# 第5章 面向对象(下)

**本章学习目标**

- 理解继承的概念。
- 掌握 final 关键字的使用。
- 熟练掌握抽象类和接口的使用。
- 理解多态的概念。
- 掌握 JDK 8.0 中 Lambda 表达式的使用。

通过第 4 章的学习,相信读者对 Java 语言面向对象的基本知识已经有了初步了解,本章将介绍 Java 面向对象的另外两大特征:继承和多态。此外,本章还将介绍 final 关键字、抽象类和接口、包、访问控制。

## 5.1 类的继承

继承是面向对象的另一大特征,用于描述类的所属关系,多个类通过继承形成一个关系体系。继承是在原有类的基础上扩展新的功能,实现了代码的复用。

### 5.1.1 继承的概念

现实生活中,继承是指下一代人继承上一代人遗留的财产,即实现财产重用。在面向对象程序设计中,继承实现代码重用,即在已有类的基础上定义新的类,新的类能继承已有类的属性与行为,并扩展新的功能,而不需要把已有类的内容再写一遍。已有的类被称为父类或基类,新的类被称为子类或派生类。例如,交通工具与公交车就属于继承关系,公交车拥有交通工具的一切特性,但同时又拥有自己独有的特性。在 Java 中,子类继承父类的语法格式如下。

```
class 子类名 extends 父类名 {
      属性和方法
}
```

Java 使用 extends 关键字指明两个类之间的继承关系。子类继承了父类中的属性和方法,也可以添加新的属性和方法,如例 5-1 所示。

**例 5-1** TestExtends.java

```
1  // 定义父类
2  class Parent {
```

```
    String name;                        // 名称
    double property;                    // 财产
    public void say() {
        System.out.println(name + "的财产:" + property);
    }
}
// 定义子类继承自父类
class Child extends Parent {
    int age;
    public void sayAge() {
        System.out.println(name + "的年龄:" + age);
    }
}
public class TestExtends {
    public static void main(String[] args) {
        // 创建 Child 对象
        Child c = new Child();
        // Child 对象本身没有 name 成员变量
// 因为 Child 的父类有 name 成员变量,所以 Child 继承了父类的成员变量和方法
        c.name = "小明";
        c.property = 100;
        c.age = 20;
        c.say ();
        c.sayAge();
    }
}
```

程序的运行结果如图 5.1 所示。


图 5.1　例 5-1 运行结果

在例 5-1 中,Child 类通过 extends 关键字继承了 Parent 类,Child 类便是 Parent 的子类。从程序运行结果可发现,Child 类虽然没有定义 name、property 成员变量和 say()成员方法,但却能访问这些成员,说明子类可以继承父类所有的成员。另外,在 Child 类里还定义了一个 sayAge()方法,说明子类可以扩展父类功能。

Java 语言只支持单继承,不允许多重继承,即一个子类只能继承一个父类,否则会引起编译错误,具体示例如下。

```
class A {}
class B {}
class C extends A, B {}
```

Java 语言虽然不支持多重继承,但它支持多层继承,即一个类的父类可以继承另外的父类。因此,Java 类可以有无限多个间接父类,具体示例如下。

```
class A {}
class B extends A {}
class C extends B {}
```

## 5.1.2 重写父类方法

在继承关系中，子类从父类中继承了可访问的方法，但有时从父类继承下来的方法不能完全满足子类需要，例如例 5-1 中，如果要求父类与子类中的 say()方法输出不同内容，这时就需要在子类的方法里修改父类的方法，即子类重新定义从父类中继承的成员方法，这个过程称为方法重写或覆盖。在进行方法重写时必须考虑权限，即被子类重写的方法不能拥有比父类方法更加严格的访问权限，如例 5-2 所示。

**例 5-2** TestOverride.java

```
1  // 定义父类
2  class Parent {
3      protected void say() {
4          System.out.println("父辈");
5      }
6  }
7  // 定义子类继承自父类
8  class Child extends Parent {
9      public void say() {
10         System.out.println("子女");
11     }
12 }
13 public class TestOverride {
14     public static void main(String[] args) {
15         // 创建 Child 对象
16         Child c = new Child();
17         c.say();
18     }
19 }
```

程序的运行结果如图 5.2 所示。


图 5.2 例 5-2 运行结果

在例 5-2 中，Child 类继承了 Parent 类的 say()方法，但在子类 Child 中重新定义的 say()方法对父类的 say()进行了重写。从程序运行结果可发现，在调用 Child 类对象的 say()方法时，只会调用子类重写的方法，并不会调用父类的 say()方法。

另外，需要注意方法重载与方法重写的区别。

(1) 方法重载是在同一个类中，方法重写是在子类与父类中。

(2) 方法重载要求：方法名相同，参数个数或参数类型不同。

(3) 方法重写要求：子类与父类的方法名、返回值类型和参数列表相同。

### 5.1.3 super 关键字



当子类重写父类方法后，子类对象将无法访问父类被重写的方法。如果在子类中需要访问父类的被重写方法，可以通过 super 关键字来实现，其语法格式如下。

```
super.成员变量
super.成员方法([实参列表])
```

接下来演示 super 关键字的作用，如例 5-3 所示。

**例 5-3** TestSuper.java

```
1  // 定义父类
2  class Parent {
3      String name = "Parent";                    // 名称
4      public void say() {
5          System.out.println("父辈");
6      }
7  }
8  // 定义子类继承自父类
9  class Child extends Parent {
10     public void say() {
11         String name = super.name;              // 访问父类成员变量
12         super.say();                           // 访问父类成员方法
13         System.out.println("姓名:" + name);
14     }
15 }
16 public class TestSuper{
17     public static void main(String[] args) {
18         // 创建 Child 对象
19         Child c = new Child();
20         c.say();
21     }
22 }
```

程序的运行结果如图 5.3 所示。


图 5.3 例 5-3 运行结果

在例 5-3 中，Child 类继承自 Parent 类，并重写了 say()方法。在子类 Child 的 say()方法中，使用 super.name 调用了父类的成员变量，使用 super.say()调用了父类被重写的成员方法。从程序运行结果可发现，通过 super 关键字可以在子类中访问被隐藏的父类成员。

在继承中，实例化子类对象时，首先会调用父类的构造方法，再调用子类的构造方法，这

与实际生活中先有父母再有孩子类似。子类继承父类时，并没有继承父类的构造方法，但子类构造方法可以调用父类的构造方法。在一个构造方法中调用另一个重载的构造方法时应使用 this 关键字，在子类构造方法中调用父类的构造方法时应使用 super 关键字，其语法格式如下。

```
super([参数列表])
```

接下来演示 super 调用父类的构造方法，如例 5-4 所示。

**例 5-4** TestSuperRefConstructor.java

```
1  // 定义父类
2  class Parent {
3      String name;                                    // 名称
4      public Parent(String name) {
5          this.name = name;
6      }
7      public void say() {
8          System.out.println("父辈");
9      }
10 }
11 // 定义子类继承自父类
12 class Child extends Parent {
13     public Child() {
14         super("Parent");
15     }
16     public void say() {
17         System.out.println("姓名:" + name);
18     }
19 }
20 public class TestSuperRefConstructor {
21     public static void main(String[] args) {
22         // 创建 Child 对象
23         Child c = new Child();
24         c.say();
25     }
26 }
```

程序的运行结果如图 5.4 所示。


图 5.4 例 5-4 运行结果

从程序运行结果可发现，实例化 Child 类对象时调用了父类的有参构造方法。super 关键字调用构造方法和 this 关键字调用构造方法类似，该语句必须位于子类构造方法的第一行，否则会编译出错。

另外,子类中如果没有显式地调用父类的构造方法,那么将自动调用父类中不带参数的构造方法,如例 5-5 所示。

**例 5-5** TestAutoCallConstructor.java

```
1  // 定义父类
2  class Parent {
3      String name;
4      public Parent(String name) {
5          this.name = name;
6          System.out.println("Parent(String name)");
7      }
8  }
9  // 定义子类继承自父类
10 class Child extends Parent {
11     public Child() {
12         System.out.println("Child()");
13     }
14     public void say() {
15         System.out.println("姓名:" + name);
16     }
17 }
18 public class TestAutoCallConstructor {
19     public static void main(String[] args) {
20         // 创建 Child 对象
21         Child c = new Child();
22     }
23 }
```

程序的运行结果如图 5.5 所示。

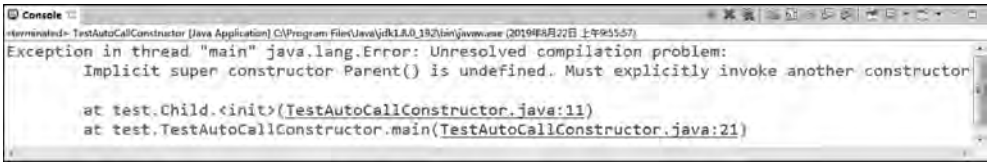


图 5.5 例 5-5 运行结果

在图 5.5 中,程序编译结果报错,原因是在 Child 类的构造方法中没有显式地调用父类构造方法,便会默认调用父类无参构造方法,而父类 Parent 中显式定义了有参构造方法,此时编译器将不再自动生成默认构造方法。因此,程序找不到无参构造方法而报错。

为了解决上述程序的编译错误,可以在子类中显式地调用父类中定义的构造方法,也可以在父类中显式定义无参构造方法,如例 5-6 所示。

**例 5-6** TestConstructorOrder.java

```
1  // 定义父类
2  class Parent {
3      String name;
4      // 定义无参构造方法
```

```
5       public Parent() {
6           System.out.println("Parent()");
7       }
8       public Parent(String name) {
9           this.name = name;
10          System.out.println("Parent(String name)");
11      }
12  }
13  // 定义子类继承自父类
14  class Child extends Parent {
15      public Child() {
16          System.out.println("Child()");
17      }
18      public void say() {
19          System.out.println("姓名:" + name);
20      }
21  }
22  public class TestConstructorOrder {
23      public static void main(String[] args) {
24          // 创建 Child 对象
25          Child c = new Child();
26      }
27  }
```

程序的运行结果如图 5.6 所示。


图 5.6　例 5-6 运行结果

在图 5.6 中，从程序运行结果可发现，子类在实例化时默认调用父类的无参构造方法，并且父类的构造方法在子类构造方法之前执行。

## 5.2　final 关键字

在 Java 中，为了考虑安全因素，要求某些类不允许被继承或不允许被子类修改，这时可以用 final 关键字修饰。它可用于修饰类、方法和变量，其具体特点如下。

（1）final 修饰的类不能被继承。

（2）final 修饰的方法不能被子类重写。

（3）final 修饰的变量是常量，初始化后不能再修改。

### 5.2.1　final 关键字修饰类

使用 final 关键字修饰的类称为最终类，表示不能再被其他的类继承，如 Java 中的 String 类。接下来演示 final 修饰类，如例 5-7 所示。

**例 5-7** TestFinalClass.java

```
1  // 使用 final 关键字修饰类
2  final class Parent {
3  }
4  // 继承 final 类
5  class Child extends Parent {
6  }
7  public class TestFinalClass {
8      public static void main(String[] args) {
9          // 创建 Child 对象
10         Child c = new Child();
11     }
12 }
```

程序的运行结果如图 5.7 所示。

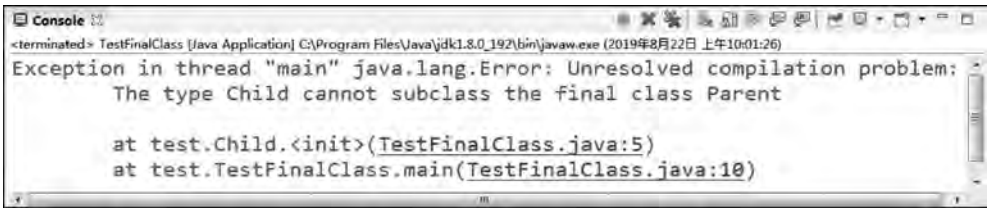


图 5.7 例 5-7 运行结果

在例 5-7 中,使用 final 关键字修饰了 Parent 类。因此,Child 类继承 Parent 类时,程序编译结果报错并提示“无法从最终 Parent 类进行继承”。由此可见,被 final 修饰的类为最终类,不能再被继承。

## 5.2.2 final 关键字修饰方法

使用 final 关键字修饰的方法,称为最终方法,表示子类不能重写此方法,接下来演示 final 修饰方法,如例 5-8 所示。

**例 5-8** TestFinalFunction.java

```
1  class Parent {
2      // final 关键字修饰方法
3      public final void say() {
4          System.out.println("final 修饰 say()方法");
5      }
6  }
7  class Child extends Parent {
8      // 重写父类方法
9      public void say() {
10         System.out.println("重写父类 say()方法");
11     }
12 }
13 public class TestFinalFunction {
```

```
14      public static void main(String[] args) {
15          // 创建 Child 对象
16          Child c = new Child();
17          c.say();
18      }
19  }
```

程序的运行结果如图 5.8 所示。

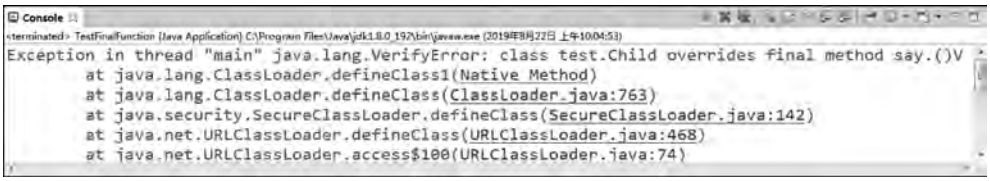


图 5.8　例 5-8 运行结果

在例 5-8 中，Parent 类中使用 final 关键字修饰了成员方法 say()，Child 类继承 Parent 类并重写了 say()方法。程序编译结果报错并提示被覆盖的方法为 final。由此可见，被 final 修饰的成员方法为最终方法，不能再被子类重写。

### 5.2.3　final 关键字修饰变量

使用 final 关键字修饰的变量，称为常量，只能被赋值一次。如果再次对该变量进行赋值，则程序在编译时会报错，如例 5-9 所示。

**例 5-9**　TestFinalLocalVar.java

```
1  package test;
2  public class TestFinalLocalVar {
3      public static void main(String[] args) {
4          final double PI = 3.14;                    // 定义并初始化
5          PI = 3.141592653;                          // 重新赋值
6      }
7  }
```

程序的运行结果如图 5.9 所示。

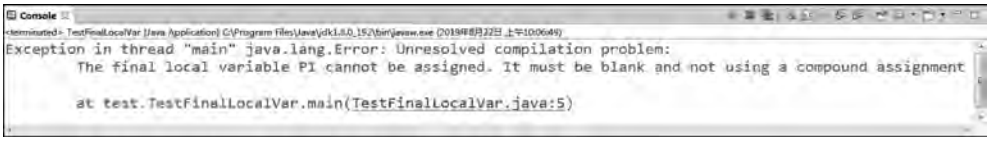


图 5.9　例 5-9 运行结果

在例 5-9 中，使用 final 修饰变量 PI，再次对其进行赋值，程序编译结果报错并提示“无法为最终变量 PI 分配值”。由此可见，final 修饰的变量为常量，只能初始化一次，初始化后不能再修改。

在例 5-9 中，使用 final 修饰的是局部变量，接下来使用 final 修饰成员变量，如例 5-10 所示。

**例 5-10** TestFinalMemberVar.java

```
1  class Parent {
2      // 使用 final 修饰成员变量
3      final double PI;
4      public void say() {
5          System.out.println(this.PI);
6      }
7  }
8  class Child extends Parent {
9  }
10 public class TestFinalMemberVar {
11     public static void main(String[] args) {
12         // 创建 Child 对象
13         Child c = new Child();
14         c.say();
15     }
16 }
```

程序的运行结果如图 5.10 所示。


图 5.10 例 5-10 运行结果

在例 5-10 中,Parent 类中使用 final 修饰了成员变量 PI,程序编译结果报错并提示可能尚未初始化变量 PI。由此可见,Java 虚拟机不会为 final 修饰的变量默认初始化。因此,使用 final 修饰成员变量时,需要在声明时立即初始化,或者在构造方法中进行初始化。下面使用构造方法初始化 final 修饰的成员变量,在 Parent 类中添加代码,具体如下。

```
public Parent() {                    // 构造方法中初始化 final 成员变量
    PI = 3.14;
}
```

程序的运行结果如图 5.11 所示。


图 5.11 例 5-10 修改后运行结果

此外,final 关键字还可以修饰引用变量,表示该变量只能始终引用一个对象,但可以改变对象的内容,有兴趣的读者可以动手验证一下。

# 5.3 抽象类和接口

## 5.3.1 抽象类

Java 中可以定义不含方法体的方法，方法的方法体由该类的子类根据实际需求去实现，这样的方法称为抽象方法，包含抽象方法的类必须是抽象类。

Java 中提供了 abstract 关键字，表示抽象的意思，用 abstract 修饰的方法，称为抽象方法，是一个不完整的方法，只有方法的声明，没有方法体。用 abstract 修饰的类，称为抽象类，抽象类可以不包含任何抽象方法，具体示例如下。

```
// 用 abstract 修饰抽象类
abstract class Parent {
    // abstract 修饰抽象方法，只有声明，没有实现
    public abstract void say();
}
```

使用抽象类时需要注意，抽象类不能被实例化，即不能用 new 关键字创建对象，因为抽象类中可包含抽象方法，抽象方法只有声明没有方法体，不能被调用。因此，必须通过子类继承抽象类去实现抽象方法，如例 5-11 所示。

**例 5-11** TestAbstractClass.java

```
1  // 用 abstract 修饰抽象类
2  abstract class Parent {
3          // abstract 修饰抽象方法，只有声明，没有实现
4          public abstract void say();
5  }
6  // 继承抽象类
7  class Child extends Parent {
8      // 实现抽象方法
9      public void say() {
10         System.out.println("Child");
11     }
12 }
13 public class TestAbstractClass {
14     public static void main(String[] args) {
15         Child c = new Child();
16         c.say();
17     }
18 }
```

程序的运行结果如图 5.12 所示。


图 5.12 例 5-11 运行结果

在例 5-11 中,子类定义时实现了抽象方法,因此在主方法实例化子类对象后,子类对象可以调用子类中实现的抽象方法。

需要注意的是,具体子类必须实现抽象父类中的所有抽象方法,否则子类必须要声明为抽象类,如例 5-12 所示。

**例 5-12** TestAbstractFun.java

```
1  // 用 abstract 修饰抽象类
2  abstract class Parent {
3          // abstract 修饰抽象方法,只有声明,没有实现
4          public abstract void say();
5          public abstract void work();
6  }
7  // 继承抽象
8  class Child extends Parent {
9      // 实现抽象方法
10     public void say() {
11         System.out.println("Child");
12     }
13 }
14 public class TestAbstractFun {
15     public static void main(String[] args) {
16         Child c = new Child();
17         c.say();
18     }
19 }
```

程序的运行结果如图 5.13 所示。


图 5.13 例 5-12 运行结果

在例 5-12 中,子类 Child 中只实现了抽象类 Parent 的 say()抽象方法,而未实现 work()抽象方法。程序编译结果报错并提示“Child 不是抽象的,并且未覆盖 Parent 中的抽象方法 work()”。错误的原因在于:子类继承了抽象类的 work()抽象方法,该方法没有方法体,不能被实例化。因此,子类必须实现抽象类的全部抽象方法,否则子类必须声明为抽象类。

另外,抽象方法不能用 static 来修饰,因为 static 修饰的方法可以通过类名调用,调用时将调用一个没有方法体的方法,肯定会出错;抽象方法也不能用 final 关键字修饰,因为被 final 关键字修饰的方法不能被重写,而抽象方法的实现需要在子类中实现;抽象方法也不能用 private 关键字修饰,因为子类不能访问带 private 关键字的抽象方法。

抽象类中可以定义构造方法,因为抽象类仍然使用的是类继承关系,而且抽象类中也可以定义成员变量。因此,子类在实例化时必须先对抽象类进行实例化,如例 5-13 所示。

**例 5-13** TestAbstractConstructor.java

```
// 用 abstract 修饰抽象类
abstract class Parent {
        // abstract 修饰抽象方法,只有声明,没有实现
        private String name;
        public Parent() {
            System.out.println("抽象类无参构造方法");
        }
        public Parent(String name) {
            this.name = name;
            System.out.println("抽象类有参构造方法");
        }
}
// 继承抽象方法
class Child extends Parent {
    public Child() {
        System.out.println("子类无参构造函数");
    }
    public Child(String name) {
        super(name);                              // 显示调用抽象类有参构造函数
        System.out.println("子类有参构造函数");
    }
}
public class TestAbstractConstructor {
    public static void main(String[] args) {
        new Child();
        new Child("张三");
    }
}
```

程序的运行结果如图 5.14 所示。

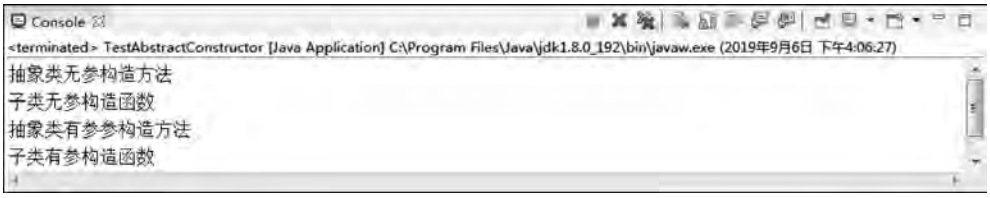


图 5.14 例 5-13 运行结果

在例 5-13 中,抽象类 Parent 中定义了无参和有参两个构造方法,从运行结果可以看出,在子类对象实例化时会默认调用抽象父类中的无参构造方法,也能直接通过 super 关键字调用抽象父类中指定参数的构造方法。

## 5.3.2 接口

接口是全局常量和公共抽象方法的集合,可被看作一种特殊的类,也属于引用类型。每个接口都被编译成独立的字节码文件。Java 提供 interface 关键字,用于声明接口,其语法

格式如下。

```
interface 接口名{
    全局常量声明
    抽象方法声明
}
```

接下来演示 interface 关键字的作用,具体示例如下。

```
// 用 interface 声明接口
interface Parent {
    String name;                          // 等价于 public static final String name;
    void say();                           // 等价于 public abstract void say();
}
```

接口中定义的变量和方法都包含默认的修饰符,其中定义的变量默认声明为“public static final”,即全局常量。另外,定义的方法默认声明为“public abstract”,即抽象方法。

### 5.3.3 接口的实现

与抽象类相似,接口中也包含抽象方法。因此,不能直接实例化接口,即不能使用 new 创建接口的实例。Java 提供 implements 关键字,用于实现多个接口,具体示例如下。

```
class 类名 implements 接口列表{
     属性和方法
}
```

接下来演示接口的实现,如例 5-14 所示。

**例 5-14** TestImplements.java

```
1  // 用 interface 声明接口
2  interface Person {
3      void say();
4  }
5  interface Parent {
6      void work();
7  }
8  // 用 implements 实现两个接口
9  class Child implements Person, Parent {
10     public void work() {
11         System.out.println("学习");
12     }
13     public void say() {
14         System.out.println("Child");
15     }
16 }
17 public class TestImplements{
18     public static void main(String[] args) {
19         Child c = new Child();
20         c.say();
```

```
21          c.work();
22      }
23  }
```

程序的运行结果如图 5.15 所示。


图 5.15　例 5-14 运行结果

在例 5-14 中，使用 interface 定义了两个接口，并在声明 Child 类的同时使用 implements 实现了接口 Person 和 Parent，接口名之间用逗号分隔，类中实现了接口中所有的抽象方法。从程序运行结果可发现，Child 类实现了接口且可以被实例化。

### 5.3.4　接口的继承

在 Java 中使用 extends 关键字来实现接口的继承，它与类的继承类似，当一个接口继承父接口时，该接口会获得父接口中定义的所有抽象方法和常量，但又与类的继承不同，接口支持多重继承，即一个接口可以继承多个父接口。其语法格式如下。

```
interface 接口名 extends 接口列表 {
    全局常量声明
    抽象方法声明
}
```

接下来演示接口之间的继承关系，如例 5-15 所示。

**例 5-15**　TestInterfaceExtend.java

```
1  // 用 interface 声明接口
2  interface Person {
3      void say();
4  }
5  // 用 extends 继承接口
6  interface Parent extends Person {
7      void work();
8  }
9  // 用 implements 实现两个接口
10 class Child implements Parent {
11     public void work() {
12         System.out.println("学习");
13     }
14     public void say() {
15         System.out.println("Child");
16     }
17 }
```

```
18 public class TestInterfaceExtend {
19     public static void main(String[] args) {
20         Child c = new Child();
21         c.say();
22         c.work();
23     }
24 }
```

程序的运行结果如图 5.16 所示。

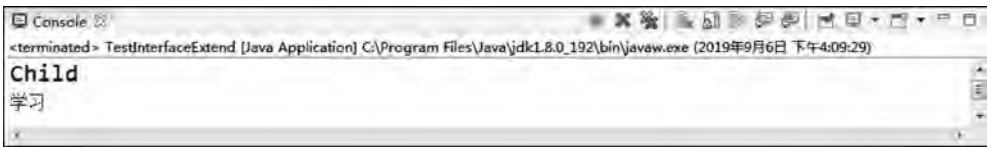


图 5.16　例 5-15 运行结果

在例 5-15 中定义了两个接口,其中 Parent 接口继承了 Person 接口,当 Child 类实现 Parent 接口时,需要实现父类接口中所有的方法。需要特别指出的是,任何实现继承接口的类,必须实现该接口继承的其他接口,除非类被声明为 abstract。

### 5.3.5　抽象类和接口的关系

抽象类与接口是 Java 语言中对于抽象类定义进行支持的两种机制,两者非常相似,初学者经常混淆这两个概念,两者的相同点可以归纳为以下三点。

(1) 都包含抽象方法。

(2) 都不能被实例化。

(3) 都是引用类型。

表 5.1 列出了两者之间的区别。

**表 5.1　接口与抽象类**

| 区别点 | 接　口 | 抽　象　类 |
|---|---|---|
| 含义 | 接口通常用于描述一个类的外围能力,而不是核心特征。类与接口之间是-able 或者 can do 的关系 | 抽象类定义了它的后代的核心特征。派生类与抽象类之间是 is-a 的关系 |
| 方法 | 接口只提供方法声明 | 抽象类可以提供完整方法、默认构造方法以及用于覆盖的方法声明 |
| 变量 | 只包含 public static final 常量,常量必须在声明时初始化 | 可以包含实例变量和静态变量 |
| 多重继承 | 一个类可以继承多个接口 | 一个类只能继承一个抽象类 |
| 实现类 | 类可以实现多个接口 | 类只从抽象类派生,必须重写 |
| 适用性 | 所有的实现只是共享方法签名 | 所有实现大同小异,并且共享状态和行为 |
| 简洁性 | 接口中的常量都被默认为 public static final,可以省略。接口中的方法被默认为 public abstract | 可以在抽象类中放置共享代码。必须用 abstract 显式声明方法为抽象方法 |
| 添加功能 | 如果为接口添加一个新的方法,则必须查找所有实现该接口的类,并为它们逐一提供该方法的实现 | 如果为抽象类提供一个方法,可以选择提供一个默认的实现,那么所有已存在的代码不需要修改就可以继续工作 |

总体来说，抽象类和接口都用于为对象定义共同的行为，两者在很大程度上是可以互相替换的，但由于抽象类只允许单继承，所以当两者都可以使用时，优先考虑接口，只有当需要定义子类的行为，并为子类提供共性功能时才考虑选用抽象类。

## 5.4 多　　态

多态是面向对象的另一大特征，封装和继承是为实现多态做准备的。简单来说，多态是具有表现多种形态能力的特征，它可以提高程序的抽象程度和简洁性，最大程度降低了类和程序模块间的耦合性。

### 5.4.1 多态的概念

多态是指同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果。在Java程序中，多态是指把类中具有相似功能的不同方法使用同一个方法名实现，从而可以使用相同的方式来调用这些具有不同功能的同名方法。接下来通过一个案例演示多态的实现，如例5-16所示。

**例5-16** TestPolymorphism.java

```
1  // 定义 Person 类
2  class Person {
3      public void say() {
4          System.out.println("Person");
5      }
6  }
7  // 定义 Parent 类继承 Person 类
8  class Parent extends Person {
9      public void say() {
10         System.out.println("Parent");
11     }
12 }
13 // 定义 Child 类实现 Parent 类
14 class Child extends Parent {
15     public void say() {
16         System.out.println("Child");
17     }
18 }
19 public class TestPolymorphism {
20     public static void main(String[] args) {
21         // 定义 Person 类型引用变量
22         Person p = null;
23         // 使用 Person 类型变量引用 Parent 对象
24         p = new Parent();
25         p.say();
26         // 使用 Person 类型变量引用 Child 对象
27         p = new Child();
28         p.say();
```

```
29          // 使用 Parent 类型变量引用 Child 对象
30          Parent p2 = new Child();
31          p2.say();
32      }
33  }
```

程序的运行结果如图 5.17 所示。

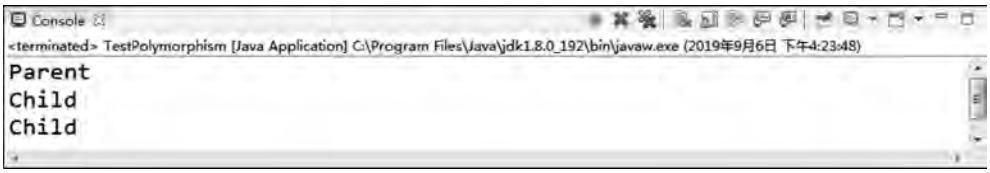


图 5.17　例 5-16 运行结果

在例 5-16 中,主方法中实现了父类类型变量引用不同的子类对象,其中第 27 行代码处,变量 p 引用的是 Parent 对象,因此,此处调用的是 Parent 类里的 say()方法;第 28 行代码处,变量 p 引用的是 Child 对象,因此,此处调用的是 Child 类里重写的 say()方法。从程序运行结果可发现,虽然执行的都是“p.say();”语句,但变量引用的对象是不同的,执行的结果也不同,这就是前面所讲的多态。

Java 中的引用变量有两种类型,即声明类型和实际类型。变量声明时被指定的类型,称为声明类型,而被变量引用的对象类型,称为实际类型。方法可以在沿着继承链的多个类中实现,当调用实例方法时,由 Java 虚拟机动态地决定所调用的方法,称为动态绑定。

动态绑定机制原理是:当调用实例方法时,Java 虚拟机从该变量的实际类型开始,沿着继承链向上查找该方法的实现,直到找到为止,并调用首次找到的实现,如例 5-17 所示。

**例 5-17**　TestDynamicBinding.java

```
1   // 定义 Person 类
2   class Person {
3       public void say() {
4           System.out.println("Person");
5       }
6   }
7   // 定义 Parent 类继承 Person 类
8   class Parent extends Person {
9       public void say() {
10          System.out.println("Parent");
11      }
12  }
13  // 定义 Child 类继承 Parent 类
14  class Child extends Parent{
15  }
16  public class TestDynamicBinding {
17      public static void main(String[] args) {
18          // 使用 Person 类型变量引用 Child 对象
19          Person p = new Child();
```

```
20          p.say();
21      }
22  }
```

程序的运行结果如图 5.18 所示。

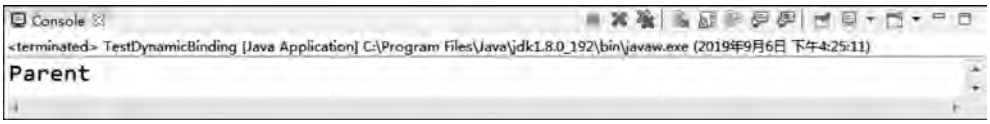


图 5.18　例 5-17 运行结果

在例 5-17 中，Person 类和 Parent 类都实现了 say()方法，Child 是空类，三个类构成了继承链。主方法中调用 say()方法时，Java 虚拟机沿着继承链向上查找实现并运行。因此，运行结果打印了“Parent”。由此可见，Java 虚拟机在运行时动态绑定方法的实现，是由变量的实际类型决定的。

## 5.4.2　对象的类型转换

对象的类型转换是指可以将一个对象的类型转换成继承结构中的另一种类型。类型转换分为两种，具体如下。

(1) 向上转型，是从子类到父类的转换，也称隐式转换。

(2) 向下转型，是从父类到子类的转换，也称显式转换。

接下来演示对象的类型转换，如例 5-18 所示。

**例 5-18**　TestTypeCast.java

```
1  // 定义 Person 类
2  class Person {
3      public void say() {
4          System.out.println("Person");
5      }
6  }
7  // 定义 Parent 类继承 Person 类
8  class Parent extends Person {
9      public void say() {
10         System.out.println("Parent");
11     }
12 }
13 // 定义 Child 类继承 Parent 类
14 class Child extends Parent{
15     public void say() {
16         System.out.println("Child");
17     }
18 }
19 public class TestTypeCast {
20     public static void main(String[] args) {
21         Person p = new Child();                    // 向上转型
22         Parent o = (Parent) p;                     // 向下转型
```

```
        o.say();
    }
}
```



程序的运行结果如图 5.19 所示。

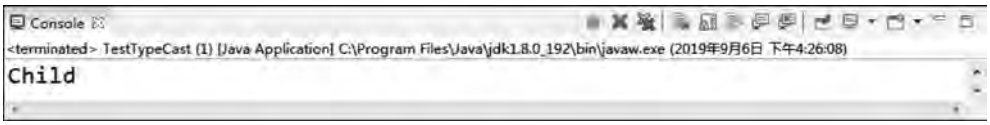


图 5.19　例 5-18 运行结果

在例 5-18 中,定义三个类构成继承链,创建 Child 对象并赋给 Person 的变量 p,是隐式的转换,称为向上转型;再将变量 p 赋给 Parent 的变量 o,必须强制转换,称为向下转型。从程序运行结果可发现,对象向下转型后,调用方法还是由实际对象决定的。

需要特别注意的是,向下转型时,被转换变量的实际类型,必须是转换类或其子类,如例 5-19 所示。

**例 5-19**　TestInstanceof.java

```
// 定义 Person 类
class Person {
    public void say() {
        System.out.println("Person");
    }
}
// 定义 Parent 类继承 Person 类
class Parent extends Person {
    public void say() {
        System.out.println("Parent");
    }
}
// 定义 Child 类继承 Person 类
class Child extends Person {
    public void say() {
        System.out.println("Child");
    }
}
public class TestInstanceof {
    public static void main(String[] args) {
        Person p = new Child();                    // 向上转型
        Parent o = (Parent) p;                     // 向下转型
        o.say();
    }
}
```

程序的运行结果如图 5.20 所示。

在图 5.20 中,程序运行结果报错并提示“Child cannot be cast to Parent”。报错的原因在于:Person 类型变量 p 先指向 Child 对象,再向下转型为 Parent 类型。在编译时,编译器


图 5.20　例 5-19 运行结果

检测的是变量的声明类型，则编译可以通过。但在运行时，转换的是变量的实际类型，而 Child 类型无法强制转换为 Parent 类型。

针对这种情况，Java 提供了 instanceof 关键字，用于判断一个对象是否是一个类（或接口）的实例，表达式返回 boolean 值，其语法格式如下。

```
变量名　instanceof　类名
```

接下来对例 5-19 的主方法进行修改，具体代码如下。

```
public static void main(String[] args) {
    Person p = new Child();                    // 向上转型
    if (p instanceof Parent) {                 // 判断对象是否是 Parent 类型
        Parent o = (Parent) p;
        o.say();
    } else if (p instanceof Child) {           // 判断对象是否是 Child 类型
        Child o = (Child) p;
        o.say();
    }
}
```

程序的运行结果如图 5.21 所示。

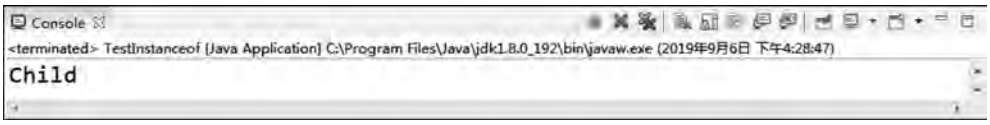


图 5.21　例 5-19 修改后运行结果

从程序运行结果可发现，instanceof 能准确判断出对象是否是某个类的实例。

### 5.4.3　Object 类

Java 中提供了一个 Object 类，是所有类的父类，如果一个类没有显式地指定继承类，则该类的父类默认为 Object。例如，下面两个类的定义是一样的。

```
class ClassName {}
class ClassName extends Object {}
```

在 Object 类中提供了很多方法，接下来分别对其中的方法进行解释，如表 5.2 所示。

本章暂时只对 toString()和 equals 方法进行讲解，而 hashCode()方法在 Java 集合中再详细讲解。

表 5.2　Object 类的方法

| 方 法 声 明 | 功 能 描 述 |
| --- | --- |
| public String toString() | 返回描述该对象的字符串 |
| public Boolean equals(Object o) | 比较两个对象是否相等 |
| public int hashCode() | 返回对象的哈希值 |

**1. toString()方法**

调用一个对象的 toString()方法会默认返回一个描述该对象的字符串,它由该对象所属类名、@和对象十六进制形式的内存地址组成,如例 5-20 所示。

**例 5-20**　TestToString.java

```
1  class Person {
2      private String name;
3      private int age;
4      public Person(String name, int age) {
5          this.name = name;
6          this.age = age;
7      }
8  }
9  public class TestToString {
10   public static void main(String[] args) {
11     Person o = new Person("张三", 18);
12         // 调用对象的 toString 方法
13         System.out.println(o.toString());
14         // 直接打印对象
15         System.out.println(o);
16     }
17 }
```

程序的运行结果如图 5.22 所示。

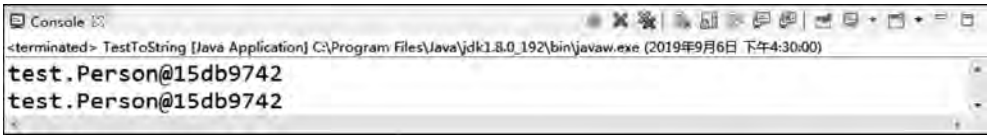


图 5.22　例 5-20 运行结果

在图 5.22 中,默认打印了对象信息,从程序运行结果可发现,直接打印对象和打印对象的 toString()方法返回值相同,也就是说,对象输出一定会调用 Object 类的 toString()方法。

通常,重写 toString()方法返回对象具体的信息。修改例 5-20 中的 Person 类,添加重写 toString()方法的代码,具体示例如下。

```
public String toString() {
    return "Person [name = " + name + ", age = " + age + "]";
}
```

程序的运行结果如图 5.23 所示。

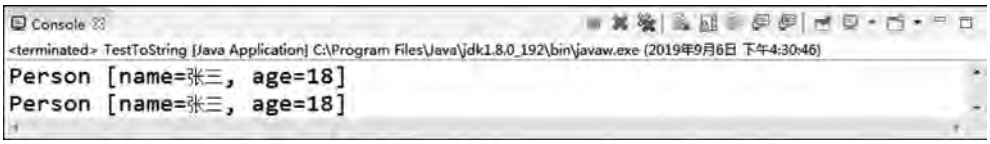


图 5.23　例 5-20 修改后运行结果

在图 5.23 中，从程序运行结果可发现，程序调用的是子类重写 Object 类的 toString() 方法，这是多态机制的体现。

**2. equals()方法**

equals()方法是用于测试两个对象是否相等，如例 5-21 所示。

**例 5-21**　TestEquals.java

```
1  class Person {
2      private String name;
3      private int age;
4      public Person(String name, int age) {
5          this.name = name;
6          this.age = age;
7      }
8      // 自定义比较方法,检查两个引用变量是否指向同一对象
9      public boolean myEquals(Object o) {
10         return (this == o);
11     }
12 }
13 public class TestEquals {
14     public static void main(String[] args) {
15       Person o1 = new Person("张三", 18);
16       Person o2 = new Person("张三", 18);
17       // 调用对象的 equals 方法
18       System.out.println(o1.equals(o2));
19       // 调用自定义 myEquals 方法
20       System.out.println(o1.myEquals(o2));
21     }
22 }
```

程序的运行结果如图 5.24 所示。


图 5.24　例 5-21 运行结果

在图 5.24 中，从程序运行结果可发现，equals()方法与直接使用==运算符检测两个对象结果相同。这是由于 equals()方法的默认实现就是用==运算符检测两个引用变量是否指向同一个对象，即比较的是地址。

如果要检测两个不同对象的内容是否相同,就必须重写 equals()方法。例如,String 类中的 equals()方法继承自 Object 类并重写,使之能够检验两个字符串的内容是否相等。修改例 5-21 中的 Person 类,添加重写 equals()方法的代码,具体示例如下。

```
public boolean equals(Object o) {
    // 比较对象是否是自己
    if (this == o)
        return true;
    // 判断是否为该类对象
    if (!(o instanceof Person))
        return false;
    Person p = (Person) o;
    // 逐个属性比较,看是否相等
    if (
            (
                // 比较 name 属性
                (null == name && null == p.name)                    // 同为 null 的情况
                || (null != name && name.equals(p.name))
            )
            && age == p.age                                         // 比较 age 属性
        )
        return true;
    return false;
}
```

程序的运行结果如图 5.25 所示。


图 5.25　例 5-21 修改后运行结果

在例 5-21 中,在 Person 类中重写了 equals()方法,首先比较两个对象的地址是否相等,如果相等,则是同一对象。因为 equals()方法的形参是 Object 类型,所以可以接收任何对象。因此,必须要判断对象是否是 Person 的实例,如果是,则依次比较各个属性。

## 5.4.4　设计模式——工厂设计模式

工厂模式(Factory)主要用来实例化有共同接口的类,它可以动态决定应该实例化哪一个类,不必事先知道每次要实例化哪一个类。工厂模式主要有三种形态:简单工厂模式、工厂方法模式和抽象工厂模式。接下来分别对这三种形态进行讲解。

**1. 简单工厂模式**

简单工厂模式(Simple Factory Pattern)又称静态工厂方法,它的核心是类中包含一个静态方法,该方法用于根据参数来决定返回实现同一接口不同类的实例,如例 5-22 所示。

**例 5-22** TestSimpleFactoryPattern.java

```
// 定义产品接口
interface Product {
}
// 定义安卓手机类
class Android implements Product {
    public Android() {
        System.out.println("安卓手机被创建!");
    }
}
// 定义苹果手机类
class Iphone implements Product {
    public Iphone() {
        System.out.println("苹果手机被创建!");
    }
}
// 定义工厂类
class SimpleFactory {
    public static Product factory(String className) {
        if ("Android".equals(className)) {
            return new Android();
        } else if ("Iphone".equals(className)) {
            return new Iphone();
        } else {
            return null;
        }
    }
}
public class TestSimpleFactoryPattern {
    public static void main(String[] args) {
        // 根据不同的参数生成产品
        SimpleFactory.factory("Android");
        SimpleFactory.factory("Iphone");
    }
}
```

程序的运行结果如图 5.26 所示。


图 5.26 例 5-22 运行结果

在例 5-22 中，定义 SimpleFactory 类就是简单工厂的核心，该类拥有必要的逻辑判断和创建对象的责任。由此可见，简单工厂就是将创建产品的操作集中在一个类中。

工厂类 SimpleFactory 有很多局限。首先，维护和新增产品时，都必须修改 SimpleFactory 源代码。其次，如果产品之间存在复杂的层次关系，则工厂类必须拥有复杂的逻辑判断。最

后,整个系统都依赖 SimpleFactory 类,一旦 SimpleFactory 类出现问题,整个系统就将瘫痪不能运行。



**2. 工厂方法模式**

工厂方法模式(Factory Method Pattern)为工厂类定义了接口,用多态来削弱了工厂类的职责,如例 5-23 所示。

**例 5-23** TestFactoryMethodPattern.java

```
1  // 定义产品接口
2  interface Product {
3  }
4  // 定义安卓手机类
5  class Android implements Product {
6      public Android() {
7          System.out.println("安卓手机被创建!");
8      }
9  }
10 // 定义苹果手机类
11 class Iphone implements Product {
12     public Iphone() {
13         System.out.println("苹果手机被创建!");
14     }
15 }
16 // 定义工厂接口
17 interface Factory {
18     public Product create();
19 }
20 // 定义 Android 的工厂
21 class AndroidFactory implements Factory {
22     public Product create() {
23         return new Android();
24     }
25 }
26 // 定义 Iphone 的工厂
27 class IphoneFactory implements Factory {
28     public Product create() {
29         return new Iphone();
30     }
31 }
32 public class TestFactoryMethodPattern {
33     public static void main(String[] args) {
34         // 根据不同的子工厂创建产品
35         Factory factory = null;
36         factory = new AndroidFactory();
37         factory.create();
38         factory = new IphoneFactory();
39         factory.create();
40     }
41 }
```

程序的运行结果如图 5.27 所示。


图 5.27　例 5-23 运行结果

在例 5-23 中，定义 Factory 工厂接口，并声明 create()工厂方法，将创建产品的操作放在了实现该方法的子工厂 AndroidFactory 类和 IphoneFactory 类中。由此可见，工厂方法模式是将简单工厂创建对象的职责，分担到子工厂类中，子工厂相互独立，互相不受影响。

工厂方法模式也有局限之处，当面对有复杂的树形结构的产品时，就必须为每个产品创建一个对应的工厂类，当达到一定数量级就会出现类爆炸。

**3. 抽象工厂模式**

抽象工厂模式(Abstract Factory Pattern)用于意在创建一系列互相关联或互相依赖的对象，如例 5-24 所示。

**例 5-24**　TestAbstractFactoryPattern.java

```
1  // 定义 Android 接口
2  interface Android {
3  }
4  // 定义 Iphone 接口
5  interface Iphone {
6  }
7  // 定义安卓手机 - A 类
8  class AndroidA implements Android {
9      public AndroidA() {
10         System.out.println("安卓手机 - A 被创建!");
11     }
12 }
13 // 定义安卓手机 - B 类
14 class AndroidB implements Android {
15     public AndroidB() {
16         System.out.println("安卓手机 - B 被创建!");
17     }
18 }
19 // 定义苹果手机 - A 类
20 class IphoneA implements Iphone {
21     public IphoneA() {
22         System.out.println("苹果手机 - A 被创建!");
23     }
24 }
25 // 定义苹果手机 - B 类
26 class IphoneB implements Iphone {
27     public IphoneB() {
28          System.out.println("苹果手机 - B 被创建!");
29     }
```

```
30 }
31 // 定义工厂接口
32 interface Factory {
33     public Android createAndroid();
34     public Iphone createIphone();
35 }
36 // 创建型号 A 的产品工厂
37 class FactoryA implements Factory {
38     public Android createAndroid() {
39         return new AndroidA();
40     }
41     public Iphone createIphone() {
42         return new IphoneA();
43     }
44 }
45 // 创建型号 B 的产品工厂
46 class FactoryB implements Factory {
47     public Android createAndroid() {
48         return new AndroidB();
49     }
50     public Iphone createIphone() {
51         return new IphoneB();
52     }
53 }
54 public class TestAbstractFactoryPattern {
55     public static void main(String[] args) {
56         // 根据不同的型号创建产品
57         Factory factory = null;
58         factory = new FactoryA();                    // 创建 A 工厂
59         factory.createAndroid();                     // 创建安卓 - A 手机
60         factory.createIphone();                      // 创建苹果 - A 手机
61         factory = new FactoryB();                    // 创建 B 工厂
62         factory.createAndroid();                     // 创建安卓 - B 手机
63         factory.createIphone();                      // 创建苹果 - B 手机
64     }
65 }
```

程序的运行结果如图 5.28 所示。

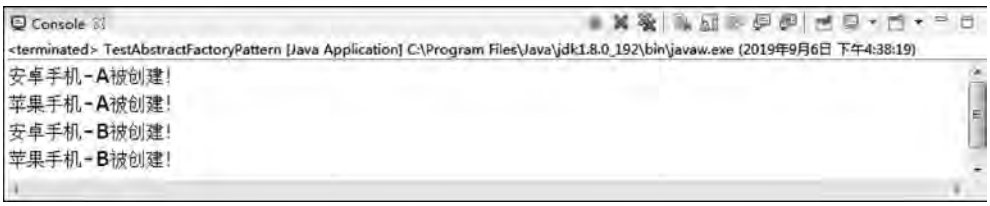


图 5.28　例 5-24 运行结果

在例 5-24 中,定义了 Factory 工厂接口,并声明两个方法,对应创建 Android 和 Iphone 两种产品。而 FactoryA 和 FactoryB 子工厂实现工厂接口,用于创建一系列产品。由此可

见，抽象工厂是在工厂方法基础上进行了分类管理。

### 5.4.5 设计模式——代理设计模式

代理模式(Proxy)是指给某一个对象提供一个代理，并由代理对象控制对原有对象的引用。如生活中，求职者找工作(真实操作)，可以让猎头帮忙去找(代理操作)，猎头把最终结果反馈给求职者。无论是真实操作还是代理操作，目的都是一样的，求职者只关心最终结果，而不关心过程，如例 5-25 所示。

**例 5-25** TestProxyPattern.java

```
1  // 定义工作接口
2  interface Work {
3      void find();
4  }
5  // 真实操作
6  class Real implements Work{
7      public void find() {
8          System.out.println("投递简历");
9      }
10 }
11 // 代理操作
12 class Proxy implements Work{
13     private Work work;
14     public Proxy(Work work) {
15         this.work = work;
16     }
17     public void find() {
18         System.out.println("合法验证");
19         work.find();
20         System.out.println("反馈结果");
21     }
22 }
23 public class TestProxyPattern {
24     public static void main(String[] args) {
25         Work work = null;
26         work = new Proxy(new Real());       // 交给猎头去找
27         work.find();
28     }
29 }
```

程序的运行结果如图 5.29 所示。


图 5.29 例 5-25 运行结果

在例 5-25 中,代理类 Proxy 完成了代理操作,由此可见,代理操作最终还是调用了真实操作。



## 5.5 包

当声明的类很多时,类名就有可能冲突,这时就需要一种机制来管理类名,因此,Java 中引入了包机制,本节将详细介绍包的用法。

### 5.5.1 包的定义与使用

包(package)是 Java 提供的一种区别类的名字空间的机制,是类的组织方式,是一组相关类和接口的集合,它提供了访问权限和命名的管理机制。

使用 package 语句声明包,其语法格式如下。

```
package 包名
```

使用时需要注意以下四点。

(1) 包名中字母一般都要小写。

(2) 包的命名规则:将公司域名反转作为包名。

(3) package 语句必须是程序代码中的第一行可执行代码。

(4) package 语句最多只有一句。

包与文件目录类似,可以分成多级,多级之间用“.”符号进行分隔,具体示例如下。

```
package com.1000phone.www;
```

如果在程序中已声明了包,就必须将编译生成的字节码文件保存到与包名同名的子目录中,可以使用带包编译命令,具体示例如下。

```
javac -d . Source.java
```

其中,“-d”表示生成以 package 定义为准的目录,“.”表示在当前所在的文件夹中生成。编译器会自动在当前目录下建立与包名同名的子目录,并将生成的.class 文件自动保存到与包名同名的子目录下。

接下来分步骤讲解包机制在 cmd 中通过命令提示符管理 Java 源文件。

(1) 在源文件首行声明包,如例 5-26 所示。

**例 5-26** TestPackage.java

```
1  // 声明类在 com.1000phone.www 包下
2  package com.1000phone.www;
3  public class TestPackage {
4      public static void main(String[] args) {
5          System.out.println("包机制");
6      }
7  }
```

(2) 使用“javac -d . TestPackage.java”编译源文件,程序的运行结果如图 5.30 所示。


图 5.30 编译源文件

(3) 执行命令完成后,在当前目录下会生成包名“com.1000phone.www”对应的目录,如图 5.31 所示。


图 5.31 包编译

(4) 使用“java com.1000phone.www.TestPackage”命令运行程序,运行带包名的字节码文件时,必须输入完整的“包.类名称”。程序的运行结果如图 5.32 所示。


图 5.32 例 5-26 运行结果

由此可见,包的管理机制让类管理更加方便。

## 5.5.2 import 语句

在实际开发中,项目都是分模块开发的,对应模块包中的类,完成相应模块的功能。但有时模块之间的类要相互调用。例如,通常开发中都是将业务逻辑层的接口和实现放在不同包中,在 Eclipse 中新建一个 chapter05 项目,在项目的 src 根目录下新建 service 和 service.impl 两

个包,在 service 包中创建 UserService 接口,在 service.impl 包下创建 UserServiceImpl 类,如例 5-27 和例 5-28 所示。

**例 5-27** UserService.java

```
1  package service;
2  public interface UserService {
3      public void say();
4  }
```

**例 5-28** UserServiceImpl.java

```
1  package service.impl;
2  public class UserServiceImpl implements UserService {
3      public void say() {
4          System.out.println("用户信息");
5      }
6      // 测试
7      public static void main(String[] args) {
8          UserService user = new UserServiceImpl();
9          user.say();
10         }
11     }
```

使用命令"javac -d. IUserService.java"和"javac -d. UserServiceImpl.java"编译源文件,程序的运行结果如图 5.33 所示。

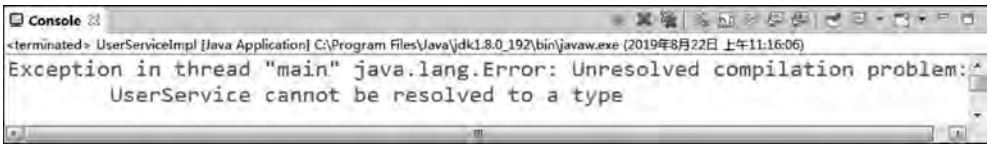


图 5.33 编译错误

在图 5.33 中,程序编译结果报错并提示"UserService cannot be resolved to a type"。这是因为类 UserService 位于 service 包中,而类 UserServiceImpl 位于 service.impl 包中,两者虽然是父包和子包的逻辑关系,但在用法上则不存在任何关系,如果要使用包中的类,则必须 import 该类所在的包。修改例 5-28 的代码,修改后的代码如下。

```
package service.impl;
import service.UserService;
public class UserServiceImpl implements UserService {
    public void say() {
        System.out.println("用户信息");
    }
    // 测试
    public static void main(String[] args) {
        UserService user = new UserServiceImpl();
        user.say();
    }
}
```

重新编译 UserServiceImpl 类，输入命令并运行 UserServiceImpl 类，命令为“java com.1000phone.www.javaTrain.user.service.impl.UserServiceImpl”，程序的运行结果如图 5.34 所示。

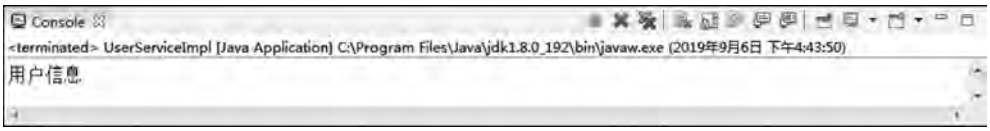


图 5.34　例 5-28 修改后运行结果

**注意**：import 关键字用于导入指定包层次下的某个类或全部类，import 语句应放在 package 语句之后，类定义之前，其语法格式如下。

```
import 包名.类名                    // 导入单类
import 包名.*                       // 导入包层次下的全部类
```

接下来在 service.impl 包下新建 UserServiceImp102 类，其内容是对例 5-28 中的修改，修改后的代码如例 5-29 所示。

**例 5-29**　UserServiceImp102.java

```
1  package service.impl;
2  import service.*;
3  public class UserServiceImpl02 implements UserService {
4      public void say() {
5          System.out.println("用户信息");
6      }
7      // 测试
8      public static void main(String[] args) {
9          UserService user = new UserServiceImpl02();
10         user.say();
11         }
12     }
```

程序的运行结果如图 5.35 所示。

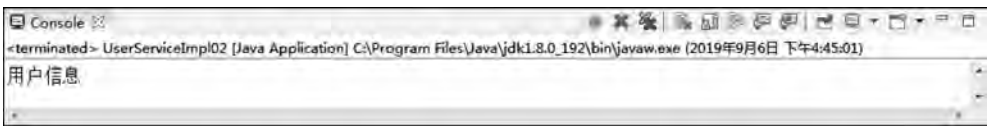


图 5.35　例 5-29 运行结果

在例 5-29 中，使用 import 关键字导入指定包层次下的全部类。因此，无须再使用包名.类名的方式。

在 JDK 5.0 后提供了静态导入功能，用于导入指定类的某个静态成员变量、方法或全部的静态成员变量、方法，具体示例如下。

```
import static 包名.类名.成员          // 导入类指定静态成员
import static 包名.类名.*             // 导入该类全部静态成员
```

import 语句和 import static 语句之间没有任何顺序要求。使用 import 导入包后,可以在代码中直接访问包中的类,即可以省略包名;而使用 import static 导入类后,可以在代码中直接访问类中的静态成员,即可以省略类名。接下来通过代码进行演示,在 chapter05 项目根目录下新建 util 包,并在该包下创建 Calc 类,然后在项目根目录下再新建 test 包,并在该包下创建 TestImportStatic 类,如例 5-30 和例 5-31 所示。

**例 5-30** Calc.java

```
1  package util;
2  public class Calc {
3      public static int add(int a, int b) {
4          return a + b;
5      }
6      public static int sub(int a, int b) {
7          return a - b;
8      }
9      public static int mul(int a, int b) {
10         return a * b;
11     }
12     public static int div(int a, int b) {
13         return a/b;
14     }
15 }
```

**例 5-31** TestImportStatic.java

```
1  package test;
2  import static util.Calc.*;
3  public class TestImportStatic {
4      public static void main(String[] args) {
5                  // 省略类名直接调用静态方法
6                  System.out.println("10 + 2 = " + add(10, 2));
7                  System.out.println("10 - 2 = " + sub(10, 2));
8                  System.out.println("10 * 2 = " + mul(10, 2));
9                  System.out.println("10/2 = " + div(10, 2));
10     }
11 }
```

程序的运行结果如图 5.36 所示。

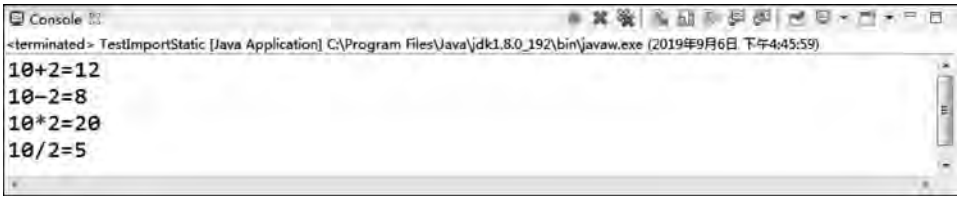


图 5.36 例 5-31 运行结果

在例 5-31 中,通过 import static 导入了 Calc 类中的全部静态方法,而在使用时省略了类名。由此可见,import 和 import static 功能相似,只是导入的对象不一样,都是用于减少代码量。

### 5.5.3 Java 的常用包

Java 的核心类都放在 java 包及其子包下，Java 扩展的类都放在 javax 包及其子包下，接下来了解一下常用的开发包，如表 5.3 所示。

**表 5.3 Java 系统包**

| 包 名 | 功 能 描 述 |
|---|---|
| java.lang | 核心包，如 String、Math、System 类等，无须使用 import 手动导入，系统自动导入 |
| java.util | 工具包，包含工具类、集合类等，如 Arrays、List 和 Set 等 |
| java.net | 包含网络编程的类和接口 |
| java.io | 包含输入、输出编程相关的类和接口 |
| java.text | 包含格式化相关的类 |
| java.sql | 数据库操作包，提供了各种数据库操作的类和接口 |
| java.awt | 包含抽象窗口工具集(Abstract Window Toolkits，AWT)相关类和接口，主要用于构建图形用户界面(GUI) |
| java.swing | 包含图形用户界面相关类和接口 |

### 5.5.4 给 Java 应用程序打包

在实际开发中，通常会将一些类提供给别人使用，直接提供字节码文件会比较麻烦，所以一般会将这些类文件打包成 jar 文件，以供别人使用。jar 文件的全称是 Java Archive File，意思就是 Java 归档文件，也称为 jar 包。将一个 jar 包添加到 classpath 环境变量中，Java 虚拟机会自动解压 jar 包，根据包名所对应的目录结构去查找所需的类。

通常使用 jar 命令来打包，可以把一个或多个路径压缩成一个 jar 文件。jar 命令在 JDK 安装目录下的 bin 目录中，直接在命令行中输入 jar 命令，即可查看 jar 命令的提示信息，如图 5.37 所示。

jar 命令主要参数如下。

(1) c：创建新的文档。

(2) v：生成详细的输出信息。

(3) f：指定归档的文件名。

下面为一个独立的类进行类打包，如例 5-32 所示。

**例 5-32** User.java

```
1  package com.1000phone.www.javaTrain.user.domain;
2  public class User {
3      private String name;
4      public User(String name) {
5          this.name = name;
6      }
7      public String toString() {
8          return "User [name=" + name + "]";
9      }
10 }
```


图 5.37　jar 命令

输入命令“javac -d . User.java”编译程序,生成目录如图 5.38 所示。

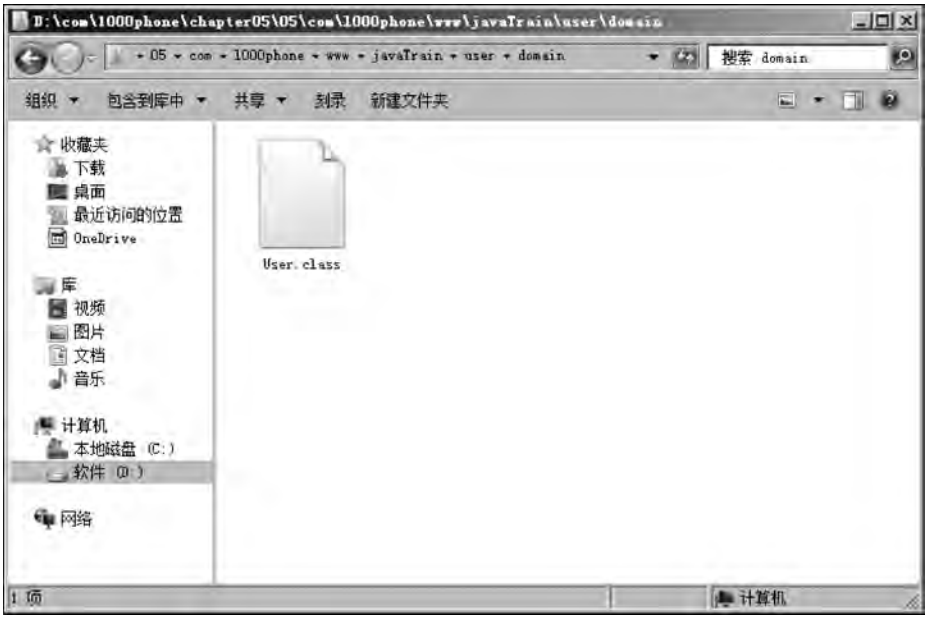


图 5.38　生成目录

接下来分步骤学习如何用 jar 命令进行文件打包及运用 jar 包。

(1) 打开命令行窗口,切换到将要被打包目录的上级目录,输入命令:

```
jar -cvf common.jar com
```

上面的命令是将 com 目录下的全部内容打包成 common.jar 文件,如果硬盘上有同名 jar 将被覆盖,运行结果如图 5.39 和图 5.40 所示。


图 5.39 压缩 jar 过程信息

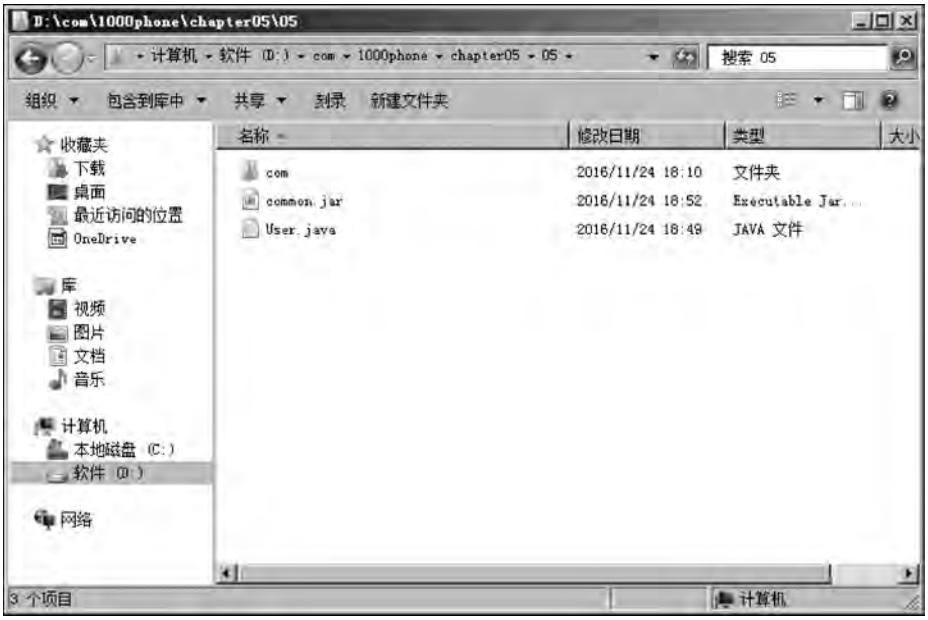


图 5.40 生成的 jar

(2) 输入“set classpath=.;D:\com\1000phone\chapter05\05\common.jar”命令,将 jar 包添加到环境变量中,如图 5.41 所示。


图 5.41 添加环境变量

(3) 删除生成的包目录及 User.class 文件。编写测试类,测试 jar 包是否可用,如例 5-33 所示。

**例 5-33** TestImportJar.java

```
1  package com.l000phone.www.javaTrain.test;
2  import com.l000phone.www.javaTrain.user.domain.*;
3
4  public class TestImportJar{
5      public static void main(String[] args) {
6          User user = new User("张三");
7          System.out.println(user);
8      }
9  }
```

程序的运行结果如图 5.42 所示。


图 5.42 例 5-33 运行结果

(4) 查看 jar 包命令如下。

```
jar -tvf jar 文件名
```

如查看 common.jar 文件,则输入命令"jar -tvf common.jar",运行结果如图 5.43 所示。

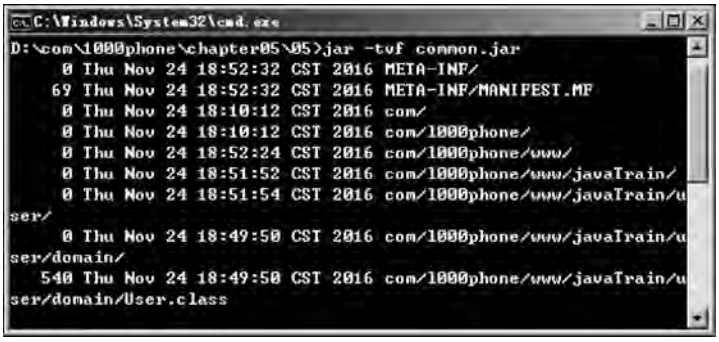


图 5.43 查看 jar

在图 5.43 中,可发现 jar 包中存在 META-INF 文件夹,在此文件夹中存在名为 MANIFEST.MF 的文件,该文件就是 jar 文件的清单文件。

(5) 解压 jar 包命令如下。

```
jar -xvf jar 文件名
```

如解压 common.jar 文件,则输入命令"jar -xvf common.jar",参数 x 代表从归档中提

取指定的文件。命令将 jar 包解压到当前目录，运行结果如图 5.44 和图 5.45 所示。


图 5.44　解压命令

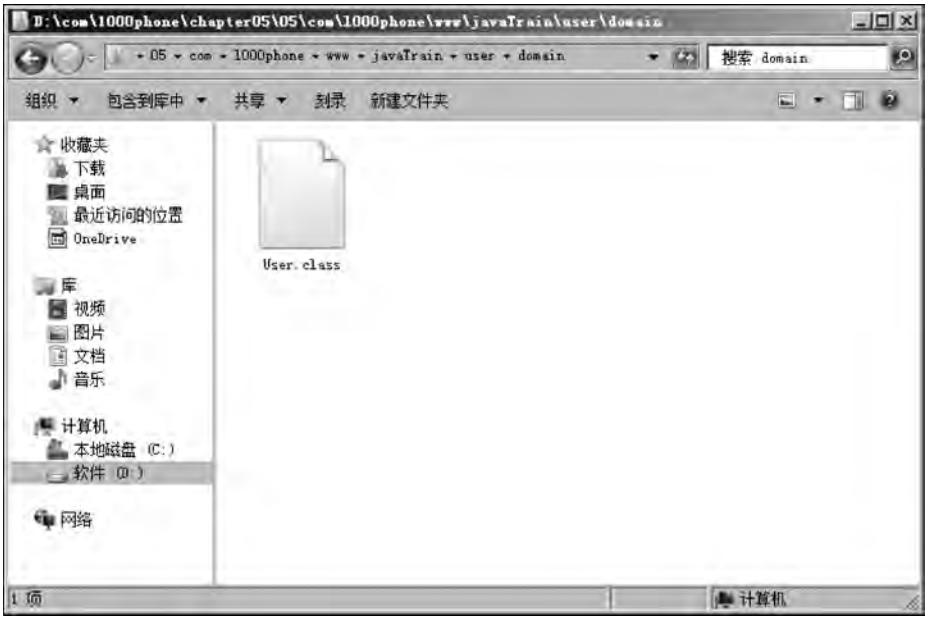


图 5.45　解压后的 jar 文件

## 5.6　Lambda 表达式

Lambda 表达式是 Java 8 发布的重要新特性，可以把它理解为一段能够像数据一样进行传递的代码，它允许把函数作为参数传递给其他的方法。在开发的过程中使用 Lambda 表达式可以使代码更加简洁，可读性更强。

### 5.6.1　Lambda 表达式语法

Java 8 中引入了一个新的操作符，“->”可以称为箭头操作符或者 Lambda 操作符。当使用 Lambda 表达式进行代码编写时就需要使用这个操作符。箭头操作符将 Lambda 表达式分成左右两部分，在操作符的左侧代表着 Lambda 表达式的参数列表（接口中抽象方法的参数列表），在操作符的右侧代表着 Lambda 表达式中所需执行的功能（是对抽象方法的具

体实现)。Lambda 表达式的语法格式如下。

```
(parameters) -> expression 或(parameters) ->{statements; }
```



上述语法还可以写成以下几种格式。

无参数无返回值:()->具体实现。

有一个参数无返回值:(x)->具体实现,或 x->具体实现。

有多个参数,有返回值,并且 Lambda 体中有多条语句:(x,y)->{具体实现}。

若方法体只有一条语句,那么大括号和 return 都可以省略。

**注意**:Lambda 表达式的参数列表的参数类型可以省略不写,可以进行类型推断。在 Java 8 之后可以使用 Lambda 表达式来表示接口的一个实现,在 Java 8 之前通常是使用匿名类实现的。

## 5.6.2 Lambda 表达式案例

接下来通过代码讲解 Lambda 表达式的使用,编写一个能够实现加、减、乘、除功能且能够实现输出字符串功能的案例。首先在 chapter05 项目的根目录下创建 lambda 包,在该包下创建 MathOne 接口,该接口中定义一个带有两个参数的 operation 方法,代码如例 5-34 所示。

**例 5-34** MathOne.java

```
1  package lambda;
2  //定义接口 MathOne
3  public interface MathOne {
4      //定义一个方法 operation
5      int operation(int a, int b);
6  }
```

然后,在 lambda 包下创建 ServiceOne 接口,在该接口中定义含有一个参数的 printMessage 方法,具体代码如例 5-35 所示。

**例 5-35** ServiceOne.java

```
1  package lambda;
2  //定义接口 ServiceOne
3  public interface ServiceOne {
4      //定义一个 printMessage 方法
5        void printMessage(String message);
6  }
```

最后,在 lambda 包下创建测试类 TestLam,在该类中实现 MathOne、ServiceOne 接口,编写功能代码,具体代码如例 5-36 所示。

**例 5-36** TestLam.java

```
1  package lambda;
2  public class TestLam {
3      public static void main(String[] args) {
4          TestLam testlam = new TestLam();
```

```
5           //实现 MathOne 接口,做加法运算,有参数类型
6           MathOne jiafa = (int a,int b) -> a + b;
7           //实现 MathOne 接口,做减法运算,无参数类型
8           MathOne jianfa = (a,b) -> b - a;
9           //实现 MathOne 接口,做乘法运算,方法体外有大括号及返回值语句
10          MathOne chengfa = (int a,int b) ->{return a * b;};
11          //实现 MathOne 接口,做除法运算,方法体外无大括号及返回值
12          MathOne chufa = (int a,int b) -> b/a;
13          //实现 ServiceOne 接口,控制台打印
14          ServiceOne service =
15 (message) -> System.out.println("Hello," + message);
16          service.printMessage("这是 Java 8 的新特性");
17          service.printMessage("这是一个 Lambda 表达式");
18          System.out.println("2 + 4 = " + testlam.operate(2,4,jiafa));
19          System.out.println("4 - 2 = " + testlam.operate(2,4,jianfa));
20          System.out.println("2 * 4 = " + testlam.operate(2,4,chengfa));
21          System.out.println("4/2 = " + testlam.operate(2,4,chufa));
22      }
23      private int operate(int a,int b,MathOne mathOne){
24          return mathOne.operation(a,b);
25      }
26 }
```

运行以上代码,可以发现控制台打印的结果中实现了加、减、乘、除和打印字符串的功能,运行结果如图 5.46 所示。


图 5.46　例 5-36 运行结果

## 5.6.3　函数式接口

**1. @FunctionalInterface 注解**

Java 8 为函数式接口引入了一个新注解@FunctionalInterface,主要用于编译级错误检查,加上该注解,当编写的接口不符合函数式接口定义的时候,编译器就会报错。函数式接口(Functional Interface)是指仅包含一个抽象方法,但是可以包含多个非抽象方法(比如,静态方法、默认方法)的接口。

正确案例:

```
package chapter05;
```

```
@FunctionalInterface
public interface ServiceOne {
       //抽象方法 printMessage
     void printMessage(String message);
}
```

错误案例:

```
package chapter05;
@FunctionalInterface
public interface ServiceOne {
       //抽象方法 printMessage
   void printMessage(String message);
       //抽象方法 printMessage2
   void printMessage2(String message);
}
```

以上的错误案例中有两个抽象方法,一个是 printMessage(),一个是 printMessage2();这与函数式接口定义有冲突,因此编译器报错。

函数式接口里是可以包含默认方法的,因为默认方法不是抽象方法,有一个默认的实现,所以是符合函数式接口的定义的,如下代码中不仅含有一个抽象方法,还有一个默认方法,代码编译时不会报错。

```
package chapter05;
@FunctionalInterface
public interface ServiceOne {
       //抽象方法
     void printMessage(String message);
       //默认方法
       default void printMessage2(String message) {
        //方法体
     };
}
```

**注意**:@FunctionalInterface 注解加或不加对于接口是不是函数式接口没有任何影响,该注解只是提醒编译器去检查该接口是否仅包含一个抽象方法。

函数式接口可以被隐式转换为 Lambda 表达式。例 5-3 中定义的 MathOne 接口中只有一个 operation 方法,通常称这种只有一个抽象方法的接口为函数式接口,可以在该接口上添加@FunctionalInterface 注解,Lambda 表达式需要函数式接口的支持。

**2. JDK 提供的函数式接口**

函数式接口可以对现有的函数友好地支持 Lambda,JDK 1.8 之前已有很多函数式接口,JDK 1.8 新增加的函数接口中包含很多类,用来支持 Java 的函数式编程,新增的 java.util.function 包中的函数式接口如表 5.4 所示。

表 5.4　JDK 1.8 新增加的函数接口

| 接　口 | 描　述 |
| --- | --- |
| BiConsumer<T,U> | 代表了一个接受两个输入参数的操作,并且不返回任何结果 |
| BiFunction<T,U,R> | 代表了一个接受两个输入参数的方法,并且返回一个结果 |
| BinaryOperator<T> | 代表了一个作用于两个同类型操作符的操作,并且返回了操作符同类型的结果 |
| BiPredicate<T,U> | 代表了一个两个参数的 boolean 值方法 |
| BooleanSupplier | 代表了 boolean 值结果的提供方 |
| Consumer<T> | 代表了接受一个输入参数并且无返回的操作 |
| DoubleBinaryOperator | 代表了作用于两个 double 值操作符的操作,并且返回了一个 double 值的结果 |
| DoubleConsumer | 代表一个接受 double 值参数的操作,并且不返回结果 |
| DoubleFunction<R> | 代表接受一个 double 值参数的方法,并且返回结果 |
| DoublePredicate | 代表一个拥有 double 值参数的 boolean 值方法 |
| DoubleSupplier | 代表一个 double 值结构的提供方 |
| DoubleToIntFunction | 接受一个 double 类型输入,返回一个 int 类型结果 |
| DoubleToLongFunction | 接受一个 double 类型输入,返回一个 long 类型结果 |
| DoubleUnaryOperator | 接受一个参数同为类型 double,返回值类型也为 double |
| Function<T,R> | 接受一个输入参数,返回一个结果 |
| IntBinaryOperator | 接受两个参数同为类型 int,返回值类型也为 int |
| IntConsumer | 接受一个 int 类型的输入参数,无返回值 |
| IntFunction<R> | 接受一个 int 类型输入参数,返回一个结果 |
| IntPredicate | 接受一个 int 类型输入参数,返回一个布尔值的结果 |
| IntSupplier | 无参数,返回一个 int 类型结果 |
| IntToDoubleFunction | 接受一个 int 类型输入,返回一个 double 类型结果 |
| IntToLongFunction | 接受一个 int 类型输入,返回一个 long 类型结果 |
| IntUnaryOperator | 接受一个参数同为类型 int,返回值类型也为 int |
| LongBinaryOperator | 接受两个参数同为类型 long,返回值类型也为 long |
| LongConsumer | 接受一个 long 类型的输入参数,无返回值 |
| LongFunction<R> | 接受一个 long 类型的输入参数,返回一个结果 |
| LongPredicate | 接受一个 long 类型的输入参数,返回一个布尔值类型结果 |
| LongSupplier | 无参数,返回一个 long 类型的结果 |
| LongToDoubleFunction | 接受一个 long 类型输入,返回一个 double 类型结果 |
| LongToIntFunction | 接受一个 long 类型输入,返回一个 int 类型结果 |
| LongUnaryOperator | 接受一个参数同为类型 long,返回值类型也为 long |
| ObjDoubleConsumer<T> | 接受一个 object 类型和一个 double 类型的输入参数,无返回值 |
| ObjIntConsumer<T> | 接受一个 object 类型和一个 int 类型的输入参数,无返回值 |
| ObjLongConsumer<T> | 接受一个 object 类型和一个 long 类型的输入参数,无返回值 |
| Predicate<T> | 接受一个输入参数,返回一个布尔值结果 |
| Supplier<T> | 无参数,返回一个结果 |
| ToDoubleBiFunction<T,U> | 接受两个输入参数,返回一个 double 类型结果 |
| ToDoubleFunction<T> | 接受一个输入参数,返回一个 double 类型结果 |
| ToIntBiFunction<T,U> | 接受两个输入参数,返回一个 int 类型结果 |
| ToIntFunction<T> | 接受一个输入参数,返回一个 int 类型结果 |
| ToLongBiFunction<T,U> | 接受两个输入参数,返回一个 long 类型结果 |
| ToLongFunction<T> | 接受一个输入参数,返回一个 long 类型结果 |
| UnaryOperator<T> | 接受一个参数为类型 T,返回值类型也为 T |

**3. 函数式接口实例**

接下来以 Predicate<T>接口为例,讲解函数式接口的使用方法,实现对集合中的元素按条件进行筛选的功能。首先创建 Test2 类,并在该类中定义一个 List 集合。由表 5.4 可知,Predicate<T>接口接受一个参数,返回一个布尔值结果。接下来演示函数式接口,具体代码如例 5-37 所示。

**例 5-37** Test2.java

```
1  package chapter05;
2  import java.util.Arrays;
3  import java.util.List;
4  import java.util.function.Predicate;
5  public class Test2 {
6      public static void main(String args[]){
7          // 定义一个 list 集合
8          List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8,
9  9,10);
10         // Predicate<Integer> predicate = n -> true
11         // n 是一个参数传递到 Predicate 接口的 test 方法
12         // n 如果存在则 test 方法返回 true
13         System.out.println("输出集合中的所有数据:");
14         // 传递参数 n
15         testA(list, n -> true);
16         // Predicate<Integer> predicate1 = n -> n%2 == 0
17         // n 是一个参数传递到 Predicate 接口的 test 方法
18         // 如果 n%2 为 0 ,test 方法返回 true
19         System.out.println();
20         System.out.println("输出集合中所有大于 4 的偶数:");
21         testA(list, n -> n%2 == 0 && n>4);
22         // Predicate<Integer> predicate2 = n -> n>3
23         // n 是一个参数传递到 Predicate 接口的 test 方法
24         // 如果 n 大于 3 ,test 方法返回 true
25         System.out.println();
26         System.out.println("输出集合中大于 6 的所有数字:");
27         testA(list, n -> n>6);
28     }
29     //定义 testA 方法
30     public static void testA(List<Integer> list, Predicate<Integer>
31 predicate) {
32      //遍历集合中的元素
33     for(Integer n: list) {
34      //满足条件的打印
35     if(predicate.test(n)) {
36        System.out.print(n + " ");
37     }
38   }
39  }
40 }
```

运行以上代码,运行结果如图 5.47 所示。


图 5.47　例 5-37 运行结果

### 5.6.4　方法引用与构造器引用

当要传递给 Lambda 体的操作已经有实现的方法了，可以使用方法引用，可以理解为方法引用是 Lambda 表达式的另外一种表现形式。实现抽象方法的参数列表，必须与引用方法的参数列表保持一致。方法引用的语法：对象::实例方法，类::静态方法，类::实例方法。构造器引用的语法：ClassName::new。

第一种语法：对象::实例方法名。

```
public void TestOne() {
        Consumer<String> con = (s) -> System.out.println(s);
        Consumer<String> consumer = System.out::println;
        consumer.accept("HelloWorld!");
}
```

第二种语法：类::静态方法名。Lambda 体中调用方法的参数列表与返回值类型要与函数式接口中抽象方法的函数列表与返回值类型保持一致。

```
public void TestTwo() {
        Comparator<Integer> comparator = (m,n) ->
Integer.compare(m,n);
        Comparator<Integer> com = Integer::compare;
}
public void TestTwo() {
        Comparator<Integer> comparator = (m,n) ->
Integer.compare(m,n);
        Comparator<Integer> com = Integer::compare;
}
```

第三种语法：类::实例方法名。如果第一个参数是调用者，第二个参数是被调用者，则可以使用这种方式。

```
public void TestThree() {
        BiPredicate<String,String> bip = (x,y) -> x.equals(y);
        BiPredicate<String,String> bp = String::equals;
}
```

在了解了方法引用与构造器引用的书写格式后，下面通过案例在 Animal 类中定义了 4

个方法作为例子来区分 Java 中 4 种不同方法的引用,具体代码如例 5-38 所示。

**例 5-38** Animal.java

```
package chapter05;
import java.util.Arrays;
import java.util.List;
class Animal {
    @FunctionalInterface
    public interface Supplier<T> {
        T get();
    }
    //Supplier 是 JDK 1.8 提供的接口,这里和 Lambda 一起使用了
    public static Animal create(final Supplier<Animal> supplier)
{
        return supplier.get();
    }
    public static void collide(final Animal animal) {
        System.out.println("Collided " + animal.toString());
    }
    public void follow(final Animal another) {
        System.out.println("Following " + another.toString());
    }
    public void repair() {
        System.out.println("Repaired " + this.toString());
    }
    public static void main(String[] args) {
//构造器引用:它的语法是 Class::new,或者更一般的 Class<T>::new,实例如下:
        Animal animal = Animal.create(Animal::new);
        Animal dog = Animal.create(Animal::new);
        Animal pig = Animal.create(Animal::new);
        Animal bear = new Animal();
        List<Animal> animals = Arrays.asList(animal,dog,pig,bear);
        System.out.println("==================== 构造器引用
========================= ");
        //静态方法引用:它的语法是 Class::static_method,实例如下:
       animals.forEach(Animal::collide);
        System.out.println("==================== 静态方法引用
======================== ");
        //特定类的任意对象的方法引用:它的语法是 Class::method,实例如下:
       animals.forEach(Animal::repair);
        System.out.println("============== 特定类的任意对象的方法引用
================ ");
        //特定对象的方法引用:它的语法是 instance::method,实例如下:
        final Animal duixiang = Animal.create(Animal::new);
       animals.forEach(duixiang::follow);
        System.out.println("==================== 特定对象的方法引用
=================== ");
    }
}
```

运行以上代码,结果如图 5.48 所示。

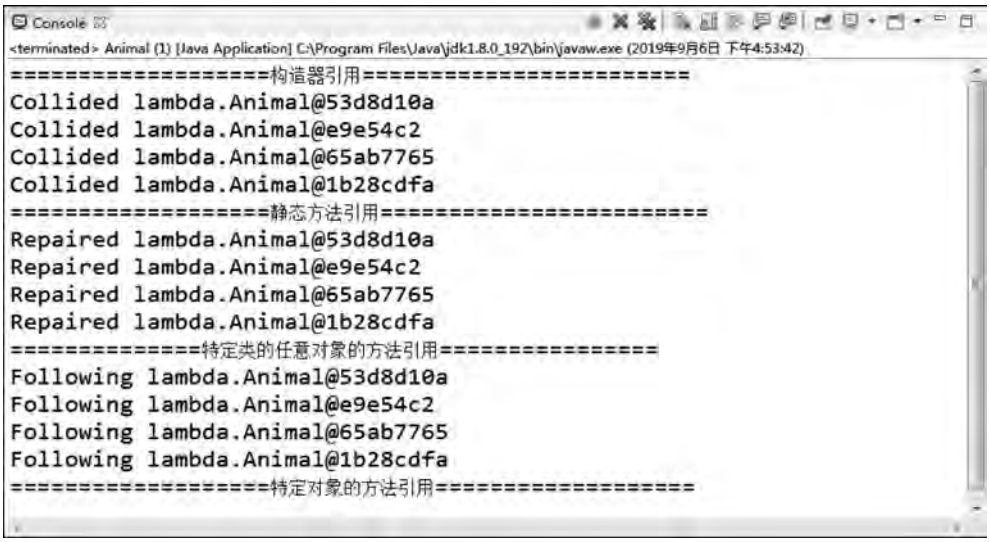


图 5.48　例 5-38 运行结果

## 小　　结

通过本章的学习，读者能够掌握 Java 面向对象的另外两大特征：继承和多态，了解 Java 8 推出的新特性 Lambda 表达式。重点要理解的是继承可以让某个类型的对象获得另一个类型的对象的属性的方法，而多态就是指一个类实例的相同方法在不同情形有不同表现形式。

## 习　　题

**1. 填空题**

(1) 如果在子类中需要访问父类的被重写方法，可以通过_____关键字来实现。

(2) Java 中使用_____关键字来表示抽象的意思。

(3) Java 中使用_____关键字来实现接口的继承。

(4) 工厂模式主要有三种形态：简单工厂模式、工厂方法模式和_____模式。

(5) Java 8 中引入了一个新的操作符，“->”可以称为箭头操作符或者_____操作符。

**2. 选择题**

(1) 以下关于 Java 语言继承的说法正确的是(　　)。

A. Java 中的类可以有多个直接父类　　B. 抽象类不能有子类

C. Java 中的接口支持多继承　　D. 最终类可以作为其他类的父类

(2) 现有两个类 A、B，以下描述中表示 B 继承自 A 的是(　　)。

A. class A extends B　　B. class B implements A

C. class A implements B　　D. class B extends A

(3) 下列选项中,用于定义接口的关键字是(　　)。

A. interface　　B. implements　　C. abstract　　D. class

(4) 下列选项中,表示数据或方法只能被本类访问的修饰符是(　　)。

A. public　　B. protected　　C. private　　D. final

(5) 在 Java 中,关于@FunctionalInterface 注解代表的含义,下列说法正确的是(　　)。

A. 主要用于编译级错误检查

B. 主要用于简化 Java 开发的工具注解

C. 检查是否含有多个非抽象方法

D. 当接口中包含默认方法时代码编译会报错

**3. 思考题**

(1) 请简述什么是继承。

(2) 请简述什么是多态,什么是动态绑定。

(3) 请简述方法的重载与重写的区别。

(4) 请简述抽象类和接口的区别。

**4. 编程题**

(1) 设计一个名为 Geometric 的几何图形的抽象类,该类包括:

① 两个名为 color、filled 的属性分别表示图形颜色和是否填充。

② 一个无参的构造方法。

③ 一个能创建指定颜色和填充值的构造方法。

④ 一个名为 getArea()的抽象方法,返回图形的面积。

⑤ 一个名为 getPerimeter()的抽象方法,返回图形的周长。

⑥ 一个名为 toString()的方法,返回圆的字符串描述。

(2) 设计一个名为 Circle 的圆类来实现 Geometric 类,该类包括:

① 一个名为 radius 的 double 属性表示半径。

② 一个无参构造方法创建圆。

③ 一个能创建指定 radius 的圆的构造方法。

④ radius 的访问器方法。

⑤ 一个名为 getArea()的方法,返回该圆的面积。

⑥ 一个名为 getPerimeter()的方法,返回圆的周长。

⑦ 一个名为 toString()的方法,返回该圆的字符串描述。

(3) 设计一个名为 Rectangle 的矩形类来实现 Geometric 类,该类包括:

① 两个名为 side1、side2 的 double 属性表示矩形的两条边。

② 一个无参构造方法创建矩形。

③ 一个能创建指定 side1 和 side2 的圆的构造方法。

④ side1 和 side2 的访问器方法。

⑤ 一个名为 getArea()的方法,返回该矩形的面积。

⑥ 一个名为 getPerimeter()的方法,返回该矩形的周长。

⑦ 一个名为 toString()的方法,返回该矩形的字符串描述。

(4) 设计一个名为 Triangle 的三角形类来实现 Geometric 类，该类包括：

① 三个名为 side1、side2 和 side3 的 double 属性表示三角形的三条边。

② 一个无参构造方法创建三角形。

③ 一个能创建指定 side1、side2 和 side3 的矩形的构造方法。

④ side1、side2 和 side3 的访问器方法。

⑤ 一个名为 getArea()的方法，返回该三角形的面积。

⑥ 一个名为 getPerimeter()的方法，返回该三角形的周长。

⑦ 一个名为 isTriangle()的方法，判断三边是否能构成三角形。

⑧ 一个名为 toString()的方法，返回三边较小的字符串描述。

(5) 编写测试类，测试图形的面积和周长。