第 5 章　接口、内部类和 Java API 基础

Java 语言的单继承机制降低了程序的复杂性，增加了系统安全性。由于在现实中子类有时需要继承多个父类的特性，因此，Java 中引入接口来实现多继承的功能。本章首先介绍接口的概念、接口的声明以及接口的实现等内容，然后介绍 Java 中其他的面向对象编程机制，包括内部类、java. lang 包和 java. util 包中常用的 Java API 以及集合类。

5.1　接　　口

5.1.1　接口的概念

Java 语言中，接口在语法上与类相似，都是由一组常量和方法组成；形式上，接口提供了一种行为框架，所提供的所有方法都是抽象的。当某个类要“继承”该接口时，类中要给出所有抽象方法的实现。因此 Java 中把对接口的“继承”称为“实现”，也就是说，接口是通过类来实现的。

一个类只能继承一个父类，同时可以实现多个接口。从这个意义上讲，借助接口机制，Java 语言实现了多继承功能。

5.1.2　接口的声明

和类一样，Java 语言中的接口也是以“包”的形式组织在一起的。接口可以是系统预定义的接口，例如 java. lang. Runnable 接口用于多线程处理，java. awt. event. ActionListener 接口用于处理命令按钮上的鼠标单击事件等。实际操作中，用户也可以自己定义接口。接口定义的语法格式如下：

```
[修饰符] interface 接口名 extends 父接口 1[,父接口 2…]
{
    常量声明
    抽象方法声明
}
```

说明：

(1) 关键字 interface 表示接口的定义。

(2) 关键字 extends 表示接口之间的继承关系，一个接口可以有多个父接口，子接口将继承父接口中的所有常量和抽象方法。

(3) 接口名为定义接口时指定的一个标识符，习惯上构成接口名的每个的单字首字母都大写，也要求尽可能“见名知义”。

(4) 接口的修饰符只有访问权限修饰符，即 public 和缺省。

① public 表示该接口为公共接口，可以被所有的类和接口使用；

② 缺省访问权限表示该接口只能被同一个包中的类和接口使用。

(5) 花括号中的部分称为接口体,接口体由常量声明和抽象方法声明两部分构成。

① 接口体中的所有常量都必须是系统默认的 public static final 修饰的常量;

② 接口体中的所有方法都必须是系统默认的 public abstract 修饰的抽象方法;

③ 无论常量和抽象方法前是否有上述默认修饰符,效果是完全相同的。

(6) Java 8 允许在接口中定义默认方法和类方法,读者可查阅相关 API 了解学习。本书不再赘述。

例如,Java 系统类库中 java.lang.Runnable 接口的定义如下:

```
public abstract interface java.lang.Runnable
{
    public abstract void run();              //线程运行代码
}
```

又如自定义一个表示“水果”的接口,代码如下:

```
public interface Fruit {
        String printName();                  //水果名称
    String printSeason();                    //成熟季节
}
```

从上面的语法可以看出,接口可理解为一种特殊的抽象类:接口和抽象类中都可以包含抽象方法且不能被实例化。但是,两者之间也存在以下几点不同:

① 接口为多个互不相关类之间的行为框架;抽象类则约定多个子类之间的共同行为。

② 一个类可以实现多个接口,为多继承机制;抽象类和子类之间只能是单继承。

③ 接口的访问权限为 public 或缺省,其成员的访问权限均为 public;抽象类和其成员的访问权限与普通类一样。

④ 接口中的方法全部是抽象方法;抽象类中则可以不包含抽象方法。

⑤ 接口中不能声明构造方法;抽象类中则可以声明构造方法。

⑥ 接口只能声明常量;抽象类中则可以声明成员变量。

5.1.3 接口的实现

接口的声明仅仅给出行为框架,需要在某个类中为接口的抽象方法定义方法体,称为类实现该接口。类实现接口的语法格式如下:

```
[修饰符] class 类名 [extends 父类名] implements 接口名 1[, 接口名 2…]
{
    …                                        //类体
}
```

例如:

```
public class Orange implements Fruit
{
    public String printName() {
```

```
        return "orange";
    }
    public String printSeason() {
        return "fall";
    }
}
```

一个类实现接口时,需要注意以下几个问题:

(1) 在用类实现接口时,类的声明部分用 implements 关键字声明该类实现哪些接口。

(2) 若实现接口的类不是 abstract 的抽象类,则在类的定义部分必须“实现”接口中的所有抽象方法,即为所有抽象方法定义方法体。

(3) 在用类实现接口中的方法时,必须使用完全相同的方法头,即有完全相同的返回值和参数列表。

(4) 因为接口中所有抽象方法的访问修饰符都默认为 public,因此在类的定义时必须显式地使用 public 修饰符,否则会出现“Cannot reduce the visibility of the inherited method”(不能降低方法的访问范围)错误。

(5) 若类要实现的接口有一个或多个父接口,则在类体中必须实现该接口及其所有父接口中的所有抽象方法。

【例 5-1】 接口的继承举例。程序代码如下:

```
1   interface InterA
2   {
3       void printA();
4   }
5   interface InterB
6   {
7       void printB();
8   }
9   //接口的继承
10  interface InterC extends InterA,InterB
11  {
12      void printC();
13  }
14  public class InterfaceInheritance5_1 implements InterC
15  {
16      //实现父接口中的抽象方法
17      public void printA()
18      {
19          System.out.println("实现父接口中的抽象方法,A------");
20      }
21      public void printB()
22      {
23          System.out.println("实现父接口中的抽象方法,B------");
24      }
```

```
25        //实现接口本身的抽象方法
26        public void printC()
27        {
28            System.out.println("实现子接口中的抽象方法,C------");
29        }
30        public static void main(String[] args)
31        {
32            InterfaceInheritance5_1 ii=new InterfaceInheritance5_1();
33            ii.printC();
34        }
35    }
```

程序运行结果如下：

```
实现子接口中的抽象方法,C------
```

程序分析如下：

由于接口 InterC 继承了接口 InterA、InterB，可以看到，第 16～24 行代码分别给出了父接口 InterA、InterB 中抽象方法 printA()、printB()的实现。类实现接口时，若接口中包含多个抽象方法，而且该类不是抽象类，则类体中必须给出所有抽象方法的实现。但是，这多个抽象方法可能不是所有方法都需要使用，如本例中仅在第 33 行代码中调用了 printC()方法。因此，对于必须给出实现又不会被调用的接口中的抽象方法，可以在实现类中给出一个空方法体的定义。例如第 16～24 行代码可以简化如下：

```
//实现父接口中的抽象方法
public void printA(){}
public void printB(){}
```

【例 5-2】 接口类型的动态绑定举例。程序代码如下：

```
1    interface Shape
2    {
3        public final static double PI=3.14159;
4        abstract double area();
5    }
6    class Circle implements Shape
7    {
8        double radius;
9        public Circle(double r)
10       {
11           radius=r;
12       }
13       public double area()              //方法实现
14       {
15           return PI * radius * radius;
16       }
17   }
```

```
18  class Rectangle implements Shape
19  {
20      double width,height;
21      public Rectangle(double w,double h)
22      {
23          width=w;
24          height=h;
25      }
26      public double area()                //方法实现
27      {
28          return width * height;
29      }
30  }
31  public class InheritancePolyphormise5_2
32  {
33      public static void main(String[] args)
34      {
35          //接口的动态绑定
36          Shape s1=new Circle(10);
37          Shape s2=new Rectangle(5,5);
38          System.out.println("圆形的面积为: "+s1.area());
39          System.out.println("长方形的面积为: "+s2.area());
40      }
41  }
```

程序运行结果如下：

```
圆形的面积为: 314.159
长方形的面积为: 25.0
```

程序分析如下：

Java 语言中，接口也可以当作一种数据类型来使用，任何实现接口的类的实例均可作为该接口的变量，并通过该变量访问类中实现的接口中的方法。第 36、37 行代码中即把 Circle 类和 Rectangle 类的对象实例赋值给了接口 Shape 的变量。程序运行时，系统动态确定应该使用哪个类中的方法。

5.1.4 常用系统接口

Java 类库中定义了不少接口，下面介绍几个常见的系统接口。

1. java.io.DataInput、java.io.DataOutput

DataInput 接口中定义了大量按照数据类型读取数据的方法。下面列举几个方法声明：

```
public abstract Boolean readBoolean();     //读入 boolean 类型数据
public abstract double readDouble();       //读入双精度类型数据
public abstract String readLine();         //读入一行数据
```

DataOutput 接口则提供大量按照数据类型写数据的方法。读者可以参阅相关文档了解这两个接口中的所有方法。

2. java. applet. AudioClip

此接口中封装有关声音播放的方法,包括如下 3 个:

```
public abstract void play();                                    //播放一遍
public abstract void loop();                                    //循环播放
public abstract void stop();                                    //停止播放
```

3. java. awt. event. ActionListener

Java 中,凡是要处理 ActionEvent 事件的监听者都必须实现 ActionListener 接口,如鼠标单击按钮的操作。该接口中声明的抽象方法为:

```
public abstract void actionPerformed(java.awt.event.ActionEvent arg0);
```

4. java. sql. Connection 接口

该接口表示与一个特定数据库的会话。下面列举几个该接口中的方法:

```
//不带参数的 SQL 语句通常用 Statement 对象执行,该方法用于产生 Statement 对象
public abstract java. sql. Statement createStatement ( ) throws java. sql.
SQLException;
public abstract void commit() throws java.sql.SQLException;       //提交更改
public abstract void close() throws java.sql.SQLException;        //关闭数据库连接
```

5.2 内部类和内部接口

5.2.1 内部类和内部接口的概念

Java 中允许在类或接口的内部声明其他的类或接口,其中被包含的类或接口称为内部类、内部接口,包含内部类、内部接口的类或接口称为外部类、外部接口。内部类、内部接口既具有类的特性,同时也作为类的成员存在。

作为类,内部类和内部接口具有以下特性。

(1) 内部类、内部接口不能与所在的外部类、外部接口同名。

(2) 内部类中可以声明成员变量和成员方法。

(3) 内部类可以继承父类或实现接口。

(4) 内部类可以声明为抽象类。

作为类的成员,内部类和内部接口在使用时有以下特点。

(1) 内部类、内部接口的访问需使用成员运算符。

(2) 内部类、内部接口可以直接访问外部类、外部接口的所有成员。

(3) 内部类在定义时可以使用 4 种访问控制权限修饰符。

(4) 内部接口只能是静态的,内部类可以是静态的。

以内部类为例,根据内部类声明的位置和形式不同,可将内部类区分为以下几种形式。

(1) 非静态内部类:内部类声明时没有指定 static 关键字,作为类的成员定义。

(2) 局部方法内部类:在方法内部声明的内部类,不是类的成员。

(3) 匿名内部类：内部类声明时可以不指定类名，不属于类成员。

(4) 静态内部类：内部类声明时指定为 static 属性，是类的成员。

因为把内部类隐藏在外部类之内，所以内部类提供了更好的封装，不允许同一包中的其他类访问该类。大部分时候，内部类都被作为成员内部类定义，分为非静态内部类和静态内部类。下面将分非静态内部类和静态内部类分别介绍内部类的定义和使用。

5.2.2 内部类的定义和使用

【例 5-3】 非静态内部类的定义和使用举例。程序代码如下：

```
1   class Line
2   {
3       Point p1,p2;
4       class Point
5       {
6           int x,y;
7           Point(int x,int y)
8           {
9               this.x=x;
10              this.y=y;
11          }
12          public void printXY()
13          {
14              System.out.print("("+x+","+y+")");
15          }
16      }
17  }
18  public class InnerClassDemo5_3
19  {
20      public static void main(String[] args)
21      {
22          Line line=new Line();
23          line.p1=line.new Point(2,10);
24          line.p2=line.new Point(3,20);
25          System.out.print("两点一线：");
26          line.p1.printXY();
27          System.out.print("-----");
28          line.p2.printXY();
29      }
30  }
```

程序运行结果如下：

```
两点一线：(2,10)-----(3,20)
```

程序分析如下：

第 23、24 行代码分别使用内部类中的构造方法创建了对象并赋值给成员变量 p1、p2；

第 26、28 行代码调用内部类的方法输出点信息。

注意：内部类编译后会生成的字节码文件为 Line$Point.class。

【例 5-4】 非静态内部类访问外部类的成员变量举例。程序代码如下：

```
1    class OutClass
2    {
3        private String str="str:外部类的私有成员变量";
4        private int x=20;
5        class InClass
6        {
7            private String str="str:内部类的私有成员变量";
8            private int y=30;
9            public void info()
10           {
11               String str="str:内部类成员方法的局部变量";
12               System.out.print(x+"\t");          //访问外部类中的私有成员变量
13               System.out.println(y);             //访问内部类中的成员变量
14               System.out.println(str);           //访问局部变量
15               System.out.println(this.str);      //访问内部类中同名的成员变量
16               System.out.println(OutClass.this.str);
                                                    //访问外部类中的同名的成员变量
17           }
18       }
19       public void info()
20       {
21           System.out.print(x+"\t");              //访问本类的成员变量
22           System.out.println(new InClass().y);   //访问内部类的成员变量
23           System.out.println("--------");
24           new InClass().info();                  //调用内部类的成员方法
25       }
26   }
27   public class InnerClassDemo5_4
28   {
29       public static void main(String[] args)
30       {
31           new OutClass().info();
32       }
33   }
```

程序运行结果如下：

```
20  30
--------
20  30
str: 内部类成员方法的局部变量
str: 内部类的私有成员变量
```

str：外部类的私有成员变量

程序分析如下：

非静态内部类的成员可以访问外部类的 private 成员，但反过来不可以。第 12～16 行代码在内部类的成员方法中显示输出了外部类成员变量 x、内部成员变量 y，以及同名的局部变量 str、内部成员变量 str、外部成员变量 str；第 21～22 行代码是在外部类的成员方法中访问 x、y；第 24 行代码是在外部类的成员方法中调用内部类的成员方法。

注意：外部类访问非静态内部类的成员时必须显示创建非静态内部类的对象。

思考：不允许在外部类的静态成员中创建和使用非静态内部类的对象。为什么？

【例 5-5】 静态内部类举例。程序代码如下：

```
1    class NonStaticOutClass
2    {
3        private String str="outer";
4        private static int x=20;
5        static class InClass
6        {
7            private static int y=30;
8            public void info()
9            {
10               System.out.println(x);       //访问外部类的静态成员变量
11               System.out.println(y);       //访问内部类中的静态成员变量
12               //System.out.println(str);  //不能访问外部类的实例成员变量
13           }
14       }
15       public void info()
16       {
17           System.out.println(x);
18           System.out.println(InClass.y);  //外部类中访问静态内部类的静态成员变量
19           System.out.println("-------");
20           new InClass().info();           //外部类中调用静态内部类的实例成员方法
21       }
22   }
23   public class StaticInnerClass5_5
24   {
25       public static void main(String[] args)
26       {
27           new NonStaticOutClass().info();
28       }
29   }
```

程序运行结果如下：

```
20
30
-------
```

```
20
30
```

程序分析如下：

由于静态成员不能访问非静态成员，静态内部类中不能访问外部类的实例成员。第 12 行代码如果执行经出现错误。另外，静态内部类可以包含非静态成员。第 18、20 行代码分别体现了如何在外部类中访问静态内部类的静态成员和非静态成员。

5.3 java.lang 包中的基础类

java.lang 包是 Java 中自动导入的包，该包为 Java 的核心包，包含利用 Java 语言编程的基础类。如 Object 类、System 类等。

5.3.1 Object 类

Object 类为 Java 中类树状结构中的根类，为所有类的直接或间接父类。因此，所有类的对象都拥有 Object 类的方法。Object 类中的常用方法见表 5-1。

表 5-1 Object 类中的常用方法

方　法	说　明
protected Object clone()	生成当前对象的一个副本
public boolean equals(Object arg0)	比较对象是否相等
public String toString()	将对象转换成字符串
protected void finalize()	销毁对象，析构方法
public final Class getClass()	获取当前对象的所属类信息

【例 5-6】 Object 类中的方法举例。程序代码如下：

```
1    public class ObjectMethodDemo5_6
2    {
3        public static void main(String[] args)
4        {
5            Object ob1="Hello Java";          //子类对象赋值给父类变量
6            Object ob2=new Object();
7            ob2=ob1;                          //引用对象的赋值
8            System.out.println(ob2.equals(ob1));
9            System.out.println(ob2.toString());
10           System.out.println(ob2.getClass());
11       }
12   }
```

程序运行结果如下：

```
true
```