第 5 章　面向对象编程

除了基本的类和对象，Java 语言的面向对象编程还包括抽象类、接口、内部类及包等高级特性。通过对这些高级特性的支持，Java 语言全面实现了面向对象的编程。本章将进一步深入介绍 Java 语言面向对象编程方面的知识。

5.1　类的三大特性

第 4 章介绍了有关类定义的基本知识。面向对象中的类还具有封装、继承与多态的特性，这也是面向对象编程所具有的重要特点。Java 语言对类的封装、继承及多态均有很好地体现，下面分别进行介绍。

5.1.1　隐藏细节——封装

封装是指隐藏对象的属性及实现细节，对外仅提供接口可见。封装实现了信息隐藏，有利于软件复用。这种技术带来的好处是达到了模块化的标准，从而提高了代码的复用程度。在某种程度上，封装也大大改善了软件开发的可维护性，降低了构建复杂软件系统的风险。

在 Java 语言中，对象的属性和方法均可以进行访问控制。使用前面介绍的访问控制修饰符 public、private、protected 和 default，可以实现不同程度的信息封装。下面通过一个具体的示例来介绍 Java 语言中封装的实现。

【示例 5-1】 定义一个电视机类 TV，其代码如下所示。

```
01  public class TV() {

02      private int volume;
03      private int color;
04
05
06      public int getVolume() {
07          return volume;
08      }
09
10
11      public void setVolume(int volume)
12          this.volume = volume;
13      }
14
15
16      public int getColor() {
```

类		TV
属性	音量	volume
	颜色	color
方法	查看音量	getVolume()
	设置音量	setVolume ()
	查看颜色	getColor()
	设置颜色	setColor()
	开机	turnOn()
	关机	turnOff()

```
17          return color;
18      }
19
20
21      public void setColor(int color) {
22          this.color = color;
23      }
24
25
26      public void turnOn() {
27          turnOn_impl();              //调用内部开机实现
28      }
29
30
31      public void turnOff() {
32          turnOff_impl();             //调用内部关机实现
33      }
34
35      //开机内部实现
36      private void turnOn_impl() {
37          //...
38          //电视机内部部件实现开机
39      }
40
41      //关机内部实现
42      private void turnOff_impl() {
43          //...
44          //电视机内部部件实现关机
45      }
46
47  }
```

分析：上面代码定义了一个电视机类 TV，包含音量 volume 和颜色 color 两个属性。还定义了一些方法，包括音量、颜色的查看、设置方法，电视机的开、关方法及开、关的内部实现方法。

从类的属性可以看到，音量 volume 和颜色 color 属性均定义成 private 私有类型，只能通过相关方法对其值进行操作。类 TV 对其属性 volume 和 color 实现了封装，只能通过外部的具体动作影响其值。在某些情况下，比如想限制用户读取其值，只要将对应的 get 方法也设置成 private 即可。

类 TV 的方法中开机、关机的内部实现也被封装成 private 方法，不允许用户直接调用。用户在外面只能看到类 TV 提供的公开开机、关机方法，不知道具体的实现细节。在程序中内部实现方法也略去了具体实现，在现实过程中这将是一个很复杂的过程，需要调用多个零部件联合完成。

在实际编程过程中，具体哪个对象的哪些属性、方法需要封装，封装到什么力度，需要视具体情况分析而定。而这种分析也已经上升为设计领域，需要读者对面向对象思想有更深的领悟及更多的相关领域经验。

5.1.2　变相“抄袭”——继承

继承是指一个类是从另一个类派生而来的，派生类自动具有了被继承类的所有特性。

其中被继承的类称为“父类”，继承父类的类称为“子类”。继承可以使子类具有父类的各种属性和方法，而不需要再次编写相同的代码。继承也极大地促进了软件的复用，而且继承的形式可以更加复杂。Java 语言中类继承的语法格式如图 5.1 所示。

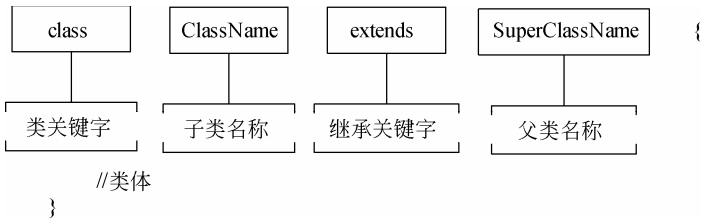
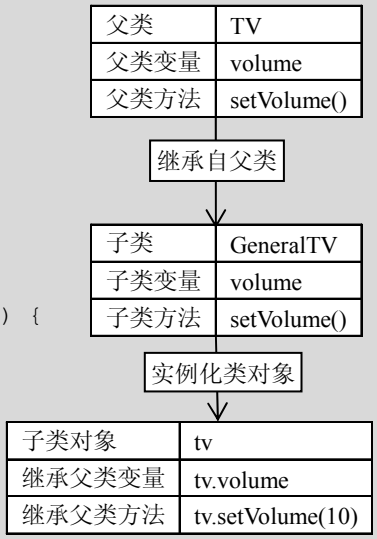


图 5.1　继承语法图

【示例 5-2】 对电视机类实现一个比较简单的继承，其代码如下所示。

```
01  class TV {
02      int volume = 1;
03
04
05      public void setVolume(int volume) {
06          this.volume = volume;
07      }
08  }
09
10  class GeneralTV extends TV {
11      public static void main(String args[]) {
12          GeneralTV tv = new GeneralTV ();
13
14          System.out.println(tv.volume);
15          tv.setVolume(10);
16          System.out.println(tv.volume);
17      }
18  }
```



程序执行结果如下：

```
1
10
```

分析：以上代码实现了一个比较简单的继承。首先定义了类 TV，代表各种电视的基类。类 TV 包含 default 类型的属性 volume 和 public 类型的方法 setVolume()。然后定义了类 GeneralTV，代表黑白电视。类 GeneralTV 继承自类 TV，作为类 TV 的子类。类 GeneralTV 没有定义自己的任何属性，方法也仅包含一个 main()方法。main()方法中创建了类 GeneralTV 的一个对象 tv，首先打印 tv 的属性 volume（该属性继承自类 TV），打印结果为 1。然后，调用继承自类 TV 的方法 setVolume()。最后再次打印 tv 的属性 volume，打印结果为 10。

注意：子类无法继承父类中使用 private 修饰符修饰的类成员。前面章节也讲过类中限定为 private 的成员只能被这个类本身访问。比如示例 5-2 中，如果属性 volume 被定义为 private，则语句 tv.volume 会报错。

子类除了继承父类的属性和方法外，还可以定义自己的属性和方法，这被称为类扩展。扩展的子类既具有父类中普遍通用的功能特性，还具有自身独特的属性和方法，从而具有更加完备的功能。

【示例 5-3】 在继承电视机类的同时，实现类的扩展，其代码如下所示。

```
01  class TV {
02      int volume = 1;
03
04      //定义 public 方法 setVolume
05      public void setVolume(int volume) {
06          this.volume = volume;
07      }
08  }
09
10  class ColourTV extends TV {
11      private int color;                    //定义 private 属性 color
12
13      public static void main(String args[]) {

14          ColourTV tv = new ColourTV();
15
16          System.out.println(tv.volume);
17          tv.setVolume(10);
18
19          System.out.println(tv.volume);
20
21          System.out.println(tv.color);
22          tv.setColor(5);
23          System.out.println(tv.getColor());
24      }
25
26      //得到属性 color
27      public int getColor() {
28          return color;
29      }
30
31      //设置属性 color
32      public void setColor(int color) {
33          this.color = color;
34      }
35  }
```

子类对象	tv
子类自身属性	color
子类自身方法	getColor()
	setColor(int color)
继承父类变量	tv.volume
继承父类方法	tv.setVolume(10)
调用自身属性	tv.color
调用自身方法	tv.setColor(5)

程序执行结果如下：

```
1
10
0
5
```

分析：以上代码在继承的同时，实现了类的扩展。同示例 5-2 一样，首先定义了类 TV，代表各种电视的基类。类 TV 包含 default 类型的属性 volume 和 public 类型的方法 setVolume。

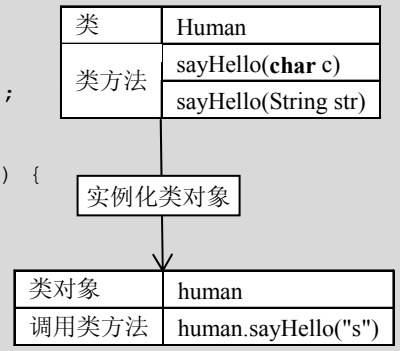
然后定义了类 ColourTV，代表彩色电视。类 ColourTV 同样继承了父类 TV 的属性 volume()和 setVolume()方法。除了继承父类外，类 ColourTV 还定义了自己的属性 color 及相应的获取、设置方法 getColor()、setColor()。Main()方法中先创建了类 ColourTV 的一个对象 tv。然后也是调用相关父类的属性和方法，并进行打印。下面一组代码则调用类 ColourTV 自身的属性和方法，并打印相关结果。

5.1.3　灵活应对——多态

多态是指对于一种服务功能，可以具有不同的实现方式，即多种形态。多态形象地描述了现实世界的复杂形态，使程序具有良好的扩展性。在继承的基础上，方法重载是实现多态的方式之一。

【示例 5-4】 用方法重载实现不同国家的人多态的问候方式的应用举例，其代码如下所示。

```
01  class Human {
02
03
04          public void sayHello(char c) {
05              System.out.println("Hello!");
06          }
07
08          public void  sayHello(String str) {
09              System.out.println("您好！");
10          }
11
12  }
13
14  public class TeatClass {
15
16      public static void main(Stringargs[])
17  {
18          Human human = new Human();
19          human.sayHello("s");
20      }
```



程序执行结果如下：

```
您好！
```

分析：以上代码首先定义了类 Human，还有一个 public 方法 sayHello(char c)。然后重载了 sayHello 方法——sayHello (String str)。在测试类 TestClass 中，创建 Human 对象，并调用了 sayHello (String str)方法，打印的结果为“您好！”。

说明：对于上述示例，父类更好的实现方式是采用抽象类或接口。有关抽象类或接口的概念在 5.2 节会详细介绍。

5.2　抽象类和接口

抽象类是 Java 语言中一种特殊的类，其中包含只进行了声明没有具体实现的抽象方

法。而接口更像是一种特殊的抽象类，其中所有方法都只进行了声明没有具体实现。抽象类和接口有着相似之处，不过也有许多区别。本节将分别对它们进行详细地介绍。

5.2.1　抽象类

抽象类是指包含了抽象方法的类。其中，抽象方法只声明方法名称，而不指明方法体。当需要表示对问题域进行分析、设计中得出的抽象概念时需要使用抽象类。抽象类比类的继承更高一层，是对类的抽象。抽象类拥有未实现的方法，不能被实例化，只能被继承扩展。通常 Java 语言中抽象类的语法格式如图 5.2 所示。

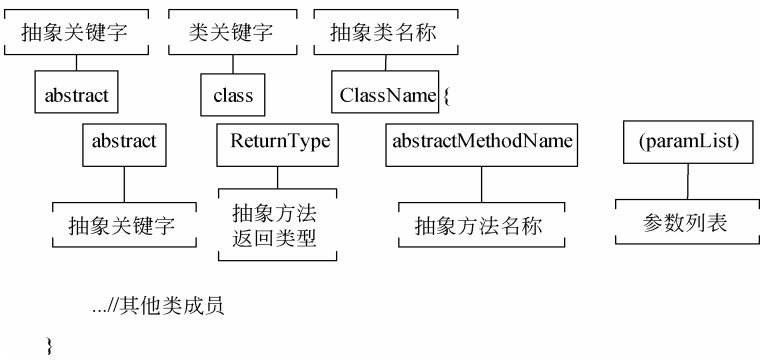
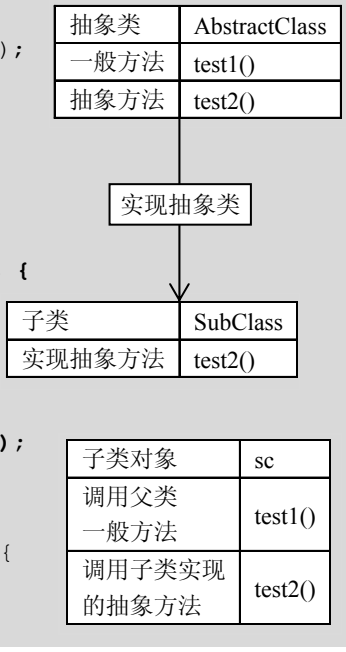


图 5.2　定义抽象类

注意：类体中至少包含一个抽象方法。抽象方法不含方法体，无需“{}”。

【示例 5-5】 定义一个简单的抽象类，其代码如下所示。

```
01  abstract class AbstractClass {
02
03      public void test1() {
04          System.out.println("this is test1!");
05      }
06
07      //定义抽象方法 test2，不含方法体
08      public abstract void test2();
09  }
10
11  public class SubClass extends AbstractClass {
12  //定义类 subClass，继承了抽象类 AbstractClass
13
14
15      //实现了抽象类中的 test2 方法
16      public void test2() {
17          System.out.println("this is test2!");
18      }
19
20      public static void main(String args[]) {
21          SubClass sc = new SubClass();
22
23          sc.test1();
```



```
24          sc.test2();
25      }
26  }
```

程序执行结果如下：

```
this is test1!
this is test2!
```

分析：此段代码首先定义了一个抽象类，类的名字为 AbstractClass。该类包含两个方法，一个为一般方法 test1()，另一个是抽象方法 test2()。需要特别注意的是，test1()方法包含方法体有具体的实现过程。test2()方法被 abstract 修饰没有方法体，只是定义了方法结构。然后定义了类 SubClass，该类继承了抽象类 AbstractClass。在类 SubClass 内，实现了抽象类 AbstractClass 中的抽象方法 test2()。最后，在 main()方法中创建了类 SubClass 的一个实例，分别调用了 test1()方法和 test2()方法。

通过语法定义和示例可以看出，抽象类有如下特点：

- 至少包含一个抽象方法。
- 不能被实例化。
- 继承抽象类的子类必须实现其所有抽象方法才能实例化，否则该子类还是抽象类。

在具体使用方面，抽象类能更准确地模拟对问题领域的理解。而且它可以提供实现的模式，使实现功能的代码更简单。抽象类一般用于关系密切的对象。其实在好多方面，抽象类和接口有着相似之处。下面介绍接口的概念，会与抽象类作一些比较。

5.2.2　接口

接口是面向对象编程中又一重要概念。在实际问题中，有些互不相关的类具有某种相同的行为，这时可以采用接口来定义这种行为。接口只定义提供什么功能，而不定义功能的具体实现，这一点与抽象类似。在 Java 语言中，不提供类的多继承，一个子类只能继承自一个父类。为了解决多继承的问题，在 Java 中可以采用接口来实现。一个类虽然不能同时继承多个类，却可以同时实现多个接口，从而可以解决现实中的多继承问题。有关接口的知识包括接口定义和接口实现，下面分别进行介绍。

1．接口定义

要使用接口，首先需要定义接口。接口定义的语法格式如图 5.3 所示。

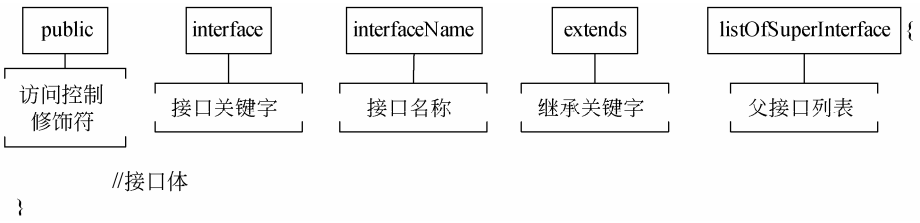


图 5.3　定义接口

一个接口可有多个父接口，用逗号隔开。另外在接口的接口体中，所有成员变量默认

为 public、static、final 类型。成员方法默认为 public 和 abstract 类型，只能包含常量及抽象方法。

【示例 5-6】 定义一个简单的接口，其代码如下所示。

```
01  public interface InterfaceExm {

02      public static final int X = 1;
03      public void test();
04  }
```

接口	InterfaceExm
接口变量	x
接口方法	test()

分析：以上代码定义了一个比较简单的接口，接口名为 InterfaceExm。接口 InterfaceExm 的接口体包含 int 型变量 X 和方法 test()。

与一般类、抽象类相比，接口在定义方面有如下特点：

- 接口体只能包含常量及抽象方法。
- 接口可以继承多个接口，而类不能多继承。
- 接口不能包含构造方法。

2．实现接口

接口定义只规定了接口对外提供的功能定义，没有对功能具体实现。要真正使用接口定义的功能，需要定义类实现接口。非抽象类实现接口主要是实现接口中定义所有抽象方法。实现接口的语法格式如图 5.4 所示。

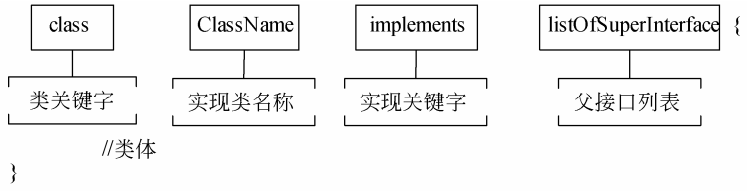
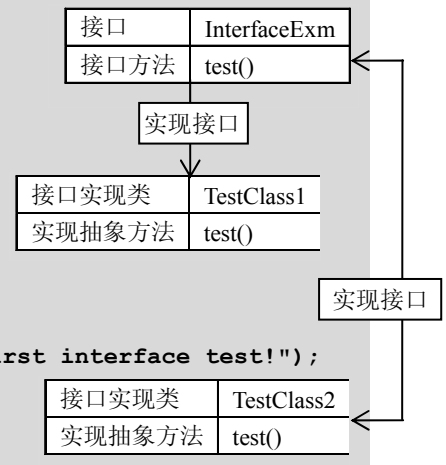


图 5.4　实现接口

一个类可以实现多个父接口，用逗号隔开。另外，在类的类体中，应该实现父接口中的所有抽象方法。

【示例 5-7】 接口实现的举例，其代码如下所示。

```
01  interface InterfaceExm {
02
03  public void test();
04  }
05
06  class TestClass1 implements
    InterfaceExm {
07
08      public void test() {
09          System.out.println("this is first interface test!");
10      }
11  }
12
```



```
13  class TestClass2 implements
    InterfaceExm {
14
15      public void test() {
16          System.out.println("this is
            second interface test!");
17      }
18  }
19
20  public class Test {
21      public static void main
        (String[] args) {
22          InterfaceExm ie;
23
24
25
26
27          ie = new TestClass1();
28
29          ie.test();
30
31
32
33
34          ie = new TestClass2();
35          ie.test();
36      }
37  }
```

类		Test
类变量	类型	InterfaceExm
	名称	ie

TestClass1 对象	ie
调用接口方法	ie.test()

TestClass2 对象	ie
调用接口方法	ie.test()

程序执行结果如下：

```
this is first interface test!
this is second interface test!
```

分析：以上代码首先定义了一个比较简单的接口，接口名为 InterfaceExm，其只包含一个抽象方法 test()。然后分别定义了两个接口实现类 TestClass1 和 TestClass2，这两个类都实现了接口 InterfaceExm。类 TestClass1 和 TestClass2 在类体内都分别实现了接口中的抽象方法 test()，实现的不同是打印不同的语句。最后定义了一个测试类 test，只包含一个 main()方法。main()方法中首先定义了接口 InterfaceExm 类型变量 ie，然后分别对变量 ie 赋予不同的值，分别为 TestClass1 对象和 TestClass2 对象，并同时调用接口 InterfaceExm 的抽象方法 test。程序执行结果为打印出不同的语句。

通过示例的执行结果可以看出，虽然调用的形式都是接口 InterfaceExm 的方法 test()，但由于来自不同的对象，具有不同的实现，所以执行的结果各不相同。这也是通常使用接口的一种方式。

最后总结一下，在实现接口方面有如下特点：

- 一个类可以同时实现多个接口。
- 接口不能被实例化，只能通过类实现接口的抽象方法。
- 虽然接口不能被实例化，但可以定义接口类型变量。
- 当类实现接口时，非抽象类必须实现所有抽象方法，否则为抽象类。

5.3　类中类——内部类

内部类指的是在类的内部嵌套定义的类。内部类本身的结构同普通类没什么不同，只不过定义的位置是在另一个类的内部。定义的内部类同样可以被外部引用。内部类可以只定义类体，没有名称，这就是匿名内部类。关于内部类的这些知识下面分别进行介绍。

5.3.1　内部类

根据内部类的概念，内部类定义的位置是在类的内部。在一个类的任意代码块内，均可以进行内部类的定义。如可以直接作为类成员定义，可以定义在类的成员方法内及循环体内等。

【示例 5-8】 定义一个简单内部的类，其代码如下所示。

```
01  public class Outer {
02      private int x = 1;
03
04
05      void sayHello() {
06          System.out.println("Hello!");
07      }
08
09      class Inner {
10          int y = 0;
11
12          void test() {
13              System.out.println(x);
14              sayHello();
15          }
16      }
17  }
```

外部类	Outer
类变量	x
类方法	sayHello()

内部类	inner
类变量	y
类方法	test()

分析：首先程序最外层定义了外部类 Outer，类 Outer 包含成员变量 x 和成员方法 sayHello()。然后，作为类 Outer 的成员，定义了内部类 Inner。内部类 Inner 有自己的成员变量 y，自己的成员方法 test()。在内部类 Inner 的成员方法 test()中，第一行代码打印了外部类 Outer 的成员变量 x 的值，第二行代码调用了外部类 Outer 的成员方法。

通过示例可以看出，内部类可以任意访问外部类中的成员。在其他代码块中定义内部类与示例类似，这里不再举例说明。

需要特别说明的是，内部类可以被修饰为 static 属性，这在普通类是不行的。修饰为 static 的内部类相当于提升了类的层次，变成了外部类，当然也不能再访问外部类的非 static 成员。

【示例 5-9】 定义一个 static 内部类，其代码如下所示。

```
01  public class Outer {
02      static class Inner {
            //定义内部类 static 成员变量 x
03          static int x = 0;
```

```
04
05
06          //定义内部类 static 成员方法 test
07          static void test() {
08              System.out.println("test");
09          }
10      }
11
12      public static void main(String
        args[]) {
13          System.out.println(Outer.Inner.x);
14
15          Outer.Inner.test();
16
17  }
18}
```

内部类	inner
类变量	x
类方法	test()

外部类	outer
调用内部类变量	Outer.Inner.x
调用内部类方法	Outer.Inner.test()

程序执行结果如下：

```
0
test
```

分析：同样，程序最外层定义了外部类 Outer。然后定义了内部类 Inner，并且被修饰为 static 属性。内部类 Inner 具有 static 类型成员变量 y，static 成员方法 test()。在外部类 Outer 中定义了 main()方法。main()方法第一行代码打印了内部类 Inner 的成员变量 x 的值，第二行代码调用了内部类 Inner 的成员方法 test()。

注意：static 内部类可以包含 static 成员，而非 static 内部类却不可以包含 static 成员。

5.3.2 “更隐蔽”的内部类——匿名内部类

匿名内部类是一种没有类名的特殊内部类。就像仅使用一次的变量可以不定义名称一样，匿名内部类也是用作定义仅使用一次的类。匿名内部类通常具有特殊的用途，也能使程序显得更加简洁。

【示例 5-10】 使用匿名内部类实现“Say Hello”功能的应用举例，其代码如下所示。

```
01  abstract class SayHello {
02
03      public abstract void hello();
04  }
05
06  public class Test {
07
08      //定义方法 hello，参数为抽象类 SayHello 类型
09      public void hello(SayHello sh) {
10          sh.hello();
11      }
12
13      //定义 main 方法
14      public static void main(String args[]) {
15          Test t = new Test();
```

抽象类	sayHello
抽象方法	hello()

类	Test
类方法	hello()

实例化类对象

类对象	t
调用类方法	t.hello()

```
16                                                  匿名内部类        new SayHello()
                                                    实现抽象类方法    hello()
17          //调用 Test 的方法 hello，参数为匿名内部类
18          t.hello(new SayHello() {
                //实现了抽象类 SayHello 的方法 hello
19              public void hello() {
20
21
22                  System.out.println("Hello!");
23              }
24          }
25          );
26  }
27}
```

程序执行结果如下：

```
Hello!
```

分析：以上代码首先定义了一个抽象类 SayHello，包含一个抽象方法 hello()。然后定义类 Test，类 Test 包含成员方法 hello()。Test 的成员方法 hello()参数为抽象类 SayHello 类型，方法体调用参数变量 sh 的 hello()方法。最后，类 Test 定义了 main()方法。main()方法体首先创建了类 Test 的对象 t，然后调用了 t 的方法 hello()。关键就在这里，调用 t 的方法 hello()时，参数为一段代码，这段代码其实就定义了一个匿名内部类。匿名内部类没有类名，直接用 new 关键字创建对象，后面{}部分为匿名内部类的类体。

通过示例可以看出，使用匿名内部类可以不用定义类名，直接创建类的对象。匿名内部类适用于为该类只创建一个对象的场合，在后续章节将介绍的图形界面部分会经常使用匿名内部类。对于初学者来说，刚接触匿名内部类会比较困惑。要解决这个问题，需要多结合具体程序实例，多多练习、揣摩，从而能够加深理解，对其很好地掌握。

5.4　类的仓库——包

在面向对象编程中，包是 Java 语言所特有的概念。包用来将相关的一组类组织在一起进行管理，类似操作系统中的文件管理。包在外在形式上就是操作系统中的文件夹，包的内容就是类文件。不同的包可以包含同名的类，包机制有效解决了命名冲突问题。有关包的知识包括创建包和使用包，下面将分别进行介绍。

5.4.1　创建包

在使用包之前，首先需要创建包。创建包的语法格式如下所示。

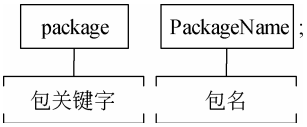


图 5.5　创建包

【示例 5-11】 创建包的实例，其代码如下所示。

```
01  package TestPackage;              //声明所属包为 TestPackage
02
03  public class Test {
04      //类体
05  }
```

分析：以上代码首先定义了一个包，包名为 TestPackage。然后定义了一个类 Test，该类所在的包即为 TestPackage。

包可以多个层次嵌套，也就是包内还可以有包。Java 语言中的包在外在形式上对应着操作系统中的目录结构。嵌套的包用“.”号分隔，对应于操作系统，也就是目录中还具有目录。比如“com.sun”，第一级包为 com，包 com 内包含第二级包 sun。在操作系统中，存在名为 com 的目录，com 目录下又具有名为 sun 的目录。

注意：在编写程序过程中，package 语句必须写在类文件的最上面。

5.4.2　使用包

创建了包之后，经常需要使用不同包中的类。使用不同包中的类主要具有两种形式：包名加类名形式和 import 导入形式。另外，在 Java SE 5.0 版本之后，还提供了静态导入功能。对这些内容，下面将分别进行介绍。

1．包名加类名

第一种形式采用包名加类名的方式，语法格式如图 5.6 所示。

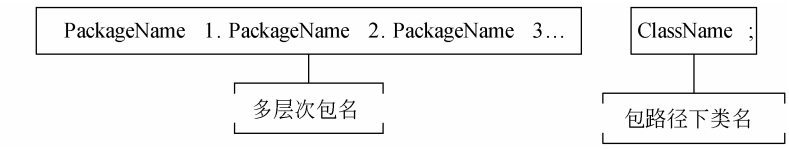


图 5.6　包名加类名使用包

其中，参数 PackageName1、PackageName2 和 PackageName3…表示多层次的包的调用，可以支持任意层次。

注意：此种形式的使用需要使用完整的包路径。完整的包路径就是从最顶层的包直到包含该类文件的最底层的包的完整路径结构。并且，所有引用该类的地方都要加完整的包路径。

【示例 5-12】 使用包中的类的第一种形式举例，其代码如下所示。

```
01  package TestPackage;
02
03  public class Test {
    //引用包路径“java.util.regex”下的类 Pattern
04      java.util.regex.Pattern p1;
05      //引用包路径“java.util.regex”下的类 Pattern
```

```
06      java.util.regex.Pattern p2;}
```

分析：以上代码定义了一个类 Test，该类属于包 TestPackage。Test 类定义了两个成员变量，变量类型为外部包“java.util.regex”下的 Pattern 类。可以注意到，虽然两个变量都是引用同一个类 Pattern，但是每个变量的类型引用前都要包含完整的包路径。

2．import导入

第二种形式采用 import 语句导入的方式，语法格式如图 5.7 所示。

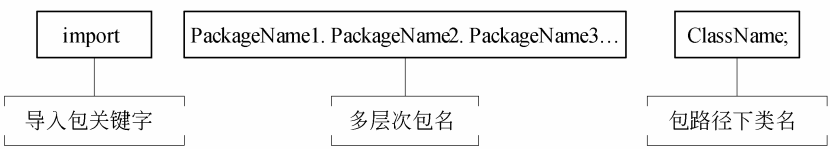


图 5.7　import 导入包

这种形式首先使用 import 语句将要使用的类所在的包导入，包为完整路径。导入包后，就可以在程序的任意位置引用包中的类，类名前不用再加第一种形式中的那种完整包路径。

【示例 5-13】 使用包中的类的第二种形式举例，其代码如下所示。

```
01  package TestPackage;
02
03  import java.util.regex.Pattern;              //导入包
04
05  public class Test {
06      Pattern p1;
07      Pattern p2;
08  }
```

分析：同形式一的示例一样，该示例定义了一个 Test 类，该类属于包 TestPackage。Test 类同样定义了两个成员变量，变量类型都为 Pattern。不同的是，变量类型 Pattern 前没有形式一示例中的完整的包名，Pattern 类所在的包在程序开始处通过 import 语句导入，使程序更加简洁。

相对于包名加类名的形式，实际编程过程中更多的是使用 import 导入形式。许多 Java 集成开发环境支持包导入的快捷键，采用 import 导入形式编写程序更方便和简洁。

技巧：如果要用到一个包中的多个类，可以采用“import PackageName1. PackageName2. PackageName3…*”的形式同时导入包中的所有类。

3．静态导入

静态导入是 Java SE 5.0 之后提供的功能，其语法格式如图 5.8 所示。

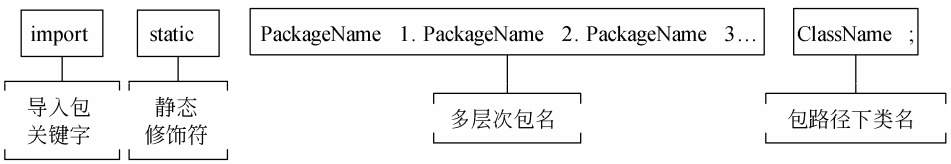


图 5.8　静态导入

静态导入与普通导入的区别就是在导入的对象之前增加了 static 修饰符。下面通过具体的实例说明静态导入的作用。

【示例 5-14】未采用静态导入，实现打印圆面积的实例，其代码如下所示。

```
01  import util.Constant;                                  //导入 Constant 类
02
03  public class Test {
04      public static void main(String args[]) {
05          //打印圆的面积
06          double r = 3.0;                                //半径
07          double zz = Constant.PI * r * r;               //面积
08
09          System.out.println(zz);
10      }
11
12  }
```

导入的 Constan 类的代码如下所示。

```
01  package util;
02
03  public class Constant {
04      public static final double PI = 3.14;             //定义常量 pai
05  }
```

程序执行结果如下：

```
28.259999999999998
```

分析：Test 类的 main()方法实现了打印圆面积的功能。其中，用到的常量 PI 在 util 包中的 Constant 类中定义。在 Test 类的头部导入了 Constant 类，使用常量 PI 的格式为“Constant.PI”。

上面的实例代码未采用静态导入的形式。下面采用静态导入的方式对原实例进行改造。改造后的代码如下所示。

```
01  import static util.Constant.PI;                       //静态导入 Constant 类
02
03  public class Test {
04      public static void main(String args[]) {
05          //打印圆的周长
06          double r = 3.0;                                //半径
07          double zz = PI * r * r;                        //周长
08
09          System.out.println(zz);
10      }
11
12  }
```

程序执行结果如下：

```
28.259999999999998
```

其中，Constant 类的代码没有变化。可以看出，在采用了静态导入之后，可以直接使用导入类中的变量，不需再采用“Constant.”的形式。对于使用导入类的方法的情况也是一样。

5.5　小　　结

本章进一步深入地介绍了 Java 语言中面向对象编程的相关知识，包括类的封装、继承、多态以及抽象类、接口、内部类和包等。本章最后给出了一个完整的实例。其中抽象类与接口的具体运用是本章的难点。抽象类与接口是面向对象编程中更加抽象的概念，要想能真正发挥它们的特点需要多结合具体需求反复实践。第 6 章将介绍 Java 语言中复合数据类型相关的内容，如数组、字符串、集合等。

5.6　习　　题

1．定义一个 Person 类，声明 id、name 和 age 成员变量，以及 getId()、getName()和 getAge()等成员方法。

【分析】本题主要考查读者对类的封装的掌握。

【核心代码】本题的关键代码如下所示。

```
public class Person {
    int id;
    String name;
    int age;

    public int getId(){
        return id;
    }

    public String getName(){
        return name;
    }

    public int getAge(){
        return age;
    }
}
```

2．定义 Person 类，声明 int 型变量 id，初始化为 7，声明方法 setId()。定义 Student 类继承于 Person 类，创建 student 对象，调用父类的 setId()方法，然调用父类的 id 变量，输出 set 后的 id 值。运行结果如下。

```
6
```

【分析】本题主要考查读者对类的继承的掌握。子类继承父类之后，就可以调用父类中不被声明为 private 的变量和方法。

【核心代码】本题的关键代码如下所示。

```
public class Student extends Person{

    public static void main(String[] args) {
        Student student = new Student();
```

```
        student.setId(6);
        System.out.println(student.id);
    }

}

class Person {

    int id = 7;

    public int setId(int id){
        this.id = id;
        return id;
    }

}
```

3．定义 Human 类，还有一个 public 方法 sayHello (String str)。然后重载了 sayHello 方法——sayHello(char c)。在测试类 TestClass 中，创建 Human 对象，并调用了 sayHello (char c) 方法，输出结果。运行结果如下。

```
Hello
```

【分析】本题主要考查读者对类的多态中方法重载的掌握。重载的方法具有相同的方法名，只是方法的参数类型不同。

【核心代码】本题的关键代码如下所示。

```
class Human {

        public void  sayHello(String str) {
            System.out.println("您好! ");
        }

        public void sayHello(char c) {
            System.out.println("Hello!");
        }
}

public class TeatClass {
    public static void main(String args[]) {
        Human human = new Human();
        human.sayHello('c');
    }
}
```