第 5 课

深入面向对象编程

在面向对象的概念中，大家可以知道所有的对象都是通过类来描绘的，但是反过来却不是这样，并不是所有的类都是用来描绘对象的。如果一个类中没有包含足够的信息来描绘一个具体的对象，那么这样的类就是抽象类。由于 Java 语言只支持单继承，那么接口的存在就弥补了单继承的不足。内部类也是嵌套的类，方法的重载和重写是 Java 语言多态性的不同表现。本课将深入讲解面向对象编程中的这些类和方法的使用以及包的概念。

本课学习目标：

- ❑ 理解抽象类的概念，掌握其用法
- ❑ 理解接口的含义，掌握接口的实现方法
- ❑ 理解内部类和匿名类的概念，掌握其用法
- ❑ 掌握方法重载和方法重写的区别
- ❑ 理解包的概念，掌握如何创建和使用包

5.1 基础知识讲解

5.1.1 抽象类与接口

因为抽象类和接口比具体类抽象，所以使用时它们总是被继承和被实现的。不过继承它们的类不只是一个，有很多类可以实现它们的抽象方法。一个方法有多种实现方式，这里用到了类的多态性，设计会变得非常清晰。因为基类是抽象类或是接口的一个描述，继承基类的子类有若干个。开发者只需要对接口或抽象类操作，而不需管有多少个实现。下面将对它们做详细的说明。

5.1.1.1 抽象类

Java 语言中有关于抽象方法的内容，抽象方法只是一个名字而没有具体的实现。包含一个或多个抽象方法的类称为抽象类。抽象类必须使用 abstract 关键字进行声明。抽象类的使用有一些限制，比如不能创建抽象类的实例。如果子类实现了抽象方法，则可以创建该子类的实例对象。要是子类也不实现的话，这个子类也是抽象类，也不能创建实例。在 Java 语言中抽象类的语法格式如下。

```
<abstract> class <class name>{
<abstract> <type> <method_name> (parameter-list);
……
}
```

抽象类是不允许被实例化的，其使用规则如下。

- 抽象类可以不包括抽象方法，它不会去实例化，所以里面的方法是不是抽象并没有本质影响。
- 含有抽象方法的类绝不能被实例化，否则这个方法无法执行。
- 如果子类是非抽象的，那么它就必须实现父类中的抽象方法，否则它继承来的抽象方法仍然没有方法体，也是个抽象方法，此时就与“含有抽象方法的类必须是抽象类”相矛盾了。

【练习 1】

例如去银行取钱时需要户主的账户和密码，两者缺一不可。虽然每个户主的账户和密码不尽相同，但是取钱都需要经过这两个条件才能成功。使用代码完成该功能的步骤如下。

（1）定义一个抽象类，在该类中写入“账户”和“密码”这两个属性，并通过构造方法给两个变量赋值。还要编写一个抽象方法，实现取钱成功的效果，其实现代码如下。

```
package dao;
abstract class Bank {
    public String account;
    public int password;
    public Bank(String account, int password){
        this.account =account;
        this.password=password;
    }
    abstract String getMoney();
}
```

（2）然后定义一个用户类“张政”，该类继承自银行类“Bank”，并重写了 getMoney()抽象方法。其实现代码如下所示。

```
package dao;
public class Zhang extends Bank{
    public Zhang(String account,int password){
        super(account,password);
    }
    String getMoney(){
        System.out.println("张政输入了他的账户:"+account+"和密码: "+password);
        return "最后得到了 2000 元";
    }
}
```

（3）接着定义一个用户类“李明”，该类也需要继承“Bank”类，并需要重写父类中的抽象方法 getMoney()，其实现代码如下。

```
package dao;
public class Li extends Bank{
    public Li(String account,int password){
        super(account,password);
    }
    String getMoney(){
        System.out.println("李明输入了他的账户:"+account+"和密码: "+password);
        return "最后得到了 3000 元";
    }
}
```

（4）最后创建一个 Test 测试类，在 main()方法中分别创建两个用户的对象，并调用各类中的 getMoney()方法，输出各账户的信息。其实现代码如下。

```
package dao;
public class Test {
    public static void main(String[] args) {
        Zhang zhang=new Zhang("张政,123456);
        Li li=new Li("李明,654321);
        System.out.println(zhang.getMoney());
        System.out.println(li.getMoney());
    }
}
```

（5）运行该程序，运行结果如图 5-1 所示。

图 5-1 抽象类代码执行结果

5.1.1.2 接口

接口就是方法定义和常量值的集合。接口又称界面，引入接口的目的是为了克服 Java 单继承机制带来的缺陷，实现类的多继承的功能。Java 的接口在语法上类似于类的一种结构，但是接口与

类有很大的区别。它只有常量定义和方法声明，没有变量和方法的实现。

1．接口的定义

接口可以用来实现不同类之间的常量共享。接口是一种特殊的类，只定义类中方法的原型，而不是直接定义方法的内容。使用 interface 来定义一个接口。接口的定义同类的定义类似，也是分为接口的声明和接口体，其中接口体由常量定义和方法定义两部分组成。定义接口的基本格式如下。

```
[修饰符] interface <接口名> [extends 父接口]{
<public> <type> <method_name> (<parameter-list>);
< public> static final <type> <var>=value;
}
```

各参数的说明如下。

- **修饰符**　用于指定接口的访问权限，可选值为 public。如果省略则使用默认的访问权限。
- **接口名**　用于指定接口的名称，接口名必须是合法的 Java 标识符。一般情况下，要求首字母大写。
- **method_name**　表示方法名，接口中的方法只有定义而没有被实现。
- **parameter-list**　表示参数列表，在接口中的方法是没有方法体的。
- **final 和 static**　用于修饰声明的常量，即常量值不能通过实现类来改变。
- **var**　表示常量名，在接口中声明常量，必须对常量值进行初始化。
- **extends 父接口**　用于指定要定义的接口继承于哪个父类接口。接口之间通过继承也可以获得父接口中的变量和方法。当使用关键字 extends 继承时，父接口名为必选参数。

定义接口时需要注意下面介绍的几个注意事项。

- Java 接口的方法只能是抽象的和公开的。
- 接口名称通常都是以“I”开头，例如 ICourse、IStudent 等。
- Java 接口不能有构造器，Java 接口可以有 public、static 和 final 属性。
- 通常都称继承了一个类，实现了一个接口。
- 如果类已经继承了一个父类，则以逗号分隔父类和接口。
- 如果某个类需要实现多个接口，也需要使用逗号进行分隔。

【练习 2】

例如定义一个接口用于计算，在该接口中定义了一个用于计算圆周率的常量 PI，还有两个分别计算面积和周长的方法 getArea()和 getGirth()，具体代码如下所示。

```
public interface IMath {
    final double PI=3.14;
    double getArea(double r);
    double getGirth(double r);
}
```

2．接口的实现

接口在定义后就可以在类中实现该接口，在类中实现接口可以使用关键字 implements，其语法格式如下。

```
[修饰符] class <类名> [extends 父类名] [implements 接口列表]{
//主体
}
```

在类中实现接口时，方法必须声明为 public，并且方法的名字、返回值类型、参数的个数及类

型必须与接口中的完全一致，并且必须实现接口中的所有方法。

【练习 3】

下面编写一个计算圆面积和圆周长的实现类，该类用于实现“练习 2”的 IMath 接口，其示例代码如下。

```
package dao;
public class Test1 implements IMath{
    public static void main(String[] args) {
        Test1 t=new Test1();
        System.out.println("圆的面积为: "+t.getArea(4));
        System.out.println("圆的周长为: "+t.getGirth(6));
    }
    @Override
    public double getArea(double r) {
        double area=PI*r*r;
        return area;
    }
    @Override
    public double getGirth(double r) {
        double girth=2*PI*r;
        return girth;
    }
}
```

程序运行结果如图 5-2 所示。

图 5-2 接口的实现执行结果

上述代码中实现类 Test1 实现了接口 IMath，同时要实现 IMath 接口中两个未实现的方法 getArea()和 getGirth()。在 main()方法中创建实现类对象，直接调用了 IMath 接口中的两个方法。

3．接口和抽象类的区别

Java 语言中的接口和抽象类最大的区别在于 Java 抽象类可以提供某些方法的部分实现，而接口则不可以。例如向一个抽象类里加入一个新的具体方法时，那么它所有的子类都得到了这个新方法。而向一个接口里加入一个新方法后，所有实现这个接口的类就无法成功通过编译了。

在语法上抽象类和接口有着以下不同。

- 抽象类在 Java 语言中表示的是一种继承关系，一个类只能使用一次继承关系。但是一个类却可以实现多个接口。
- 继承抽象类使用的是 extends 关键字，实现接口使用的是 implements 关键字。继承写在前面，实现接口写在后面。
- 在抽象类中可以有自己的数据成员，也可以有非抽象类的成员方法。而在接口中只能够有静态的不能被修改的数据成员，所有的成员方法都是抽象的。
- 实现抽象类和接口的类必须实现其中的所有方法。抽象类中可以有非抽象方法，接口中则不能有实现方法。

- 抽象类中的变量默认是 friendly 型，其值可以在子类中重新定义，也可以重新赋值。而接口中定义的变量默认是 public static final 型，且必须赋与其初值，所以实现类中不能重新定义，也不能改变其值。

除了语法上的不同，还有抽象类的实现只能由这个抽象类的子类给出，即这个实现处在抽象类所定义出的继承的等级结构中，而由于 Java 语言的单继承性，所以抽象类作为类型定义工具的效能就有限制了。而 Java 接口弥补了这一点，任何一个实现了 Java 接口所规定方法的类都可以具有这个接口的类型，而一个类可以实现任意多个 Java 接口，从而这个类就有了多种类型。

5.1.2 内部类和匿名类

掌握 Java 语言高级编程的一部分，就是要学会使用内部类。它可以让程序的设计结构更加优雅。内部类中的匿名内部类（即匿名类），可以使代码更加简洁、紧凑，模块化程度更高。内部类能够访问外部类的一切成员变量和方法，包括私有的，而实现接口或继承类做不到。本节将详细介绍内部类和匿名类。

5.1.2.1 内部类

内部类是指在一个外部类的内部再定义一个类。内部类作为外部类的一个成员，是依附于外部类而存在的。内部类可为静态，可用 protected 和 private 修饰。

简单的内部类定义形式如下所示。

```
class Test1{
    class Test2{}
}
```

在上述代码中，在 Test1 类中创建了一个 Test2 类，这里的 Test2 类就可以称为内部类，Test1 类称为外部类。Test2 类此时是作为 Test1 类的一个成员存在的，也就说，在类中不但可以存在成员方法和成员变量，还可以存在成员类。同样也可以在方法中创建内部类。

使用内部类最大的优势在于：每个内部类都能独立地继承自一个接口的实现，所以无论外部类是否已经继承了某个接口的实现，对于内部类都没有影响。如果没有内部类提供的可以继承多个具体的或抽象的类的能力，一些设计与编程问题就很难解决。从这个角度看，内部类使得多重继承的解决方案变得完整。接口解决了部分问题，而内部类有效地实现了“多重继承”。

内部类主要有以下几类：局部内部类、静态内部类、匿名内部类。下面将着重讲解前两个内部类。

1．局部内部类

局部内部类是一个在方法中定义的内部类，它的可见范围是当前方法。和局部变量一样，局部内部类不能用访问控制修饰符及 static 修饰符来修饰。

使用局部内部类具有以下特点。

- 局部内部类只能在当前方法中使用。
- 局部内部类和实例内部类一样，不能包含静态成员。
- 在局部内部类中定义的内部类也不能被public、protected和private这些访问控制修饰符修饰。
- 局部内部类和实例内部类一样，可以访问外部类的所有成员，此外，局部内部类还可以访问所在方法中的 final 类型的参数和变量。不可以访问没有被 final 修饰的局部变量。

【练习 4】

下面通过实例来具体讲解，代码如下所示。

```
public class Test2 {
    private int a = 50;
    private int out_i = 1;
    public void f(final int b) {
        final int s = 200;
        int i = 1;
        final int j = 10;
        class Inner {
            int a = 30;
            Inner(int b) {
                inner_f(b);
            }
            int inner_i = 100;
            void inner_f(int b) {
                System.out.println(out_i);
                System.out.println(j);
                System.out.println(a);
                System.out.println(this.a);
                System.out.println(Test2.this.a);
            }
        }
        new Inner(b);
    }
    public static void main(String[] args) {
        Test2 out = new Test2();
        out.f(3);
    }
}
```

上述代码中，在无返回值的方法 f()中定义了一个局部内部类 inner，在内部类 inner 中又定义了与外部类 Test2 同名的变量 a。在静态方法 inner_f()中调用输出内容时，如果内部类没有与外部类同名的变量，在内部类中可以直接访问外部类的实例变量 out_i；也可以访问外部类的局部变量 j 用 final 修饰；a 变量是内部类中与外部类同名的变量，直接用变量名访问的是内部类的变量；同样 a 变量用 this 关键字访问的也是内部类变量，但是 a 变量用外部类名.this.的内部类变量名访问的是外部类变量。

程序运行结果如图 5-3 所示。

图 5-3 局部内部类的访问执行结果

2．静态内部类

和普通的类一样，内部类也可以有静态的。如果不需要内部类对象与其外部类对象之间有联系，那就可以将内部类声明为 static 类型，这通常称为静态内部类。

静态内部类有以下特点。

- 静态内部类的实例不会自动持有外部类的特定实例的引用，在创建内部类的实例时，不必创建外部类的实例。
- 静态内部类可以直接访问外部类的静态成员。如果访问外部类的实例成员，就必须通过外部类的实例去访问。
- 在静态内部类中可以定义静态成员和实例成员。
- 外部类可以通过完整的类名直接访问静态内部类的静态成员。

【练习 5】

下面通过实例来具体讲解静态内部类，代码如下所示。

```
public class Test3 {
    private static int i = 1;
    public static void outer_f1() {
    }
    static class Inner {
        static int inner_i = 100;
        static void inner_f1() {
            System.out.println("Outer.i" + i);
            outer_f1();
        }
    }
    public void outer_f2() {
        System.out.println(Inner.inner_i);
        Inner.inner_f1();
    }
    public static void main(String[] args) {
        new Test3().outer_f2();
    }
}
```

在上述代码中，首先在外部类 Test3 中定义一个静态内部类 inner，在静态内部类中定义了静态成员 inner_i。inner 类中只能访问外部类的静态成员（包括静态变量和静态方法），即变量 i，然后外部类通过点运算符“.”来访问内部类的静态成员，而访问内部类的非静态成员通过实例化内部类 inner。

程序运行结果如图 5-4 所示。

图 5-4　静态内部类的访问执行结果

5.1.2.2　匿名类

匿名类是没有名字的内部类，一般没办法直接引用。匿名类必须在创建时作为 new 语句的一部分来声明它们。使用这种形式的 new 语句声明一个新的匿名类，可以对一个给定的类进行扩展，或者实现一个给定的接口。它还创建那个类的一个新实例，并把它作为语句的结果返回。其语法如下

所示。

```
new <类或接口>{
//类的主体
}
```

匿名类有两种实现方式：第一种是继承一个类，重写其方法；第二种是实现一个接口（可以是多个），实现其方法。

【练习 6】

下面以继承一个类为例，编写一段代码如下。

```
public class Anonymous {
    public static void main(String args[]){
        Anonymous test=new Anonymous();
        test.show();
    }
    private void show(){
        Out anonyInter=new Out(){
            void show(){
                System.out.println("我是匿名类！");
            }
        };
        anonyInter.show();
    }
}
class Out{
    void show(){
        System.out.println("我是外部类！");
    }
}
```

程序运行的输出结果如图 5-5 所示。

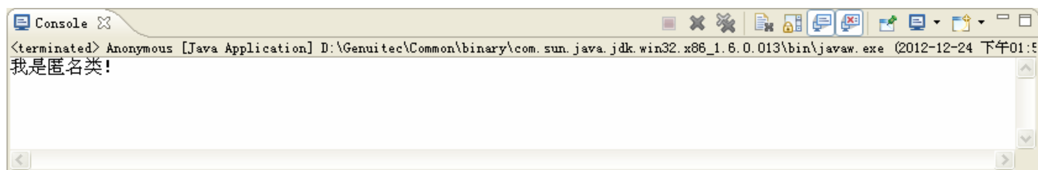

图 5-5　匿名类继承实现的执行结果

上述代码在 Anonymous 类的方法 show()中，构造了一个匿名类并实例化一个匿名类对象 anonyInter，同时重写父类的方法。在外部类中已经存在类 out，匿名类通过重写其方法，获得了另外的实现，即输出了“我是匿名类！”的结果。

使用匿名类时，只是对某个类有特殊的实现，只能在匿名类中编写其他的方法。在匿名类中编写的自己的方法是不可见的，而且此种做法毫无意义，一般不会用。在这里只是告诉初学者对于匿名内部类不要想得太多，简单地说，匿名类就是重写父类或接口的方法。

提示

接口的方式和继承相似，只要把父类换成接口就行了。

5.1.3 方法的重载和重写

继承和多态都是面向对象程序设计的特点。使用继承可以在一个父类的基础上再创建一个子类，这个子类不但拥有父类已有的属性和方法，还可以创建属于自己的属性和方法。由于子类和父类之间的关系，从而引出了方法重载和方法重写的问题。重写是父类与子类之间多态性的一种表现，重载是一个类中多态性的表现。掌握方法的重载和重写的区别为学会使用多态的方式编写程序、提高程序的可维护性奠定了基础。

5.1.3.1 方法的重载

命名转换是程序开发工程的重要部分，但当处理特别的名称时命名转换也会变得相当繁琐。简化这一过程的其中一个方法是通过重载而重新使用方法的名称。重载能够使具有相同名称但不相同数目和类型参数的类传递给方法。

下面来看一段简单的代码。

```
public class Lesson {
    private String math;
    private String english;
    Lesson(){
        math="";
        english="";
    }
}
```

该代码声明了一个名为 Lesson 的类，还有两个成员变量参数以存储课程名。分配给成员变量参数的名称就符合它们本身的含义。当调用一个 Lesson 类时，用户可以很直观地使用这些成员变量参数。

当一个重载方法被调用时，重载方法的参数列表必须和被重载的方法不同，并且这种不同必须足以清楚地确定要调用哪一种方法。而且重载方法的返回值类型可以和被重载的方法相同，也可以不同，但是只有返回值类型不同是不够的。

【练习 7】

例如公司餐厅在计算员工一天的消费情况时，消费的价格类型、总和都是不尽相同的。价格类型有零有整，每个人的消费情况都不同，结果总和也不会相同。这时候就可以使用方法的重载来解决这些问题，示例代码如下所示。

```
public class Overload {
    int n1=500;
    int n2=700;
    public int sum(){
        return n1+n2;
    }
    public int sum(int a,int b){
        return a+b;
    }
    public int sum(int i,int j,int k){
        return i+j+k;
    }
    public float sum(int m,float n){
```

```
            return m+n;
        }
        public static void main(String[] args) {
            Overload ol=new Overload();
            System.out.println("第一天的消费总额为: "+ol.sum()+"元");
            System.out.println("第二天的消费总额为: "+ol.sum(560,350)+"元");
            System.out.println("第三天的消费总额为: "+ol.sum(340,240,540)+"元");
            System.out.println("第四天的消费总额为: "+ol.sum(350,420.5f)+"元");
        }
    }
```

该程序执行结果如图 5-6 所示。

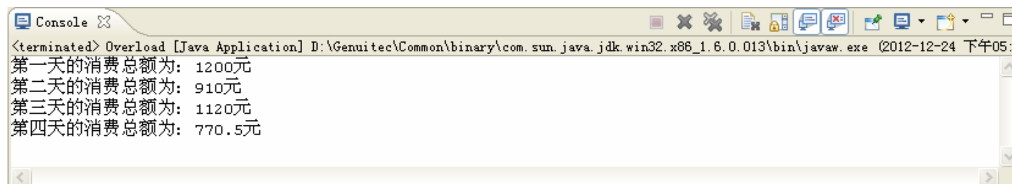

图 5-6 方法的重载执行结果

上述代码中，在几个类中定义的方法名称都相同，但是不同的是它们的参数列表和返回值类型。程序运行时，会根据参数调用不同的方法，最后输出每天不同的消费情况。

5.1.3.2 方法的重写

方法的重写是与方法的重载不同，但又容易混淆的另一个概念。方法重写是指在子类中定义的一个方法，其名称、返回值类型和参数列表正好与父类中某个方法的名称、返回值类型和参数列表相匹配，这样的情况就是子类的方法重写了父类的方法。方法的重写又可以称为方法的覆盖。

方法重写的规则如下。

- ❑ 重写方法的标志必须要和被重写方法的标志完全匹配，才能达到重写的效果。
- ❑ 重写方法的返回值必须和被重写方法的返回值一致。
- ❑ 重写方法所抛出的异常必须和被重写方法所抛出的异常一致，或者是其子类。
- ❑ 被重写方法不能为 private，否则在其子类中只是新定义了一个方法，并没有对其进行重写。

【练习 8】

例如 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。其语法格式如下。

```
public String toString()
```

下面编写一个重写 toString()的例子，实例代码如下。

```
public class Student {
    private String name="张三";
    private int age=18;
    private String sex="男";
    public String toString (){
        return "学生姓名为: "+name+"\n 年龄为: "+age+"\n 性别为: "+sex;
    }
    public static void main(String[] args) {
```

```
            Student s=new Student();
            System.out.println(s);
        }
    }
```

该程序运行结果如图 5-7 所示。

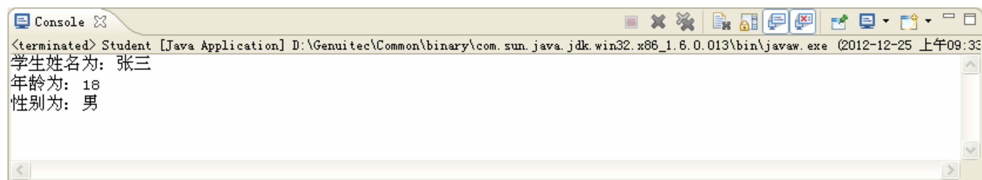

图 5-7　重写 toString()方法执行结果

上述代码中重写了方法 toString()，输出了学生对象的信息。而在平常的开发中，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()。

【练习 9】

equals()方法也是 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。该方法的语法如下。

```
public boolean equals(Object obj)
{
    return (this==obj);
}
```

下面编写一个重写 equals()方法的例子，实例代码如下。

```
public class Student1 {
    private String name;
    private int age;
    private String sex;
    public Student1(String name,int age,String sex){
        this.name=name;
        this.age=age;
        this.sex=sex;
    }
    public static void main(String[] args) {
        Student1 s1 =new Student1("王五",21,"男");
        Student1 s2=new Student1("赵六",23,"女");
        if(s1.equals(s2)){
            System.out.println("两个同学的信息相同");
        }
        else{
            System.out.println("两个同学的信息不同");
        }
    }
}
```

该程序执行结果如图 5-8 所示。

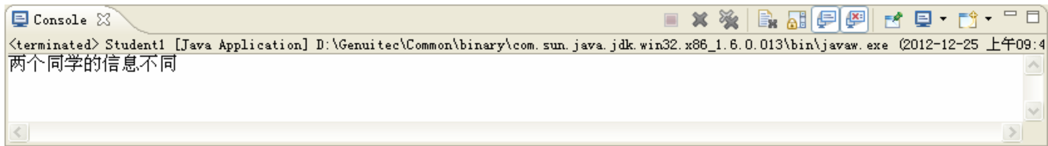

图 5-8 重写 equals()方法执行结果

以上程序将对象引用 s1 和对象引用 s2 进行了比较。对象引用就相当于内存的地址，在此处地址不同；同时对象的属性比较也不同，所以输出的内容是“两个同学的信息不同”。如果用方法 equals()比较两个对象，不能比较内容是否相等时，就是没有重写该方法。

5.1.4 super 关键字

super 表示调用父类的构造方法，但只是调用方法，不构造对象。如果子类的构造方法要引用 super 的话，必须把 super 放在方法体的首个语句，来完成子类对象的部分初始化工作。

如果想用 super 继承父类构造的方法，但是没有放在第一行的话，那么在 super 之前的语句肯定是为了想要完成某些行为，但是又用了 super 继承父类的构造方法。那么以前所做的修改就都回到以前了，就是说又成了父类的构造方法了。

super 关键字主要有以下两种用途。

1．调用父类的构造方法

子类可以调用由父类声明的构造方法。但是必须在子类的构造方法中使用 super 关键字。其具体的语法格式如下。

```
super([参数列表]);
```

如果父类的构造方法中包括参数，则参数列表为必选项，用于指定父类构造方法的入口参数。

【练习 10】

例如在 Animals 类中定义不同的构造方法，在它的子类中使用 super 关键字调用，定义 Animals 类的示例代码如下。

```
package dao;
public class Animals {
    private String skin;
    private int foot;
    public Animals(){
    }
    public Animals(String skin,int foot){
        this.skin=skin;
        this.foot=foot;
    }
    public String toString(){
        return "外表是: "+skin+"脚有: "+"只";
    }
}
```

接着在该类中新建一个子类 bird，示例代码如下。

```
package dao;
public class Bird extends Animals{
```

```
        public String birdName;
        public Bird(){
            super();
        }
        public Bird(String skin,int foot,String birdName){
            super(birdName,foot,skin);
            this.birdName=birdName;
        }
    }
```

上述代码中，在子类 bird 中使用 super 关键字直接调用父类的带参数的构造方法，使代码操作更加简便。

2．操作被隐藏的成员变量和被覆盖的成员方法

在子类中可操作被隐藏的成员变量和被覆盖的成员方法。该方法与 this 关键字相似，其基本形式如下。

```
super.member
```

其中 member 表示类中的成员方法或属性名称，表示子类的成员名隐藏了父类中的同名成员的情况。

【练习 11】

修改“练习 10”中的 Animals 类和 Bird 类，示例代码如下。

```
package dao;
public class Animals {
    public int foot;
}
```

然后在 Bird 类创建一个与父类 Animals 相同名称的变量，并在 Bird 类的构造方法中分别赋予父类的属性和本类的属性不同的值。示例代码如下。

```
package dao;
public class Bird extends Animals{
    private int foot;
    public Bird(int a,int b){
        super.foot=a;
        this.foot=b;
    }
    public void intr(){
        System.out.println("Animals 中的 foot="+super.foot);
        System.out.println("Bird 中的 foot="+this.foot);
    }
    public static void main(String[] args) {
        Bird b=new Bird(4,2);
        b.intr();
    }
}
```

该程序执行结果如图 5-9 所示。

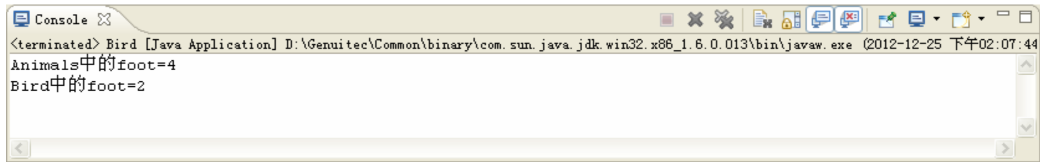

图 5-9 super 访问父类成员执行结果

该实例说明了 super 子类成员名隐藏了父类中的同名成员的情况，尽管 Bird 类中的 foot 隐藏了 Animals 类中的 foot 属性，但是 super 允许操作被隐藏的成员变量和被覆盖的成员方法，允许访问定义与父类中的属性 foot。

5.1.5 包的概念

在编写 Java 项目时，随着程序架构越来越大，类的个数会越来越多，这时就会发现管理程序中维护类名称也会是一件麻烦的事情，尤其在发生一些类重名的问题时。因此有时需要将处理同一个方面的问题的类放在同一个文件夹下，以便于管理。Java 为了解决这些问题，提供了包的机制。

包允许将类组合成较小的单元（类似文件夹），使其易于找到，同时使用相应的类文件有助于避免命名冲突。在使用许多类时，类和方法的名称很难决定，有时需要使用与其他类相同的名称。包基本上隐藏了类，并避免了名称上的冲突。包允许在更广的范围内保护类、数据和方法，可以在包内定义类，而在包外的代码不能访问该类。包将类名空间划分为更加容易管理的块，用于管理接口和类。

5.1.5.1 创建包

创建一个包，应使用关键字 package。在 Java 源程序的第一行输入 package 命令，任何在此文件中声明的类都属于这个包。包名通常是小写，而类名的第一个字母一般是大写。如果在 Java 源文件中没有使用 package 定义包名，那么所创建的类就属于默认的包（default package）。

创建包的语法格式如下。

```
package 包名 1[.包名 2][.包名 3......]
```

在 Java 中可以创建一个多层次包，包名之间使用点进行分隔即可。包名的个数没有限制，其中前面的包名包含后面的包名。

注意
创建包的语句必须是程序中的第一条可执行语句，它的前面只能有注释行或空行。创建包的语句只能有一句。

【练习 12】

下面以创建一个多层次的包为例，编写一个 Course 类，在该类中定义两个变量课程名称和课程成绩。其代码如下所示。

```
package com.school.student;
public class Course {
    private String name;
    private int grade;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
```

```
        }
        public int getGrade() {
            return grade;
        }
        public void setGrade(int grade) {
            this.grade = grade;
        }
    }
```

上述代码在 Course 类的首行创建了一个包后，Course 类就属于该包，就可以在当前目录下的路径“com\school\student”中找到该类，如图 5-10 所示。

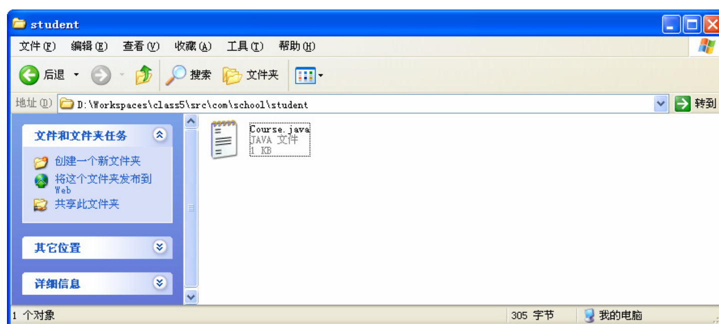

图 5-10　Course 类的路径

5.1.5.2　使用包

需要使用包时，同一个包内部的类会默认引入。如果要在其他程序中，调用包中的 Java 类需要使用 import 关键字，告知编译器所要使用的类是位于哪一个包中。

import 语句定义在 package 语句之后，任何定义类的语句之前。使用 import 引入包的语法格式如下。

```
import 包名 1[.包名 2].(类名|*);
```

其中 import 关键字后面是分层包名，接下来是要引用的类名或星号（*），星号表示 Java 编译器将引用整个包中的所有类。

【练习 13】

下面在新建包“com.school.teacher”中调用创建 Get 类，该类可以调用“练习 12”的“com.school.student”包中的 Course 类。首先创建 Get 类，然后在该类的 main()方法中直接创建 Course 类的对象并给该对象中的属性赋值。其示例代码如下所示。

```
package com.school.teacher;
public class Get {
    private Course course;
    public static void main(String[] args) {
        Course course=new Course();
        course.setName("英语课");
        course.setGrade(89);
        Get get=new Get();
        get.course=course;
        System.out.println("钟国"+get.course.getName()+"的成绩是:"+get.course.
```

```
        getGrade());
        }
}
```

运行该程序结果如图 5-11 所示。

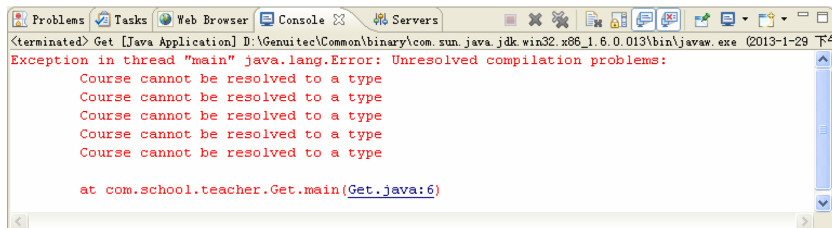

图 5-11　错误代码执行结果

上述代码中调用了 Course 类，但由于没有导入该类所在的包所以编译出现错误。下面使用 import 导入“com.school.student”包到源代码中，导入代码如下所示。

```
import com.school.student.Course;
```

再次运行程序，结果输出如下。

```
钟国英语课的成绩是: 89
```

Java 语言中提供类的许多包，有 java.lang 包、java.awt 包、java.util 包、java.io 包、java.sql 包、javax.swing 包和 java.net 包等。每个包存储的是某一个方面常用的类，如 lang 包存储的是 Java 的基本语法的类，awt 包中存储的是图形用户界面方面的包，util 包存储的是基本工具类，io 包存储的是文件方面的类，sql 包存储的是操作数据库方面的类。

5.2 实例应用：模拟公司奖励制度

5.2.1 实例目标

本课深入讲解了面向对象编程中的类和方法等的使用。而类中接口的出现，使 Java 语言弥补了单继承的不足。一个类不仅可以继承另一个类，还可以实现多个接口。本节将综合应用这些知识点来实现一个某电器公司奖励制度的案例。

假如某电器公司为鼓励员工提高销售业绩，实行多劳多得，制订了累进制的三级员工奖励。三等奖励是在一月内售出 100000～200000 元业绩的员工，奖金为其业绩的 5%；二等奖励是售出 200000～500000 元业绩的员工，奖金在加上三等奖的基础上，即扣除 100000 元的部分后，加上剩下部分的 10%；售出 500000 元以上业绩的员工，奖金在加上二等奖的基础上，即扣除 200000 元的部分后，加上剩下部分的 20%。需完成的主要功能如下。

- 计算三等奖励的金额。
- 计算二等奖励的金额。
- 计算一等奖励的金额。

5.2.2 技术分析

本案例设定根据员工销售情况的不同，分等级进行奖励。分别计算三级奖励的金额，其中与技

术相关的最主要的知识点如下所示。

- 通过声明不同的变量保存员工的销售额和对应的比率。
- 使用接口和继承关系，使类之间可以继承，并且实现多个接口。
- 使用运算符（即算术运算符）计算不同的操作。
- 使用 this 或 super 关键字调用本类或父类的方法。

5.2.3 实现步骤

实现某电器公司奖励制度的相关功能的步骤如下。

（1）创建 ICompany 接口，在该接口中定义 3 个未实现的方法，分别完成计算一、二、三等奖励的金额，其实现代码如下。

```
package dao;
public interface ICompany {
    public double firstAward();
    public double secondAward();
    public double thirdAward();
}
```

（2）创建父类奖金 MoneyAward，在该类中定义两个变量 num（销售额）和 rate（比率），并创建用来计算三等奖金的方法 thirdAward()。其代码如下所示。

```
package dao;
public class MoneyAward {
    private double num;
    private double rate;
    public MoneyAward(double num,double rate){
        this.num=num;
        this.rate=rate;
    }
    //计算三等奖金的多少
    public double thirdAward(){
        return num*rate;
    }
}
```

（3）创建一个类 SecondAward 使该类继承 MoneyAward 类，并在类中定义新的变量 num2 和 rate2，创建用来计算二等奖金的方法 second()。其代码如下所示。

```
package dao;
public class SecondAward extends MoneyAward{
    private double num2;
    private double rate2;
    public SecondAward(double num,double rate,double num2,double rate2){
        super(num,rate);
        this.num2=num2;
        this.rate2=rate2;
    }
    public double second(){
```

```
        return (num2-100000)*rate2+super.thirdAward();
    }
}
```

（4）接着创建类 FirstAward。该类继承了 SecondAward 类，并实现了 ICompany 接口。在 FirstAward 类中定义了变量 num3 和 rate3，然后实现了计算一等奖金的方法 firstAward()。其代码格式如下。

```
import dao.ICompany;
public class FirstAward extends SecondAward implements ICompany{
    private double num3;
    private double rate3;
    public FirstAward(double num3,double rate3){
        super(num3,rate3);
    }
    @Override
    public double firstAward() {
        return (num3-200000)*rate3+super.second();
    }
    @Override
    public double secondAward() {
        return 0;
    }
}
```

（5）最后创建测试类 SendMoney。创建了一个 FirstAward 类的对象 cp，用于调用间接父类和父类中的方法，并在类中定义程序的主方法 main()，分别打印输出一、二、三等奖金，代码如下所示。

```
public class SendMoney {
    public static void main(String[] args) {
        FirstAward cp = new FirstAward(150000, 0.05,300000, 0.1,750000, 0.2);
        SecondAward cp1 = new SecondAward(150000, 0.05,300000, 0.1);
        MoneyAward cp2 = new MoneyAward(150000, 0.05);
        System.out.println("三等奖励的金额为: "+cp2.thirdAward());
        System.out.println("二等奖励的金额为: "+ cp1.second());
        System.out.println("一等奖励的金额为: "+cp.firstAward());
    }
}
```

（6）该程序执行结果如图 5-12 所示。

图 5-12　员工奖励金额执行结果

5.3 拓展训练

1．实现不同食物不同味道的应用程序

编写一个程序，实现不同的食物输出不同味道的功能。首先定义一个食物接口，在该接口中定义一个未实现的方法。然后分别定义三种不同食物的类，使它们实现该食物接口和该接口的方法，使不同的食物分别打印出不同的味道。

程序运行结果如图 5-13 所示。

图 5-13　拓展训练 1 运行效果图

2．使用方法重载实现简单购车流程的应用程序

编写一个程序，实现简单购车的流程功能。首先创建一个主类，在该类中定义几个方法，这几个方法的名称相同，不同的是它们的参数列表和返回值类型。程序运行时，根据参数调用不同的方法。最后创建一个测试类对主类进行实例化，然后调用不同的方法，实现简单的购车介绍和购买过程。

程序运行结果如图 5-14 所示。

图 5-14　拓展训练 2 运行效果图

5.4 课后练习

一、填空题

1. 抽象类中不能直接创建抽象方法，而是要通过__________关键字来创建的。
2. 为了弥补 Java 单继承机制带来的缺陷，引入了新的特殊类__________。
3. 内部类能够访问外部类的一切__________。
4. super 关键字表示调用__________的构造方法，只是调用方法，不构造对象。
5. 创建一个包，应使用关键字__________。
6. 一个包创建好之后，需要在其他程序中调用该包中的 Java 类，需要关键字__________引入。

二、选择题

1. 接口被继承的类的个数为__________。
   A. 一个
   B. 一个或多个
   C. 两个
   D. 两个以上
2. 内部类主要有___________。
   A. 局部内部类、静态内部类、匿名内部类
   B. 局部内部类、静态内部类、抽象内部类
   C. 静态内部类、抽象内部类
   D. 局部内部类、抽象内部类
3. 一个重载方法被调用时，重载方法的参数列表必须和被重载的方法__________。
   A. 相同
   B. 不同
   C. 相同或不同
   D. 以上都可以
4. 使用 super 关键字继承父类的构造方法，必须把 super 放在方法体的__________语句。
   A. 首条
   B. 第二条
   C. 末条
   D. 随便一条
5. 创建一个包，需要注意包的命名规则通常是___________。
   A. 大写
   B. 首字母小写，其余字母大写
   C. 小写
   D. 首字母大写，其余字母小写

三、简答题

1. 简述抽象类和接口的区别。
2. 简述方法重载和方法重写的区别。
3. 简述 super 关键字的用途。
4. 简述包的概念，以及如何创建包和使用包。