第四章 接口与多态

接口在语法上与抽象类类似，它声明了若干**抽象方法**和**常量**，其主要作用是帮助实现类的多重继承功能。

多态性是面向对象设计语言的重要特性之一。

4.1 接口

接口允许创建者规定一个类的基本形式：方法名、自变量列表以及返回类型，但不规定方法主题。接口也包含了数据成员，但默认都为 static 和 final。接口只提供一种形式并不提供具体的细节。

4.1.1 接口的作用及语法

1 接口的作用

Java 的接口也是面向对象的一个重要机制。它的引进是为了实现多继承，同时免除 C++中的多继承那样的复杂性。接口中的所有方法都是抽象的，这些抽象方法由实现这一接口的不同类来具体完成。

Java 可以建立类与类之间的“协议”。将类根据其实现的功能分组，用接口代表，而不必顾虑它所在的类继承层次；这样可以最大限度地利用动态绑定，隐藏实现细节。

还可以实现常量的共享。

如图 1 所示。Java 还可以在看起来不相干的对象之间定义共同的行为。

图 1

2 接口的语法

声明格式如下：

[接口修饰符] interface 接口名称 [extends 父接口名]{

..//方法的**原型声明**或者**静态变量**

}

接口与一般类一样，本身也有数据成员与方法，但数据成员一定要赋予初值，且不可修改（因为是常量），此时 final 可以省略。方法必须是“抽象方法”，public 和 abstract 也可以省略。

例：声明一个接口 Shape2D，包括π和计算面积的方法原型：

```
interface Shape2D{//声明 Shape2D 接口
    final double pi =3.14;//数据成员一定要初始化
    public abstract double area();//抽象方法
}
/*如前文所讲，在接口声明中，我们可以省略 final,
public,abstract 等关键字。因此可以修改为以下格式：*/
interface Shape2D{
    double pi = 3.14;
    double area();
}
```

接口不能实例化，即不可能 new 一个出来。

4.1.2 实现接口

接口的使用

利用接口设计类的过程，被称为接口的实现，使用 implements 关键字，语法如下：

```
public class 类名称 implements 接口名称{
  //在类体中实现接口的方法
  //本来声明的更多变量和方法
```

```
}
```
注意：

（1）必须实现接口中的所有方法；

（2）来自接口的方法必须是 public。

例：实现接口 Shape2D

```
//接上面的接口声明
class Circle implements Shape2D{
    double radius;
    public Circle(double r
    {
        radius = r;
    }
    public double area(){
        return(pi*radius*radius;
    }
}
class Rectangle implements Shape2D{
    int width,height;
    public Rectangle(int w, int h){
        width = w;
        height = hl
    }
    public double area(){
        return(width*height);
    }
}
```
测试一下：

```
public class InterfaceTester{
    public static void main(String args[]){
        Rectangle rect = new Rectangle(5,6);
        System.out.println("Area of rect = " + rect.area());
        Circle cir = new Circle(2.0);
        System.out.println("Area of cir = " + cir.area());
    }
}
```
结果为

```
Area of rect = 30.0
Area of cir =12.56
```
例：接口类型的引用变量:

```
public class InterfaceTester{
    public static void main(String args[]){
        Shape2D var1,var2;
        var1 = new Rectangle(5,6);
        System.out.println("Area of rect = " + var1.area());
```

```
            var2 = new Circle(2.0);
            System.out.println("Area of cir = " + var2.area());
        }
}
```

4.1.3 多重继承

一个类可以实现多个接口，通过这种机制可以实现对设计的多重继承。

实现多个接口的语法如下：

[类修饰符] class 类名 implements 接口 1,接口 2,…{……}

4.1.4 接口的扩展

已有的接口称为超接口，父接口，基本接口等，扩展出来的接口为子接口。

接口扩展的语法：

interface 子接口的名称 extends 超口 1,超口 2,…{……}

说明：

（1）首先声明父接口，然后声明其子接口；

（2）之后声明类实现子接口，因此必须在此类内明确定义子接口中抽象方法的处理方式；

（3）最后在主类中我们声明了类型的变量并创建对象，最后通过对象调用那些方法。

4.2 塑型（类型转换）

4.2.1~4.2.2 类型转换

1.转换方式：**隐式**的类型转换和**显式**的类型转换。

2.从方向来看：**向上**转型和**向下**转型。

3.类型转换规则：

（1）基本类型转换：将值一种类型转换成另一种类型。

（2）引用变量的转换：将引用转换向另一类型的引用，并不改变对象本身的类型。

它只能被转换为：

任意一个（直接或间接）超类的类型（向上转型）；

对象所属的类（或其超类）实现的一个接口（向上转型）；

被转为引用指向的对象的类型（唯一可以向下转型的情况）。

（3）当一个引用被转为其超类引用后，通过他能够访问的只有在超类中声明过的方法。

如图 2 所示：

图 2

我们看这里定义了一个 person 类，隐含的继承或者扩展了 Object 类，然后 Employee 雇员类继承了 person 类，customer 顾客类也继承了 person 类，manager 继承了雇员类，person 实现了 insurable 接口，另外 company，car 都继承了 insurable 接口。所以把这一组可以被保险的对外接口规定在 insurable 这个接口里面。

下面来看，manager 类型的引用可以被转换成什么呢？按照继承的层次，它可以被转型为 employee 雇员类，子类对象总是可以充当超类对象用的；还可以转换成 person 类型和 object 类型。由于 person 实现了 insurable 接口，所以 manager 课转型为 insurable 接口。由于没有继承关系，也不是实现接口的关系，manager 又不能被塑型为 customer 还有 company 或者 car。

1.隐式转换（自动转换）

基本数据类型：可以转换的类型直接，存储容量低自动向高的类型转换。

引用变量：被转成更一般的类（向上的超类）。例如：

```
Employee emp;
emp = new Manager();//将 manager 类型对象直接赋
```

//给 Employee 类的饮用对象，系统会自动将 manage 转换
//为 employee 类

2.显式类型转换

引用变量
Employee emp;
Manager man;
emp = new Manager();
man = (Manager)emp;//将 emp 显式转换为它指向的对象的类型
类型转换的主要应用场合：
（1）赋值转换：赋值运算符右边的表达式或对象类型转换为左边的类型；
（2）方法调用转换：实参的类型转换为形参的类型；
（3）算术表达式转换：算术混合运算时，不同类型的操作数转换为相同的类型再进行运算；
（4）字符串转换：字符串连接运算时，如果一个操作数为字符串，另一个操作数为其他类型，则会自动将其他类型转换为字符串。

4.2.3 查找方法

如果转换前后两个类都提供了相同方法，那么系统会调用哪个类的方法呢？

1. 实例方法查找

我们通过图 3 来看一下实例方法的查找。实例方法是非静态方法。对于实例方法，从对象创建时的类开始，沿类层次向上查找。例如，

Manager man = new Manager();
Employee emp1 = new Employee();
Employee emp2 = (Employee)man;
emp1.computePay();//调用 employee 类中的 computePay()中的方法
man.computePay();//调用 manager 类中的 computePay()方法；
emp2.computePay();//调用 manager 类中的 computePay()方法。

图 3

2.类方法查找

这里，我们以图 4 为例来讲解。对于类方法，查找在编译时进行，所以总是在变量声明时所属的类中进行查找。

例如：

Manager man = new Manager();
Employee emp1 = new Employee();
Employee emp2 = (Employee)man;
Manager.expenseAllowance();//in manager
man. .expenseAllowance();//in manager
Employee.expenseAllowance();//in employee
emp1.expenseAllowance();//in employee
emp2.expenseAllowance();//in employee 因为总是在变量声明时
//所属的类中进行查找。emp2 声明的是 employee 类

图 4

4.3 多态的概念

超类对象和从相同的超类派生出来的多个子类的对象，可被当作同一种类型的对象对待；

实现统一接口的不同类型对象，也可以被当作同一种类型的对象对待；

可向这些不同类型的对象发送同样的消息，由于多态性，这些不同类的对象响应同一消息时的行为可能有所差别。

多态的目的：

（1）使代码变得更简单且容易理解；

（2）使程序具有很好的可扩展性。

接下来结合一个例子来进行介绍。如图 5 所示。我们来看这个继承关系图，最上面的这个和超类 shape 类里面，声明了一个绘图方法 draw()还有一个擦出方法 erase()。子类 circle、square、triangle 中都覆盖了这两个方法。以后绘图可以如下进行：

Shape s =new Circle();

s.draw();//实际调用的是 circle 对象的 draw()

绑定就是将一个方法调用表达式与方法体的代码结合起来。它分为：

早绑定：程序运行之前执行绑定（编译时）

晚绑定：基于对象的类别，在程序运行时执行。又被称为动态绑定或者运行绑定。若一种语言实现了后期绑定，同时必须提供一些机制，课在运行期间判断对象的类型，并分别调用适当的方法。

4.4 多态的应用举例

例子：二次分发

有不同种类的交通工具（vehicle），如公共汽车（bus）及小汽车（car），由此可声明一个抽象类 Vehicle 和两个子类 Bus，Car。

声明一个抽象类 Driver 和两个子类 FemaleDriver 及 MaleDriver；

在 Driver 中声明抽象方法 drives，在两个子类中对这个方法进行覆盖；

drives 接受一个 Vehicle 类的参数，当不同类型的交通工具被传送到此方法时，可以输出具体的交通工具；

所有的类放在 drive 包中。

希望在输入测试代码时得到的结果如图 6 所示。

具体代码详见课本。

二次分发即对输出消息的请求被分发两次，首先根据驾驶员的类型发送给一个类，之后根据交通工具的类型发送给另一个类。

5.5 构造方法与多态

构造方法不具有多态性。

构造子类对象时构造方法的调用顺序：

首先是调用超类的构造方法；这个步骤会不断重复下去，首先被执行的是最远超类的构造方法；

执行当前子类对象的构造方法体的其它语句。

注意：**子类不能直接存取父类中声明的私有数据成员。**所以在该类方法中要调用它超类的 toString 时用 super.toString。

如果构造方法中调用多态方法会发生什么呢？

会造成一些难以发现的程序错误。从概念上讲，构造方法的职责是让对象实际进入存在状态。在任何构造方法内部，整个对象可能只是得到部分初始化，但却不知道哪些类已经继承。然而，一个动态绑定的方法调用却会调用位于派生类里的一个方法。如果在构造方法内部做这件事情，那么对于调用方法，它要操纵的成员可能尚未得到正确的初始化，就会造成一些难以发现的程序错误。

实现构造方法的注意事项：

（1）用尽可能少的动作把对象的状态设置好。构造方法就是用来初始化的，别干别的事情。

（2）如果可以避免不要调用任何方法。

（3）在构造方法内唯一能够安全调用的是在基类中具有 final 属性的哪些方法（private 方法也可以，因为它们自动具有 final 属性）。这些方法不能被覆盖，所以不会出现问题。