的，所以接口就好比是一个规定。下面介绍如何定义一个接口，语法如下：

```
接口修饰符 interface 接口名称
{
    //成员变量和方法的声明
}
```

- 接口修饰符和类的修饰符是一样的。
- interface 是定义接口的关键字。
- 接口里的成员变量默认为 public static final 类型的。
- 接口不能声明为 final 的，因为 final 类型的必须要实现。

**【范例 9-1】**下面通过代码演示如何定义一个接口。

示例代码 9-1

```
01 //定义一个 bike 自行车的接口
02 public interface bike
03 {
04     //成员变量和方法的声明
05 }
06
```

```

07 //定义一个 sacing 公路赛车的接口
08 public interface sacing
09 {
10     //成员变量和方法的声明
11 }

```

接口的定义遵循类的定义原则，接口也同样可以继承，下面用代码来演示公路赛车继承于自行车。

```

01 //定义一个 bike 自行车的接口
02 public interface bike
03 {
04     //成员变量和方法的声明
05 }
06
07 //定义一个 sacing 公路赛车的接口继承于 bike
08 interface sacing extends bike
09 {
10     //成员变量和方法的声明
11 }

```

- 类不能多继承，但可以多层继承。
- 接口既可以多层继承，也可以多继承。

**【范例 9-2】**下面用代码来演示接口的多继承。

示例代码 9-2

```

01 //定义一个自行车的接口 bike
02 public interface bike
03 {
04     //成员变量和方法的声明
05 }
06
07 //定义一个越野车的接口 cross
08 public interface cross
09 {
10     //成员变量和方法的声明
11 }
12
13 //定义一个变速车的接口 shift
14 public interface shift
15 {
16     //成员变量和方法的声明
17 }
18
19 //定义一个 sacing 公路赛车的接口，它继承于 bike 和 cross
20 public interface sacing extends bike cross
21 {
22     //成员变量和方法的声明
23 }

```

**【范例 9-3】**下面用一段完整的代码来演示接口的定义。

示例代码 9-3

```

01 //创建一个包，名字为 a
02 package a;
03
04 //创建一个接口名字为 aaa

```



```
05  public interface aaa
06  {
07      int i = 2;
08
09      //创建一个接口方法 getMax
10      public int getMax();
11
12      //创建一个接口方法 getMes
13      public String getMes();
14 }
```

【代码解析】在定义接口的时候，接口里的方法是没有方法体的。接口里定义的方法需要在其子类去实现。接口的变量默认被修饰为 `public static final` 类型。

### 9.1.2 访问接口里的常量

在接口里定义的成员变量为常量，因为接口为每个成员变量默认的修饰是 `public static final` 类型，即便不显式地修饰也默认地加上了。下面通过代码演示接口里的常量。

【范例 9-4】示例代码 9-4 演示接口里的成员变量为常量。

示例代码 9-4

```
01 //创建一个包，名字为 a
02 package a;
03
04 //创建一个接口名字为 aaa
05 public interface aaa
06 {
07     //定义一个成员变量
08     int i = 2;
09 }
10
11 import a.aaa;
12
13 //test 类描述的是访问接口的常量
14 public class test
15 {
16     //Java 程序的主入口方法
17     public static void main(String[] args)
18     {
19         //取得接口里的值
20         aaa.i = aaa.i + 1;
21
22         //打印并显示结果
23         System.out.println(aaa.i);
24     }
}
```

【运行结果】使用 `javac` 编译程序将产生一个和该程序对应的 `class` 程序，然后使用 Java 运行编译产生的 `class` 程序，运行出现如下异常。

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
  The final field aaa.i cannot be assigned
  at test.main(test.java:10)
```

【代码解析】错误提示为成员变量 `aaa.i` 不能赋值。这是因为在接口中的变量默认就是 `final` 修饰的，所以该值是不能够改变的，从而不能进行赋值。



**【范例 9-5】**下面代码将上述代码进行修改使其运行成功。

示例代码 9-5

```

01 //创建一个包,名字为 a
02 package a;
03
04 //创建一个接口名字为 aaa
05 public interface aaa
06 {
07     int i = 2;
08 }
09
10 import a.aaa;
11
12 //Test 类描述的是访问接口的常量
13 public class test
14 {
15     //Java 程序的主入口方法
16     public static void main(String[] args)
17     {
18         //取得接口里的值
19         int n = aaa.i + 1;
20
21         //打印并显示结果
22         System.out.println(n);
23     }
24 }
```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序,然后使用 Java 运行编译产生的 class 程序,运行结果如图 9-2 所示。

**【代码解析】**final 关键字修饰的成员变量的值不能改变其值,如果改变其值将出现编译错误。static 关键字修饰的成员变量的值是属于类的,不基于对象的存在而存在。在 test 类中没有创建对象就直接调用了接口中的成员变量。



图 9-2 访问接口中变量



## 9.2 接口的使用

接口就是一个特殊的抽象类,抽象类里有抽象的方法和普通的方法,而接口里方法全为抽象的,需要在其子类进行具体的方法实现。类就是一个产品的详细功能说明,而接口就是这些功能的简要说明。本节将详细说明接口是如何使用的,以及它们的注意事项。

### 9.2.1 接口里的方法如何创建

在接口里创建方法和一个类里的方法很相似,不同点就是接口里的方法没有具体的方法体,而类里的方法必须实现其方法体。下面用一段代码演示在接口里如何定义方法。

```

01 //创建一个接口名字为 aaa
02 public interface aaa
03 {
```



```
04     //创建一个接口方法 getMax
05     public int getMax();
06
07     //创建一个接口方法 getMes
08     String getMes();
09 }
```

接口里的方法是不能被默认修饰为 static final 类型的，因为接口里的方法就是需要继承实现，如果加上这些关键字在编译时将提示错误。

```
01 //创建一个接口名字为 aaa
02 public interface aaa
03 {
04     //创建一个接口方法 getMax
05     public int getMax();
06
07     //创建一个接口方法 getMes
08     static final String getMes();
09 }
```

总结一下接口的方法和类里的方法的区别。在定义接口里的方法和类里的方法都是有一定规则的，但是它们之间的定义是有一定区别，分别如下：

- 接口里的方法默认被修饰为 public、abstract 类型。
- 类里的方法如果修饰为 abstract 类型，将提示错误。
- 接口里的方法不能是 static、final 类型，只能为 public、abstract 类型。
- 类里的方法不能为 final、abstract 类型。

## 9.2.2 接口引用怎么使用

在前面几节里介绍了接口创建及其注意事项，创建接口就是为了使用。下面介绍如何使用接口，以及使用接口的注意事项。接口的实现语法如下：

类的修饰符 class 类名称 implements 接口名称

通过上面的语法结构可以看出和类的继承很相似，下面通过一段代码来演示接口是如何实现的。

**【范例 9-6】**下面是通过代码演示接口是如何实现的。

### 示例代码 9-6

```
01 //创建一个包，名字为 a
02 package a;
03
04 //创建一个接口名字为 aaa
05 public interface aaa
06 {
07     //创建一个接口方法 getMax
08     public int getMax();
09
10     //创建一个接口方法 getMes
11     String getMes();
12 }
13
14 import a.aaa;
15
16 //test 类描述的是实现接口的方法
```

```

17 public class Test implements aaa
18 {
19     //实现接口里的方法
20     public int getMax()
21     {
22         //具体的方法体
23         return 0;
24     }
25
26     //实现接口里的方法
27     public String getMes()
28     {
29         //具体的方法体
30         return null;
31     }
32 }

```

**【范例 9-7】**通过一个完整的例子来演示接口的方法是怎么使用的。首先创建一个接口。

示例代码 9-7

```

01 //创建一个包,名字为 a
02 package a;
03
04 //创建一个接口名字为 aaa
05 public interface aaa
06 {
07     //创建一个接口方法 getMax
08     public int getMax();
09
10     //创建一个接口方法 getMes
11     String getMes();
12 }

```

接下来开发使用该接口的类。

```

01 //引用接口的包
02 import a.aaa;
03
04 //test 类描述的是实现接口的方法
05 public class Test implements aaa
06 {
07     //实现接口里的方法
08     public int getMax()
09     {
10         //定义 int 类型的私有变量 i
11         int i = 123;
12
13         //将变量 i 返回并退出方法
14         return i;
15     }
16
17     //实现接口里的方法
18     public String getMes()
19     {
20         //定义 String 类型的私有变量 s
21         String s = "实现接口里的方法";
22
23         //将变量 s 返回并退出方法

```



```
24         return s;
25     }
26
27     //main方法为Java程序的入口方法
28     public static void main(String args[])
29     {
30         //创建test类的对象实例,引用为t
31         test t = new test();
32
33         //实现了接口里的方法并进行调用
34         int i = t.getMax();
35         String a = t.getMes();
36
37         //打印并显示结果
38         System.out.println(i);
39         System.out.println(a);
40     }
41 }
```



图 9-3 接口引用的使用

**【运行结果】**使用 `javac` 编译程序将产生一个和该程序对应的 `class` 程序, 然后使用 Java 运行编译产生的 `class` 程序, 运行结果如图 9-3 所示。

**【代码解析】**接口里的方法通过 `implements` 关键字实现后, 有了具体的方法体, 就和正常的方法使用没有任何区别了。实现的方法是随着类的创建而存在的, 因此需要将对象实例后再使用。



### 9.3 什么是抽象类

抽象类和接口有些类似, 抽象类需要其他类继承来实现抽象类中的方法, 以及给出更多的方法。在日常生活中, 一个产品的简要介绍和详细介绍结合, 说明了产品具有什么功能, 和这个功能都完成了什么。在 Java 中也是类似的, 接口是抽象类的特殊版本。接口里的方法必须都为抽象的, 而抽象类里可以为抽象的也可以有其他形式的存在。

#### 9.3.1 抽象类的使用和特点

抽象类和一般的类很相似, 只是在类里存在一些没有方法体的方法, 即抽象方法。下面通过一段代码演示抽象类。

**【范例 9-8】**下面用代码演示抽象类是如何使用。首先开发一个抽象类。

示例代码 9-8

```
01 //创建一个包, 名字为 a
02 package a;
03
04 //创建一个接口名字为 aaa
05 public abstract class aaa
06 {
07     //抽象类里的抽象方法
08     abstract public void show();
09
10     //抽象类里的方法
```

```

11     public int getMax()
12     {
13         //定义 int 类型的私有变量 i
14         int i = 123;
15
16         //将变量 i 返回并退出方法
17         return i;
18     }
19 }
```

再定义一个使用该抽象类的类。

```

01 //引入抽象类的包
02 import a.aaa;
03
04 //test 类描述的是实现接口的方法
05 public class test extends aaa
06 {
07     //main 方法为 Java 程序的入口方法
08     public static void main(String args[])
09     {
10         //创建 test 类的对象实例，引用为 t
11         test t = new test();
12
13         //实现了抽象类的方法并调用
14         t.show();
15     }
16
17     //实现了抽象类的方法
18     public void show()
19     {
20         System.out.println("实现了抽象类里的方法");
21     }
22 }
```

**【运行结果】**使用 `javac` 编译程序将产生一个和该程序对应的 `class` 程序，然后使用 `Java` 运行编译产生的 `class` 程序，运行结果如图 9-4 所示。

**【代码解析】**抽象类和接口一样，没有实现的方法必须在其子类去实现，如果不实现就必须把类声明为 `abstract` 抽象类型的。抽象类里的方法的使用就和一般类里的使用没什么区别了。

**【范例 9-9】**下面通过代码演示抽象类不能被实例化。首先定义一个抽象类。

示例代码 9-9

```

01 //创建一个包，名字为 a
02 package a;
03
04 //创建一个接口名字为 aaa
05 public abstract class aaa
06 {
07     //抽象类里的抽象方法
08     abstract public void show();
09
10    //抽象类里的方法
```



图 9-4 抽象类的使用



```
11 public int getMax()
12 {
13 //定义 int 类型的私有变量 i
14 int i = 123;
15
16 //将变量 i 返回并退出方法
17 return i;
18 }
19 }
```

再定义使用该抽象类的类。

```
01 //引入抽象类的包
02 import a.aaa;
03
04 //test 类描述的是实现接口的方法
05 public class Test extends aaa
06 {
07 //main 方法为 Java 程序的入口方法
08 public static void main(String args[])
09 {
10 //创建 test 类的对象实例，引用为 t
11 Test t = new Test();
12
13 //实现了抽象类的方法并调用
14 t.show();
15
16 aaa a = new aaa();
17 }
18
19 //实现了抽象类的方法
20 public void show()
21 {
22 System.out.println("实现了抽象类里的方法");
23 }
24 }
```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行出现如下异常。

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
  Cannot instantiate the type aaa
  at test.main(test.java:16)
```

【代码解析】此代码说明了抽象类不能实例化，抽象类和接口一样是为了继承用的。

通过上面代码的演示后，下面总结抽象类的特点，如下所示。

- 抽象类一般在父类中使用，而它的子类实现父类中的抽象方法。
- 如果父类有一个或多个抽象方法，那么父类必须为抽象类。
- 抽象类里的抽象方法没有任何方法体，子类要实现父类的所有抽象方法。如果没有实现抽象方法，其子类即为抽象类。
- 抽象类是用来继承的，不能被实例化。
- 所有的抽象方法都必须声明在抽象类中。
- 抽象类里的抽象方法，只有在子类实现了才能使用。
- 抽象类里允许有抽象方法和普通方法。



- 抽象类里的普通方法可以被子类调用。

### 9.3.2 抽象类与接口区别

前面几节学习了抽象类和接口，它们之间很相似但是也有区别。它们之间的区别如下：

- 抽象类中有一个抽象方法或多个抽象方法。
- 如果抽象类的子类里有一个没有实现的抽象方法，那么这个类也是抽象类。
- 实现抽象类里的方法可以实现部分方法，也可以实现所有方法。
- 抽象类里可以有成员变量。
- 抽象类里可以有私有的方法和私有的成员变量。
- 接口中的方法全部都被修饰为抽象方法。
- 接口里的方法都被默认修饰为 public abstract 类型的。
- 接口里的变量都被默认修饰为 public static final 类型的，即常量。
- 一个类可以实现一个接口，也可以实现多个接口。
- 接口里的方法必须要全部实现。
- 接口里没有成员变量。
- 接口里的方法全部都是 public，即公共类型的。



## 9.4 接口的多态

所谓多态，就好比日常生活中的橘子和羊肉都是食物的一种，而橘子又是水果的一种，羊肉是肉类的一种，橘子和羊肉是两种不相同的食物，但是食物可以同时指向它们这两种物品。这就是日常生活中多态的形式。在 Java 中也是如此，食物的对象可以指向橘子和羊肉对象，这给编写代码带来了很大的灵活性。

**【范例 9-10】**下面通过一个例子演示接口是如何实现多态的。首先创建一个 food 接口。

示例代码 9-10

```

01 //创建一个 food 接口
02 interface food
03 {
04     //得到食物的名称
05     public void getname();
06
07     //吃食物的方法
08     public void eat();
09 }
```

再来编写一个继承 food 接口的 fruit 接口和 meat 接口。

```

01 //创建一个 fruit 接口，即水果，继承于 food 接口
02 interface fruit extends food
03 {
04     //此接口继承了父接口的方法
05 }
```

```

01 //创建一个 meat 接口，即肉类，继承于 food 接口
02 interface meat extends food
```



```
03  {
04      //此接口继承了父接口的方法
05  }
```

接下来定义一个实现 fruit 接口的 ora 类。

```
01 //ora 类描述的是橘子，继承于 fruit 接口
02 class ora implements fruit
03 {
04     //此类实现了接口里的所有方法
05     public void eat()
06     {
07         //打印水果名称
08         System.out.println("此方法是吃橘子的方法");
09     }
10
11     public void getname()
12     {
13         //打印水果名称
14         System.out.println("吃的水果名称为橘子");
15     }
16 }
```

同时创建一个实现 meat 接口的 hotpot 类。

```
01 //hotpot 类描述的是羊肉，继承于 meat 接口
02 class hotpot implements meat
03 {
04     //此类实现了接口里的所有方法
05     public void eat()
06     {
07         //打印羊肉名称
08         System.out.println("此方法是吃羊肉的方法");
09     }
10
11     public void getname()
12     {
13         //打印羊肉名称
14         System.out.println("吃的水果名称为羊肉");
15     }
16 }
```

最后编写一个总类。

```
01 //test 类描述的是实现多态接口
02 public class test
03 {
04     //main 方法为 Java 程序的入口方法
05     public static void main(String args[])
06     {
07         //创建橘子的实例
08         food f1 = new ora();
09         f1.eat();
10         f1.getname();
11
12         //创建羊肉的实例
13         food f2 = new hotpot();
14         f2.eat();
15         f2.getname();
16     }
17 }
```

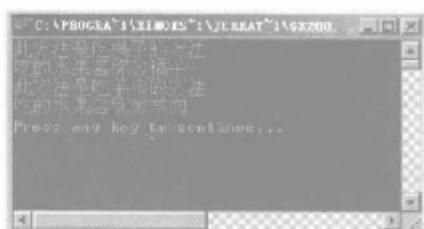


图 9-5 接口多态

**【运行结果】** 使用 `javac` 编译程序将产生一个和该程序对应的 `class` 程序，然后使用 Java 运行编译产生的 `class` 程序，运行结果如图 9-5 所示。

**【代码解析】** 此例子演示的定义接口、继承接口及实现接口并实现接口里的方法。在 `main` 方法里用来创建 `ora` 的对象实例，其引用是 `food` 接口类型的，`ora` 对象在创建的时候为 `food` 接口类型的，而在运行期为 `ora` 类型的，这就是多态。而本例应用的是接口，即接口的多态，创建 `hotpot` 对象和 `ora` 对象是一样的效果。在代码中，读者可以看到一个对象有两种不同的状态，这就是多态的好处。它可以使代码更简单，更灵活。



## 9.5 判断类型

`instanceof` 一般使用于多态的时候，在代码中判断对象的引用类型具体为哪一种类型。根据不同的对象类型来执行不同对象中的方法。本节将介绍什么是 `instanceof`，以及使用它的注意事项。

### 9.5.1 什么是 `instanceof`

所谓 `instanceof` 在字面上可以理解为检查实例。`instanceof` 在 Java 中是一个二元操作符，也是 Java 所保留的关键字。`instanceof` 的语法结构如下：

对象的引用 `instanceof` 类或接口

`instanceof` 语句的返回结果是 `boolean` 类型的。如果返回 `true`，说明对象的引用是该对象所指的类或接口；如果返回 `false`，说明对象的引用是该对象所指的类或接口。

**【范例 9-11】** 下面通过一段代码演示如何使用 `instanceof`。

示例代码 9-11

```

01 //创建一个 fruit 接口，即水果
02 interface fruit
03 {
04     //得到食物的名称
05     public void getname();
06
07     //吃食物的方法
08     public void eat();
09 }
10
11 //ora 类描述的是橘子，继承于 fruit 接口
12 class ora implements fruit
13 {
14     //此类实现了接口里的所有方法
15     public void eat()
16     {
17         //打印水果名称
18         System.out.println("此方法是吃橘子的方法");
19     }
20 }
```



## 21 天学通 Java

```
21     public void getname()
22     {
23         //打印水果名称
24         System.out.println("吃的水果名称为橘子");
25     }
26 }
27
28 //apple 类描述的是苹果类
29 class apple
30 {
31     String name;
32 }
33
34 //test 类描述的是测试 instanceof
35 public class test
36 {
37     //main 方法为 Java 程序的入口方法
38     public static void main(String args[])
39     {
40         //创建 ora 类的对象实例，其引用为 o
41         ora o = new ora();
42
43         //判断对象引用 o 的类型
44         if (o instanceof ora)
45         {
46             System.out.println("对象引用 o 指向的类为 ora");
47         }
48
49         if (o instanceof fruit)
50         {
51             System.out.println("对象引用 o 指向的接口为 fruit");
52         }
53
54         //创建 apple 类的对象实例，其引用为 a
55         apple a = new apple();
56
57         //判断对象引用 a 的类型
58         if (a instanceof fruit)
59         {
60             System.out.println("对象引用 a 指向的接口为 fruit");
61         }
62         else
63         {
64             System.out.println("对象引用 a 没有指向任何类或对象");
65         }
66     }
67 }
```

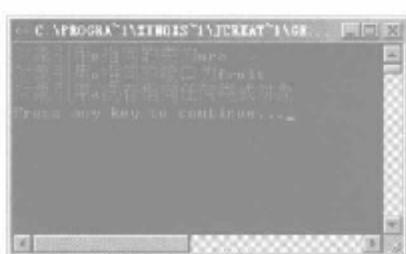


图 9-6 使用 instanceof

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 9-6 所示。

**【代码解析】**通过 if 语句来判断对象引用的类型是否为 ora 类和 fruit 接口。如果是，就打印相应的信息。ora 类是继承与 fruit 接口的，用 instanceof 来检测是能通过的。判断对象引用 a 是否能指向 fruit，结果为 false。原因是对象 apple 不是 fruit 接口的子类，

所以说多态是基于继承的。

**【范例 9-12】**下面的程序讲解如何使用 instanceof 测试对象是否已经创建成功。

示例代码 9-12

```

01 //test 类描述的是测试 instanceof 判断对象创建
02 public class test
03 {
04     //创建 test 类的对象引用 t1
05     static test t1;
06
07     //创建 Test 类的对象引用 t2
08     static test t2;
09
10    //main 方法为 Java 程序的入口方法
11    public static void main(String args[])
12    {
13        //创建 test 类的对象实例
14        t1 = new test();
15
16        //判断对象引用 o 的类型
17        if (t1 instanceof test)
18        {
19            System.out.println("对象引用 t1 指向的类为 test");
20        }
21        else
22        {
23            System.out.println("对象引用 t1 不指向的类为 test");
24        }
25
26        //判断对象引用 o 的类型
27        if (t2 instanceof test)
28        {
29            System.out.println("对象引用 t2 指向的类为 test");
30        }
31        else
32        {
33            System.out.println("对象引用 t2 不指向的类为 test");
34        }
35    }
36 }
37

```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 9-7 所示。

**【代码解析】**被 static 关键字修饰的成员变量是随着类的创建而创建的。在类的生命周期中只存在一份。对象引用 t1 在 main 方法里进行了实例引用，指向了具体的对象。对象引用 t2 没有指向任何对象，所以被系统赋予默认值，即 null。因此 t2 没有指向任何对象，所以说第二个 if 语句的结果为 false。

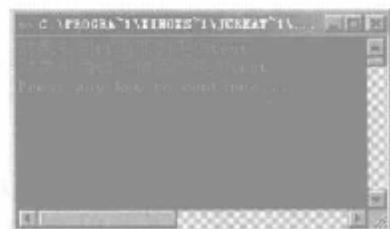


图 9-7 测试对象是否创建



### 9.5.2 使用 instanceof 的注意事项

在使用 instanceof 进行对象类型判断的时候也是有规则要遵循的，下面总结使用 instanceof 有哪些规则。

- instanceof 关键字不能比较基本类型的数据。
- instanceof 关键字可以对对象和接口使用。
- instanceof 关键字的比较是基于多态的。
- 不推荐使用 instanceof 关键字，要多多应用多态。
- instanceof 关键字右边比较的类型只能为类和接口。

**【范例 9-13】**下面是一个小例子，通过它来说明比较对象时提示的错误。

示例代码 9-13

```
01 //test 类描述的是测试 instanceof 判断对象
02 public class test
03 {
04     //main 方法为 Java 程序的入口方法
05     public static void main(String args[])
06     {
07         //创建 test 类的对象实例，其引用为 t1
08         test t1 = new test();
09
10         //创建 Test 类的对象实例，其引用为 t2
11         Test t2 = new Test();
12
13         //判断对象引用 o 的类型
14         if (t1 instanceof test)
15         {
16             System.out.println("对象引用 t1 指向的类为 test");
17         }
18         else
19         {
20             System.out.println("对象引用 t1 不指向的类为 Test");
21         }
22
23         //判断对象引用 o 的类型
24         if (t1 instanceof t2)
25         {
26             System.out.println("对象引用 t1 指向的对象为 t2");
27         }
28         else
29         {
30             System.out.println("对象引用 t1 不指向的对象为 t2");
31         }
32     }
33 }
```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行出现如下异常。

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
  t2 cannot be resolved to a type
  at test.main(test.java:24)
```

**【代码解析】**t1 通过 instanceof 关键字的判断，将显示正确结果。t2 为不可识别的符号。

因为 `instanceof` 关键字的右边只能为类或接口类型。



## 9.6 综合练习

说明抽象类和接口的区别。

【提示】抽象类中有一个抽象方法或多个抽象方法。接口中的方法全部都被修饰为抽象方法，并且接口里的方法都被默认修饰为 `public abstract` 类型的。

如果抽象类的子类里有一个没有实现的抽象方法，那么这个类也是抽象类。一个类可以实现一个接口，也可以实现多个接口。

实现抽象类里的方法可以实现部分方法，也可以实现所有方法。接口里的方法必须要全部实现。

抽象类里可以有成员变量。抽象类里可以有私有的方法和私有的成员变量。接口里的变量都被默认修饰为 `public static final` 类型，即常量。接口里没有成员变量。接口里的方法全部都是 `public`，即公共类型的。



## 9.7 小结

本章学习了一个更重要的知识——接口，对象通过多态可以变得灵活，而接口可以使对象变得更加灵活。接口是基于继承的，通过在编译期和运行期来扮演不同的类型。学好本章知识能更好地了解面向对象的知识。读者如果想了解更多的关于本章的内容，可以参考电子工业出版社出版的《深入浅出 JDK 6.0》一书进行学习。



## 9.8 习题

### 一、填空题

- \_\_\_\_\_是定义接口的关键字，接口里的成员变量默认为\_\_\_\_\_类型的。
- 接口里的方法默认被修饰为\_\_\_\_\_、\_\_\_\_\_类型的。
- 接口里的方法不能是 `static`、`final` 类型的，只能为\_\_\_\_\_、\_\_\_\_\_类型的。
- `instanceof` 关键字的比较是基于\_\_\_\_\_的。

### 二、选择题

- 选择下面正确定义接口方法的选项（\_\_\_\_\_）。

- A. `private int getArea();`
- B. `public static void main(String args[]);`
- C. `public void main(String args[])`
- D. `public int getInt(int x);`

- 选择下面程序的运行结果（\_\_\_\_\_）。

```
01 interface MyHello
02 {
03     int x=0;
```



```
04     void myVoid1();
05 }
06 public class Hello implements MyHello
07 {
08     public static void main(String args[])
09     {
10         Hello a=new Hello();
11         a.myVoid2();
12     }
13     public void myVoid2()
14     {
15         for(int y=4;y>x;y--,++x)
16         {
17             System.out.println(x);
18         }
19     }
20     public void myVoid1()
21     {
22     }
23 }
```

- A. 0 1 2 3      B. 0 1 2      C. 0 1      D. 编译发生错误

# 第 10 章 构造器

在前面的章节里，构造器的代码读者已经看到很多次了。在 Java 中，通过用 `new` 关键字来调用构造器使其对象在内存中创建出来。本章将详细介绍构造器的一些知识。通过本章的学习，读者应该能够实现如下几个目标。

- 了解什么是构造器。
- 熟练掌握如何创建构造器。
- 熟练掌握构造器的使用，包括构造器如何调用等问题。
- 了解构造器的一些基本机制。



## 10.1 什么是构造器

在日常生活中，盖房子需要工具和工人，通过工人使用这些工具，来修建一个房子。在 Java 中，构造器就好比是工具，而 `new` 关键字就是工人，通过 `new` 关键字和构造器结合来创建对象。

### 10.1.1 构造器的使用

要建立对象就要使用 `new` 关键字，这是建立对象唯一的方法。介绍构造器的语法组成如下所示。

```
类的修饰符 类的名称(参数列表)
{
    //方法体
}
```

- 构造器可以使用的修饰符有 `public`、`protected`、`default`、`private`，不写即为 `default` 类型的。
- 构造器的名称必须要和类的名称相同。
- 不能有返回值，`void` 也不行。
- 构造器的参数可有可无，可以有一个也可有多个参数。

**【范例 10-1】**下面代码演示了一个错误的构造器。

示例代码 10-1

```
01  public class Test
02  {
03      //创建一个构造器
04      public void test()
05      {
06          //方法体
07      }
08  }
```



【代码解析】这段代码在编译时和运行时都没有错误，因为 `test()` 在这里不被认为是一个构造器了，而被认为是一种方法，此时系统会默认生成一个构造器。构造器必须没有返回类型。修改示例代码 10-1 使其正确，如下所示。

```
01  public class test
02  {
03      //创建一个构造器
04      public test()
05      {
06          //方法体
07      }
08  }
```

构造器有多种形式，包括无参构造器和有参构造器。有参构造器中又分为具有一个参数的和具有多个参数的。参数的类型可以相同，也可以不同。下面的程序中给出了这几种形式。

```
01  public class test
02  {
03      //创建一个无参数的构造器
04      public test()
05      {
06          //构造器里的方法
07      }
08
09      //创建一个有一个参数的构造器
10      public test(String s)
11      {
12          //构造器里的方法
13      }
14
15      //创建一个有两个参数的构造器
16      public test(String s1, String s2)
17      {
18          //构造器里的方法
19      }
20
21      //创建一个有两个参数但类型不同的构造器
22      public test(String s, int i)
23      {
24          //构造器里的方法
25      }
26  }
```

【范例 10-2】下面是一个构造器的完整应用的程序。

示例代码 10-2

```
01  //test 类描述的是测试构造器
02  public class test
03  {
04      private String s;
05
06      public test(String s)
07      {
08          this.s = s;
09      }
10
11      public String getMea()
12      {
```

```

13         return s;
14     }
15
16     //main 方法为 Java 程序的入口方法
17     public static void main(String args[])
18     {
19         //创建 test 类的对象实例, 其引用为 t
20         test t = new test("构造器测试");
21
22         //调用方法并赋值给 String 类型的引用 s
23         String s = t.getMes();
24
25         //打印并显示结果
26         System.out.println(s);
27     }
28 }

```

**【运行结果】** 使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 10-1 所示。

**【代码解析】** 通过使用 new 关键字来调用一个有参数的构造器, 并在构造器里对其私有成员变量 s 进行赋值。调用 test 对象的方法来显示其私有成员变量的值。

### 10.1.2 被修饰的构造器

构造器是可以被修饰符修饰的, 不同的修饰符修饰构造器也具有不同的效果, 本节将通过使用不同的修饰符来进行代码演示。

#### 【范例 10-3】被 public 修饰符修饰的构造器。

示例代码 10-3

```

01  //test 类描述的是测试 public 类型的构造器
02  public class test
03  {
04      //声明一个私有的成员变量
05      private String s;
06
07      //test 类的构造器
08      public test()
09      {
10          System.out.println("构造器运行了");
11      }
12
13      //给私有成员变量赋值
14      public String gets()
15      {
16          return s;
17      }
18
19      //取得私有成员变量的值
20      public void sets(String s)
21      {
22          this.s = s;
23      }
24
25  }

```



图 10-1 构造器



```
23     }
24
25     //main方法为Java程序的入口方法
26     public static void main(String args[])
27     {
28         //创建test类的对象实例,其引用为t
29         Test t = new Test();
30
31         //调用方法并赋值
32         t.setS("给私有的成员变量赋值");
33
34         //取得私有的成员变量值
35         String s = t.gets();
36
37         //打印并显示结果
38         System.out.println(s);
39     }
40 }
```



图 10-2 public 修饰构造器

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序,然后使用 Java 运行编译产生的 class 程序,运行结果如图 10-2 所示。

**【代码解析】**被修饰为 public 类型的构造器是最常见的构造器,此时的构造器能让所有的类访问到。如果要让所有类都能访问到此类,构造器要修饰为 public 类型,而类的修饰符也要为 public 类型的。

**【范例 10-4】**访问被 private 修饰符修饰的构造器时的错误。

#### 示例代码 10-4

```
01 //test类描述的是测试private类型的构造器
02 class apple
03 {
04     //声明一个私有的成员变量
05     private String s;
06
07     //被修饰为private类型的Test类的构造器
08     private apple()
09     {
10         System.out.println("构造器运行了");
11     }
12
13     //给私有成员变量赋值
14     public String gets()
15     {
16         return s;
17     }
18
19     //取得私有成员变量的值
20     public void sets(String s)
21     {
22         this.s = s;
23     }
24 }
25
26 public class testApple
```

```

27  {
28      //main方法为Java程序的入口方法
29      public static void main(String args[])
30      {
31          //创建test类的对象实例,其引用为a
32          apple a = new apple();
33
34          //调用方法并赋值
35          a.sets("给私有的成员变量赋值");
36
37          //取得私有的成员变量值
38          String s = a.gets();
39
40          //打印并显示结果
41          System.out.println(s);
42      }
43  }

```

【运行结果】使用javac编译程序,将产生一个和该程序对应的class程序,然后使用Java运行编译产生的class程序,运行出现如下异常。

```

Exception in thread "main" java.lang.Error: Unresolved compilation problem:
  The constructor apple() is not visible
  at test.main(test.java:39)

```

【代码解析】apple类的构造器被修饰为private类型的,在别的类进行调用的时候是访问不到的。在本类里是可以访问的。要解决这个问题有两种方法,一是修改构造器的修饰符,二是调用其他的方法返回其对象的类型。

【范例10-5】修改示例代码10-4,使其能够正确访问private类型的构造器。

#### 示例代码10-5

```

01 //test类描述的是测试private类型的构造器
02 class apple
03 {
04     //声明一个私有的成员变量
05     private String s;
06
07     //被修饰为private类型的test类的构造器
08     private apple()
09     {
10         System.out.println("构造器运行了");
11     }
12
13     //通过一个static方法来返回apple的对象实例
14     public static apple getTest()
15     {
16         return new apple();
17     }
18
19     //给私有成员变量赋值
20     public String gets()
21     {
22         return s;
23     }
24

```

```
25 //取得私有成员变量的值
26 public void setS(String s)
27 {
28     this.s = s;
29 }
30 }
31 public class testApple
32 {
33     //main方法为Java程序的入口方法
34     public static void main(String args[])
35     {
36         //创建Test类的对象实例，其引用为a
37         apple a = apple.getTest();
38
39         //调用方法并赋值
40         a.setS("给私有的成员变量赋值");
41
42         //取得私有的成员变量值
43         String s = a.gets();
44
45         //打印并显示结果
46         System.out.println(s);
47     }
48 }
```



图 10-3 修改后

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 10-3 所示。

**【代码解析】**构造器被修饰为 private 类型后，除本类外其他类是访问不到的。需要声明一个 public 类型的方法来返回其对象引用。通过取得方法返回的对象引用，再执行并打印结果。这种方式也叫单子模式。单子模式将在第 10.4 节介绍。

### 10.1.3 构造器方法与普通方法的区别

构造器方法和普通的方法是有一定区别，主要是功能、修饰符、返回值和命名上有本质的区别。区别如下：

- 构造器是为了创建一个类的对象实例，也可以在创建对象的时候使用。
  - 方法是为了执行相应的方法体，即 Java 代码。
  - 构造器可以被修饰为 public、protected、default、private 类型，但不能修饰为 abstract、final、native、static、synchronized。
  - 方法可以修饰为除了 protected、native 外的修饰符。
  - 构造器没有返回值，也没有 void 型。
  - 方法没有返回值或有任何类型的返回值。
  - 构造器的名称要和类的名称相同。
  - 方法的名称可以任意起，但要注意标识符的命名规则，使其更具有意义。



## 10.2 如何实例化一个对象

实例化就是在内存中实实在在地创建一个对象，在日常生活中就好比创造了一个东西出来。而在 Java 中，实例化一个对象用 new 关键字完成。下面先介绍 new 关键字的语法并通过一个例子来演示。

new 构造器的名称(参数列表)

- new 为 Java 关键字，要注意大小写。
- 构造器的名称要和类的名称相同。
- 通过调用构造器方法来对这个对象进行一些必要的初始化。
- 用 new 关键字实例化对象后返回该对象的引用。

**【范例 10-6】**下面是一个完整的例子，来演示如何实例化对象。

示例代码 10-6

```

01 //test 类描述的是 new 关键字实例化对象
02 public class test
03 {
04     //创建 String 类型的字符串 s
05     private String s;
06
07     //创建 get 方法读取私有的成员变量的值
08     public String getS()
09     {
10         return s;
11     }
12
13     //创建 set 方法给私有的成员变量赋值
14     public void setS(String s)
15     {
16         this.s = s;
17     }
18
19     //main 方法为 Java 程序的入口方法
20     public static void main(String args[])
21     {
22         //创建 test 类的对象实例，其引用为 a
23         Test a = new test();
24
25         //调用方法并赋值
26         a.setS("给私有的成员变量赋值");
27
28         //取得私有的成员变量值
29         String s = a.getS();
30
31         //打印并显示结果
32         System.out.println(s);
33     }
34 }
```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 10-4 所示。

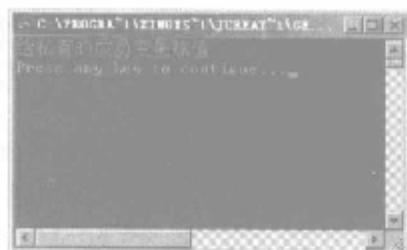


图 10-4 实例化对象

【代码解析】成员变量和方法都是属于对象的。随着对象的存在，成员变量和方法才存在。通过 new 关键字来创建对象后，调用其引用的方法来打印结果。new 关键字所调用的构造器方法必须要存在，否则会提示错误。

【范例 10-7】下面代码演示了调用一个没有事先定义的构造器方法。

#### 示例代码 10-7

```
01 //test 类描述的是 new 关键字实例化对象
02 public class test
03 {
04     //创建 String 类型的字符串 s
05     private String s;
06
07     //创建 get 方法读取私有的成员变量的值
08     public String getS()
09     {
10         return s;
11     }
12
13     //创建 set 方法给私有的成员变量赋值
14     public void sets(String s)
15     {
16         this.s = s;
17     }
18
19     //main 方法为 Java 程序的入口方法
20     public static void main(String args[])
21     {
22         //创建 test 类的对象实例，其引用为 a
23         test a = new test("2");
24
25         //调用方法并赋值
26         a.sets("给私有的成员变量赋值");
27
28         //取得私有的成员变量值
29         String s = a.gets();
30
31         //打印并显示结果
32         System.out.println(s);
33     }
34 }
```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行出现如下异常。

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
  The constructor test(String) is undefined
  at test.main(test.java:24)
```

【代码解析】错误提示说明是，没有一个参数类型为 String 类型的构造器方法。声明一个参数类型为 String 类型的构造器就可以解决此问题。



## 10.3 构造器的使用

通过前面的介绍，读者已经对构造器有了基本的了解。但是使用构造器时有一些注意事项。本节将介绍构造器在父子类中是如何使用的。

### 10.3.1 构造器的调用

构造器的调用一般有两种情况，一般是在本类里调用或在同包下的另一个类调用，另一种情况是子类调用父类的构造器的。下面通过代码分别演示。

**【范例 10-8】** 在本类里调用构造器方法。

示例代码 10-8

```

01 //test 类描述的是在本类里调用构造器
02 public class test
03 {
04     //创建 String 类型的字符串 str
05     private String str;
06
07     //创建 get 方法读取私有的成员变量的值
08     public String getStr()
09     {
10         return str;
11     }
12
13     //创建 set 方法给私有的成员变量赋值
14     public void setStr(String str)
15     {
16         this.str = str;
17     }
18
19     //main 方法为 Java 程序的入口方法
20     public static void main(String args[])
21     {
22         //创建 test 类的对象实例，其引用为 a
23         test a = new test();
24
25         //调用方法并赋值
26         a.setStr("给私有的成员变量赋值");
27
28         //取得私有的成员变量值
29         String s = a.getStr();
30
31         //打印并显示结果
32         System.out.println(s);
33     }
34 }
```

**【运行结果】** 使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 10-5 所示。

**【代码解析】** 在本类里调用构造器，读者在前面的章节里见到过最多，也是最常见的。

**【范例 10-9】** 在不同包下调用构造器方法。



## 示例代码 10-9

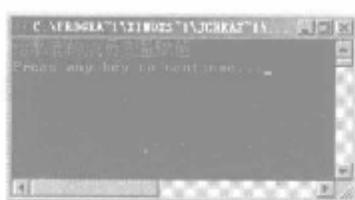


图 10-5 在本类中调用

```
01 package b;
02
03 public class apple
04 {
05     //创建 String 类型的字符串 str
06     private String str;
07
08     //创建 get 方法读取私有的成员变量的值
09     public String getStr()
10     {
11         return str;
12     }
13
14     //创建 set 方法给私有的成员变量赋值
15     public void setStr(String str)
16     {
17         this.str = str;
18     }
19
20     //创建一个无参数的构造器
21     public test()
22     {
23         System.out.println("无参的构造器运行了");
24     }
25
26     //创建一个有一个参数的构造器
27     public test(String str)
28     {
29         System.out.println("有参的构造器运行了");
30         this.str = str;
31     }
32 }
33
34 //引入 b 包下的 apple 类
35 import b.apple;
36
37 //test 类描述的是在不同包下调用构造器
38 public class testApple
39 {
40     //main 方法为 Java 程序的入口方法
41     public static void main(String args[])
42     {
43         //创建 test 类的对象实例, 其引用为 a1
44         apple a1 = new apple();
45
46         //调用方法并赋值
47         a1.setStr("给私有的成员变量赋值");
48
49         //取得私有的成员变量值
50         String s1 = a1.getStr();
51
52         //打印并显示结果
53         System.out.println(s1);
54
55         //创建 test 类的对象实例, 其引用为 a2
56         apple a2 = new apple("调用的是一个有参数的构造器");
57
58 }
```

```

25     //取得私有的成员变量值
26     String s2 = a2.getStr();
27
28     //打印并显示结果
29     System.out.println(s2);
30 }
31 }

```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 10-6 所示。

【代码解析】在不同包下只要用 import 关键字引入所需要的类后，使用构造器方法就和在本类一样了。在调用包里的构造器方法时，要确定类的修饰符是否为 public 类型。如果不是将出现下列错误提示。

```

Exception in thread "main" java.lang.Error: Unresolved compilation problems:
  The type test is not visible
  The type Test is not visible
  The type test is not visible
  The type test is not visible

  at test1.main(test1.java:11)

```

### 【范例 10-10】子类调用父类的构造器方法。

示例代码 10-10

```

01 //Fruit 类描述的是水果
02 class fruit
03 {
04     //创建 String 类型的字符串 color，表示水果的颜色
05     private String color;
06
07     //创建 get 方法读取私有的成员变量的值
08     public String getColor()
09     {
10         return color;
11     }
12
13     //创建 set 方法给私有的成员变量赋值
14     public void setColor(String color)
15     {
16         this.color = color;
17     }
18
19     //创建一个无参数的构造器
20     public fruit()
21     {
22         System.out.println("父类的无参的构造器运行了");
23     }
24 }
25
26 //apple 类描述的是苹果继承与 fruit
27 public class apple extends fruit
28 {
29     //创建一个无参数的构造器
30     public apple()

```

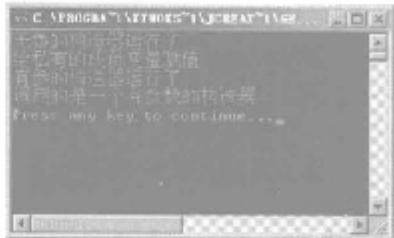


图 10-6 不同包下调用



```
31     [
32         super();
33     }
34
35     //main 方法为 Java 程序的入口方法
36     public static void main(String args[])
37     {
38         //创建 test 类的对象实例, 其引用为 a1
39         fruit f = new apple();
40
41         //调用方法并赋值
42         f.setColor("给私有的成员变量赋值");
43
44         //取得私有的成员变量值
45         String s1 = f.getColor();
46
47         //打印并显示结果
48         System.out.println(s1);
49     }
50 }
```

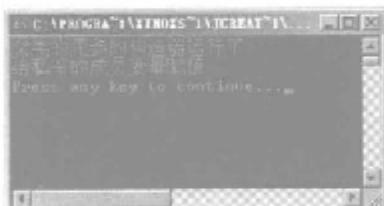


图 10-7 子类调用父类构造器  
不会调用子类的构造器方法。修改子类构造器方法如下所示, 即可看到效果。

```
public apple()
{
    //System.out.println("子类的无参的构造器运行了");
}
```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 10-7 所示。

**【代码解析】**在发生继承关系时, 子类的构造器方法默认调用父类的构造器方法, 调用完后再调用子类的构造器方法。此例中子类的构造器方法调用了方法 super, 而方法 super 就是调用了父类的构造器方法, 但

### 10.3.2 构造器重载

构造器重载和方法重载是一样的, 重载就好比日常生活中人的名字, 有大名有小名, 但这些名字都指的是这个人, 不同点就是让这个人去做的事情可能不一样。这个和 Java 里的方法的重载很相似。下面看一下构造器重载的要求。

- 构造器重载的方法名称相同, 但参数列表不相同。如参数个数和参数类型等。
- 构造器重载的方法是没有返回值的。
- 构造器不能被继承, 这和方法有所区别。
- 构造器重载的修饰符只有 public、private、protected 这三个。

**【范例 10-11】**下面通过示例代码 10-11 演示构造器重载。

示例代码 10-11

```
01 class bike
02 {
03     public bike()
04     {
05         //该构造器的方法体
06     }
07 }
```

```
06  }
07
08 public bike(String color, int Size)
09 {
10     //该构造器的方法体
11 }
12 }
```

【代码解析】虽然这两个构造器的名称相同，但是构造器的参数列表，即个数和类型，是不相同的。判断构造器是否被重载，看参数列表是否相同很重要。

【范例 10-12】下面通过示例代码演示构造器重载是如何应用的。

示例代码 10-12

```
01 //bike 类描述的是自行车
02 public class bike
03 {
04     //bike 类的私有成员变量，描述的是自行车的颜色
05     private String color;
06
07     //bike 类的私有成员变量，描述的是自行车的大小尺寸
08     private int size;
09
10    //取得私有成员变量 color 的值
11    public String getColor()
12    {
13        return color;
14    }
15
16    //设置私有成员变量 color 的值
17    public void setColor(String color)
18    {
19        this.color = color;
20    }
21
22    //取得私有成员变量 size 的值
23    public int getSize()
24    {
25        return size;
26    }
27
28    //设置私有成员变量 size 的值
29    public void setSize(int size)
30    {
31        this.size = size;
32    }
33
34    //创建一个无参数的构造器
35    public bike()
36    {
37        //打印并显示信息
38        System.out.println("无参的构造器运行了");
39    }
40
41    public bike(String color, int size)
42    {
43        //对私有成员变量进行赋值
44        this.color = color;
45        this.size = size;
```



```
46 //打印并显示信息
47 System.out.println("有两个参数的构造器运行了");
48 }
49 }
50 //main方法为Java程序的入口方法
51 public static void main(String args[])
52 {
53     //创建test类的对象实例,其引用为b1
54     bike b1 = new bike();
55
56     //调用方法并赋值
57     b1.setColor("黄色的自行车");
58     b1.setSize(26);
59
60     //取得私有成员变量的值
61     String color1 = b1.getColor();
62     int size1 = b1.getSize();
63
64     //打印并显示结果
65     System.out.println(color1 + " : " + size1);
66
67
68     //创建bike类的对象实例,其引用为b2
69     bike b2 = new bike("绿色的自行车", 28);
70
71     //取得私有成员变量的值
72     String color2 = b2.getColor();
73     int size2 = b2.getSize();
74
75     //打印并显示结果
76     System.out.println(color2 + " : " + size2);
77 }
78 }
```

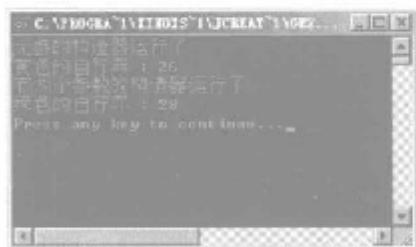


图 10-8 构造器重载

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 10-8 所示。

**【代码解析】**在此例子中拥有两种构造器方法, 在调用的时候根据参数列表的不同由虚拟机来决定使用哪种构造器方法。构造器重载和方法重载的注意事项是一样的, 但构造器重载没有返回值。判断构造器是否重载, 主要是看构造器的参数列表是否相同。

### 10.3.3 父子类间的构造器的调用流程

前面学习了用 new 关键字来创建一个对象, 但在继承关系发生时, 父类与子类是如何创建对象的呢? 它们的顺序又是什么样的呢? 下面通过一个例子来演示构造器是如何调用的。

**【范例 10-13】**下面使用示例代码 10-13 来演示构造器是如何调用的。

示例代码 10-13

```
01 //bike 类描述的是自行车
02 class bike
03 {
```

```

04      //创建一个无参的父类构造器
05  public bike()
06  {
07      //打印并显示信息
08      System.out.println("父类中无参的构造器被调用了");
09  }
10 }
11
12 //aceing 类描述的是公路赛车继承与 bike 类
13 public class aceing extends bike
14 {
15     //创建一个无参的子类构造器
16     public aceing()
17     {
18         //打印并显示信息
19         System.out.println("子类中无参的构造器调用了");
20     }
21
22 //main 方法为 Java 程序的入口方法
23 public static void main(String args[])
24 {
25     //创建 aceing 类的对象实例, 其引用为 a
26     aceing a = new aceing();
27 }
28 }

```

**【运行结果】** 使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 10-9 所示。

**【代码解析】** 在上个例子中声明了两个类, bike 类和 aceing 类, bike 类是 aceing 类的父类。用 new 关键字来创建 aceing 对象的时候, 因为有继承的关系, 虚拟机会先去创建父类的对象, 对父类的成员变量进行初始化, 再回到子类来调用其构造器方法并进行初始化。

详细的步骤如下所述。

- 在用 new 关键字创建对象 aceing 的时候, 执行 new aceing() 会进入到 aceing 对象的构造器方法体内。
- 因为继承的关系, 会默认调用方法 super 进入到父类 bike 对象的构造器方法体内。
- 对父类 bike 对象进行初始化。父类的构造器方法执行完毕后回到子类的构造器继续执行。
- 执行子类的构造器方法, 并初始化数据。

**【范例 10-14】** 利用方法 super 来改变构造器的执行顺序。

示例代码 10-14

```

01 //bike 类描述的是自行车
02 class bike
03 {
04     //bike 类的私有成员变量
05     private String color;
06
07     //创建一个无参的父类构造器

```



图 10-9 调用构造器



```
08     public bike()
09     {
10         //打印并显示信息
11         System.out.println("父类中无参的构造器被调用了");
12     }
13
14     // 创建一个有参数的父类构造器
15     public bike(String color)
16     {
17         //对私有成员变量赋值
18         this.color = color;
19
20         //打印并显示信息
21         System.out.println("父类中有一个参数的构造器被调用了");
22     }
23 }
24
25 //aceing类描述的是公路赛车继承与 bike 类
26 class aceing extends bike
27 {
28     //创建一个无参的子类构造器
29     public aceing()
30     {
31         super("黄色");
32         //打印并显示信息
33         System.out.println("子类中无参的构造器被调用了");
34     }
35 }
36
37 //test 类是用来测试 aceing 类的
38 public class test
39 {
40     //main方法为 Java 程序的入口方法
41     public static void main(String args[])
42     {
43         //创建 aceing 类的对象实例, 其引用为 a
44         aceing a = new aceing();
45     }
46 }
```



图 10-10 利用 super 方法

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 10-10 所示。

【代码解析】本例和上一个例子中的不同点就是使用方法 super, super 方法能显式地指定调用哪个构造器来创建对象。本段代码中方法 super 的参数为一个 String 类型的字符串, 在创建对象的时候虚拟机会去找一个参数类型为 String 类型的构造器, 如果没有就会出现错误提示。通过调用方法 super 的参数指定的构造器来创建对象, 而无参的那个构造器没有进行调用。

如果方法 super 指定的构造器不存在就会出现如下错误。

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
  The constructor bike(String) is undefined
  at aceing.<init>(test.java:31)
  at test.main(test.java:44)
```



### 10.3.4 如何自定义构造器

自定义的构造器已在前面的代码中多次使用。所谓自定义构造器就是不显式地定义构造器，编译器就是自动地生成一个无参的构造器，但是一旦显式地定义了一个构造器，编译器就不会自动生成了。下面用代码演示如何自定义构造器。

```
public class Test
{
    //定义一个无参的构造器
    public Test()
    {
        //该构造器的方法体
    }
    //具有两个参数的构造器
    public Test(String i, int n)
    {
        //该构造器的方法体
    }
}
```

**【范例 10-15】**如果调用的构造器没有显式的声明，那么将会出现编译错误。

示例代码 10-15

```
01 //aceing 类描述的是公路赛车
02 class aceing
03 {
04     //创建一个无参的子类构造器
05     public aceing(String color)
06     {
07         //打印并显示信息
08         System.out.println("aceing 类中的无参构造器调用了");
09     }
10 }
11
12 //test 类是用来测试 aceing 类的
13 public class test
14 {
15     //main 方法为 Java 程序的入口方法
16     public static void main(String args[])
17     {
18         //创建 aceing 类的对象实例，其引用为 a
19         aceing a = new aceing();
20     }
21 }
```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行出现如下异常。

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
  The constructor aceing() is undefined
  at test.main(test.java:19)
```

**【代码解析】**错误提示显示一个无参的构造器没有定义，也就是说没有找到一个无参的构造器。本段代码中显式地创建了一个有一个参数的构造器，虚拟机认为已经有构造器了，就不会去自动生成了。



【范例 10-16】修改示例代码 10-15 使其运行正确。

#### 示例代码 10-16

```
01 //aceing 类描述的是公路赛车
02 class aceing
03 {
04     //创建一个无参的构造器
05     public aceing()
06     {
07         //打印并显示信息
08         System.out.println("aceing 类中的无参构造器调用了");
09     }
10
11     //创建一个无参的子类构造器
12     public aceing(String color)
13     {
14         //打印并显示信息
15         System.out.println("aceing 类中有一个参数的构造器调用了");
16     }
17 }
18
19 //test 类是用来测试 aceing 类的
20 public class test
21 {
22     //main 方法为 Java 程序的入口方法
23     public static void main(String args[])
24     {
25         //创建 aceing 类的对象实例, 其引用为 a
26         aceing a = new aceing();
27     }
28 }
```



图 10-11 修改后

【运行结果】使用 `javac` 编译程序将产生一个和该程序对应的 `class` 程序, 然后使用 `Java` 运行编译产生的 `class` 程序, 运行结果如图 10-11 所示。

【代码解析】在示例代码 10-16 中, 添加了无参构造器, 从而使程序能够正常运行, 这一点是需要特别注意的。同时为每一个类写无参构造器是很好的编程习惯, 读者应该从初学时养成该习惯。



## 10.4 什么是单子模式

单子模式是 Java 模式工厂里的一种, 所谓单子模式, 就是在一个时间段内对象只存在一份。单子模式就是把构造器修饰为 `private` 类型的, 用一个 `public` 类型的方法返回该对象的引用。下面通过代码演示什么是单子模式。

【范例 10-17】下面是使用单子模式的代码。

#### 示例代码 10-17

```
01 //aceing 类描述的是公路赛车
02 class aceing
03 {
```

```

04 // 定义一个私有, 静态类型的成员变量 ace
05 private static aceing ace;
06
07 // 创建一个无参的构造器
08 private aceing()
09 {
10     // 打印并显示信息
11     System.out.println("aceing 类中的无参构造器调用了");
12 }
13
14 // 该方法返回 aceing 对象的实例
15 public static aceing getAceing()
16 {
17     // 判断 ace 是否为 null, 即没有指向任何对象.
18     if (ace == null)
19     {
20         // 创建 aceing 类的对象实例, 并把引用赋值给 ace
21         ace = new aceing();
22     }
23
24     // 返回并退出 ace 的值
25     return ace;
26 }
27
28 // aceing 类的私有方法
29 public void showMes()
30 {
31     System.out.println("执行 aceing 类的方法");
32 }
33 }
34
35 // test 类是用来测试 aceing 类的
36 public class test
37 {
38     // main 方法为 Java 程序的入口方法
39     public static void main(String args[])
40     {
41         // 通过 getAceing 方法返回 aceing 类的对象实例
42         aceing a = aceing.getAceing();
43
44         // 调用 aceing 对象里的方法 showMes
45         a.showMes();
46     }
47 }
48 }
```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 10-12 所示。

【代码解析】aceing 类为单子模式, 该类的构造器被修饰为 private 类型的。aceing 类的 getAceing 方法可以看成一个工厂方法, 如果要获得该类的引用, 都需要调用进行返回类引用。在 getAceing 方法中先进行比较是否已经创建了该类的引用, 如果没有再进行创建。

【范例 10-18】利用单子模式比较生成的两个对象是否一样。

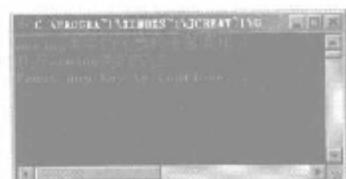


图 10-12 单子模式



## 示例代码 10-18

```
01 //aceing 类描述的是公路赛车
02 class aceing
03 {
04     //定义一个私有, 静态类型的成员变量 ace
05     private static aceing ace;
06
07     //创建一个无参的构造器
08     private aceing()
09     {
10         //打印并显示信息
11         System.out.println("aceing 类中的无参构造器调用了");
12     }
13
14     //该方法返回 aceing 对象的实例
15     public static aceing getAceing()
16     {
17         //判断 ace 是否为 null, 即没有指向任何对象,
18         if (ace == null)
19         {
20             //创建 aceing 类的对象实例, 并把引用赋值给 ace
21             ace = new aceing();
22         }
23
24         //返回并退出 ace 的值
25         return ace;
26     }
27 }
28
29 //test 类是用来测试 aceing 类的
30 public class test
31 {
32     //main 方法为 Java 程序的入口方法
33     public static void main(String args[])
34     {
35         //通过 getAceing 方法返回 aceing 类的对象实例
36         aceing a1 = aceing.getAceing();
37
38         //通过 getAceing 方法返回 aceing 类的对象实例
39         aceing a2 = aceing.getAceing();
40
41         //比较两个对象引用是否相等
42         if (a1 == a2)
43         {
44             //打印并显示结果
45             System.out.println("创建的两个对象引用指向的是一个对象");
46         }
47         else
48         {
49             //打印并显示结果
50             System.out.println("创建的两个对象引用指向的不是一个对象");
51         }
52     }
53 }
54 }
```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 10-13 所示。



图 10-13 比较对象



## 10.5 构造器在程序中是何时运行的

前面的章节里介绍了构造器是为了创建对象并对其对象的成员变量进行初始化等操作，那么在构造器运行前系统执行什么呢，之后又执行什么呢？下面总结如下：

- 加载要创建该对象的父类，以及成员变量和其他继承关系。
- 加载该类的静态块和静态成员变量，并对其进行初始化等操作。
- 静态块和静态成员变量加载完毕后创建对象并加载非静态成员变量，并对其进行初始化等操作。
- 执行构造器里的方法体，完成后该类的对象创建完毕。
- 父类的运行顺序和该类的顺序是一样的。

**【范例 10-19】**通过示例代码 10-19 来演示构造器是何时运行的。

示例代码 10-19

```

01 //aceing 类描述的是公路赛车
02 class aceing
03 {
04     //静态块
05     static
06     {
07         System.out.println("子类的静态块语句执行。");
08     }
09
10     //定义一个私有，静态类型的成员变量 acc
11     public int size;
12
13     //创建一个无参的构造器
14     public aceing()
15     {
16         //构造器的方法体
17     }
18
19     //创建一个有一个参数的构造器
20     public aceing(int size)
21     {
22         //对该类的成员变量进行赋值。
23         this.size = size;
24
25         //打印并显示信息
26         System.out.println("子类 aceing 的无参构造器调用了");
27     }
28
29     //方法 show
30     public void show()
31     {
32         System.out.println(size);
33     }
34 }
35
36 //test 类是用来测试 aceing 类的
37 public class test
38 {

```



```
39     //main方法为Java程序的入口方法
40     public static void main(String args[])
41     {
42         //通过getAcing方法返回Acing类的对象实例
43         Acing a1 = new Acing(26);
44         a1.show();
45     }
46 }
47
48 }
```

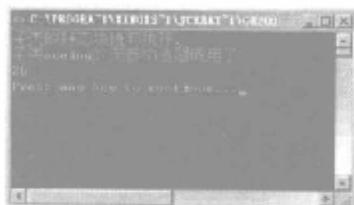


图 10-14 运行构造器

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 10-14 所示。

【代码解析】在用 new 关键字去创建对象的时候，先初始化该类的静态块和静态成员变量。静态块和静态成员变量是属于类，是类级别的，不随着对象的创建而创建。静态块和静态成员变量初始化完毕后，执行构造器方法里的方法体。构造器方法执行完毕后，创建对象，并调用该对象的方法。

**【范例 10-20】**下面是一段复杂的代码，来说明有继承关系的构造器是何时运行的。

#### 示例代码 10-20

```
01 //bike 类描述的是自行车
02 class bike
03 {
04     //父类的静态块
05     static
06     {
07         System.out.println("父类的静态块语句执行。");
08     }
09
10     //创建 bike 的成员变量
11     public String color;
12
13     //创建 bike 的无参构造方法
14     public bike()
15     {
16         //打印并显示
17         System.out.println("bike 类的无参构造器被调用了。");
18     }
19
20     //创建 bike 的有一个参数的构造器
21     public bike(String color)
22     {
23         //对该类的成员变量进行赋值
24         this.color = color;
25
26         //打印并显示
27         System.out.println("父类 bike 的构造器方法被执行。");
28     }
29
30     //父类的方法
31     public void show()
32     {
```

```

33         System.out.println("该类的其他方法被执行。");
34     }
35 }
36
37 //aceing 类描述的是公路赛车
38 class aceing extends bike
39 {
40     //子类的静态块
41     static
42     {
43         System.out.println("子类的静态块语句执行。");
44     }
45
46     //定义一个私有, 静态类型的成员变量 acc
47     public int size;
48
49     //创建一个无参的构造器
50     public aceing()
51     {
52         super("黄色");
53         //对该类的成员变量进行赋值。
54         this.size = size;
55
56         //打印并显示信息
57         System.out.println("子类 aceing 的无参构造器调用了");
58     }
59
60     //子类的方法 show, 重写了父类的方法
61     public void show()
62     {
63         System.out.println(color + " : " + size);
64     }
65 }
66
67 //test 类是用来测试 aceing 类的
68 public class test
69 {
70     //main 方法为 Java 程序的入口方法
71     public static void main(String args[])
72     {
73         //通过 getAceing 方法返回 aceing 类的对象实例
74         aceing a1 = new aceing();
75
76         //调用该类的方法 show
77         a1.show();
78     }
79 }

```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 10-15 所示。

【代码解析】在有继承关系的时候, 是先初始化父类的各个成员。创建 aceing 对象的时候先初始化父类的静态块和静态成员变量, 再初始化了类的静态块和静态成员变量。静态块和静态成员变量初始化完毕后, 进入到父类的



图 10-15 有继承关系的构造器



构造器方法里执行该构造器的方法体。父类构造器的方法体执行完毕后，执行子类的构造器方法体。子类执行完毕后，执行相应的方法。



## 10.6 综合练习

1. 看下面的程序有什么错误。

```
01 public class LianXi1
02 {
03     public LianXi1()
04     {
05         System.out.println("调用无参构造器");
06         new LianXi1("hello");
07     }
08     public LianXi1(String s)
09     {
10         System.out.println("调用有参构造器");
11         new LianXi1();
12     }
13     public static void main(String args[])
14     {
15         new LianXi1();
16     }
17 }
```

**【提示】**运行该程序，首先是交替显示两条语句的，但是最后是会发生异常的。这是因为在该程序的第 15 行首先调用无参构造器，在无参构造器中首先显示“调用无参构造器”，然后调用有参构造器。在有参构造器中，首先显示“调用有参构造器”，然后调用无参构造器。这样就形成一个循环，从而使程序不能终止，直到 Java 虚拟机发生错误。

2. 编写一个构造器重载的程序，在每一个构造器中显示一条语句。

**【提示】**可以采用构造器间调用。

```
01 public class LianXi2
02 {
03     public LianXi2()
04     {
05         new LianXi2("A");           //调用具有一个参数的构造器
06     }
07     public LianXi2(String s)
08     {
09         System.out.println("参数值为: "+s);
10         new LianXi2("B","C");    //调用具有两个参数的构造器
11     }
12     public LianXi2(String s,String ss)
13     {
14         System.out.println("第一个参数值为: "+s+", 第二个参数值为: "+ss);
15         new LianXi2("D","E","F"); //调用具有三个参数的构造器
16     }
17     public LianXi2(String s,String ss,String sss)
18     {
19         System.out.println("参数包括: "+s+", "+ss+", "+sss);
20     }
21     public static void main(String args[])
22     {
```

```

23         new LianXiz(); //调用无参构造器
24     )
25 }

```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 10-16 所示。



## 10.7 小结

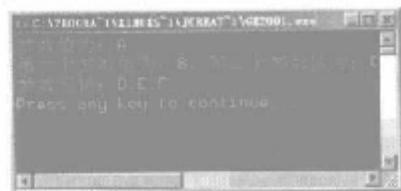


图 10-16 练习 2

本章介绍了构造器的知识，以及它的使用和注意事项。了解构造器方法的执行顺序对了解程序的执行有很好的帮助。希望读者重点了解构造器的使用这一节里的内容。如果读者想了解更多的关于本章的内容，可以参考电子工业出版社出版的《深入浅出 JDK 6.0》一书进行学习。



## 10.8 习题

### 一、填空题

1. 构造器可以使用的修饰符有\_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_。
2. 构造器有多种形式，包括\_\_\_\_\_和\_\_\_\_\_。
3. 构造器的重载的方法名称相同，但\_\_\_\_\_不相同。
4. 构造器的名称必须要和\_\_\_\_\_的名称相同。
5. 在 Java 中，实例化一个对象用\_\_\_\_\_关键字来完成。
6. 构造器的重载的方法是没有\_\_\_\_\_的。
7. 所谓单子模式，就是在一个时间段内\_\_\_\_\_只存在一份。

### 二、选择题

1. 选择下面程序的运行结果（ ）。

```

01 public class Hello
02 {
03     public static void main(String args[])
04     {
05         A a=new B();
06     }
07 }
08 class A
09 {
10     public A()
11     {
12         super();
13         System.out.print("A");
14     }
15 }
16 class B extends A
17 {
18     public B()
19     {
20         System.out.print("B");
21     }
22 }

```



21        )  
22 }

- A. A              B. B              C. AB              D. BA

2. 选择下面程序的运行结果 ( )。

```
01  public class Hello
02  {
03      public static void main(String args[])
04      {
05          A a=new B();
06      }
07  }
08  class A
09  {
10      public A()
11      {
12          System.out.print("A");
13          super();
14      }
15  }
16  class B extends A
17  {
18      public B()
19      {
20          System.out.print("B");
21      }
22  }
```

- A. 第 12 行发生编译错误              B. 第 18 行发生编译错误  
C. AB              D. BA

### 三、简答题

1. 什么是单子模式，如何使用单子模式？
2. 简述构造器的运行机制。

### 四、编程题

1. 编写一个构造器间互相调用的程序。
2. 编写一个父子类之间构造器调用的程序。

# 第 11 章 异常处理

每个人都不能保证写的程序没有错误，如果程序中可能发生错误就需要进行异常处理。在学校中，老师批改作业通常要指出学生所犯的错误，可能是准确地指出错误，也可能是给出一个错误范围，让学生在这个范围内自己查找。在 Java 中，异常处理也是这样的，通过异常处理来指出程序中的错误，可以给出一个具体异常，也可以给出一个异常范围。在本章中就来学习如何进行异常处理。通过本章的学习，读者应该实现如下几个目标。

- 了解什么是异常处理。
- 熟练掌握如何进行异常处理。
- 掌握异常的分类和区别不同的异常。
- 能够自定义异常和使用自定义异常。



## 11.1 异常处理基本介绍

本节将对异常做一个大概的介绍。异常发生的原因有很多，可能是软件的问题，也可能是硬件的问题。在 Java 程序中，对异常的处理都是一样的，一般情况下是通过 try-catch 语句来进行异常处理。该语句还可以存在 finally 语句。本节中将对这些最简单的异常处理语句进行介绍。

### 11.1.1 try 和 catch 捕获异常

通常情况下，在 Java 程序中就是采用 try-catch 语句进行异常处理的。这种方法既好用，又容易让开发员理解。try-catch 语句的基本语法如下所示。

```
try
{
    //此处是可能出现异常的代码
}
catch(Exception e)
{
    //此处是如果发生异常的处理代码
}
```

在 try 语句中放可能出现异常的代码；在 catch 语句中需要给出一个异常的类型和该类型的引用，并在 catch 语句中放当出现该异常类型时需要执行的代码。

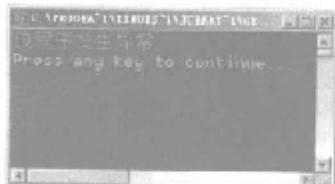
**【范例 11-1】**示例代码 11-1 演示如何使用 try-catch 语句来捕获异常。

示例代码 11-1

```
01  public class Yичang1
02  {
03      public static void main(String args[])
04      {
```



```
05     try
06     {
07         int[] a=new int[5];      //定义一个长度为 5 的数组
08         a[5]=6;                //为其中元素赋值
09     }
10     catch(Exception e)
11     {
12         //发生异常时执行的语句
13         System.out.println("该程序发生异常");
14     }
15 }
16 }
```



**【运行结果】**使用 `javac` 编译程序将产生一个和该程序对应的 `class` 程序，然后使用 Java 运行编译产生的 `class` 程序，运行结果如图 11-1 所示。

**【代码解析】**在该程序中，使用 `try` 语句将可以出现异常的语句包含起来，在这段程序中定义了一个长度为 5 的数

组，而进行赋值时为数组下标为 5 的数值进行赋值。已经知道数组下标是从 0 开始的，所以长度为 5 的数组，下标最大应该为 4，所以该程序将发生异常。当发生异常就会执行 `catch` 语句中的程序，从而显示“该程序发生异常”。

### 11.1.2 try-catch 语句使用注意点

使用 `try-catch` 语句时有很多注意点和技巧，在一开始学时就应该了解这些。有些初学者会认为使用了 `try-catch` 语句的程序就会发生异常，这是不对的。`try-catch` 语句是对有可能发生异常的程序进行查看，如果没有发生异常，就不会执行 `catch` 语句中的内容。

在程序中如果不使用 `try-catch` 语句，则当程序发生异常的时候，会自动退出程序的运行；而使用 `try-catch` 语句后，当程序发生异常的时候，会执行 `catch` 语句中的程序，从而使程序不自动退出。在前面的学习中经常会看到出现异常的情况，如果在其中使用 `try-catch` 语句就不会出现那种异常信息。



**注意：** `try-catch` 语句是对有可能发生异常的程序进行查看，如果没有发生异常，就不会执行 `catch` 语句中的内容。在程序中如果不使用 `try-catch` 语句，则当程序发生异常的时候，会自动退出程序的运行。

`try-catch` 语句中的 `catch` 语句可以不只是一个，可以存在多个 `catch` 语句来定义可能发生的多个异常。当处理任何一个异常时，则将不再执行其他 `catch` 语句。

**【范例 11-2】**在示例代码 11-2 中给出了一个使用多个 `catch` 语句进行异常处理的程序。

#### 示例代码 11-2

```
01 public class YiChang2
02 {
03     public static void main(String args[])
04     {
05         try
06         {
07             int[] a=new int[5];      //定义一个长度为 5 的数组
08             a[5]=6;                //为其中元素赋值
09         }
10         catch(ArrayIndexOutOfBoundsException e)
11         {
12             System.out.println("数组越界异常");
13         }
14     }
15 }
```

```

09     }
10     catch(ArrayIndexOutOfBoundsException e)
11     {
12         //发生数组下标越界异常时执行的语句
13         System.out.println("该程序发生数组下标越界异常");
14     }
15     catch(Exception ee)
16     {
17         //发生异常时执行的语句
18         System.out.println("该程序发生异常");
19     }
20 }
21 }

```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 11-2 所示。

**【代码解析】**本程序中的第 7 行和第 8 行同样还是定义了一个发生数组下标越界的程序。但是在该程序中使用了两个 catch 语句。在第一个 catch 语句中 ArrayIndexOutOfBoundsException 就是数组下标异常, 它是 Exception 异常的子类。当发现程序发生数组下标越界异常, 则会执行第一个 catch 语句中的语句, 执行该 catch 语句后将不再执行其他的 catch 语句。从而使第二个 catch 语句中的语句不再执行。

当对程序使用多个 catch 语句进行异常处理时, 特别需要注意的是要将范围相对小的异常放在前面, 将范围相对大的异常放在后面, 这通过程序是很容易理解的。

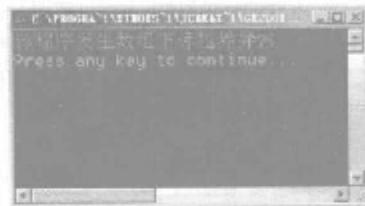


图 11-2 多个 catch 语句



**注意:** 当对程序使用多个 catch 语句进行异常处理时, 要将范围相对小的异常放在前面, 将范围相对大的异常放在后面。

**【范例 11-3】**示例代码 11-3 是一个错误的使用多个 catch 语句的程序。

#### 示例代码 11-3

```

01 public class YiChang3
02 {
03     public static void main(String args[])
04     {
05         try
06         {
07             int[] a=new int[5];          //定义一个长度为 5 的数组
08             a[5]=6;                   //为其中元素赋值
09         }
10         catch(Exception ee)
11         {
12             //发生异常时执行的语句
13             System.out.println("该程序发生异常");
14         }
15         catch(ArrayIndexOutOfBoundsException e)
16         {
17             //发生数组下标越界异常时执行的语句
18             System.out.println("该程序发生数组下标越界异常");
19         }
20 }
21 }

```



```
20      )  
21  }
```

【代码解析】该程序只是将示例代码 11-2 的程序中的两个 catch 语句换了一下位置。该程序是一个错误的程序。因为其中第一个 catch 语句将对所有的异常进行处理，执行该 catch 语句后将不执行后面的 catch 语句，从而使第二个 catch 语句永远执行不到。这一点在初学时是要特别注意的。

### 11.1.3 finally 语句的使用

在 try-catch 语句中还可以具有 finally 语句。在实际开发中经常要使用到 finally 语句，尤其是将在后面学习到的数据库操作中。连接数据库是可以发生异常的，当然也是可能不发生异常的。但是有一点，不管是否发生异常，连接数据库所用到的资源都是需要关闭的，这些操作是必须执行的，这些执行语句就可以放在 finally 语句中。即在 finally 语句中就是放肯定会被执行的语句。



**提示：**不管是否发生异常，连接数据库所用到的资源都是需要关闭的，这些操作是必须执行的。

finally 语句的语法形式如下所示。

```
try  
{  
    //此处是可能出现异常的代码  
}  
catch(Exception e)  
{  
    //此处是如果发生异常的处理代码  
}  
finally  
{  
    //此处是肯定被执行的代码  
}
```

【范例 11-4】示例代码 11-4 是一个使用 finally 语句的程序。

示例代码 11-4

```
01  public class YiChang4  
02  {  
03      public static void main(String args[])  
04      {  
05          try  
06          {  
07              int[] a=new int[5];      //定义一个长度为 5 的数组  
08              a[5]=6;                //为其中元素赋值  
09          }  
10          catch(ArrayIndexOutOfBoundsException e)  
11          {  
12              //发生数组下标越界异常时执行的语句  
13              System.out.println("该程序发生数组下标越界异常");  
14          }  
15          catch(Exception ee)  
16          {  
17              //发生异常时执行的语句  
18              System.out.println("该程序发生异常");  
19          }  
20      }  
21  }
```

```

19
20
21
22     //将必须要执行的语句放在 finally 语句中
23     System.out.println("该语句是肯定执行的");
24
25 }
26 }

```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 11-3 所示。

【代码解析】在本程序中定义了一个 finally 语句，finally 语句中的第 23 行代码语句是肯定会被执行的。为了更好地看到这一点，再来看一个和示例代码 11-4 相对的程序示例代码 11-5。

**【范例 11-5】**示例代码 11-5 是一个和示例代码 11-4 相对的程序。

示例代码 11-5

```

01 public class Yичang5
02 {
03     public static void main(String args[])
04     {
05         try
06         {
07             int[] a=new int[5];          //定义一个长度为 5 的数组
08             a[0]=6;                   //为其中元素赋值
09         }
10     catch(ArrayIndexOutOfBoundsException e)
11     {
12         //发生数组下标越界异常时执行的语句
13         System.out.println("该程序发生数组下标越界异常");
14     }
15     catch(Exception ee)
16     {
17         //发生异常时执行的语句
18         System.out.println("该程序发生异常");
19     }
20     finally
21     {
22         //将必须要执行的语句放在 finally 语句中
23         System.out.println("该语句是肯定执行的");
24     }
25 }
26 }

```



图 11-4 finally 语句 2

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 11-4 所示。

【代码解析】该程序是和示例代码 11-4 相对的。在该程序的第 7 行和第 8 行并没有发生异常，所以不会执行 catch 语句中的语句。但是从运行结果中可以看出，第 23 行中的 finally 语句中的程序被执行。由本段

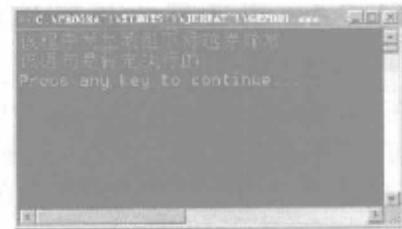


图 11-3 finally 语句



程序和示例代码 11-4 可以知道, finally 语句是无论程序中是否发生异常都被执行的。

finally 语句虽然在程序中肯定执行, 但是为了确保知识的严谨性, 这里也给出了几个可能会中断 finally 语句执行的情况。首先是 finally 语句中本身就产生异常; 再者就是执行 finally 语句的线程死亡, 线程的问题会在后面的学习中学习到; 还有一种情况, 那就是程序执行到 finally 语句时停电了。

#### 11.1.4 再谈异常处理注意点

学习 finally 语句后, 又多出了很多在写 try-catch-finally 语句时需要注意的地方。这些在开发中是比较少见的, 但是在考试中经常会出现。第一个注意点就是当不存在 catch 语句时, finally 语句必须存在并且紧跟在 try 语句后面。读者可以自己写程序来验证这一点。

还有一个需要注意的格式是在 try 语句和 catch 语句间不能存在任何语句, 同样在 catch 语句和 finally 语句中也不能存在任何语句, 这里的语句不包括注释语句。



**注意:**当不存在 catch 语句时, finally 语句必须存在并且紧跟在 try 语句后面。

在 try 语句和 catch 语句间不能存在任何语句, 同样在 catch 语句和 finally 语句中也不能存在任何语句。

**【范例 11-6】**示例代码 11-6 是一个讲解 try-catch-finally 语句格式注意点的程序。

示例代码 11-6

```
01  public class YiChang5
02  {
03      public static void main(String args[])
04      {
05          try
06          {
07              int[] a=new int[5];          //定义一个长度为 5 的数组
08              a[5]=6;                      //为其中元素赋值
09          }
10          //这里放注释语句是可以的
11          catch(ArrayIndexOutOfBoundsException e)
12          {
13              //发生数组下标越界异常时执行的语句
14              System.out.println("该程序发生数组下标越界异常");
15          }
16          catch(Exception ee)
17          {
18              //发生异常时执行的语句
19              System.out.println("该程序发生异常");
20          }
21          //这里不能放除注释外的其他语句
22          System.out.println("该语句放在这是不对的");
23          finally
24          {
25              //将必须要执行的语句放在 finally 语句中
26              System.out.println("该语句是肯定执行的");
27          }
28      }
29  }
```

【代码解析】运行该程序是会发生错误的，这是因为第 22 行程序存在于 catch 语句和 finally 语句间。在学习时不要只学一半知识，这一点还是需要强调的，这里不能有任何语句，不包括注释语句。注释语句是不会在程序执行时被执行的。



## 11.2 异常的分类

可以对异常进行分类，从大的角度将异常分为捕获异常和未捕获异常两类。在 Java 类库中有一个叫做 `Throwable` 类，该类继承于 `Object` 类。所有的异常类都是继承 `Throwable` 类，`Throwable` 类有两个直接子类，`Error` 类和 `Exception` 类。在 `Exception` 类中又有一个 `RuntimeException` 类。在 `Exception` 类中的直接和间接子类中除去 `RuntimeException` 类的直接和间接子类，都是捕获异常。其他的都为未捕获异常。

### 11.2.1 捕获异常

捕获异常是在翻译外文书时译者给起的名字，如果直接翻译的话是必须处理异常。捕获异常通常是由外部因素造成的，不是由程序造成的。例如连接网络等操作，这些是和很多因素有关系的，有可能并不是程序的错误。虽然这些错误并不是程序的错误，但也是必须要进行处理的。



**提示：**捕获异常是在翻译外文书时译者给起的名字，如果直接翻译的话是必须处理异常。

在 Java 程序中对捕获异常的处理是必须进行异常处理，虽然有可能程序并不会出现异常。这种设计大大提高了程序的稳定性。下面通过程序代码来进行捕获异常的讲解。

**【范例 11-7】**示例代码 11-7 是一个对可能发生捕获异常语句没有进行异常处理的程序。

示例代码 11-7

```

01 import java.net.*;
02 public class YiChang{
03 {
04     public static void main(String args[])
05     {
06         //一个可能出现捕获异常的语句
07         ServerSocket s=new ServerSocket(8080);
08     }
09 }
```

【代码解析】运行该程序是会发生错误的。错误信息为“未报告的异常 `java.io.IOException`：必须对其进行捕捉或声明以便抛出”。该错误信息的意思就是 `IOException` 异常是一个捕获异常，必须对该异常进行异常处理。为了能让程序运行，将该程序修改成示例代码 11-8 的形式。

**【范例 11-8】**示例代码 11-8 是一个对捕获异常进行处理的程序。

示例代码 11-8

```

01 import java.net.*;
02 import java.io.*;
```



```
03  public class YiChang8
04  {
05      public static void main(String args[])
06      {
07          try
08          {
09              //一个可能出现捕获异常的语句
10              ServerSocket s=new ServerSocket(8080);
11          }
12          catch(IOException e)
13          {
14              System.out.println("程序发生捕获异常 IOException");
15          }
16      }
17  }
```

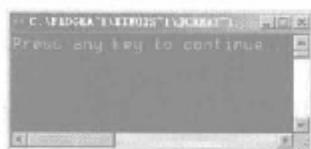


图 11-5 处理捕获异常

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 11-5 所示。

**【代码解析】**从运行结果中可以看出，程序是能够正常运行的。在该程序中虽然没有出现 IOException 捕获异常，但是如果不用 try-catch 语句进行处理也是不能够运行通过的。

在对捕获异常进行处理时要谨慎，如果语句中不可能出现捕获异常，但是程序中仍然对该语句进行捕获异常处理，这样该程序运行时是会发生错误的。



**注意：**如果语句中不可能出现捕获异常，但是程序中仍然对该语句进行捕获异常处理，这样该程序运行时是会发生错误的。

**【范例 11-9】**示例代码 11-9 是一个对错误进行捕获异常处理的程序。

示例代码 11-9

```
01  import java.io.*;
02  public class YiChang9
03  {
04      public static void main(String args[])
05      {
06          try
07          {
08              //一个不可能出现捕获异常的语句
09              System.out.println("我不会发生捕获异常");
10          }
11          catch(IOException e)
12          {
13              System.out.println("程序发生捕获异常 IOException");
14          }
15      }
16  }
```

**【代码解析】**运行该程序是会发生错误的。错误信息为“在相应的 try 语句主体中不能抛出异常 java.io.IOException”。该程序中的第 9 行是不可能发生捕获异常的，而本程序进行了捕获异常处理，所以这是没有必要的，从而造成程序发生错误，该情况是针对捕获异常来说的。在下一节中将对未捕获异常进行讲解，未捕获异常是不会出现这种情况的。



## 11.2.2 未捕获异常

在异常中，除了捕获异常以外的都是未捕获异常。未捕获异常包括 `Error` 类以及它的直接子类和间接子类和 `RuntimeException` 类，以及它的直接子类和间接子类。`Error` 类及它的子类通常是由硬件运行错误所导致的错误。这些是很严重的错误，通常是不能通过程序来进行修改的。`RuntimeException` 类以及它的子类通常是程序运行时引起的异常。



**提示：**`Error` 类及它的子类通常是由硬件运行错误所导致的错误。这些是很严重的错误，通常是不能通过程序来进行修改的。

前面使用最多的就是数组下标越界异常，该异常就是未捕获异常。未捕获异常是可以不进行异常处理的，如果不进行异常处理编译的时候是完全没有问题的，但是在运行时会发生错误。

**【范例 11-10】**下面是没有对未捕获异常进行处理的示例代码 11-10。

示例代码 11-10

```
01 public class YiChang10
02 {
03     public static void main(String args[])
04     {
05         int[] a=new int[5]; //定义一个长度为 5 的数组
06         a[5]=6; //为其中元素赋值
07     }
08 }
```

**【运行结果】**使用 `javac` 编译程序将产生一个和该程序对应的 `class` 程序，然后使用 `Java` 运行编译产生的 `class` 程序，运行结果如图 11-6 所示。

**【代码解析】**从程序中可以看到该程序是会发生数组下标越界的未捕获异常的，该程序在编译时是能够正常编译的，但是在运行时就会出现如图 11-6 所示的错误信息。所以通常情况下未捕获异常也是需要进行异常处理的。对未捕获异常进行异常处理的程序在前面已经介绍了很多，这里就不再对该程序进行异常处理，读者可以自己把该工作完成。

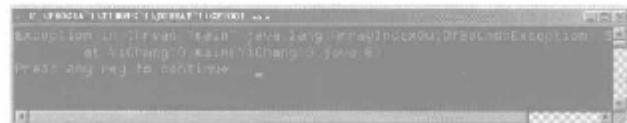


图 11-6 没有处理未捕获异常



## 11.3 抛出异常

对异常的处理不是只有前面讲的使用 `try-catch` 语句来进行处理的。在有些情况下，异常是不需要立即进行处理的，但是也必须要进行异常处理，这时候就用到抛出异常的内容。

### 11.3.1 抛出异常的简单介绍

日常生活中，例如学校中有什么问题都会先去问老师，但是有一些问题例如转学是不



能由老师来解决的，这时候老师就需要再去问校长，由校长来解决这个问题。可能校长还有不能解决的问题，就需要去问教育部。抛出异常也是这样的，当一个程序段发生异常时，如果自己不能够进行异常处理，就可以抛出异常给上一层。如果上一层也不能解决就可以一直向上抛出异常，直到抛出给 main 方法。如果仍然不能解决，就会中断程序，将异常显示出来。

**提示：**当一个程序段发生异常时，如果自己不能够进行异常处理，就可以抛出异常给上一层。如果上一层也不能解决就可以一直向上抛出异常，直到抛出给 main 方法。

**【范例 11-11】**示例代码 11-11 是一个抛出异常的程序。

示例代码 11-11

```
01  public class YiChang11
02  {
03      public static void main(String args[])
04      {
05          YiChang11 yc=new YiChang11();
06          yc.a();
07      }
08      //在 a 方法中进行异常处理
09      public void a()
10      {
11          try
12          {
13              b();
14          }
15          catch(Exception e)
16          {
17              System.out.println("在该处进行异常处理");
18          }
19      }
20      //该方法没有进行异常处理，抛出异常给 a 方法
21      public void b()
22      {
23          c();
24      }
25      //该方法没有进行异常处理，抛出异常给 b 方法
26      public void c()
27      {
28          int[] a=new int[5];          //定义一个长度为 5 的数组
29          a[5]=6;                      //为其中元素赋值
30      }
31  }
```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 11-7 所示。

**【代码解析】**在该程序中，调用顺序是 main 方法调用 a 方法，a 方法调用 b 方法，b 方法调用 c 方法。在 c 方法中发生异常，但是在 c 方法中并没有进行异常处理，c 方法就会将异常抛出给 b 方法。在 b 方法中同样没有进行异常

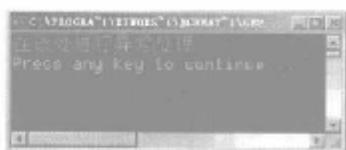


图 11-7 抛出异常

处理, b 方法就会将异常抛出给 a 方法中。在 a 方法中进行了异常处理, 从而使程序运行正常。

### 11.3.2 使用 throws 和 throw 语句抛出异常

在抛出异常的操作中, 不但可以使用上一节中的抛出异常的方法, 还可以使用 throws 语句和 throw 语句进行抛出异常。throws 语句是在方法的声明中使用来抛出异常, 而 throw 语句是在方法体内使用抛出异常。

**提示:** throws 语句是在方法的声明中使用来抛出异常, 而 throw 语句是在方法体内使用抛出异常。

**【范例 11-12】**示例代码 11-12 是一个使用 throws 和 throw 语句抛出异常的程序。

#### 示例代码 11-12

```

01 import java.net.*;
02 public class YiChang12
03 {
04     public static void main(String args[]) throws IOException
05     {
06         try
07         {
08             //一个可能出现捕获异常的语句
09             ServerSocket s=new ServerSocket(8080);
10         }
11         catch(IOException e)
12         {
13             throw e;
14         }
15     }
16 }
```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 11-8 所示。

**【代码解析】**该程序是能够正常运行的。该程序虽然对第 9 行的可能出现捕获异常的语句进行了处理, 但是使用 throw 语句在抛出该 IOException 异常, 并在 main 方法中使用 throws 语句声明该方法抛出 IOException 异常。



图 11-8 抛出异常

## 11.4 自定义异常

在 Java 中定义了非常多的异常类, 几乎覆盖了可能出现的所有问题, 但是再多的定义也不可能满足所有的情况, 这时候就需要来进行自定义异常。本节就来学习如何创建自定义异常, 并介绍如何使用自定义异常。

### 11.4.1 创建和使用自定义异常类

创建自定义异常类需要继承 Exception 类。在自定义的异常类中通过具有一个无参构造器和一个带有字符串参数的有参构造器。



**【范例 11-13】**示例代码 11-13 是一个最简单的自定义异常类。

示例代码 11-13

```
class MyException extends Exception
{
    public MyException()
    {
    }
    public MyException(String s)
    {
        super(s);
    }
}
```

**【代码解析】**在该自定义异常类的程序中，让自定义的类继承 Exception 类，其中定义了一个无参构造器和一个需要字符串类型的有参构造器。这是一个最简单的自定义异常类。

在 Exception 类中定义很多方法，这里讲解一些最常见的方法。使用 printStackTrace 方法可以显示异常调用栈的信息。使用 toString 方法可以得到异常对象的字符串表示。使用 getMessage 方法可以得到异常对象中携带的出错信息。在自定义的异常类中因为继承了 Exception 类，所以同时拥有这些方法。

**【范例 11-14】**示例代码 11-14 是一个使用自定义异常类的程序。

示例代码 11-14

```
public class Yichang13
{
    public static void main(String args[])
    {
        //创建一个自定义异常对象
        MyException me=new MyException("自定义异常");
        //获取自定义异常信息
        System.out.println("自定义异常对象携带的错误信息为：" +me.getMessage());
        System.out.println("自定义异常的字符串表示为：" +me.toString());
    }
}
```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 11-9 所示。



图 11-9 使用自定义异常

**【代码解析】**该程序是一个简单的使用自定义异常类的程序。在该程序中首先创建了一个自定义异常对象，然后调用该异常类中的方法，来显示异常的信息。

### 11.4.2 自定义异常的实际应用

本节学习如何在实际应用中使用自定义异常。这里看一个非常简单的实际应用。在百

分制的考试当中, 0 到 100 的得分都是可能发生的, 但是超出这个范围, 则肯定是发生了错误。这里就可以使用自定义异常, 当使用不在 0 到 100 范围内的数的时候就会发生自定义异常。自定义异常类仍然还是采用上一节中定义的自定义异常。

**【范例 11-15】**示例代码 11-15 是一个在实际应用中使用自定义异常的程序。

示例代码 11-15

```
public class YiChang14
{
    // 定义一个可能发生自定义异常的方法
    public String deiFen(int fen) throws MyException
    {
        if(fen>=0&&fen<=100)
        {
            return "正常";
        }
        else
        {
            // 当分数不在 0 到 100 的范围内时抛出自定义异常
            throw new MyException("错误输入");
        }
    }
    public static void main(String args[])
    {
        YiChang14 yc=new YiChang14();
        try
        {
            String s=yc.deiFen(68);           // 68 在范围内, 不会发生异常
            System.out.println(s);
            String ss=yc.deiFen(123);        // 123 不在范围内, 会发生异常
            System.out.println(ss);
        }
        catch(MyException e)
        {
            System.out.println("异常信息为: "+e.getMessage());
        }
    }
}
```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 11-10 所示。

**【代码解析】**在示例代码 11-15 中使用了自定义异常。在该程序的主方法中, 首先使用 68 为参数调用方法, 因为 68 在 0 到 100 之间, 所以是不会发生异常的, 从而显示“正常”。第二次是使用 123 为参数调用方法, 由于已经超出了范围, 所以是会抛出自定义异常的。在主方法中将对自定义异常进行异常处理, 在 catch 语句中将显示该异常信息。

读者在学习示例代码 11-15 时也许感觉不到使用自定义异常的优越性, 这是因为在实际应用中是不会将抛出异常的方法和调用方法写在一起的。



图 11-10 自定义异常应用



**提示:** 读者在学习示例代码 11-15 时也许感觉不到使用自定义异常的优越性, 这是因为在实际应用中是不会将抛出异常的方法和调用方法写在一起的。



**【范例 11-16】**示例代码 11-16 是开发中最常见的使用自定义异常的程序。首先开发一个定义可能抛出异常方法的类。

示例代码 11-16

```
public class YiChang15
{
    //定义一个可能发生自定义异常的方法
    public String deiFen(int fen) throws MyException
    {
        if(fen>=0&&fen<=100)
        {
            return "正常";
        }
        else
        {
            //当分数不在 0 到 100 的范围内时抛出自定义异常
            throw new MyException("错误输入");
        }
    }
}
```

该方法定义后，其他人就可以进行使用了。先看一个不会发生自定义异常的程序。

```
public class YiChang16
{
    public static void main(String args[])
    {
        YiChang15 yc=new YiChang15();
        try
        {
            String s=yc.deiFen(68);           //68 在范围内，不会发生异常
            System.out.println(s);
        }
        catch(MyException e)
        {
            System.out.println("异常信息为: "+e.getMessage());
        }
    }
}
```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 11-11 所示。

再看一个会发生自定义异常的程序。

```
public class YiChang17
{
    public static void main(String args[])
    {
        YiChang14 yc=new YiChang14();
        try
        {
            String ss=yc.deiFen(123); //123 不在范围内，会发生异常
            System.out.println(ss);
        }
        catch(MyException e)
        {
            System.out.println("异常信息为: "+e.getMessage());
        }
    }
}
```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 11-12 所示。



图 11-11 没有发生异常

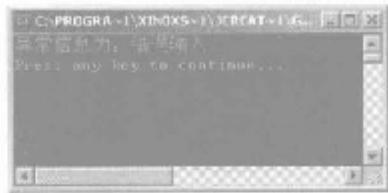


图 11-12 发生异常

【代码解析】上面的两个程序都调用了可能抛出自定义异常的类。该示例代码和上一个示例代码相比并没有更多的功能。但是读者会发现该程序更适合当有很多人同时使用自定义异常的时候。这种设计在实际的团队开发中经常使用, 该团队为了完成自己需要的功能, 定义了一个自定义异常类和抛出该异常类的方法。这些只需要一次编写, 其他人就可以直接来使用该方法和自定义异常类。



## 11.5 综合练习

1. 判断下面的程序是否能够正常运行。

```

01 import java.io.*;
02 class Fu
03 {
04     public void myVoid()throws IOException
05     {
06         System.out.println("父类");
07     }
08 }
09 public class LianXii extends Fu
10 {
11     public void myVoid()throws Exception
12     {
13         System.out.println("子类");
14     }
15     public static void main(String args[])
16     {
17     }
18 }
```

【提示】编译该程序是会发生错误的, 这是因为在子类中重写了父类中的 myVoid 方法, 但是在父类中的方法使用 throws 抛出 IOException 异常, 而子类的方法抛出 Exception 异常, 这样违反了重写规则, 所以是编译会发生错误的。

2. 判断下面的程序是否能够正常运行。

```

01 import java.io.*;
02 class Fu
03 {
04     public void myVoid()throws IOException
05     {
06         System.out.println("父类");
07     }
08 }
```



```
09  public class LianXi2 extends Fu
10  {
11      public void myVoid(String s) throws Exception
12      {
13          System.out.println("子类");
14      }
15      public static void main(String args[])
16      {
17      }
18  }
```

【提示】有些读者可能会认为该程序和上一个程序一样也会发生编译错误的，这里不是这样的，该程序能够正常编译。因为在该程序子类中的 myVoid 方法并不是重写的父类中的 myVoid 方法，而是重载的方法，从而不会发生编译错误。



## 11.6 小结

本章主要对 Java 中的异常处理进行了详细的讲解。首先讲解了异常处理的基本结构，然后讲解了异常处理的分类，最后还对如何定义自己的异常进行了讲解。如果读者想了解更多的关于本章的内容，可以参考电子工业出版社出版的《深入浅出 JDK 6.0》一书进行学习。



## 11.7 习题

### 一、填空题

1. 所有的异常类都是继承 \_\_\_\_\_ 类，`Throwable` 类有两个直接子类，\_\_\_\_\_ 类和 \_\_\_\_\_ 类。
2. 未捕获异常包括 \_\_\_\_\_ 类及它的直接子类和间接子类和 \_\_\_\_\_ 类及它的直接子类和间接子类。
3. 使用 \_\_\_\_\_ 方法可以显示异常调用栈的信息。使用 \_\_\_\_\_ 方法可以得到异常对象的字符串表示。使用 \_\_\_\_\_ 方法可以得到异常对象中携带的出错信息。
4. 当不存在 `catch` 语句时，`finally` 语句必须存在并且紧跟在 \_\_\_\_\_ 语句后面。
5. \_\_\_\_\_ 语句是在方法的声明中使用来抛出异常，而 \_\_\_\_\_ 语句是在方法体内使用抛出异常。
6. 创建自定义异常类需要继承 \_\_\_\_\_ 类。在自定义的异常类中通过具有一个无参构造器和一个带有字符串参数的有参构造器。

### 二、选择题

1. 选择下面程序的运行结果（ ）。

```
01  public class Hello
02  {
03      public static void main(String args[])
04      {
05          try
06          {
07              System.out.println("A");
08          }
09      }
10  }
```

```

08     }
09     finally
10     {
11         System.out.println("B");
12     }
13 }
14

```

- A. 没有任何输出      B. A  
 C. AB      D. 发生编译错误
2. 选择下面程序的运行结果 ( )。

```

01 public class Hello
02 {
03     public static void main(String args[])
04     {
05         try
06         {
07             int [] a=new int[3];
08             a[3]=4;
09             System.out.println("A");
10         }
11         catch(ArrayIndexOutOfBoundsException e)
12         {
13             System.out.println("B");
14         }
15         catch(Exception ee)
16         {
17             System.out.println("C");
18         }
19         finally
20         {
21             System.out.println("D");
22         }
23     }
24 }

```

- A. ABD      B. BD      C. BCD      D. ABCD

### 三、简答题

1. 如何定义和使用自定义异常。
2. try-catch 有哪些注意点。

### 四、编程题

1. 编写一个程序来说明多个 catch 语句顺序的重要性。
2. 创建一个自定义异常，并使用该异常。
3. 编写一个异常再抛出的程序。

# 第 12 章 内 部 类

以前看到过这样一条新闻，说在一个鸡蛋中发现里面有一个小鸡蛋，这个小鸡蛋同样有蛋清和蛋黄，当时感觉很奇怪。后来学习 Java 后，发现在 Java 中竟然也有这种奇怪的事，那就是内部类。内部类就好像刚提到的鸡蛋中的小鸡蛋一样，包含在另一个类中。通过本章的学习，读者会发现内部类还有很多和该小鸡蛋相似的地方。读者通过本章的学习，应该实现如下几个目标。

- 了解什么是非静态内部类和如何进行非静态内部类和外部类之间的访问。
- 了解什么是局部内部类和如何进行局部内部类和外部类之间的访问。
- 了解什么是静态内部类和如何进行静态内部类和外部类之间的访问。
- 了解什么是匿名内部类和如何进行匿名内部类和外部类之间的访问。



## 12.1 非静态内部类

当一个类作为另一个类的非静态成员时，则这个类就是一个非静态内部类。在本节中将学习如何创建和使用非静态内部类，同时也讲解如何在内部类中访问外部类和在外部类中如何访问内部类。

### 12.1.1 创建非静态内部类

创建非静态内部类很容易，只需要定义一个类让该类作为其他类的非静态成员。该非静态内部类和成员变量或成员方法没有区别，同样可以在非静态内部类前面加可以修饰成员的修饰符。

**提示：**创建非静态内部类只需要定义一个类让该类作为其他类的非静态成员。该非静态内部类和成员变量或者成员方法没有区别，同样可以在非静态内部类前面加可以修饰成员的修饰符。

创建非静态内部类的基本语法如下所示。

```
class Wai
{
    class Nei
    {
        // 内部类成员
    }
    // 外部类成员
}
```

**【范例 12-1】**示例代码 12-1 是一个创建非静态内部类的程序。

## 示例代码 12-1

```

01  public class NeiBuLei1
02  {
03      private class Nei          //创建非静态内部类
04      {
05          int i=1;            //内部类成员
06      }
07      int ii=2;              //外部类成员
08      public static void main(String args[])
09      {
10          System.out.println("创建一个非静态内部类");
11      }
12  }

```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 12-1 所示。

【代码解析】对一个内部类进行分析，需要从两个方面来看。首先从内部类本身来看，它仍然是一个类，第 5 行为该类的成员变量。其次从外部类来看，内部类就是一个成员，它和外部类第 7 行的成员变量是并列关系。正因为内部类在外部看来是一个成员，所以能使用修饰成员的 private 修饰符来修改内部类，这对于一个正常的类是不可能做到的。

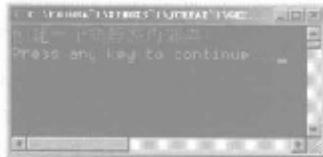


图 12-1 非静态内部类

### 12.1.2 在外部类中访问内部类

在内部类的程序中，经常会进行外部类和内部类之间访问。在外部类中访问内部类是很容易的，只要把内部类看成一个类，然后创建该类的对象，使用对象来调用内部类中的成员就可以。

【范例 12-2】示例代码 12-2 是一个在外部类中访问内部类的程序。

## 示例代码 12-2

```

01  class Wai
02  {
03      class Nei          //创建非静态内部类
04      {
05          int i=5;        //内部类成员
06      }
07      public void myVoid() //外部类成员
08      {
09          Nei n=new Nei(); //创建一个内部类对象
10          int ii=n.i;      //访问内部类成员
11          System.out.println("内部类的变量值为: "+ii);
12      }
13  }
14  public class NeiPutLei2
15  {
16      public static void main(String args[])
17      {
18          Wai w=new Wai();
19          w.myVoid();
20      }
21  }

```

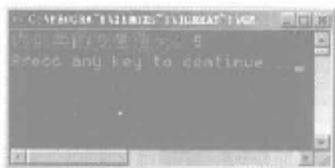


图 12-2 访问内部类

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 12-2 所示。

【代码解析】从程序的第 16 行主方法讲起，在 main 方法中，首先创建一个外部类对象，然后访问外部类的成员方法。在外部类的成员方法中，创建了一个内部类对象，然后使用内部类对象调用内部类的成员变量，从而得到结果。编译该程序将产生三个 class 文件，分别是主类、外部类和内部类。内部类产生的 class 文件的名称为 Wai\$Nei.class，在该名称中可以区分该内部类到底是哪一个类的内部类。

### 12.1.3 在外部类外访问内部类

不但可以在外部类中访问内部类，还可以在外部类外访问内部类。读者肯定会觉得非常难的是，要想访问类成员中的成员怎么访问呢？其实在 Java 中，这很容易做到。在外部类外访问内部类的基本语法如下所示。

```
Wai.Nei wn=new Wai().new Nei();
```

使用该方法就能够创建一个内部类对象，使用该内部类对象就可以访问内部类的成员。该方法是不容易理解的，该方法也是可以分为两条语句的。

```
Wai w=new Wai();
Wai.Nei wn=w.new Nei();
```

这样就很容易理解了。首先是创建一个外部类的对象，然后让该外部类对象调用创建一个内部类对象。

【范例 12-3】示例代码 12-3 是一个在外部类外访问内部类的程序。

示例代码 12-3

```
01  class Wai
02  {
03      class Nei           //创建非静态内部类
04      {
05          int i=5;        //内部类成员
06          int ii=6;
07      }
08  }
09  public class NeiBuLei3
10  {
11      public static void main(String args[])
12      {
13          Wai.Nei wn1=new Wai().new Nei();
14          Wai w=new Wai();
15          Wai.Nei wn2=w.new Nei();
16          System.out.println("内部类中的变量i的值为: "+wn1.i);
17          System.out.println("内部类中的变量ii的值为: "+wn2.ii);
18      }
19  }
```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 12-3 所示。

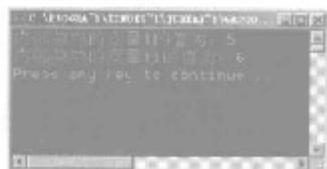


图 12-3 在外部类外访问内部类

【代码解析】在示例代码 12-3 中使用了两种方法来从外部类外访问内部类。在外部类外访问内部类时，是不能够直接创建内部类对象的，因为内部类只是外部类的一个成员。所以要想创建内部类对象，首先要创建外部类对象，然后以外部类对象为标识来创建内部类对象。



**注意：**在外部类外访问内部类时，不一定是肯定会访问成功的。因为内部类是外部类的成员，它可能存在 `private` 修饰符，使该内部类变成一个私有的成员，这样就不能访问的。

**【范例 12-4】**示例代码 12-4 是一个在外部类外访问内部类失败的程序。

示例代码 12-4

```

01 class Wai
02 {
03     private class Nei           //创建非静态内部类
04     {
05         int i=5;              //内部类成员
06         int ii=6;
07     }
08 }
09 public class NeiBuLei4
10 {
11     public static void main(String args[])
12     {
13         Wai.Nei w1=new Wai().new Nei();
14         Wai w=new Wai();
15         Wai.Nei w2=w.new Nei();
16         System.out.println("内部类中的变量 i 的值为: "+w1.i);
17         System.out.println("内部类中的变量 ii 的值为: "+w12.ii);
18     }
19 }
```

【代码解析】运行该程序是会发生错误的，错误信息是不能访问 `private` 修饰的成员。从该程序也能再一次看出非静态成员内部类从外部看来完全是一个外部类的成员。示例代码 12-4 中发生的错误是经常会在考试中出现的，在实际开发中同样需要特别注意。

#### 12.1.4 在内部类中访问外部类

在内部类中访问外部类，就像所有的同一个类中成员互相访问一样，这样是没有限制的，包括将成员声明为 `private` 私有的。

**【范例 12-5】**示例代码 12-5 是一个在内部类中访问外部类的程序。

示例代码 12-5

```

01 class Wai
02 {
03     int i=8;                  //外部类成员变量
04     class Nei                //创建非静态内部类
05     {
06         public void myVoid()  //内部类成员变量
07         {
08             System.out.println("外部类中的成员变量值为: "+i);
09         }
10     }
11 }
```



```
09         }
10     }
11 }
12 public class NeiBuLei5
13 {
14     public static void main(String args[])
15     {
16         Wai w=new Wai();           //创建外部类对象
17         Wai.Nei wn2=w.new Nei(); //创建内部类对象
18         wn2.myVoid();           //调用内部类中成员
19     }
20 }
```

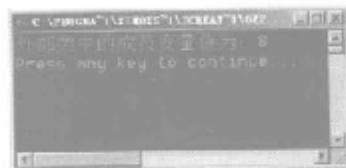


图 12-4 内部类中访问外部类

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 12-4 所示。

**【代码解析】**在示例代码 12-5 中，在内部类中定义了一个 myVoid 来访问外部类中的成员变量 i。可以看到从内部类中访问外部类是非常容易的，不需要添加任何内容，就像成员方法调用一样。

有些读者学习完示例代码 12-5 后，会有疑问，如果外部类中也有一个成员变量 i 怎么办呢？读者可以进行实验，从结果中可以看到得到的是内部类成员变量的值。下面通过示例代码 12-6 解决这个问题。

**【范例 12-6】**示例代码 12-6 是一个在内部类和外部类中具有同名称变量访问的程序。

#### 示例代码 12-6

```
01 class Wai
02 {
03     int i=8;           //外部类成员变量
04     class Nei
05     {
06         int i=9;
07         Wai ww=new Wai();
08         public void myVoid() //内部类成员变量
09         {
10             System.out.println("内部类中的成员变量值为: "+i);
11             System.out.println("外部类中的成员变量值为: "+ww.i);
12         }
13     }
14 }
15 public class NeiBuLei6
16 {
17     public static void main(String args[])
18     {
19         Wai w=new Wai();           //创建外部类对象
20         Wai.Nei wn2=w.new Nei(); //创建内部类对象
21         wn2.myVoid();           //调用内部类中成员
22     }
23 }
```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 12-5 所示。

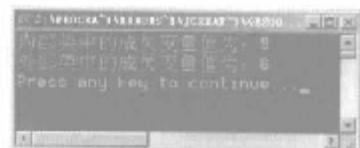


图 12-5 同名称访问

**【代码解析】**在本程序中的第3行定义了一个外部类的成员变量，第6行定义了一个内部类的成员变量，这两个成员变量的名称是相同的。而在内部直接访问时，将访问的是内部类的成员变量。要想访问外部类成员变量，就需要首先创建一个外部类对象，然后使用该对象调用外部类成员变量。这是访问内部类和外部类成员变量名称相同时的一个方法，还有另一种方法。

使用 this 可以指向当前对象的引用，这里使用 this 可以轻松完成在内部类中访问同名成员变量的情况，如下面的示例代码 12-7 所示。



**技巧：**使用 this 可以指向当前对象的引用，使用 this 可以轻松完成在内部类中访问同名成员变量的情况。

**【范例 12-7】**示例代码 12-7 是一个使用 this 访问同名成员变量的程序。

示例代码 12-7

```

01  class Wai
02  {
03      int i=8;           //外部类成员变量
04      class Nei         //创建非静态内部类
05      {
06          int i=9;       //内部类成员变量
07          public void myVoid() //内部类成员变量
08          {
09              System.out.println("内部类中的成员变量值为: "+this.i);
10              System.out.println("外部类中的成员变量值为: "+Wai.this.i);
11          }
12      }
13  }
14  public class NeiBuLei7
15  {
16      public static void main(String args[])
17      {
18          Wai w=new Wai();           //创建外部类对象
19          Wai.Nei wn2=w.new Nei(); //创建内部类对象
20          wn2.myVoid();           //调用内部类中成员
21      }
22  }

```

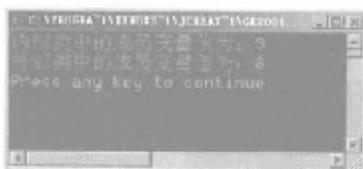


图 12-6 使用 this 访问

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 12-6 所示。

**【代码解析】**在本程序中，使用 this 来访问内部类和外部类具有同名成员变量的情况。在内部类中，直接使用 this 来访问成员变量，则访问的是本类中的成员变量，也就是内部类中的成员变量。如果要访问外部类的成员变量，就需要使用类名加 this 来访问指定的该类中的成员变量。在该程序中使用的类名是外部类的类名，所以访问的是外部类的成员变量。



## 12.2 局部内部类

上一节中介绍了非静态成员内部类，以及如何对非静态成员内部类进行操作。本节中将学习局部内部类的知识，通过非静态成员内部类的学习，学习局部内部类也变得很容易。从名称就可以看出局部内部类是作为一个类的局部变量来定义的。

### 12.2.1 创建局部内部类

局部内部类的作用范围是和局部变量的作用范围相同的，只在局部中起作用，所以对局部内部类进行访问时，只能在该局部内部类的作用范围内。

**【范例 12-8】**示例代码 12-8 是一个创建和访问局部内部类的程序。

示例代码 12-8

```
01 class Wai
02 {
03     public void myVoid()
04     {
05         class Nei           // 定义一个局部内部类
06         {
07             int i=5;        // 局部内部类的成员变量
08         }
09         Nei n=new Nei();
10         System.out.println("局部内部类的成员变量为: "+n.i);
11     }
12 }
13 public class NeiBuLei8
14 {
15     public static void main(String args[])
16     {
17         Wai w=new Wai();    // 创建外部类对象
18         w.myVoid();        // 调用内部类中成员
19     }
20 }
```

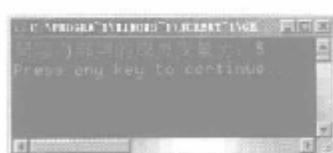


图 12-7 创建局部内部类

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 12-7 所示。

**【代码解析】**在本程序中定义了一个局部内部类，并进行了对该局部内部类的访问。对该内部类进行访问必须在该内部类所在的方法中通过创建内部类对象来进行访问。

这是因为这里的内部类是作为局部成员的形式出现的，只能在它所在的方法中进行调用。

### 12.2.2 在局部内部类中访问外部类成员变量

在局部内部类中访问外部类成员变量很容易实现，并不需要进行过多操作。在局部内部类中可以直接调用外部类的成员变量。

**【范例 12-9】**示例代码 12-9 是一个在局部内部类中访问外部类成员变量的程序。

示例代码 12-9

```

01  class Wai
02  {
03      int i=9;           //定义一个外部类的成员变量
04      public void myVoid()
05      {
06          class Nei           //定义一个局部内部类
07          {
08              public void myNeiVoid()
09              {
10                  System.out.println("外部类的成员变量值为: "+i); //访问外部类的成
11                  //员变量
12              }
13          }
14          Nei n=new Nei();      //创建内部类对象
15          n.myNeiVoid();        //调用内部类中的成员方法
16      }
17  public class NeiBuLei9
18  {
19      public static void main(String args[])
20      {
21          Wai w=new Wai();      //创建外部类对象
22          w.myVoid();           //调用内部类中成员
23      }
24  }

```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 12-8 所示。

【代码解析】在示例代码 12-9 中定义了一个局部内部类, 在该局部内部类中定义了一个方法来访问外部类的成员变量。从运行结果中可以看出在内部类中可以成功访问外部类的成员变量。在该程序中同样需要注意的是, 对内部类进行访问需要和内部类在同一方法中。

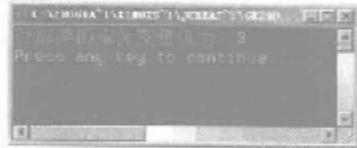


图 12-8 访问外部类成员变量

### 12.2.3 在局部内部类中访问外部类的局部变量

和访问外部类的成员变量不同, 在局部内部类中访问外部类中和局部内部类在同一局部的局部变量是不能够直接访问的。

【范例 12-10】示例代码 12-10 是一个错误的访问外部类局部变量的程序。

示例代码 12-10

```

01  class Wai
02  {
03      public void myVoid()
04      {
05          int i=9;           //定义一个外部类的局部变量
06          class Nei           //定义一个局部内部类
07          {
08              public void myNeiVoid()
09              {
10                  System.out.println("外部类的局部变量值为: "+i); //访问外部类的成
11                  //员变量
12              }
13          }
14      }
15  }

```



```

11         }
12     }
13     Nei n=new Nei();           //创建内部类对象
14     n.myNeiVoid();            //调用内部类中的成员方法
15   }
16 }
17 public class NeiBuLei10
18 {
19     public static void main(String args[])
20     {
21         Wai w=new Wai();        //创建外部类对象
22         w.myVoid();            //调用内部类中成员
23     }
24 }

```

【代码解析】运行该程序是会发生错误的，错误信息为“从内部类中访问局部变量 i，需要被声明为最终类型”。在局部内部类中访问外部类的局部变量是不能够访问普通的局部变量的，必须将该局部变量声明为 final。将该示例修改成示例代码 12-11 的形式就可以正常运行。

**【范例 12-11】**示例代码 12-11 是一个能够正常运行的在内部类中访问外部类成员变量的程序。

示例代码 12-11

```

01 class Wai
02 {
03     public void myVoid()
04     {
05         final int i=9;           //定义一个外部类的局部变量
06         class Nei                //定义一个局部内部类
07         {
08             public void myNeiVoid()
09             {
10                 System.out.println("外部类的局部变量值为: "+i); //访问外部类的成
11                 //员变量
12             }
13             Nei n=new Nei();        //创建内部类对象
14             n.myNeiVoid();          //调用内部类中的成员方法
15         }
16     }
17 public class NeiBuLei11
18 {
19     public static void main(String args[])
20     {
21         Wai w=new Wai();        //创建外部类对象
22         w.myVoid();              //调用内部类中成员
23     }
24 }

```

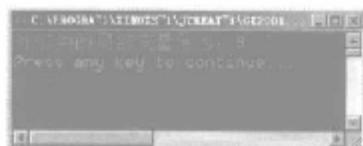


图 12-9 正确访问局部变量

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 12-9 所示。

【代码解析】从运行结果中可以看出，从局部内部类中能够正常地访问外部类的局部变量，这是由于局部变

量被声明为 final 类型。普通的局部变量在所在的方法中执行结束后将消失，所以在内部类访问该普通的局部变量时将不能够正常访问。但是将局部变量用 final 修饰，这样该局部变量就不会在方法结束时消失，从而能够正常地访问。

#### 12.2.4 静态方法中的局部内部类

局部内部类定义在非静态方法和静态方法中是不同的，前面的两节中都是将局部内部类定义在非静态方法中，本节将学习静态方法中定义局部内部类的情况。在静态方法中定义的局部内部类要想访问外部类中的成员，该程序必须是静态成员。静态成员和非静态成员之间的访问是不变的。



**注意：**在静态方法中定义的局部内部类要想访问外部类中的成员，该程序必须是静态成员。静态成员和非静态成员之间的访问是不变的。

**【范例 12-12】**示例代码 12-12 是一个错误的访问成员的程序。

示例代码 12-12

```

01  class Wai
02  {
03      int i=3;
04      public static void myVoid()
05      {
06          class Nei           //定义一个局部内部类
07          {
08              public void myNeiVoid()
09              {
10                  System.out.println("外部类的局部变量值为：" + i); //访问外部类的成
11                  //员变量
12              }
13              Nei n=new Nei();           //创建内部类对象
14              n.myNeiVoid();           //调用内部类中的成员方法
15          }
16      }
17  public class NeiBuLei12
18  {
19      public static void main(String args[])
20      {
21          Wai w=new Wai();           //创建外部类对象
22          w.myVoid();               //调用内部类中成员
23      }
24  }

```

**【代码解析】**运行该程序是会发生错误的，错误信息为“无法从静态上下文中引用非静态变量 i”。该程序主要错误原因是第三行定义的外部类变量是一个非静态成员变量。而本程序中定义的局部变量是定义在静态的方法中，所以是不能够正常访问的。如果想正常访问，就需要将程序修改成示例代码 12-13 的形式。

**【范例 12-13】**示例代码 12-13 是一个正确进行访问的程序。

示例代码 12-13

```

01  class Wai
02  {

```



```
03     static int i=3;
04     public static void myVoid()
05     {
06         class Nei           //定义一个局部内部类
07         {
08             public void myNeiVoid()
09             {
10                 System.out.println("外部类的局部变量值为: "+i); //访问外部类的成
11                 //员变量
12             }
13             Nei n=new Nei();           //创建内部类对象
14             n.myNeiVoid();           //调用内部类中的成员方法
15         }
16     }
17     public class NeiBuLei13
18     {
19         public static void main(String args[])
20         {
21             Wai w=new Wai();           //创建外部类对象
22             w.myVoid();               //调用内部类中成员
23         }
24     }
}
```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序,然后使用 Java 运行编译产生的 class 程序,运行结果如图 12-10 所示。

【代码解析】该程序是能够正常运行的,从运行结果中也可以看到这一点。示例代码 12-13 只是在示例代码 12-12 的基础上做了很小的改动,将第 3 行外部类成员变量改为静态的成员变量,从而使在静态方法中的局部内部类能够访问该静态变量。

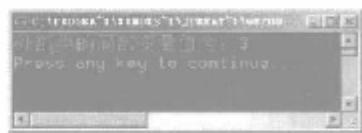


图 12-10 正确访问外部类

### 12.3 静态内部类

在第 12.1 节中已经讲解了非静态内部类,本节将讲解什么是静态内部类。静态内部类就是在外部类中扮演一个静态成员的角色。在本节中就来学习如何创建静态内部类和关于静态内部类访问的问题。

#### 12.3.1 创建静态内部类

创建静态内部类的形式和创建非静态内部类的形式很相似的,只是需要将该内部类使用 static 修饰成静态的形式。使用 static 修饰类,这在正常类中是不可能的。定义静态内部类的语法如下所示。

```
class Wai
{
    static class Nei
    {
        //内部类成员
    }
    //外部类成员
}
```

【范例 12-14】示例代码 12-14 是一个创建静态内部类的程序。

示例代码 12-14

```

01  public class NeiBuLei14
02  {
03      static class Nei          //创建静态内部类
04      {
05          int i=1;            //内部类成员
06      }
07      int ii=2;              //外部类成员
08      public static void main(String args[])
09      {
10          System.out.println("创建一个静态内部类");
11      }
12  }

```

【运行结果】使用 `javac` 编译程序将产生一个和该程序对应的 `class` 程序，然后使用 Java 运行编译产生的 `class` 程序，运行结果如图 12-11 所示。

【代码解析】在示例代码 12-14 中，从第 3 行到第 6 行定义了一个静态内部类。第 5 行是定义的一个静态内部类成员变量，而第 7 行是一个外部类的成员变量。通过学习非静态内部类，静态内部类是很容易学习的。

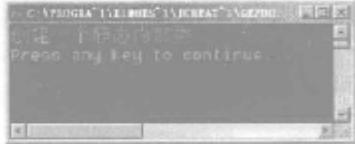


图 12-11 创建静态内部类

### 12.3.2 在外部类中访问静态内部类

在外部类中访问静态内部类和在外部类中访问非静态内部类一样的，只需要从成员间访问的角度就可以考虑到这一点。

【范例 12-15】示例代码 12-15 是一个在外部类中访问静态内部类的程序。

示例代码 12-15

```

01  class Wai
02  {
03      static class Nei          //创建静态内部类
04      {
05          int i=5;            //内部类成员
06      }
07      public void myVoid()    //外部类成员
08      {
09          Nei n=new Nei();   //创建一个内部类对象
10          int ii=n.i;        //访问内部类成员
11          System.out.println("静态内部类的变量值为: "+ii);
12      }
13  }
14  public class NeiBuLei15
15  {
16      public static void main(String args[])
17      {
18          Wai w=new Wai();
19          w.myVoid();
20      }
21  }

```

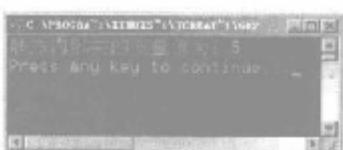


图 12-12 在外部类中访问静态内部类

通过上一节的学习，读者知道在外部类中访问静态内部类和访问非静态内部类是相同的，但是在外部类中访问静态内部类和非静态内部类就不再相同。因为静态内部类是外部类的静态成员，静态成员是不需要外部类对象而存在的，所以在外部类外，对静态内部类进行访问时是不需要创建外部类对象的。

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 12-12 所示。

【代码解析】将示例代码 12-15 和示例代码 12-2 相对比，会发现在外部类中不管是访问非静态内部类还是静态内部类都是一样的。这里只需要遵守成员间访问的规则就可以。在外部类中访问静态内部类时同样需要，首先创建内部类对象，然后使用内部类对象调用内部类成员。

### 12.3.3 在外部类外访问静态内部类

通过上一节的学习，读者知道在外部类中访问静态内部类和访问非静态内部类是相同的，但是在外部类中访问静态内部类和非静态内部类就不再相同。因为静态内部类是外部类的静态成员，静态成员是不需要外部类对象而存在的，所以在外部类外，对静态内部类进行访问时是不需要创建外部类对象的。



**注意：**因为静态内部类是外部类的静态成员，静态成员是不需要外部类对象而存在的，所以在外部类外，对静态内部类进行访问时是不需要创建外部类对象的。

在外部类外访问静态内部类的基本语法如下所示。

**【范例 12-16】**示例代码 12-16 是一个在外部类外访问静态内部类的程序。

示例代码 12-16

```

01  class Wai
02  {
03      static class Nei           // 创建静态内部类
04      {
05          int i=13;           // 内部类成员
06      }
07  }
08  public class NeiBuLei16
09  {
10     public static void main(String args[])
11     {
12         Wai.Nei wn=new Wai.Nei(); // 创建内部类对象
13         System.out.println("内部类中的变量 i 的值为: "+wn.i);
14     }
15 }
```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 12-13 所示。

【代码解析】在示例代码 12-16 中的第 12 行创建了一个内部类对象，可以看出访问静态内部类是不需要创建外部类的，可以直接创建内部类对象。需要注意的是，创建内部类对象也是需要指明该内部类是哪一个外部类的内部类。

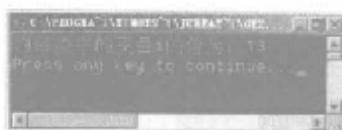


图 12-13 在外部类外访问静态内部类

## 12.4 匿名内部类

在所有的内部类中最难的就应该是匿名内部类。匿名内部类从名字上看就知道是没有类名的类。本节将介绍如何创建匿名内部类和如何进行关于匿名内部类的访问问题。

### 12.4.1 创建匿名内部类

在创建匿名内部类中将使用到继承父类或实现接口的知识，匿名内部类是没有名字的，所以在创建匿名内部类时同时创建匿名内部类的对象。创建匿名内部类的语法格式如下所示。

```
new NeiFather()
{
    //匿名内部类
}
```

在创建匿名内部类的语法中，NeiFather 是匿名内部类继承的父类的类名，使用 new 同时创建了匿名内部类的对象。在匿名内部类中可以重写父类中的方法，也可以定义自己的方法。



**注意：**由于匿名内部类是没有类名的，所以匿名内部类中自定义的方法只能在匿名内部类中使用。

**【范例 12-17】**示例代码 12-17 是一个创建匿名内部类的程序。

示例代码 12-17

```
01 //创建匿名内部类将继承的父类
02 class NeiFather
03 {
04     //父类中的方法
05     public void myVoid()
06     {
07         System.out.println("这是内部类父类的方法");
08     }
09 }
10 public class NeiBuLei17
11 {
12     public static void main(String args[])
13     {
14         //创建匿名内部类
15         NeiFather nf=new NeiFather()
16         {
17             //重写父类中的方法
18             public void myVoid()
19             {
20                 System.out.println("这是匿名内部类的方法");
21             }
22         };
23         nf.myVoid(); //调用匿名内部类中的方法
24     }
25 }
```

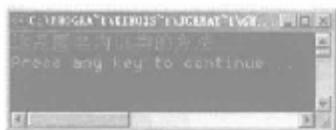


图 12-14 创建匿名内部类

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 12-14 所示。

【代码解析】在创建匿名内部类时需要注意的是，在创建匿名内部类时必须同时创建匿名内部类对象，否则以后将不能创建匿名内部类对象。在对匿名内部类进行访问时，是使用引用通过多态来访问的，这也是由于匿名没有类名不能创建自身类型引用造成的。

**提示：**在创建匿名内部类时必须同时创建匿名内部类对象，否则以后将不能创建匿名内部类对象。

读者学习创建匿名内部类后可能会有疑问，这里是创建的一个类吗？这是很容易解释的。编译该程序，将产生一个 NeiBuLei17\$1.class 文件，该文件就是匿名内部类编译后产生的文件。NeiBuLei17 是外部类的类名，但是匿名内部类是没有类名的，则将使用数字来表示是第几个匿名内部类。因为该程序中只有一个匿名内部类，所以就出现“NeiBuLei17\$1.class”名称。从这里也可以看出，在程序中的确创建了一个类，只是该类没有名称。

创建匿名内部类还可以通过实现接口来完成的，学习完前面的知识，是很容易理解这一点的。只需要将父类的名称改为接口的名称就可以完成。

**【范例 12-18】**示例代码 12-18 是一个通过实现接口创建匿名内部类的程序。

示例代码 12-18

```

01 //创建匿名内部类将实现的接口
02 interface NeiWeiKou
03 {
04     //接口中的方法
05     public void myVoid();
06 }
07 public class NeiBuLei18
08 {
09     public static void main(String args[])
10     {
11         //创建匿名内部类
12         NeiWeiKou njk=new NeiWeiKou()
13         {
14             //实现接口中的方法
15             public void myVoid()
16             {
17                 System.out.println("这是匿名内部类的方法");
18             }
19         };
20         njk.myVoid(); //调用匿名内部类中的方法
21     }
22 }
```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 12-15 所示。

【代码解析】从示例代码 12-18 中可以看出，也可以通

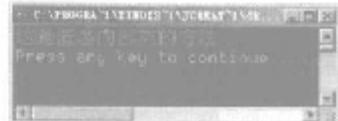


图 12-15 创建匿名内部类

通过实现接口来创建匿名内部类。只需要在创建匿名内部类时创建接口的引用和使用该引用调用匿名内部类中的方法。当然接口中是不能够像父类一样创建具体的方法的，接口中只能是抽象的方法。

#### 12.4.2 匿名内部类的初始化

匿名内部类是没有名称的，所以匿名内部类也是不可能具有构造器的，这就出现一个问题。有时在匿名内部类中也是要定义成员变量的，但是该成员变量应该放在什么位置呢。这里的解决方法就是创建一个非静态语句块，将所有的初始化的成员变量都放在该非静态语句块中。这样在匿名内部类中的方法中就可以来调用这些成员变量。

**【范例 12-19】**示例代码 12-19 是一个处理匿名内部类初始化问题的程序。

示例代码 12-19

```

01 //创建匿名内部类将继承的父类
02 class NeiFather
03 {
04     int i;
05     //父类中的方法
06     public void myVoid()
07     {
08         System.out.println("这是内部类父类的方法");
09     }
10 }
11 public class NeiBuLei19
12 {
13     public static void main(String args[])
14     {
15         //创建匿名内部类
16         NeiFather nf=new NeiFather()
17         {
18             {
19                 i=9;
20             }
21             //重写父类中的方法
22             public void myVoid()
23             {
24                 System.out.println("匿名内部类中成员变量的值为: "+i);
25             }
26         };
27         nf.myVoid(); //调用匿名内部类中的方法
28     }
29 }
```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 12-16 所示。

**【代码解析】**在该程序中的匿名内部类继承了 NeiFather 类，实现了 NeiFather 类的 myVoid 方法，同时使用非静态语句块为 myVoid 类中的 i 变量进行初始化。进行初始化后就可以在重写的 myVoid 方法中来调用该变量。

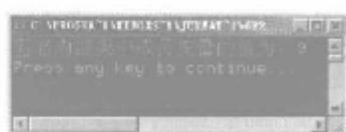


图 12-16 初始化



## 12.5 综合练习

在下面的程序中运行哪一条语句可以正常运行。

```
01  public class LianXii
02  {
03      static int h=1;
04      private int i=2;
05      public void myVoid()
06      {
07          final int j=3;
08          int k=4;
09          class Nei
10          {
11              public void myNeiVoid()
12              {
13                  //System.out.println(h);
14                  //System.out.println(i);
15                  //System.out.println(j);
16                  //System.out.println(k);
17              }
18          }
19          Nei n=new Nei();
20          n.myNeiVoid();
21      }
22      public static void main(String args[])
23      {
24          LianXii lx=new LianXii();
25          lx.myVoid();
26      }
27  }
```

**【提示】**在该程序的第 13 行到第 16 行注释了 4 条访问程序中变量的语句，释放这些语句，可以发现访问 h、i 和 j 都是能够正常访问的，但是访问变量 k 就会发生编译异常。这是因为在局部内部类中，只能访问方法中的最终变量。



## 12.6 小结

本章讲解了 Java 中的内部类，内部类包括非静态内部类、局部内部类、静态内部类和匿名内部类，并分别对这些内部类进行了讲解。在讲解每一种内部类时，首先讲解如何创建该内部类，并且讲解了如何对内部类进行访问。如果读者想了解更多的关于本章的内容，可以参考电子工业出版社出版的《深入浅出 JDK 6.0》一书进行学习。



## 12.7 习题

### 一、填空题

1. 内部类可以分为\_\_\_\_\_，\_\_\_\_\_，\_\_\_\_\_，\_\_\_\_\_。
2. 静态内部类是外部类的\_\_\_\_\_，静态成员是不需要外部类对象而存在的，所以在外部类外，对静态内部类进行访问时是不需要创建外部类对象的。
3. 非静态内部类和\_\_\_\_\_没有区别，同样可以在非静态内部类前面加可以修饰成员的修饰符。

4. 在静态方法中定义的局部内部类要想访问外部类中的成员，该程序必须是\_\_\_\_\_。静态成员和非静态成员之间的访问是不变的。

## 二、选择题

1. 选择访问下面程序中 B 类的正确方法（ ）。

```
01  public class A
02  {
03      public static class B
04      {
05          public static void myVoid()
06          {
07          }
08      }
09  }
```

- A. A.B a=new A.B();                    B. A.B a=new B();  
 C. B a=new B();                        D. A a=new A.B();

2. 选择下面程序的运行结果（ ）。

```
01  class A
02  {
03      A()
04      {
05          System.out.println("A");
06      }
07      class B
08      {
09          B()
10          {
11              System.out.println("B");
12          }
13          public void myVoid1()
14          {
15              System.out.println("C");
16          }
17      }
18      public static void main(String args[])
19      {
20          A a=new A();
21          a.myVoid2();
22      }
23      public void myVoid2()
24      {
25          B b=new B();
26          b.myVoid1();
27      }
28  }
```

- A. ABC                    B. AB                    C. AC                    D. B

## 三、简答题

1. 简述非静态内部类和静态内部类的异同。  
 2. 简述局部内部类和成员内部类的异同。

## 四、编程题

1. 编写一个匿名内部类，以及如何访问匿名内部类。  
 2. 编写一个同时具有非静态内部类和局部内部类的程序。

# 第 13 章 多 线 程

多线程是 Java 中的并发机制，表示能够在同一时间内同时执行多个操作。在日常生活中，边上网边听歌就是一个多线程。随着 CPU 进入双核，甚至多核时代，多线程的优势越来越明显。Java 本身就是一门支持多线程的语言，在 Java 中使用多线程是很方便的，同样也是很高效的。通过本章的学习，读者应该能够实现如下几个目标。

- 了解什么是多线程。
- 熟练掌握如何定义和使用多线程。
- 了解多线程的生命周期。
- 掌握多线程的调用的几个情况。
- 了解多线程的同步问题。



## 13.1 多线程简介

多线程就好像日常生活中同时做几件事一样，例如早上起床，要烧水洗脸，在烧水时就可以刷牙，还可以边刷牙边看早间新闻，这样就同时做着烧水、刷牙、看电视三件事。多线程也是一样的，在同一时刻有可能在执行多个线程，这样能够更好地提高办事效率。

实际开发中也是在很多地方使用多线程的，例如在很多网站中，当用户注册后，系统一方面会通知用户已经注册成功，一方面向用户在注册时填写的 E-mail 中发送邮件。这里就需要使用多线程，如果使用的是单线程，系统就会向用户注册的 E-mail 中发送邮件后才显示用户注册成功，由于发送邮件可能需要很长的时间，从而影响整个注册进度。

在前面的学习中，虽然没有使用多线程，但是同样使用到了线程的知识。在每一个程序中的 main 方法就是一个线程，它一般被称为主线程。在主线程中可以启动多个子线程来执行。



## 13.2 定义线程和创建线程对象

上一节讲解了什么是多线程，本节将讲解怎样来定义线程和如何创建线程对象。定义线程有两种方法，一种是继承 Thread 类，一种是实现 Runnable 接口，这两种方法存在各自优缺点。和定义线程对应的就是创建线程对象，也有两种方法。在本节中就来学习使用这两种方法来定义线程，以及相对应的创建线程对象的方法。

### 13.2.1 继承 Thread 类定义线程

定义一个线程可以通过继承 Thread 类来实现，这是一种相对简单的定义线程的方法。在 Thread 类中有一个 run 方法，在定义的线程中需要重写这个方法。在重写的 run 方法中，可以定义该线程所要执行的语句。当线程启动时，run 方法中的程序就成为一条独立的执行线程。



**【范例 13-1】**示例代码 13-1 是一个通过继承 Thread 类定义线程的程序。

示例代码 13-1

```

01 public class XianCheng1 extends Thread
02 {
03     public void run()
04     {
05         System.out.println("通过继承 Thread 定义线程");
06     }
07 }

```

**【代码解析】**该程序是无法运行的，因为没有 main 方法，也就是没有启动线程的方法。在该程序中创建了一个线程类继承于 Thread 类，并且在该类中重写了 run 方法，在其中定义了该线程的功能是显示一条语句。



**注意：**重写的 run 方法也是可以作为一般的方法来调用的，但是这种调用并不是作为一个线程出现的，它只是主线程中的一部分。同样，run 方法也是可以被重载的，但是重载后的 run 方法不作为一个线程，也是主线程的一部分。

讲解完定义线程后，就可以来学习如何创建线程对象。通过继承 Thread 类创建线程，是很容易创建线程对象的。在这种定义线程的方法中，创建线程对象和创建普通对象是一样的。下面是创建示例代码 13-1 中线程对象的代码。

```
XianCheng1 xc=new XianCheng1();
```

从创建线程对象的程序可以看出，创建线程对象的方法和创建普通对象的方法是一样的。但是这只是对于使用继承 Thread 类创建线程的方法来说的。

### 13.2.2 实现 Runnable 接口定义线程

定义线程除了通过继承 Thread 类来实现，还可以通过实现 Runnable 接口来实现。在 Runnable 接口中具有一个抽象的 run 方法，在实现 Runnable 接口时，需要实现该 run 方法。该 run 方法就会作为一个执行线程的方法。

**【范例 13-2】**示例代码 13-2 是一个通过实现 Runnable 接口定义线程的程序。

示例代码 13-2

```

01 public class XianCheng2 implements Runnable
02 {
03     public void run()
04     {
05         System.out.println("通过实现 Runnable 接口定义线程");
06     }
07 }

```

**【代码解析】**示例代码 13-2 和示例代码 13-1 唯一不同的就是，示例代码 13-1 是通过继承 Thread 类定义线程，示例代码 13-2 是通过实现 Runnable 接口来定义线程。这两种方法中都需要定义一个 run 方法，不管该方法是通过重写父类方法，还是实现接口方法。run 方法是一个线程的入口，是线程必须具有的。

在使用通过实现 Runnable 接口定义的线程中，要想创建线程对象就不是很容易做到的。因



为直接创建类对象，创建的并不是一个线程对象。要想创建线程对象，必须要借助 Thread 类。



**注意：**因为直接创建类对象，创建的并不是一个线程对象。要想创建线程对象，必须要借助 Thread 类。

Thread 类具有 4 个构造器，最常用的就是具有一个参数，该参数是实现 Runnable 接口类对象的构造器。创建线程对象的程序如下所示。

```
XianCheng2 xc=new XianCheng2();
Thread t=new Thread(xc);
```

在该程序中，首先创建了一个实现 Runnable 接口的类对象，然后将该对象作为 Thread 类的参数，从而创建了一个线程对象。创建的类对象是可以作为多个 Thread 类构造器参数的，这样就创建了多个线程。这一点将在以后的学习中多次使用。



## 13.3 运行线程

上一节学习了如何定义线程，并且知道了如何创建线程对象。读者对这些都了解后，就需要来学习如何运行线程。本节将分两个部分来讲解，先讲解如何启动线程，然后讲解如何运行多个线程。

### 13.3.1 启动线程

有些读者会认为启动线程就是调用线程类中的 run 方法，但这个观点是错误的，例如示例代码 13-3 中所演示的。

**【范例 13-3】**示例代码 13-3 是一个错误的启动线程的程序。

示例代码 13-3

```
01 class MyRunnable implements Runnable
02 {
03     //定义一个 run 线程方法
04     public void run()
05     {
06         System.out.println("这是一个错误的启动线程的程序");
07     }
08 }
09 public class XianCheng3
10 {
11     public static void main(String args[])
12     {
13         MyRunnable mr=new MyRunnable();
14         mr.run(); //调用 run 方法
15     }
16 }
```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 13-1 所示。

**【代码解析】**从该程序可以看出，run 方法是可以通过方法调用执行的，但是这并不代表创建了一个新线程。

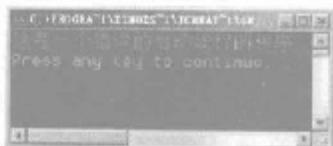


图 13-1 错误启动线程



这是一个错误的启动线程的方法。

如果想正确地启动一个线程，需要调用线程对象的 start 方法，下面通过程序来演示如何正确的启动一个线程。

**【范例 13-4】**示例代码 13-4 是一个正确的启动线程的程序。

示例代码 13-4

```

01  class MyRunnable implements Runnable
02  {
03      //定义一个 run 线程方法
04      public void run()
05      {
06          System.out.println("这是一个正确的启动线程的程序");
07      }
08  }
09  public class XianCheng4
10  {
11      public static void main(String args[])
12      {
13          MyRunnable mr=new MyRunnable();
14          Thread t=new Thread(mr);
15          t.start(); //启动线程
16      }
17  }

```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 13-2 所示。

**【代码解析】**第一次看到该程序时，读者可能会感到有些奇怪，为什么调用的是 start 方法，而执行的是 run 方法，这就是 Java 对多线程的设计。在调用 start 方法后，就启动了线程，该线程是和 main 方法并列执行的线程。这样该程序就变为一个多线程程序。



**注意：**线程只能被启动一次，也就是只能调用一次 start 方法。当多次启动线程，也就是多次调用 start 方法时，就会发生异常。

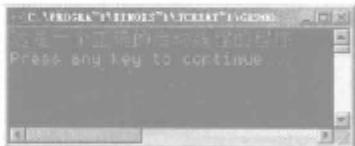


图 13-2 正确启动线程

**【范例 13-5】**示例代码 13-5 是一个多次启动线程的程序。

示例代码 13-5

```

01  class MyRunnable implements Runnable
02  {
03      //定义一个 run 线程方法
04      public void run()
05      {
06          System.out.println("这是一个正确的启动线程的程序");
07      }
08  }
09  public class XianCheng5
10  {
11      public static void main(String args[])
12      {
13          MyRunnable mr=new MyRunnable();
14          Thread t=new Thread(mr);

```



```
15         t.start(); //启动线程
16         t.start(); //再次启动线程
17     }
18 }
```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 13-3 所示。

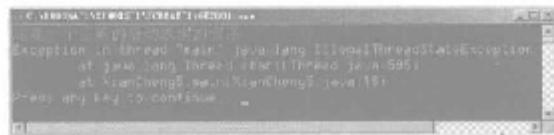


图 13-3 多次启动线程

**【代码解析】**编译示例代码 13-5 程序是能够正常编译的, 但是运行时, 就会发生运行异常。从运行结果中可以看出, 当第一次启动线程时, 是能够正常运行 run 方法的。但是第二次启动线程就会发生 IllegalThreadStateException 异常。多次启动线程的问题在后面的学习中还会涉及, 将对其做进一步的讲解。

### 13.3.2 同时运行多个线程

学习了如何启动线程, 接下来就来学习如何同时运行多个线程。首先通过示例代码 13-6 来看一下如何同时运行多个线程。

**【范例 13-6】**示例代码 13-6 是一个同时运行多个线程的程序。

示例代码 13-6

```
01 //定义一个实现 Runnable 接口的类
02 class MyRunnable1 implements Runnable
03 {
04     //定义一个 run 线程方法
05     public void run()
06     {
07         for(int i=0;i<100;i++)
08         {
09             System.out.print("#");
10         }
11     }
12 }
13 //再定义一个实现 Runnable 接口的类
14 class MyRunnable2 implements Runnable
15 {
16     //定义一个 run 线程方法
17     public void run()
18     {
19         for(int i=0;i<100;i++)
20         {
21             System.out.print("$");
22         }
23     }
24 }
25 public class XianCheng6
26 {
27     public static void main(String args[])
28     {
```

```

29         MyRunnable1 mr1=new MyRunnable1();
30         MyRunnable2 mr2=new MyRunnable2();
31         Thread t1=new Thread(mr1);
32         Thread t2=new Thread(mr2);
33         t1.start();           //启动第一个线程
34         t2.start();           //启动第二个线程
35     }
36 }

```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 13-4 所示。

【代码解析】在示例代码 13-6 中首先定义了两个实现 Runnable 接口的类，在两个类中都定义了 run 方法，显示多个不同的符号。从运行结果中可以看出，不同的符号是交替显示的。

在同时运行多个线程时，运行结果不是唯一的，因为有很多不确定的因素。首先先执行哪一个线程就是不确定的，线程间交替也是不确定的。但是确定的是每一个线程都将启动，每一个线程都执行结束。



图 13-4 同时运行多个线程

## 13.4 线程生命周期

线程是存在生命周期的。线程的生命周期分为 5 种不同的状态，分别是新建状态、准备状态、运行状态、等待/阻塞状态和死亡状态。本节将对每一个状态进行讲解。

### 13.4.1 新建状态

当一个线程对象被创建后，线程就处于新建状态。在新建状态中的线程对象从严格意义上还只是一个普通的对象，它还不是一个独立的线程。处于新建状态中的线程被调用 start 方法后就会进入准备状态。从新建状态中只能进入准备状态，并且不能从其他状态进行新建状态。新建状态是线程生命周期的第一个状态。

### 13.4.2 准备状态

处于新建状态中的线程被调用 start 方法就会进入准备状态。处于准备状态下的线程随时都可能被系统选择进入运行状态，从而执行线程。可能同时有多个线程处于准备状态，对于哪一个线程将进入运行状态是不确定的。线程从新建状态进入到准备状态后是不可能再进入新建状态的。在等待/阻塞状态中的线程被解除等待和阻塞后将不直接进入运行状态，而是首先进入准备状态，让系统来选择哪一个线程进入运行状态。

### 13.4.3 运行状态

处于准备状态中的线程一旦被系统选中，使线程获取了 CPU 时间，就会进入运行状态。在运行状态中将执行线程类 run 方法中的程序语句。线程进入运行状态后也不是一下执行结束的，线程在运行状态下随时都可能被调度程序调度回准备状态。在运行状态下还可以让线程进入到等待/阻塞状态。

在通常的单核 CPU 中，在同一时刻只有一个线程处于运行状态的。在多核的 CPU 中，



就可能两个线程或更多的线程同时处于运行状态，这也是多核 CPU 运行速度快的原因。

#### 13.4.4 等待/阻塞状态

Java 中定义了许多线程调度的方法，包括睡眠、阻塞、挂起和等待，这些方法将在后面的调度章节中讲解。使用这些方法都会将处于运行状态的线程调度到等待/阻塞状态。处于等待/阻塞状态的线程被解除后，不会立即回到运行状态，而是首先进入准备状态，等待系统的调度。

#### 13.4.5 死亡状态

当线程中的 `run` 方法执行结束后，或者程序发生异常终止运行后，线程会进入死亡状态。处于死亡状态的线程不能再使用 `start` 方法启动线程，这在前面的学习中已经学到了这一点。但是这不代表处于死亡状态的线程不能再被使用，它也是可以再被使用的，只是将被作为普通的类来使用。



**注意：**线程生命周期的问题，有些读者会觉得很容易的。线程生命周期的问题在后面的学习中会经常使用到，只有能充分了解线程的生命周期，才能更好地理解后面的内容。



### 13.5 线程的调度

通过系统自动调度，线程的执行顺序是没有保障的。Java 中定义了一些线程调度的方法，使用这些方法在一定程度上对线程进行调度，使用这些方法只是给线程一个调度的建议，具体是否能够成功，也是没有保障的。线程调度的方法有几个，包括睡眠方法、设置优先级、让步方法等，在本节中就来学习这些方法的使用。

#### 13.5.1 睡眠方法

当线程处于运行状态时，调用 `sleep` 睡眠方法将使线程从运行状态进入等待/阻塞状态，从而使程序停止运行。`sleep` 睡眠方法是具有一个时间参数的，当经过这么长时间后，线程将进入准备状态，等待系统的调度。从而可以看出，当线程调用睡眠方法后，要想回到运行状态，需要的时间要比指定的睡眠时间长。

`sleep` 方法被重载，存在两种形式，`sleep` 方法的基本语法格式如下所示。

```
public static void sleep(long millis) throws InterruptedException;  
public static void sleep(long millis, int nanos) throws InterruptedException;
```

使用这两个 `sleep` 方法都能使线程进入睡眠状态，`millis` 参数表示线程睡眠的毫秒数，`nanos` 参数表示线程睡眠的纳秒数。`sleep` 方法是一个静态的方法，所以 `sleep` 方法不是依赖于某一个对象的，它的位置是比较随意的。当在线程中执行 `sleep` 方法，则该线程就进入睡眠状态。要想让某一个线程进入睡眠状态，并不是让该线程调用 `sleep` 方法，而只是让该线程执行 `sleep` 方法。`sleep` 方法是可能发生捕获异常的，所以在使用 `sleep` 方法时必须进行异常处理。

【范例 13-7】示例代码 13-7 是一个使用 sleep 方法调度线程的程序。

示例代码 13-7

```

01 //定义一个实现 Runnable 接口的类
02 class MyRunnable1 implements Runnable
03 {
04     //定义一个 run 线程方法
05     public void run()
06     {
07         for(int i=0;i<100;i++)
08         {
09             System.out.print("@");
10             try
11             {
12                 Thread.sleep(50);      //使用 sleep 方法是线程进入睡眠状态 50 毫秒
13             }
14             catch(InterruptedException e)
15             {
16                 e.printStackTrace();
17             }
18         }
19     }
20 }
21 //再定义一个实现 Runnable 接口的类
22 class MyRunnable2 implements Runnable
23 {
24     //定义一个 run 线程方法
25     public void run()
26     {
27         for(int i=0;i<100;i++)
28         {
29             System.out.print("$");
30             try
31             {
32                 Thread.sleep(50);      //使用 sleep 方法是线程进入睡眠状态 50 毫秒
33             }
34             catch(InterruptedException e)
35             {
36                 e.printStackTrace();
37             }
38         }
39     }
40 }
41 public class XianCheng7
42 {
43     public static void main(String args[])
44     {
45         MyRunnable1 mr1=new MyRunnable1();
46         MyRunnable2 mr2=new MyRunnable2();
47         Thread t1=new Thread(mr1);
48         Thread t2=new Thread(mr2);
49         t1.start();          //启动第一个线程
50         t2.start();          //启动第二个线程
51     }
52 }

```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 13-5 所示。



图 13-5 sleep 方法调度线程

**【代码解析】**从运行结果中可以看出，符号是交替显示的，这主要是使用 `sleep` 方法的作用。在两个线程类中都定义了线程方法来循环显示符号。但是每显示一个符号，线程就会调用 `sleep` 方法，使该线程进入睡眠状态 50 毫秒，从而将 CPU 时间让给另一个线程。这样就出现了交替显示符号的运行结果。



**注意：** `sleep` 方法只是给线程一个调度的建议，是否调度成功是不能确定的。在该程序中只有两个线程，所以运行结果出现交替显示的结果。当程序中存在多个线程时，运行结果就可以发生变化，甚至出现意外的结果。

### 13.5.2 线程优先级

在大部分的系统中，对进程的调度都是采用优先级的方式来进行的。在 Java 中对线程进行调度时，也是可以采用优先级来调度的。不同的线程可以具有不同的优先级，优先级高的线程就会占用更多的 CPU 资源和被执行概率。

Java 中的优先级是采用从 1 到 10 来表示的，数字越大表示优先级越高。如果没有为线程设置优先级，则线程的优先级为 5，这也是线程的默认值。但是对于子线程来说，它的优先级是和其父线程优先级相同的。

当需要对线程的优先级进行设置时，可以通过调用 `setPriority` 方法来设置。`setPriority` 方法的语法格式如下所示。

```
public final void setPriority(int i);
```

其中参数 `i` 表示的就是优先级的等级数，它可以从 1 到 10。除了可以使用数字来表示优先级，Java 还在 `Thread` 类中定义了三个表示优先级的常量。`MAX_PRIORITY` 表示线程的最高优先级，`NORM_PRIORITY` 表示线程的默认优先级，`MIN_PRIORITY` 表示线程的最低优先级。

**【范例 13-8】**示例代码 13-8 是一个通过设置线程优先级来对线程进行调度的程序。

示例代码 13-8

```
01 //定义一个实现 Runnable 接口的类
02 class MyRunnable1 implements Runnable
03 {
04     //定义一个 run 线程方法
05     public void run()
06     {
07         for(int i=0;i<100;i++)
08         {
09             System.out.print("#");
10         }
11     }
12 }
13 //再定义一个实现 Runnable 接口的类
14 class MyRunnable2 implements Runnable
15 {
16     //定义一个 run 线程方法
17     public void run()
```

```

18     {
19         for(int i=0;i<100;i++)
20         {
21             System.out.print("$");
22         }
23     }
24 }
25 public class XianCheng8
26 {
27     public static void main(String args[])
28     {
29         MyRunnable1 mr1=new MyRunnable1();
30         MyRunnable2 mr2=new MyRunnable2();
31         Thread t1=new Thread(mr1);
32         Thread t2=new Thread(mr2);
33         //设置线程一为最高优先级
34         t1.setPriority(Thread.MAX_PRIORITY);
35         //设置线程一为最低优先级
36         t2.setPriority(Thread.MIN_PRIORITY);
37         t1.start();           //启动第一个线程
38         t2.start();           //启动第二个线程
39     }
40 }

```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 13-6 所示。

**【代码解析】**在示例代码 13-8 中，在第 34 行设置线程一的优先级为最高优先级，在第 36 行设置线程二的优先级为最低优先级。因为优先级高的线程将获得更多的 CPU 时间，从而出现本程序中的先运行完线程一，再运行线程二的运行结果。

### 13.5.3 yield 让步方法

在 Java 中具有两种线程让步方法，先来介绍第一种 yield 让步方法。yield 让步方法是让线程让出当前 CPU，而将 CPU 让给哪一个线程是不确定的，由系统来进行选择。使用 yield 让步方法的线程将从运行状态进入到准备状态。



**注意：**yield 让步操作是可能不成功的。因为在线程中使用 yield 方法，使该线程进入准备状态。但是系统是有可能再次选择该线程，使该线程进入运行状态的。

yield 让步方法的基本语法格式如下所示。

```
public static void yield();
```

可以看出 yield 让步方法是一个静态方法，所以该方法也是和对象无关的。当在正在运行的线程中运行该方法时，该线程将回到准备状态。

**【范例 13-9】**示例代码 13-9 是一个使用 yield 让步方法对线程进行调度的程序。

#### 示例代码 13-9

```

01 //定义一个实现 Runnable 接口的类
02 class MyRunnable1 implements Runnable

```



图 13-6 线程优先级



```
03  {
04      //定义一个 run 线程方法
05      public void run()
06      {
07          for(int i=0;i<100;i++)
08          {
09              System.out.print("@");
10              Thread.yield(); //调用 yield 方法让当前线程让步
11          }
12      }
13  }
14 //再定义一个实现 Runnable 接口的类
15 class MyRunnable2 implements Runnable
16 {
17     //定义一个 run 线程方法
18     public void run()
19     {
20         for(int i=0;i<100;i++)
21         {
22             System.out.print("$");
23             Thread.yield(); //调用 yield 方法让当前线程让步
24         }
25     }
26 }
27 public class XianCheng9
28 {
29     public static void main(String args[])
30     {
31         MyRunnable1 mr1=new MyRunnable1();
32         MyRunnable2 mr2=new MyRunnable2();
33         Thread t1=new Thread(mr1);
34         Thread t2=new Thread(mr2);
35         t1.start(); //启动第一个线程
36         t2.start(); //启动第二个线程
37     }
38 }
```



图 13-7 yield 让步方法

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 13-7 所示。

**【代码解析】**在该程序中的第 10 行和第 23 行，让当前线程执行 yield 让步方法，并让当前线程回到准备状态。从运行结果中可以看出，运行一个线程时，先显示一个符号，显示就执行 yield 让步方法，从而使线程交替执行，从而显示的符号交替显示。

#### 13.5.4 join 让步方法

使用 join 让步方法，可以将当前线程的 CPU 资源让步给指定的线程。join 让步方法的语法格式如下所示。

```
public final void join() throws InterruptedException;
public final void join(long millis) throws InterruptedException;
public final void join(long millis, int nanos) throws InterruptedException;
```

join 让步方法是具有三种形式的，没有参数表示指定的线程执行完成后再执行其他线程，参数表示在参数的时间内执行让步给的执行线程。join 让步方法也是可能发生捕获异

常的，所以在使用时要进行异常处理。

**【范例 13-10】**示例代码 13-10 是一个使用 join 让步方法调度线程的程序。

示例代码 13-10

```

01 //定义一个实现 Runnable 接口的类
02 class MyRunnable1 implements Runnable
03 {
04     //定义一个 run 线程方法
05     public void run()
06     {
07         for(int i=0;i<100;i++)
08         {
09             System.out.print("#");
10         }
11     }
12 }
13 public class XianCheng10
14 {
15     public static void main(String args[])
16     {
17         MyRunnable1 mrl=new MyRunnable1();
18         Thread t1=new Thread(mrl);
19         t1.start();           //启动第一个线程
20         for(int i=0;i<100;i++)
21         {
22             if(i==10)
23             {
24                 try
25                 {
26                     t1.join(); //让步给 t1 线程
27                 }
28                 catch(InterruptedException e)
29                 {
30                     e.printStackTrace();
31                 }
32             }
33             System.out.print("#");
34         }
35     }
36 }

```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 13-8 所示。

**【代码解析】**有些读者会认为该程序只有一个线程，这种认为是不对的，因为在该程序中 main 方法也是一个线程，同样也是来显示一些符号。在该程序 main 方法中，判断当 i 等于 10 时，执行 join 方法，让步与 t1 线程。从运行结果中可以看出，先显示 10 个主线程中的符号，然后一直显示 t1 线程中的符号，当 t1 线程执行完成后，再进行执行 main 方法线程。



图 13-8 join 让步方法



## 13.6 综合练习

### 1. 判断下面程序是否能够正常运行。

```
01  public class LianXi1 extends Thread
02  {
03      public void run()
04      {
05          for(int i=0;i<5;i++)
06          {
07              System.out.println(i);
08          }
09      }
10  public static void main(String args[])
11  {
12      LianXi1 lx=new LianXi1();
13      lx.run();
14  }
15 }
```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 13-9 所示。



图 13-9 练习 1

### 2. 判断下面的程序是否能够正常运行。

```
01  public class LianXi2 extends Thread
02  {
03      public void run()
04      {
05          for(int i=0;i<5;i++)
06          {
07              Thread.sleep(100);
08          }
09      }
10  public static void main(String args[])
11  {
12      LianXi1 lx=new LianXi1();
13      lx.run();
14  }
15 }
```

【提示】运行该程序是会发生异常的，异常信息为“未报告的异常 java.lang.Interrupted Exception；必须对其进行捕捉或声明以便抛出”。该程序需要修改成如下形式。

```
01  public class LianXi3 extends Thread
02  {
03      public void run()
04      {
05          for(int i=0;i<5;i++)
06          {
07              try
08              {
09                  Thread.sleep(100);           //需要对 sleep 方法进行异常处理
10              }
11              catch(Exception e)
12              {
13              }
14  }
```

```

14      }
15  }
16  public static void main(String args[])
17  {
18      LianXil lx=new LianXil();
19      lx.run();
20  }
21 }

```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 13-10 所示。

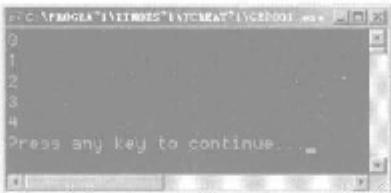


图 13-10 修改后运行结果



## 13.7 小结

本章学习了 Java 中的多线程, 首先对线程进行了简单的介绍, 然后讲解如何定义、创建和运行多线程。接下来还讲解了线程的生命周期和对线程的调度。读者如果想了解更多的关于本章的内容, 可以参考电子工业出版社出版的《深入浅出 JDK 6.0》一书进行学习。



## 13.8 习题

### 一、填空题

1. 定义线程有两种方法, 一种是\_\_\_\_\_，一种是\_\_\_\_\_，这两种方法是存在各自优缺点的。
2. 在 Runnable 接口中具有一个抽象的\_\_\_\_\_方法, 在实现 Runnable 接口时, 需要实现该方法。
3. 线程的生命周期分为 5 种不同的状态, 分别是\_\_\_\_\_状态、\_\_\_\_\_状态、\_\_\_\_\_状态、\_\_\_\_\_状态和\_\_\_\_\_状态。
4. Java 还在 Thread 类中定义了三个表示优先级的常量。\_\_\_\_\_表示线程的最高优先级, \_\_\_\_\_表示线程的默认优先级, \_\_\_\_\_表示线程的最低优先级。
5. 因为直接创建类对象, 创建的并不是一个线程对象。要想创建线程对象, 必须要借助\_\_\_\_\_。
6. 处于新建状态中的线程被调用 start 方法后就会进入\_\_\_\_\_状态。
7. 在 Java 中定义了许多线程调度的方法, 包括\_\_\_\_\_、\_\_\_\_\_、\_\_\_\_\_和\_\_\_\_\_。
8. 当线程处于运行状态时, 调用\_\_\_\_\_将使线程从运行状态进入等待/阻塞状态, 从而使程序停止运行。

## 二、选择题

1. 启动线程的方法是 ( )。  
A. run 方法 B. start 方法  
C. sleep 方法 D. join 方法
  2. 选择下面选项中哪些是线程调度的方法 ( )。  
A. 线程优先级 B. start 方法  
C. sleep 方法 D. join 方法
  3. 选择下面程序的运行结果 ( )。

```
01 class MyThread extends Thread
02 {
03     public static void main(String args[])
04     {
05         MyThread mt=new MyThread();
06         mt.run();
07     }
08     public void run()
09     {
10         System.out.println("A");
11     }
12 }
```



• 选择合适的运行策略 • 29

```
02
03     public static void main(String args[])
04     {
05         MyThread mt=new MyThread();
06         mt.start();
07         mt.start();
08     }
09     public void run()
10     {
11         System.out.println("A");
12     }
13 }
```

- A. 发生编译错误
  - B. 只有一个线程运行
  - C. 运行两个线程
  - D. 没有任何输出

5. 选择下面程序的运行结果 ( )。

```
01 class MyThread extends Thread
02 {
03     public static void main(String args[])
04     {
05         MyThread mt=new MyThread();
06         Thread t=new Thread(mt);
07         t.start();
08     }
09     public void run()
10     {
11         for(int i=0;i<4;i++)
12     }
13 }
```

```
12     {  
13         System.out.println(i);  
14     }  
15 }  
16 }
```

- A. 发生编译错误                            B. 0  
C. 0 1 2 3                                    D. 没有任何输出

### 三、简答题

1. 简述定义线程的两种方法，并说明它们有什么异同。
2. 简述线程的生命周期，并说明状态间的转换。
3. 线程中有哪些调度方法，其中哪些是静态方法。

### 四、编程题

1. 编写一个多线程程序，定义两个不同优先级的线程，通过使用调度方法，使优先级较低的线程获得更多的运行机会。
2. 编写一个使用让步方法进行线程调度的程序。

# 第 14 章 Swing 桌面程序开发

Swing 是一门开发桌面程序的技术。在本章中读者将学到如何开发界面程序，这要比前面学习的程序有意思得多。本章将对 Swing 的知识按从浅到深的顺序依次进行讲解。通过本章的学习，读者应该实现如下几个目标。

- 了解 Swing 开发的基本过程。
- 掌握如何创建窗口、面板、标签和按钮。
- 掌握和熟练使用 Swing 中的事件。



## 14.1 开发第一个 Swing 程序

本节首先使用一个简单的程序，让读者知道什么是 Swing 程序，以及 Swing 程序的功能。该程序中很多知识是以前没有介绍过的，在后面的学习中将详细地分析各个地方。

**【范例 14-1】**示例代码 14-1 是一个简单的 Swing 程序。

示例代码 14-1

```
01 import javax.swing.*; //导入 Swing 包
02 //继承 JFrame 类
03 public class Swing1 extends JFrame
04 {
05     //定义构造器
06     public Swing1()
07     {
08         this.setLayout(null); //设置布局管理器
09         JLabel jl=new JLabel(); //定义一个标签
10         jl.setText("第一个 Swing 程序"); //设置显示的文字
11         jl.setBounds(50,50,400,50); //设置标签的大小和位置
12         this.add(jl); //将标签放到窗口中
13         this.setBounds(300,250,500,200); //设置窗口的大小和位置
14         this.setVisible(true); //设置窗口是可见的
15     }
16     public static void main(String args[])
17     {
18         Swing1 s=new Swing1();
19     }
20 }
```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 14-1 所示。

**【代码解析】**第一次看到该程序可能会觉得很复杂，其实其中都是很基础的内容，在以后的 Swing 程序中也会重复使用。在该程序中，首先要导入 Swing 包，然后继承该包中的 JFrame 类，使用该类才能使运行结果出现界面的形式。在程序中需要定义一个构造器，

在构造器中首先要设置布局管理器，该程序没有使用布局管理器，布局管理器的知识会在后面用一章的内容来进行讲解。然后就是定义了一个用于显示文字的标签。在最后还需要设置窗口的大小和位置，以及可见性。

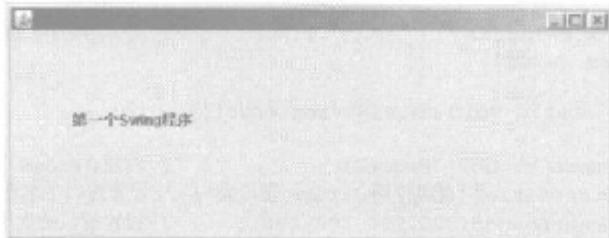


图 14-1 第一个 Swing 程序

从第一个 Swing 程序可以看出，运行结果不再是以前在黑屏中显示信息，而是在界面中显示信息。这里的信息不只包括文字信息，也包括以后将要学到的一些组件信息。



## 14.2 JFrame 窗口类

在 Swing 程序中，窗口是一个容器，在该容器中可以放其他一些组件。学习 JFrame 窗口类是学习其他组件的基础。在 Swing 程序中创建窗口可以使用继承 JFrame 类来完成。

### 14.2.1 JFrame 窗口类简介

在开发的 Swing 程序中，通常是通过继承 JFrame 窗口类来实现窗口的。在该类中具有很多很有用的方法，包括定义窗口标题、标框，以及窗口的大小和位置。在介绍这些方法之前，先来介绍一下 JFrame 窗口类的构造器。JFrame 窗口类具有 4 种构造器。

最常用的 JFrame 窗口类的构造器是无参构造器，使用该构造器将创建一个初始不可见的新窗体。除此之外还有一个 String 类参数的构造器，使用该构造器能够在初始时就创建一个具有标题的新窗体。还有两种需要给出图形配置参数的构造器，这两种构造器在本书中不进行介绍。

创建新窗口后，就可以通过 JFrame 窗口类的方法来设置新窗口。首先使用无参构造器创建的是一个不可见的新窗体，所以要使用方法来将窗体设置为可见的形式。在 JFrame 窗口类中定义了一个 setVisible 方法来设置窗口的可见性，该方法具有一个布尔型参数，true 表示可见，false 表示不可见。将初始状态下的窗口设置为不可见是有原因的，因为有很多对窗口的操作需要在窗口不可见的状态下执行，从而 setVisible 方法通常在程序的最后执行。

在 JFrame 窗口类中有个 setTitle 方法，该方法需要一个字符型参数。使用 setTitle 方法可以设置窗口的名称；还有一个 setBounds 方法，该方法具有 4 个参数，前两个参数分别表示窗口位置的横坐标和纵坐标，后两个参数分别表示窗口大小的宽度和高度。JFrame 窗口类最重要的方法就是 add 方法，使用该方法可以将组件添加到窗口中。这些都是比较常用的 JFrame 窗口类的方法。

### 14.2.2 创建简单窗体

通过前面对 JFrame 窗口类的学习，可以来创建一个简单的窗体。创建窗体有两种方



法，先来介绍第一种方法，可以直接使用 `JFrame` 窗口类的构造器来创建一个简单的窗体。

**【范例 14-2】**示例代码 14-2 是一个直接使用 `JFrame` 窗口类创建简单窗体的程序。

#### 示例代码 14-2

```
01 import javax.swing.*; //导入 Swing 包
02 public class Swing2
03 {
04     public static void main(String args[])
05     {
06         JFrame jf=new JFrame(); //创建 JFrame 类构造器
07         jf.setTitle("直接使用 JFrame 窗口类"); //设置窗口的名称
08         jf.setBounds(300,250,300,200); //设置窗口的大小和位置
09         jf.setVisible(true); //设置窗口可见性
10     }
11 }
```

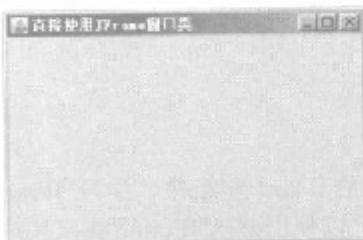


图 14-2 直接使用 `JFrame` 窗口类创建

**【运行结果】**使用 `javac` 编译程序将产生一个和该程序对应的 `class` 程序，然后使用 `Java` 运行编译产生的 `class` 程序，运行结果如图 14-2 所示。

**【代码解析】**在本程序中直接使用 `JFrame` 窗口类来创建一个窗体。首先创建一个 `JFrame` 类对象，然后调用 `JFrame` 类中的方法。在本程序中使用 `setTitle` 方法来设置窗口的名称，使用 `setBounds` 方法来设置窗口的大小和位置，使用 `setVisible` 方法来设置窗口的可见性，这种方法只是创建一个简单窗口时需要的，如果创建一个复杂的窗口，使用这种方法就会使程序变的非常复杂难理解。

创建窗体还有另外一种方法，那就是通过继承 `JFrame` 类来创建窗体，这也是实际开发中使用的。使用这种方法的好处是能够更好的使程序具有条理性。

**【范例 14-3】**示例代码 14-3 是一个通过继承 `JFrame` 类创建窗体的程序。

#### 示例代码 14-3

```
01 import javax.swing.*; //导入 Swing 包
02 //继承 JFrame 类
03 public class Swing3 extends JFrame
04 {
05     //定义构造器
06     public Swing3()
07     {
08         this.setTitle("通过继承创建窗口"); //设置窗口标题
09         this.setBounds(300,250,300,200); //设置窗口的大小和位置
10         this.setVisible(true); //设置窗口是可见的
11     }
12     public static void main(String args[])
13     {
14         Swing3 s=new Swing3();
15     }
16 }
```

**【运行结果】**使用 `javac` 编译程序将产生一个和该程序对应的 `class` 程序，然后使用 `Java` 运行编译产生的 `class` 程序，运行结果如图 14-3 所示。

**【代码解析】**该程序是通过继承 JFrame 类来创建窗体的。在定义类时继承 JFrame 类，在定义本类的构造器中对窗口进行设置。在其中同样设置了窗口的标题、大小、位置和可见性。该程序的运行结果和上一个程序的运行结果相比只是窗体的标题不同。

### 14.2.3 设置窗体

除了上一节中学习的在创建窗体时必要的设置窗体的方法外，还有一些设置窗体的方法。例如 setResizable 方法，使用该方法可以设置创建的窗口是否可以调整大小。



**注意：**默认状态下窗体是能够调整大小的，也就是 setResizable 方法的默认值为 true。

**【范例 14-4】**示例代码 14-4 是一个设置窗体不能被调整大小的程序。

示例代码 14-4

```

01 import javax.swing.*;                                //导入 Swing 包
02 //继承 JFrame 类
03 public class Swing4 extends JFrame
04 {
05     //定义构造器
06     public Swing4()
07     {
08         this.setTitle("通过继承创建窗口");      //设置窗口标题
09         this.setBounds(300, 250, 300, 200);    //设置窗口的大小和位置
10         this.setResizable(false);                //设置窗口不能被调整大小
11         this.setVisible(true);                  //设置窗口是可见的
12     }
13     public static void main(String args[])
14     {
15         Swing4 s=new Swing4();
16     }
17 }
```



图 14-4 不能调整大小的窗体

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 14-4 所示。

**【代码解析】**有些读者会认为该结果是和图 14-3 一样的，其实不然。该运行结果中的窗体是不能最大化的，这主要是由于在程序中使用 setResizable 方法的参数为 false，使得窗口不能调整大小，从而也就使窗口不能最大化。如果试图通过鼠标来调整大小，也是不能成功的。



**提示：**在 JFrame 窗口类中还有一个 setUndecorated 方法，使用该方法可以将窗体的边框和标题栏去掉。

下面是使用 setUndecorated 方法的程序。

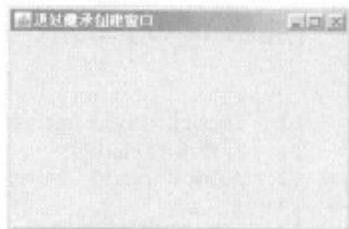


图 14-3 通过继承创建



**【范例 14-5】**示例代码 14-5 是一个使用 setUndecorated 方法对窗口进行设置的程序。

示例代码 14-5

```
01 import javax.swing.*; //导入 Swing 包
02 //继承 JFrame 类
03 public class Swing5 extends JFrame
04 {
05     //定义构造器
06     public Swing5()
07     {
08         this.setTitle("通过继承创建窗口"); //设置窗口标题
09         this.setBounds(300, 250, 300, 200); //设置窗口的大小和位置
10         this.setUndecorated(true); //设置窗口没有边框和标题栏
11         this.setVisible(true); //设置窗口是可见的
12     }
13     public static void main(String args[])
14     {
15         Swing5 s=new Swing5();
16     }
17 }
```



图 14-5 没有边框的窗体

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 14-5 所示。

**【代码解析】**读者看到该程序运行结果可能会感到很奇怪。该窗体是没有边框和标题栏的。这主要是因为在程序中使用了 setUndecorated 方法，并设置该方法的参数为 true，从而使窗体不再有边框和标题栏。

很多窗体中都是有标题图片的，同样使用 Swing 编写的窗体也不例外。在 JFrame 窗口类中具有一个 setIconImage 方法，使用该方法能够在标题中出现图片。

**【范例 14-6】**示例代码 14-6 是一个为标题添加图片的程序。

示例代码 14-6

```
01 import javax.swing.*;
02 import java.awt.*;
03 //继承 JFrame 类
04 public class Swing6 extends JFrame
05 {
06     //定义构造器
07     public Swing6()
08     {
09         this.setTitle("通过继承创建窗口"); //设置窗口标题
10         this.setBounds(300, 250, 300, 200); //设置窗口的大小和位置
11         Image i=this.getToolkit().getImage("a.jpg"); //指定一张图片
12         this.setIconImage(i); //设置窗口标题
13         this.setVisible(true); //设置窗口是可见的
14     }
15     public static void main(String args[])
16     {
17         Swing6 s=new Swing6();
18     }
19 }
```



【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 14-6 所示。

【代码解析】在本程序的第 11 行, 首先指定一张图片, a.jpg 图片和该程序处于同一目录下, 所以可以直接给出该图片的名称就可以; 然后就可以使用 setIconImage 方法将该图片设置为窗体的图标。

在 JFrame 窗口类中还有一个很重要的方法, 那就是 setDefaultCloseOperation 方法。使用该方法可以设置当单击关闭按钮关闭窗口时所执行的动作。这里的动作包括 4 种情况, 分别对应着 4 个常量。对应关系如表 14-1 所示。

表 14-1 窗体关闭对应的静态常量

常量	含义
DO NOTHING ON CLOSE	不执行任何动作
DISPOSE ON CLOSE	释放窗体对象
HIDE ON CLOSE	隐藏窗体
EXIT ON CLOSE	退出 JVM



**提示:** 如果不使用 setDefaultCloseOperation 方法进行设置, 默认值为 HIDE ON CLOSE, 也就是在默认情况下单击关闭按钮将会使窗口隐藏。



## 14.3 JPanel 面板类

上一节讲解的 JFrame 窗口类是一个容器类, 从本节开始将讲解一些控件。首先要讲解的就是 JPanel 面板类。面板可以说是控件, 但它同样是一种容器, 只不过它不是顶层容器, 所以本节要先了解一下什么是容器, 然后再介绍 JPanel 面板类。

### 14.3.1 容器介绍

Swing 中的控件可以分为三类, 顶层容器、非顶层容器和普通控件。在前面介绍的 JFrame 窗口类就是一个顶层容器。顶层容器是一种可以直接显示在系统桌面上的控件, 其他控件必须直接或间接地借助顶层容器进行显示。顶层容器除了包括 JFrame 窗口类外, 还包括 JWindow 和 JDialog 等不常用的类。

本节中将介绍的 JPanel 面板类是一个非顶级容器, 非顶级容器具有两面性。非顶级容器是要放到顶级容器中使用的, 对于顶级容器来说, 非顶级容器是一般控件。在非顶级容器中还可以添加控件, 对于这些控件来看, 非顶级容器就是一个容器。

普通控件在控件中占大部分, 使用这些控件可以实现特定的功能, 但它们不具有容器的作用, 它们只能放在容器中进行显示。普通控件包括按钮、文本框等很多控件。

有些读者会认为将普通控件直接放到顶级容器中不也可以完成显示功能吗? 这种说法在语法上是正确的, 但是正确不一定是合理的。这种设计将会使程序变的非常复杂, 而且难以维护。通常在设计界面时, 都会先定义顶级容器, 然后向顶级容器中添加非顶级容

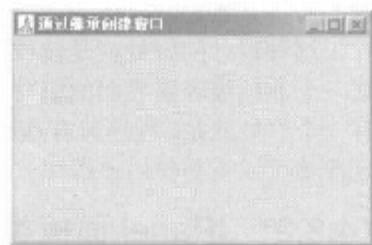


图 14-6 设置窗体图标



器，而将普通控件放在非顶级容器中。

这种设计的好处就是使程序开发变得简单，在开发时开发员在某一时间内只需要关注某一个非顶级容器界面的编写，最后将所有的非顶级容器添加到顶级容器中。这种设计还有一个好处就是，程序具有重用性，因为有可能在多个界面中使用同一个非顶级容器程序，这样就可以重复使用该程序。

### 14.3.2 JPanel 面板类简介

JPanel 面板类是一个非顶级容器，使用 JPanel 面板类可以搭建一个子界面。JPanel 面板类同样具有 4 种构造器，最常用的仍然是无参构造器。使用有参构造器可以在初始时设置面板采用什么布局管理器和是否使用双缓冲。

JPanel 面板类本身没有特殊功能，它的作用就是作为非顶级容器来添加普通控件，搭建子界面。所以 JPanel 面板类的方法也是很少很简单的。首先 JPanel 面板类具有一个添加控件的 add 方法，使用该方法能够将普通控件添加到面板中。getHeight 方法和 getWidth 方法分别是返回当前面板的高度和宽度。



**提示：**JPanel 面板类还有一个 setToolTipText 方法，该方法具有一个字符串参数，该方法的作用主要是当鼠标指针停留在面板上时显示文本，字符串内容就是要显示的内容。

### 14.3.3 创建面板

在前面的学习中读者已经知道，面板必须添加到窗口中，而面板中还可以添加普通的控件。本节就来学习如何创建面板，以及如何进行添加操作。

**【范例 14-7】**示例代码 14-7 是一个创建面板的程序。

示例代码 14-7

```
01 import javax.swing.*; //导入 swing 包
02 //继承 JFrame 类
03 public class Swing7 extends JFrame
04 {
05     JPanel jp=new JPanel(); //创建一个面板
06     JButton jb=new JButton("按钮"); //创建一个按钮
07     //定义构造器
08     public Swing7()
09     {
10         this.setTitle("创建面板"); //设置窗口名称
11         jp.add(jb); //将按钮添加到面板中
12         this.add(jp); //将面板添加到窗口中
13         this.setBounds(300,250,300,200); //设置窗口的大小和位置
14         this.setVisible(true); //设置窗口是可见的
15     }
16     public static void main(String args[])
17     {
18         Swing7 s=new Swing7();
19     }
20 }
```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 14-7 所示。

【代码解析】在本程序中是一个创建面板的程序。在示例代码 14-7 中的第 5 行创建了一个面板，在第 12 行是让窗体调用 add 方法将该面板添加到窗体中。在第 6 行是创建的一个按钮，在第 11 行将该按钮添加到面板中。有些读者可能会有疑问了，在运行结果中只有一个按钮，怎么没有看到面板。这是因为面板不是普通的控件，它是一个放置控件的容器，所以它是不显示的。



图 14-7 创建面板



## 14.4 JLabel 标签类

标签是 Swing 中最基本的控件，它是一种非交互的控件，也就是不需要进行操作的控件。标签虽然通常只起到一个显示功能，但是它是界面编程中必不可少的。使用标签能够给用户提供更多的相关信息。

### 14.4.1 JLabel 标签类简介

JLabel 标签类的知识点要比前面所学到的 JPanel 面板类的知识点多很多。首先 JLabel 标签类具有 6 个构造器来创建标签，在表 14-2 中列出了这 6 个构造器。

表 14-2 标签类构造器

标签构造器	说 明
public void JLabel()	创建没有图像和标题的标签
public void JLabel(Icon image)	创建具有图像的标签
public void JLabel(Icon image,int horizontalAlignment)	创建具有图像和指定对齐方式的标签
public void JLabel(String text)	创建指定文本的标签
public void JLabel(String text,int horizontalAlignment)	创建指定文本和对齐方式的标签
public void JLabel(String text,Icon image,int horizontalAlignment)	创建指定文本、图像和对齐方式的标签

使用表 14-2 中的标签类构造器都能够创建标签，其中最常见的还是无参构造器。在一些构造器中使用到了对齐方式的参数，对齐方式是通过 JLabel 标签类的静态常量来定义的，在表 14-3 中给出了表示对齐方式的常量。

表 14-3 对齐方式常量

常量名	说 明
LEADING	水平方向对齐到左边界，垂直方向上对齐到上边界
TRAILING	水平方向对齐到右边界，垂直方向上对齐到下边界
LEFT	左对齐
RIGHT	右对齐
TOP	上对齐
BOTTOM	下对齐
CENTER	居中



JLabel 标签类的方法有很多，这些方法都是对应的形式，分别是获取和设置方法。这里给出一些比较常用的方法，其中 `setText` 方法已经在前面的学习中使用过，表示设置标签要显示的文本。同时和这个方法相对应的就是 `getText` 方法，使用该方法来获取标签显示的文本。除了这两个方法外，还有对图像、对齐方式等进行操作的方法，这些在以后的学习中使用时将进行讲解。

#### 14.4.2 创建标签

学习完了 JLabel 标签类后，创建标签就是很容易的问题。示例代码 14-8 就是一个创建简单标签的程序。

**【范例 14-8】**示例代码 14-8 是一个创建标签的程序。

示例代码 14-8

```
01 import javax.swing.*; //导入 Swing 包
02 //继承 JFrame 类
03 public class Swing8 extends JFrame
04 {
05     JLabel jl=new JLabel(); //创建一个标签
06     //定义构造器
07     public Swing8()
08     {
09         this.setTitle("创建标签"); //设置窗口名称
10         jl.setText("这是一个标签"); //设置标签显示的内容
11         jl.setVerticalAlignment(JLabel.CENTER); //设置标签垂直居中
12         jl.setHorizontalAlignment(JLabel.CENTER); //设置标签水平居中
13         this.add(jl); //将标签添加到窗口中
14         this.setBounds(300,250,300,200); //设置窗口的大小和位置
15         this.setVisible(true); //设置窗口是可见的
16     }
17     public static void main(String args[])
18     {
19         Swing8 s=new Swing8();
20     }
21 }
```



图 14-8 创建标签

**【运行结果】**使用 `javac` 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 14-8 所示。

**【代码解析】**其中第 5 行使用 `JLabel` 标签类的无参构造器创建了一个标签。在第 10 行使用 `setText` 方法设置标签上要显示的内容。在第 11 行使用 `setVerticalAlignment` 方法设置标签在容器中垂直居中。在第 12 行使用 `setHorizontalAlignment` 方法设置标签在容器中水平居中。在第 13 行将该标签添加到窗口中。在该程序中为了使程序简单，就直接将标签放在窗体中，而没有再定义非顶级容器。



#### 14.5 JButton 按钮类

为了更好地学习下一章的布局管理器，下面介绍一个 Swing 中最常见的控件，那就是

按钮。按钮是进行交互操作使用最多的控件，同时按钮也是相对简单的控件。在下一章中学习布局管理器时，将使用按钮来进行举例说明，所以该节也是学习布局管理器的基础。

### 14.5.1 JButton 按钮类简介

使用 JButton 按钮类可以创建最常用的按钮控件。JButton 按钮类同样具有多个构造器，使用这些构造器都能够创建按钮控件。最常用的仍然是使用无参构造器创建一个不带文本和图标的按钮。

在 JButton 按钮类中具有几个很常用的方法。其中 setText 方法是用来设置按钮上显示的文本，和其对应的是用 getText 方法来获取按钮上显示的文本。在 JButton 按钮类中还有一个经常被使用，也是非常有意思的 setMnemonic 方法，使用该方法可以设置按钮的助记符。助记符就是使用键盘中的 Alt 加该助记符就能起到相应功能的特殊符号。例如在 Word 中，使用 Alt+F 就能打开文件菜单。为按钮添加助记符后就可以使用 Alt 加该助记符来代替单击按钮的操作。

 **提示：**助记符就是使用键盘中的 Alt 加该助记符就能起到相应功能。

### 14.5.2 创建按钮

学习完了 JButton 按钮类后，创建按钮就是很容易的问题。示例代码 14-9 就是一个创建简单按钮的程序。

**【范例 14-9】**示例代码 14-9 是一个创建按钮的程序。

示例代码 14-9

```

01 import javax.swing.*; //导入 Swing 包
02 //继承 JFrame 类
03 public class Swing9 extends JFrame
04 {
05     JButton jb=new JButton(); //创建一个按钮
06     //定义构造器
07     public Swing9()
08     {
09         this.setTitle("创建按钮"); //设置窗口名称
10         jb.setText("这是一个按钮"); //设置按钮上显示的内容
11         jb.setMnemonic('a'); //设置按钮的助记符
12         this.add(jb); //将按钮添加到窗口中
13         this.setBounds(300,250,300,200); //设置窗口的大小和位置
14         this.setVisible(true); //设置窗口是可见的
15     }
16     public static void main(String args[])
17     {
18         Swing9 s=new Swing9();
19     }
20 }
```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 14-9 所示。

**【代码解析】**在示例代码 14-9 程序的第 5 行使用空构

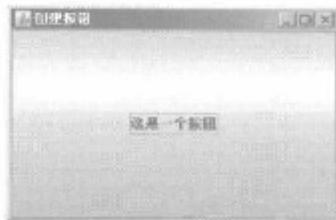


图 14-9 创建按钮



造器创建了一个按钮。在第 10 行使用 `setText` 方法设置了按钮上显示的内容。在第 11 行使用 `setMnemonic` 方法设置了按钮的助记符。在窗口中单击, 可以看出是该界面中是一个按钮, 同样使用 `Alt+A` 同样能起到单击按钮的作用。有些读者可能会感到奇怪, 为什么整个窗口中就只有这一个按钮, 而且占满整个窗口。这个问题在学完下一章就会明白是怎么回事。

### 14.5.3 按钮动作事件

按钮是具有动作事件的, 单击按钮时触发动作事件, 也就是 `ActionEvent` 事件。但是如果想让按钮在触发事件后执行程序, 就需要为按钮添加动作事件监听器, 并且需要为按钮注册动作事件监听器。编写动作事件监听器是通过实现 `ActionListener` 监听接口来完成的。

在 `ActionListener` 监听接口中只有一个 `actionPerformed` 方法, 所以在动作事件监听器中只需要实现这一个方法。将触发事件后将执行的程序都写在 `actionPerformed` 方法中。定义完监听器后, 还需要向按钮注册该监听器, 注册监听器是通过 `addActionListener` 方法来实现的。

**【范例 14-10】**示例代码 14-10 是一个演示按钮动作事件的程序。

示例代码 14-10

```
01 import javax.swing.*; //导入 Swing 包
02 import java.awt.event.*; //导入事件包
03 //继承 JFrame 类
04 public class Swing10 extends JFrame
05 {
06     JButton jb=new JButton(); //创建一个按钮
07     int i=0; //定义一个表示按下次数的变量
08     //定义构造器
09     public Swing10()
10     {
11         this.setTitle("创建按钮"); //设置窗口名称
12         jb.setText("按钮按下了 0 次"); //设置按钮上显示的内容
13         jb.setMnemonic('a'); //设置按钮的助记符
14         this.add(jb); //将按钮添加到窗口中
15         //为按钮注册监听器
16         jb.addActionListener(new ActionListener()
17         {
18             //触发动作事件时, 执行的方法
19             public void actionPerformed(ActionEvent e)
20             {
21                 Swing10.this.jb.setText("按钮按下了
22                 "+(++i)+" 次");
23             }
24         });
25         this.setBounds(300, 250, 300, 200); //设置窗口的大小和位置
26         this.setVisible(true); //设置窗口是可见的
27     }
28     public static void main(String args[])
29     {
30         Swing10 s=new Swing10();
31     }
32 }
```

**【运行结果】**使用 `javac` 编译程序将产生一个和该程序对应的 `class` 程序, 然后使用 `Java` 运行编译产生的 `class` 程序, 运行结果如图 14-10 所示。

在窗口中单击一次按钮，运行结果如图 14-11 所示。



图 14-10 按钮动作事件

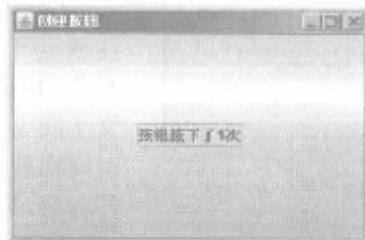


图 14-11 单击按钮

**【代码解析】**在示例代码 14-10 中第 16 行为按钮注册了一个事件监听器。该程序中的动作事件监听器是通过匿名内部类来定义的，这里使用匿名内部类使得该程序非常简单。在动作事件监听器中重写了 ActionListener 监听接口的 actionPerformed 方法。在该方法中将表示单击按钮次数的变量增加 1，并且显示在按钮上。在本程序中同样也为按钮定义了助记符，使用 Alt+A 同样能起到单击按钮的作用。

有些读者可能对前面学过的匿名内部类不是很明白，下面也给出一个不使用匿名内部类来创建按钮动作事件的程序。

**【范例 14-11】**示例代码 14-11 是一个不使用匿名内部类实现动作事件监听的程序。

#### 示例代码 14-11

```

01 import javax.swing.*; //导入 Swing 包
02 import java.awt.event.*; //导入事件包
03 //继承 JFrame 类
04 public class Swing11 extends JFrame implements ActionListener
05 {
06     JButton jb=new JButton(); //创建一个按钮
07     int i=0; //定义一个表示按下次数的变量
08     //定义构造器
09     public Swing11()
10     {
11         this.setTitle("创建按钮"); //设置窗口名称
12         jb.setText("按钮按下了 0 次"); //设置按钮上显示的内容
13         jb.setMnemonic('a'); //设置按钮的助记符
14         this.add(jb); //将按钮添加到窗口中
15         jb.addActionListener(this); //为按钮注册监听器
16         this.setBounds(300,250,300,200); //设置窗口的大小和位置
17         this.setVisible(true); //设置窗口是可见的
18     }
19     //触发动作事件时，执行的方法
20     public void actionPerformed(ActionEvent e)
21     {
22         this.jb.setText("按钮按下了 "+(i)+" 次");
23     }
24     public static void main(String args[])
25     {
26         Swing11 s=new Swing11();
27     }
28 }
```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java

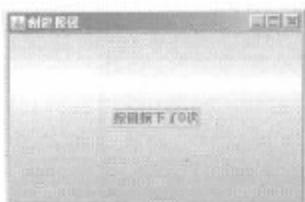


图 14-12 另一种方法

件，这种方法更直观。

运行编译产生的 class 程序，运行结果如图 14-12 所示。

**【代码解析】**在示例代码 14-11 的第 15 行同样需要为按钮注册动作事件监听器。在程序的第 20 行到 23 行实现了 ActionListener 接口中的 actionPerformed 方法，在该方法中定义了同上一个程序同样的功能。这两种定义事件的方法都是实现同样的功能，可以自己来选择以后使用哪一种方法来开发事件程序。笔者的建议是通过匿名内部类的方法来定义和注册事件。



## 14.6 Swing 中的事件

虽然在对按钮的讲解中已经使用了事件，但是还是有必要对事件进行一个总体的讲解。对于一个界面程序来说，如果只能显示一些控件，这是完全不能满足功能要求的。通过事件的使用，就可以使界面具有更加丰富的功能。

### 14.6.1 事件简介

事件是一种很好的让界面和用户进行交互的手段。当用户和界面交互时，经常会进行一些操作，例如单击按钮，按下指定键盘键，都会触发事件。事件触发后会告诉程序发生的事情，程序会根据不同的事件做出响应。在事件的发生和响应的过程中需要两个对象，事件源和事件监听器。

事件源就是触发事件的控件，这里包括按钮、文本框、窗体等很多种控件。但是不同的控件存在不同的事件，事件信息被封装在事件对象中。事件监听器是指实现专门的监听接口的类的对象。每一个事件都有对应的监听接口，同时在该接口中给出了处理事件的方法。在编写监听器时需要事件监听接口，同时实现其中的方法，在方法中编写触发事件后执行的程序。在编写程序时，还需要将监听器注册给事件源，这样才能执行事件。



**提示：**事件源和监听器之间是多对多的关系，一个事件源可以对应多个监听器，一个监听器可以为多个事件源服务，这在后面将会给出具体的程序进行讲解。

### 14.6.2 同一个事件源注册多个监听器

同一个事件源可以同时注册多个监听器，这种情况下触发事件，所有的监听器都将执行事件方法，对事件进行处理。

**【范例 14-12】**示例代码 14-12 是一个演示同一个事件源可以同时注册多个监听器的程序。

#### 示例代码 14-12

```
01 import javax.swing.*;           //导入 Swing 包
02 import java.awt.event.*;         //导入事件包
03 //继承 JFrame 类
04 public class Swing12 extends JFrame
05 {
```

```

06     JButton jb=new JButton();           //创建一个按钮
07     int i=0;                          //定义一个表示按下次数的变量
08     //定义构造器
09     public Swing12()
10     {
11         this.setTitle("创建按钮");        //设置窗口名称
12         jb.setText("按钮按下了 0 次");    //设置按钮上显示的内容
13         jb.setMnemonic('a');            //设置按钮的助记符
14         this.add(jb);                  //将按钮添加到窗口中
15         //为按钮注册监听器
16         jb.addActionListener(new ActionListener())
17         {
18             //触发动作事件时, 执行的方法
19             public void actionPerformed(ActionEvent e)
20             {
21                 Swing12.this.jb.setText("按钮按下了
22                 "+(++i)+" 次");
23             }
24         );
25         //为按钮注册第二个监听器
26         jb.addActionListener(new ActionListener()
27         {
28             //触发动作事件时, 执行的方法
29             public void actionPerformed(ActionEvent e)
30             {
31                 Swing12.this.jb.setText("按钮按下了
32                 "+(++i)+" 次");
33             }
34         );
35         this.setBounds(300,250,300,200);    //设置窗口的大小和位置
36         this.setVisible(true);           //设置窗口是可见的
37     }
38     public static void main(String args[])
39     {
40         Swing12 s=new Swing12();
41     }
42 }

```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 14-13 所示。

在图 14-13 的运行结果中, 单击一次按钮, 运行结果如图 14-14 所示。



图 14-13 多个监听器



图 14-14 按下一次按钮

【代码解析】首先从运行结果来看, 当单击一次按钮时, 运行结果中显示“按钮按下了 2 次”, 这是因为该按钮注册了多个监听器。在示例代码 14-12 中的第 16 到第 24 行为按



钮注册了第一个监听器。从第 26 行到第 34 行为按钮注册了第二个监听器。在两个监听器中都是将表示次数的变量增加 1，因为按钮触发事件后都会执行监听器中的方法，从而使结果增加 2。

当为同一个事件源注册多个监听器时，监听器的执行顺序并不是先注册先执行的顺序，而是先注册后执行的顺序来执行的。读者可以写一个程序来演示这一点。

### 14.6.3 同一个监听器注册给多个事件源

在同一个监听器注册给多个事件源的情况下，所有的事件源中的任意一个触发事件都会通知监听器，并执行监听器中的事件处理方法。

**【范例 14-13】**示例代码 14-13 是一个演示同一个监听器注册给多个事件源的程序。

示例代码 14-13

```
01 import javax.swing.*; //导入 Swing 包
02 import java.awt.event.*; //导入事件包
03 //继承 JFrame 类
04 public class Swing13 extends JFrame implements ActionListener
05 {
06     JButton jb1=new JButton(); //创建一个按钮
07     JButton jb2=new JButton(); //创建第二个按钮
08     int i=0; //定义一个表示第一个按钮按下次数的变量
09     int j=0; //定义一个表示第二个按钮按下次数的变量
10     JPanel jp=new JPanel(); //定义构造器
11     public Swing13()
12     {
13         this.setTitle("创建按钮"); //设置窗口名称
14         jb1.setText("按钮按下了 " + i + " 次"); //设置按钮上显示的内容
15         jb2.setText("按钮按下了 " + j + " 次"); //设置按钮上显示的内容
16         jb1.setMnemonic('a'); //设置按钮的助记符
17         jb2.setMnemonic('b'); //设置按钮的助记符
18         jp.add(jb1); //将按钮添加到面板中
19         jp.add(jb2); //将按钮添加到面板中
20         this.add(jp); //将面板添加到窗体中
21         jb1.addActionListener(this); //为按钮注册监听器
22         jb2.addActionListener(this); //为按钮注册监听器
23         this.setBounds(300, 250, 300, 200); //设置窗口的大小和位置
24         this.setVisible(true); //设置窗口是可见的
25     }
26     //触发动作事件时，执行的方法
27     public void actionPerformed(ActionEvent e)
28     {
29         //如果事件源是第一个按钮
30         if(e.getSource() == jb1)
31         {
32             //设置第二个按钮的显示内容
33             this.jb2.setText("按钮按下了 " + (++j) + " 次");
34         }
35         //如果事件源是第二个按钮
36         else if(e.getSource() == jb2)
37         {
38             //设置第一个按钮的显示内容
39             this.jb1.setText("按钮按下了 " + (++i) + " 次");
40         }
41     }
42 }
```

```

41      }
42  }
43  public static void main(String args[])
44  {
45      Swing13 s=new Swing13();
46  }
47 }

```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序,然后使用 Java 运行编译产生的 class 程序,运行结果如图 14-15 所示。

**【代码解析】**在该运行结果中,如果单击左边的按钮,则右边按钮显示“按钮按下了 0 次”。同样如果单击右边的按钮,则左边按钮显示“按钮按下了 1 次”。在本程序中,将一个监听器注册给多个事件源,从而不管是单击哪一个按钮,都将执行监听器方法。为了辨别是哪一个按钮被按下,在监听器方法中需要判断是哪一个按钮触发了事件,执行的效果是使另一个按钮的显示内容发生变化。



图 14-15 注册多个事件源

#### 14.6.4 窗体获取和失去焦点事件

在 Swing 中,针对窗体的事件有很多,但是这些事件都是很容易理解的。窗体中的所有事件都是使用 WindowEvent 类来表示。在本节中就先来介绍窗体获取和失去焦点事件,该事件是通过实现 WindowFocusListener 监听接口实现的。

**【范例 14-14】**示例代码 14-14 是一个演示窗体焦点的事件。

##### 示例代码 14-14

```

01 import javax.swing.*; //导入 Swing 包
02 import java.awt.event.*; //导入事件包
03 //继承 JFrame 类并实现 WindowFocusListener 接口
04 public class Swing14 extends JFrame implements WindowFocusListener
05 {
06     //定义构造器
07     public Swing14()
08     {
09         this.setTitle("窗体获取和失去焦点事件"); //设置窗口标题
10         this.addWindowFocusListener(this); //注册窗体获取和失去焦点事件
11         this.setBounds(300,250,300,200); //设置窗口的大小和位置
12         this.setVisible(true); //设置窗口是可见的
13     }
14     //当窗体获取焦点时执行该方法
15     public void windowGainedFocus(WindowEvent e)
16     {
17         //在后台显示"窗体获取焦点"
18         System.out.println("窗体获取焦点");
19     }
20     //当窗体失去焦点时执行该方法
21     public void windowLostFocus(WindowEvent e)
22     {
23         //在后台显示"窗体失去焦点"
24         System.out.println("窗体失去焦点");

```



```
25      }
26      public static void main(String args[])
27      {
28          Swing14 s=new Swing14();
29      }
30  }
```

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 14-16 所示。

该程序的运行结果主要体现在后台中, 后台的运行结果如图 14-17 所示。



图 14-16 窗体焦点事件



图 14-17 后台运行结果

**【代码解析】**示例代码 14-14 是一个演示窗体获取和失去焦点事件的程序。首先在程序的第 4 行需要实现 WindowFocusListener 监听接口, 然后在第 10 行为窗体注册该事件监听。在程序中还需要实现 WindowFocusListener 监听接口中的 windowGainedFocus 方法和 windowLostFocus 方法。windowGainedFocus 方法是当窗体获取焦点时需要执行的方法, windowLostFocus 方法是当窗体失去焦点时执行的方法。这里只是当触发事件后, 在后台中显示一句话。

#### 14.6.5 窗体打开、关闭和激活事件

要实现窗体打开、关闭和激活事件只需要实现 WindowListener 监听接口的监听器。WindowListener 监听接口中同样具有几种方法, 这里还是通过程序来讲解这些方法。

**【范例 14-15】**示例代码 14-15 是一个演示窗体打开、关闭和激活事件的程序。

示例代码 14-15

```
31  import javax.swing.*;
32  import java.awt.event.*;
33  //继承 JFrame 类并实现 WindowListener 接口
34  public class Swing15 extends JFrame implements WindowListener
35  {
36      //定义构造器
37      public Swing15()
38      {
39          this.setTitle("窗体打开、关闭和激活事件"); //设置窗口标题
40          this.addWindowListener(this); //注册窗体打开、关闭和激活事件
41          this.setBounds(300,250,300,200); //设置窗口的大小和位置
42          this.setVisible(true); //设置窗口是可见的
43      }
44      //当窗体首次变为可见执行该方法
45      public void windowOpened(WindowEvent e)
46      {
47          //在后台显示"窗体首次变为可见"
48      }
49  }
```



```

18         System.out.println("窗体首次变为可见");
19     }
20     //当关闭窗口时执行该方法
21     public void windowClosing(WindowEvent e)
22     {
23         //在后台显示"关闭窗体"
24         System.out.println("关闭窗体");
25     }
26     //当关闭窗口后执行该方法
27     public void windowClosed(WindowEvent e)
28     {
29         //在后台显示"关闭窗口后"
30         System.out.println("关闭窗口后");
31     }
32     //当窗体从正常状态变为最小化状态时执行该方法
33     public void windowIconified(WindowEvent e)
34     {
35         //在后台显示"从正常状态变为最小化状态"
36         System.out.println("从正常状态变为最小化状态");
37     }
38     //当窗体从最小化状态变为正常状态执行该方法
39     public void windowDeiconified(WindowEvent e)
40     {
41         //在后台显示"从最小化状态变为正常状态"
42         System.out.println("从最小化状态变为正常状态");
43     }
44     //当窗体被激活时执行该方法
45     public void windowActivated(WindowEvent e)
46     {
47         //在后台显示"窗体被激活"
48         System.out.println("窗体被激活");
49     }
50     //当窗体变为非激活时执行该方法
51     public void windowDeactivated(WindowEvent e)
52     {
53         //在后台显示"窗体变为非激活"
54         System.out.println("窗体变为非激活");
55     }
56     public static void main(String args[])
57     {
58         Swing15 s=new Swing15();
59     }
60 }

```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 14-18 所示。

同样该程序的运行结果也主要体现在后台上, 后台的运行结果如图 14-19 所示。

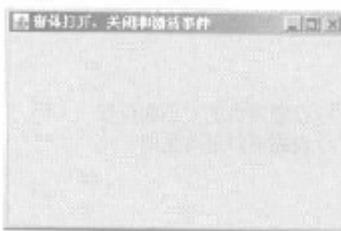


图 14-18 打开、关闭和激活事件

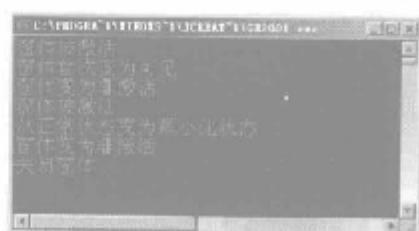


图 14-19 后台运行结果



【代码解析】示例代码 14-15 是一个演示窗体打开、关闭和激活事件的程序。要使窗体能够发生打开、关闭和激活事件必须实现 WindowListener 监听接口，在该接口中有多个方法，这些方法都是非常容易理解的，这里就需要读者自己动手操作，来看每一个方法的功能。其中需要讲解的就是 windowClosing 方法和 windowClosed 方法。windowClosing 方法是指当窗口正在关闭时执行的方法，而 windowClosed 方法是指窗口完全关闭后执行的方法。



## 14.7 综合练习

### 1. 开发一个同一个事件源注册多个监听器的程序。

【提示】

```
01 import javax.swing.*;           //导入 Swing 包
02 import java.awt.event.*;         //导入事件包
03 //继承 JFrame 类
04 public class LianXil extends JFrame
05 {
06     JButton jb=new JButton();      //创建一个按钮
07     int i=0;                      //定义一个表示按下次数的变量
08     //定义构造器
09     public LianXil()
10     {
11         this.setTitle("创建按钮");    //设置窗口名称
12         jb.setText("按钮按下了 0 次"); //设置按钮上显示的内容
13         jb.setMnemonic('a');         //设置按钮的助记符
14         this.add(jb);              //将按钮添加到窗口中
15         //为按钮注册监听器
16         jb.addActionListener(new ActionListener()
17         {
18             //触发动作事件时，执行的方法
19             public void actionPerformed(ActionEvent e)
20             {
21                 LianXil.this.jb.setText("按钮按下了
22                     "+(++i)+" 次");
23             }
24         });
25         //为按钮注册第二个监听器
26         jb.addActionListener(new ActionListener()
27         {
28             //触发动作事件时，执行的方法
29             public void actionPerformed(ActionEvent e)
30             {
31                 LianXil.this.jb.setText("按钮按下了
32                     "+(++i)+" 次");
33             }
34         });
35         this.setBounds(300,250,300,200); //设置窗口的大小和位置
36         this.setVisible(true);          //设置窗口是可见的
37     }
38     public static void main(String args[])
39     {
40         LianXil s=new LianXil();
```

```

41      )
42  }

```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 14-20 所示。

## 2. 开发一个同一个监听器注册给多个事件源的程序。

### 【提示】

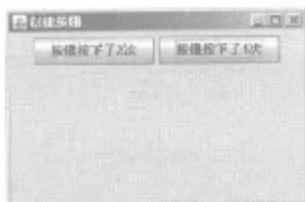
```

01  import javax.swing.*;           //导入 Swing 包
02  import java.awt.event.*;        //导入事件包
03  //继承 JFrame 类
04  public class LianXi2 extends JFrame implements ActionListener
05  {
06      JButton jbl=new JButton();    //创建一个按钮
07      JButton jb2=new JButton();   //创建第二个按钮
08      int i=0;                     //定义一个表示第一个按钮按下次数的变量
09      int j=0;                     //定义一个表示第二个按钮按下次数的变量
10      JPanel jp=new JPanel();     //定义构造器
11      //定义构造器
12      public LianXi2()
13      {
14          this.setTitle("创建按钮");    //设置窗口名称
15          jbl.setText("按钮按下了 0 次"); //设置按钮上显示的内容
16          jb2.setText("按钮按下了 0 次"); //设置按钮上显示的内容
17          jbl.setMnemonic('a');        //设置按钮的助记符
18          jb2.setMnemonic('b');        //设置按钮的助记符
19          jp.add(jbl);              //将按钮添加到面板中
20          jp.add(jb2);              //将按钮添加到面板中
21          this.add(jp);              //将面板添加到窗体中
22          jbl.addActionListener(this); //为按钮注册监听器
23          jb2.addActionListener(this); //为按钮注册监听器
24          this.setBounds(300,250,300,200); //设置窗口的大小和位置
25          this.setVisible(true);      //设置窗口是可见的
26      }
27      //触发动作事件时，执行的方法
28      public void actionPerformed(ActionEvent e)
29      {
30          //如果事件源是第一个按钮
31          if(e.getSource()==jbl)
32          {
33              //设置第二个按钮的显示内容
34              this.jb2.setText("按钮按下了"+(++j)+"次");
35          }
36          //如果事件源是第二个按钮
37          else if(e.getSource()==jb2)
38          {
39              //设置第一个按钮的显示内容
40              this.jbl.setText("按钮按下了"+(++i)+"次");
41          }
42      }
43      public static void main(String args[])
44      {
45          LianXi2 s=new LianXi2();
46      }
47  }

```



图 14-20 练习 1



【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 14-21 所示。



## 14.8 小结

图 14-21 练习 2

本章是对 Java 中 Swing 程序入门的一章，本章只是对界面开发做了一个简单的介绍。在本章中，首先介绍了如何进行界面开发，然后分别介绍了窗口类、面板类、标签类和按钮类。在本章的最后对界面开发中非常重要的事件开发进行了讲解。如果读者想了解更多的关于本章的内容，可以参考电子工业出版社出版的《深入浅出 JDK 6.0》一书进行学习。



## 14.9 习题

### 一、填空题

- 在开发的 Swing 程序中，通常是通过继承 \_\_\_\_\_ 类来实现窗口的。
- 在 JFrame 窗口类中定义了一个 setVisible 方法来设置窗口的可见性，该方法具有一个布尔型参数，\_\_\_\_\_ 表示可见，\_\_\_\_\_ 表示不可见。
- 使用 setResizable 方法的参数为 \_\_\_\_\_，使得窗口不能调整大小，从而也就使窗口不能最大化。
- Swing 中的控件可以分为三类，\_\_\_\_\_、\_\_\_\_\_ 和 \_\_\_\_\_。
- JPanel 面板类是一个 \_\_\_\_\_，使用 JPanel 面板类可以搭建一个子界面。
- 使用 \_\_\_\_\_ 可以创建最常用的按钮控件。

### 二、选择题

- 设置窗体是否能够调整大小的方法为（ ）。  
A. setResizable 方法      B. setUndecorated 方法  
C. setIconImage 方法      D. setDefaultCloseOperation 方法
- 向面板中添加控件的方法为（ ）。  
A. add 方法      B. getHeight 方法  
C. getWidth 方法      D. setToolTipText 方法
- 设置按钮助记符的方法为（ ）。  
A. setText 方法      B. getText 方法  
C. setMnemonic 方法      D. add 方法
- 在下面程序运行结果中，各单击一次按钮，单击顺序发生变化，运行结果为（ ）。

```
01 import javax.swing.*;           //导入 Swing 包
02 import java.awt.event.*;         //导入事件包
03 //继承 JFrame 类
04 class Swing13 extends JFrame implements ActionListener
05 {
06     JButton jbt=new JButton();     //创建一个按钮
07 }
```

```

07     JButton jb2=new JButton();           //创建第二个按钮
08     int i=0;                           //定义一个表示第一个按钮按下次数的变量
09     int j=0;                           //定义一个表示第二个按钮按下次数的变量
10     JPanel jp=new JPanel();           //定义构造器
11     //定义构造器
12     public Swing13()
13     {
14         this.setTitle("创建按钮");        //设置窗口名称
15         jb1.setText("按钮按下了 0 次");   //设置按钮上显示的内容
16         jb2.setText("按钮按下了 0 次");   //设置按钮上显示的内容
17         jb1.setMnemonic('a');            //设置按钮的助记符
18         jb2.setMnemonic('b');            //设置按钮的助记符
19         jp.add(jb1);                  //将按钮添加到面板中
20         jp.add(jb2);                  //将按钮添加到面板中
21         this.add(jp);                  //将面板添加到窗体中
22         jb1.addActionListener(this);    //为按钮注册监听器
23         jb2.addActionListener(this);    //为按钮注册监听器
24         this.setBounds(300,250,300,200); //设置窗口的大小和位置
25         this.setVisible(true);         //设置窗口是可见的
26     }
27     //触发动作事件时, 执行的方法
28     public void actionPerformed(ActionEvent e)
29     {
30         //如果事件源是第一个按钮
31         if(e.getSource()==jb1)
32         {
33             //设置第二个按钮的显示内容
34             this.jb2.setText(j+3);
35         }
36         //如果事件源是第二个按钮
37         else if(e.getSource()==jb2)
38         {
39             //设置第一个按钮的显示内容
40             this.jb1.setText(++j);
41         }
42     }
43     public static void main(String args[])
44     {
45         Swing13 s=new Swing13();
46     }
47 }

```

- A. 相同, 都为 1、3      B. 相同, 都为 1、4  
 C. 不同, 一种为 1、3, 另一种 1、4      D. 发生编译错误

### 三、简答题

1. 简述创建窗体必须要定义的几个方法, 以及这些方法的含义。
2. 简述如何将同一个事件源注册给多个监听器。

### 四、编程题

编写一个使用同一个监听器注册给多个事件源的程序。

# 第 15 章 布局管理器

在日常生活中，超市已经变为一个必不可少的基础设施。在超市中，所有的商品都被超市管理人员有条理地分好类，摆在指定的位置，日常用品放在一起，食品放在一起。在 Java Swing 界面开发中，就用到了超市原理。其中窗体就好像一个超市，窗体中的控件就好像是商品，而布局管理器就是超市的管理人员。在 Swing 编程中使用布局管理器能够非常有效地对容器中的控件进行有条理并且美观的摆放。布局管理器也是有很多种的，包括流布局、网格布局、边框布局和空布局等，本章将学习这些布局管理器。通过本章的学习，读者应该实现如下几个目标。

- 了解各种布局管理器的样式。
- 掌握每一种布局管理器的使用。



## 15.1 流布局

流布局是相对比较简单的一种布局管理器，也是最常用的布局管理器。在流布局中放置控件时，将按照控件的添加顺序，依次将控件从左到右进行摆放，并且在一行的最后会进行自动换行放置。在一行中，控件是默认居中放置的。

### 15.1.1 流布局介绍

布局管理器也是通过构造器来创建的。流布局是通过 `FlowLayout` 类来创建，`FlowLayout` 类具有三种构造器。首先是无参构造器，使用无参构造器能够创建一个默认的以居中对齐方式，控件间水平和垂直间距为 5 个像素的流布局。

`FlowLayout` 类还具有一个需要整型参数的构造器，使用该构造器能够创建一个指定对齐方式的流布局管理器，它的控件间水平和垂直间距仍然是默认的 5 个像素。流布局管理器的对齐方式如表 15-1 所示。

表 15-1 流布局管理器对齐方式

对齐方式	说 明
LEFT	控件左对齐
CENTER	控件居中，这也是默认值
RIGHT	控件右对齐
LEADING	控件与容器开始边对齐
TRAILING	控件与容器结束边对齐

在创建流布局管理器时，就可以给出这些常量，来定义该流布局管理器的对齐方式。`FlowLayout` 类还有一个具有三个参数的构造器，第一个参数表示流布局管理器的对齐方

式, 第二个参数表示流布局管理器中控件间水平间距, 第三个参数表示流布局管理器中控件间垂直间距。

**提示:** FlowLayout 类还有一个具有三个参数的构造器, 第一个参数表示流布局管理器的对齐方式, 第二个参数表示流布局管理器中控件间水平间距, 第三个参数表示流布局管理器中控件间垂直间距。

FlowLayout 类中还具有一些比较常用的方法, 使用这些方法能够很有效地对流布局管理器进行操作。getAlignment 方法和 setAlignment 方法分别获取和设置流布局管理器的对齐方式。getHgap 方法和 setHgap 方法分别获取和设置流布局管理器中控件和控件之间的水平间距。getVgap 方法和 setVgap 方法分别获取和设置流布局管理器中控件和控件之间的垂直间距。

## 15.1.2 使用流布局

学习完如何创建流布局后, 读者就可以自己动手来使用流布局。由于只学过按钮控件, 所以这里只使用按钮来演示流布局管理器。

**【范例 15-1】**示例代码 15-1 是一个简单的使用流布局管理器的程序。

示例代码 15-1

```

01 import javax.swing.*; //导入 Swing 包
02 import java.awt.event.*; //导入事件包
03 import java.awt.*;
04 //继承 JFrame 类
05 public class BuJul extends JFrame
06 {
07     JButton jb1=new JButton("第一个按钮"); //创建第一个按钮
08     JButton jb2=new JButton("第二个按钮"); //创建第二个按钮
09     JButton jb3=new JButton("第三个按钮"); //创建第三个按钮
10     JButton jb4=new JButton("第四个按钮"); //创建第四个按钮
11     JButton jb5=new JButton("第五个按钮"); //创建第五个按钮
12     JPanel jp=new JPanel(); //创建一个面板
13     //定义构造器
14     public BuJul()
15     {
16         this.setTitle("使用流布局管理器"); //设置窗口名称
17         jp.setLayout(new FlowLayout()); //设置面板的布局为流布局
18         jp.add(jb1); //将按钮添加到面板中
19         jp.add(jb2);
20         jp.add(jb3);
21         jp.add(jb4);
22         jp.add(jb5);
23         this.add(jp); //将面板添加到窗口中
24         this.setBounds(300,250,300,200); //设置窗口的大小和位置
25         this.setVisible(true); //设置窗口是可见的
26     }
27     public static void main(String args[])
28     {
29         BuJul s=new BuJul();
30     }
31 }
```

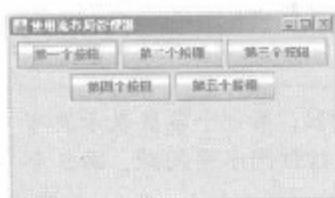


图 15-1 使用流布局

**【运行结果】**使用 `javac` 编译程序将产生一个和该程序对应的 `class` 程序, 然后使用 Java 运行编译产生的 `class` 程序, 运行结果如图 15-1 所示。

**【代码解析】**从运行结果中可以看出在流布局管理器中放置控件的方式, 放置顺序是按照控件的先后顺序, 从左到右依次摆放, 当一行放不下时会进行自动换行。当控件不满一行时, 会将该行中的控件居中显示。

上面的程序看起来是很麻烦的, 为了更好地让读者了解流布局管理器的使用, 这里写一个让流布局管理器和事件相结合的程序。

**【范例 15-2】**示例代码 15-2 是一个动态使用流布局的程序。

示例代码 15-2

```
01 import javax.swing.*; //导入 Swing 包
02 import java.awt.event.*; //导入事件包
03 import java.awt.*;
04 //继承 JFrame 类
05 public class BuJu2 extends JFrame implements ActionListener
06 {
07     int i;
08     JButton jb=new JButton("创建按钮"); //创建按钮
09     JPanel jp=new JPanel(); //创建一个面板
10     //定义构造器
11     public BuJu2()
12     {
13         this.setTitle("使用流布局管理器"); //设置窗口名称
14         jp.setLayout(new FlowLayout()); //设置面板的布局为流布局
15         jp.add(jb); //将按钮添加到面板中
16         jb.addActionListener(this); //为控件注册监听器
17         this.add(jp); //将面板添加到窗口中
18         this.setBounds(300,250,300,200); //设置窗口的大小和位置
19         this.setVisible(true); //设置窗口是可见的
20     }
21     public void actionPerformed(ActionEvent e) //实现监听接口方法
22     {
23         ++i; //为变量加 1
24         JButton jbi=new JButton("按钮"+i); //创建新按钮
25         jp.add(jbi); //将新按钮添加到面板中
26         this.show(true); //执行后刷新
27     }
28     public static void main(String args[])
29     {
30         BuJu2 s=new BuJu2();
31     }
32 }
```

**【运行结果】**使用 `javac` 编译程序将产生一个和该程序对应的 `class` 程序, 然后使用 Java 运行编译产生的 `class` 程序, 运行结果如图 15-2 所示。

在图 15-2 的运行结果中, 多次单击按钮, 运行结果如图 15-3 所示。



图 15-2 通过事件说明流布局

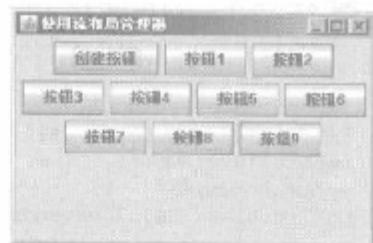


图 15-3 多次按按钮后

**【代码解析】**示例代码 15-2 是一个事件和流布局管理器结合的程序。在程序的第 16 行为初始的按钮注册了事件监听器，同时实现了监听接口中的方法。当触发事件后都会执行监听方法，在监听方法中，将首先创建一个新按钮，在按钮中显示不同的值来区别，并将新创建的按钮添加到面板中。其中还有一个比较关键的 `show` 方法，使用该方法起到刷新的作用。读者可以自己动手来操作一下，这样就更能够了解流布局中控件的摆放。

## 15.2 网格布局

网格布局也是一种比较常见的布局管理器。使用网格布局管理器后，会将所有的控件尽量按照给出的行数和列数来排列，同时网格布局管理器也会对控件进行尺寸的调整，使所有的控件具有相同的尺寸。在网格布局中，也会尽量使使用的空间成矩形的形式来显示。当窗体发生大小变化时，所有的空间也将自动改变大小来填充窗体。

### 15.2.1 网格布局介绍

网格布局是通过 `GridLayout` 类来创建的。`GridLayout` 类具有三个构造器，使用无参构造器将创建具有默认行和默认列的网格布局。在创建网格布局管理器时最常用的就是具有两个整型参数的构造器，第一个参数表示网格布局管理器的行数，第二个参数表示网格布局管理器的列数。还有一个具有 4 个参数的构造器，除了可以定义行数和列数外，还可以定义控件间水平间距和垂直间距。

**提示：**在创建网格布局管理器时最常用的就是具有两个整型参数的构造器，第一个参数表示网格布局管理器的行数，第二个参数表示网格布局管理器的列数。

`GridLayout` 类中还定义了一些方法来对创建的网格布局进行操作。`getRows` 方法和 `setRows` 方法分别是获取和设置网格布局的行数。`getColumns` 方法和 `setColumns` 方法分别是获取和设置网格布局的列数。`getHgap` 方法和 `setHgap` 方法分别是获取和设置网格布局中控件间水平间距。`getVgap` 方法和 `setVgap` 方法分别是获取和设置网格布局中的控件间垂直间距。

### 15.2.2 使用网格布局

学习完如何创建网格布局后，就可以自己动手来使用网格布局。由于只学过按钮控件，所以这里还是使用按钮来演示网格布局管理器。



**【范例 15-3】**示例代码 15-3 是一个使用网格布局的程序。

示例代码 15-3

```

01 import javax.swing.*; //导入 Swing 包
02 import java.awt.event.*; //导入事件包
03 import java.awt.*;
04 //继承 JFrame 类
05 public class BuJu3 extends JFrame
06 {
07     JButton jb1=new JButton("第一个按钮"); //创建第一个按钮
08     JButton jb2=new JButton("第二个按钮"); //创建第二个按钮
09     JButton jb3=new JButton("第三个按钮"); //创建第三个按钮
10     JButton jb4=new JButton("第四个按钮"); //创建第四个按钮
11     JButton jb5=new JButton("第五个按钮"); //创建第五个按钮
12     JPanel jp=new JPanel(); //创建一个面板
13     //定义构造器
14     public BuJu3()
15     {
16         this.setTitle("使用网格布局管理器"); //设置窗口名称
17         jp.setLayout(new GridLayout(3,2)); //设置面板的布局为网格布局
18         jp.add(jb1); //将按钮添加到面板中
19         jp.add(jb2);
20         jp.add(jb3);
21         jp.add(jb4);
22         jp.add(jb5);
23         this.add(jp); //将面板添加到窗口中
24         this.setBounds(300,250,300,200); //设置窗口的大小和位置
25         this.setVisible(true); //设置窗口是可见的
26     }
27     public static void main(String args[])
28     {
29         BuJu3 a=new BuJu3();
30     }
31 }

```

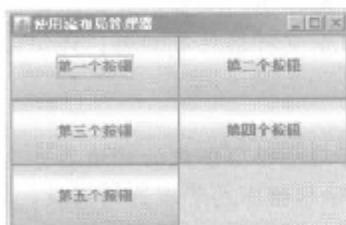


图 15-4 使用网格布局

**【运行结果】**使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 15-4 所示。

**【代码解析】**在示例代码 15-3 中将面板的布局管理器设置为网格布局, 并将网格布局设置为三行两列, 从而出现如图 15-4 的运行结果。如果在定义网格布局管理器时改变行数或列数, 从而就会改变运行结果。

**【范例 15-4】**为了更好地说明网格布局, 示例代码 15-4 就是一个让事件和网格布局相结合的程序。

示例代码 15-4

```

01 import javax.swing.*; //导入 Swing 包
02 import java.awt.event.*; //导入事件包
03 import java.awt.*;
04 //继承 JFrame 类
05 public class BuJu4 extends JFrame implements ActionListener
06 {
07     int i;

```

```

08     JButton jb=new JButton("创建按钮");           //创建按钮
09     JPanel jp=new JPanel();                      //创建一个面板
10     //定义构造器
11     public BuJu4()
12     {
13         this.setTitle("使用网格布局管理器");        //设置窗口名称
14         jp.setLayout(new GridLayout(3,2));           //设置面板的布局为网格布局
15         jp.add(jb);                                //将按钮添加到面板中
16         jb.addActionListener(this);                 //为控件注册监听器
17         this.add(jp);                                //将面板添加到窗口中
18         this.setBounds(300,250,300,200);           //设置窗口的大小和位置
19         this.setVisible(true);                     //设置窗口是可见的
20     }
21     public void actionPerformed(ActionEvent e)      //实现监听接口方法
22     {
23         ++i;                                     //为变量加 1
24         JButton jbi=new JButton("按钮"+i);        //创建新按钮
25         jp.add(jbi);                            //将新按钮添加到面板中
26         this.show(true);                         //执行后刷新
27     }
28     public static void main(String args[])
29     {
30         BuJu4 s=new BuJu4();
31     }
32 }

```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序，然后使用 Java 运行编译产生的 class 程序，运行结果如图 15-5 所示。

多次单击按钮后将出现如下几个运行结果，如图 15-6、图 15-7、图 15-8 所示。

【代码解析】从运行结果中可以看出，首先会一个控件占一行，将要添加的控件添加到该控件的所在列中。因为规定的为三行，所以当有 4 个控件时将新创建一列，但是不会出现第一列三个控件，第二列一个控件的情况，而是将所有的控件尽量以矩形的形式排列。当控件多时，同样也是按照这个规律。

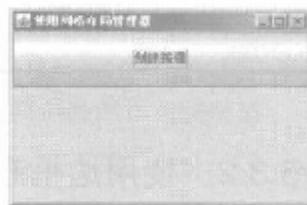


图 15-5 网格布局 1

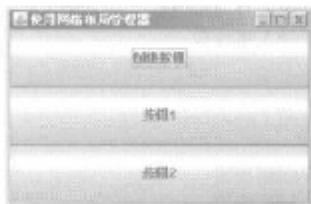


图 15-6 网格布局 2



图 15-7 网格布局 3

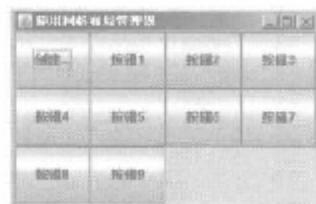


图 15-8 网格布局 4



### 15.3 边框布局

前面学习的流布局和网格布局具有很多相似的地方，但是边框布局就和它们存在很大的不同。在使用边框布局时，通常都会由程序员来为控件指定在容器中的位置。边框布局



将容器分为 5 个部分，包括东南西北中 5 部分。在每一个部分中只能放置一个控件，所以如果控件超过 5 个将不能完全显示。在使用边框布局时需要注意的是，当容器的大小发生变化时，四周的控件是不会发生变化的，只有中间的控件将发生变化。

### 15.3.1 边框布局介绍

边框布局是通过 `BorderLayout` 类创建的。`BorderLayout` 类具有两个构造器，一个是无参构造器，另一个是指定控件间间距的构造器，通常使用无参构造器来创建边框布局管理器。

在前面将控件添加到容器中都是通过 `add` 方法，将控件作为 `add` 方法的参数来进行添加的。但是在向边框布局容器中添加控件时，这样是不完全的。在向边框布局容器中添加控件是使用具有两个参数的 `add` 方法。其中第一个参数表示要添加的控件，第二个参数表示要添加到边框布局中的哪一个位置。边框布局的位置表示是通过常量来表示的，具体常量如表 15-2 所示。

表 15-2 边框布局位置

位 置	说 明
NORTH	容器的顶部
SOUTH	容器的底部
EAST	容器的左边
WEST	容器的右边
CENTER	容器的中间

在将控件添加到边框布局管理器时，就可以使用这些常量将控件添加到指定的位置。

### 15.3.2 使用边框布局

学习完如何创建边框布局后，读者就可以自己动手来使用边框布局。这里由于只学过按钮控件，所以还是使用按钮来演示边框布局管理器。

【范例 15-5】示例代码 15-5 是一个使用边框布局的程序。

示例代码 15-5

```
01 import javax.swing.*; //导入 Swing 包
02 import java.awt.event.*; //导入事件包
03 import java.awt.*;
04 //继承 JFrame 类
05 public class BuJu5 extends JFrame implements ActionListener
06 {
07     JButton jb1=new JButton("北边"); //创建按钮
08     JButton jb2=new JButton("南边"); //创建按钮
09     JButton jb3=new JButton("东边"); //创建按钮
10     JButton jb4=new JButton("西边"); //创建按钮
11     JLabel jl=new JLabel("进行按钮操作"); //创建标签
12     JPanel jp=new JPanel(); //创建一个面板
13     //定义构造器
14     public BuJu5()
15     {
16         this.setTitle("使用边框布局管理器"); //设置窗口名称
17         jp.setLayout(new BorderLayout()); //设置面板的布局为边框布局
```



```

18         jb1.addActionListener(this);           //为按钮注册监听器
19         jb2.addActionListener(this);
20         jb3.addActionListener(this);
21         jb4.addActionListener(this);
22         jp.add(jb1, BorderLayout.NORTH);      //将按钮添加到面板中的指定位置
23         jp.add(jb2, BorderLayout.SOUTH);
24         jp.add(jb3, BorderLayout.EAST);
25         jp.add(jb4, BorderLayout.WEST);
26         jl.add(jl, BorderLayout.CENTER);
27         this.add(jp);                      //将面板添加到窗口中
28         this.setBounds(300, 250, 300, 200);    //设置窗口的大小和位置
29         this.setVisible(true);              //设置窗口是可见的
30     }
31     public void actionPerformed(ActionEvent e) //实现监听接口方法
32     {
33         if(e.getSource()==jb1)              //如果事件源为第一个按钮
34         {
35             jl.setText("北边");           //设置标签显示内容
36         }
37         else if(e.getSource()==jb2)
38         {
39             jl.setText("南边");
40         }
41         else if(e.getSource()==jb3)
42         {
43             jl.setText("东边");
44         }
45         else if(e.getSource()==jb4)
46         {
47             jl.setText("西边");
48         }
49     }
50     public static void main(String args[])
51     {
52         BuJu5 s=new BuJu5();
53     }
54 }

```

【运行结果】使用 javac 编译程序将产生一个和该程序对应的 class 程序, 然后使用 Java 运行编译产生的 class 程序, 运行结果如图 15-9 所示。

在图 15-9 的运行结果中, 任意单击一个按钮都会改变中间标签所显示的内容。例如单击上面的按钮, 运行结果如图 15-10 所示。



图 15-9 边框布局



图 15-10 按下上边按钮后

【代码解析】在该程序中使用到了将一个监听器注册给多个事件源的知识。在该程序中定义了 4 个按钮和一个标签, 并将这些都添加到面板中, 在添加时为它们指定位置, 将