第 3 章　继承、接口与多态

求职者在介绍自己的基本情况时不必“从头说起”，比如，不必介绍自己所具有的人的一般属性等，因为人们已经知道求职者肯定是一个人，已经具有了人的一般属性，求职者只要介绍自己独有的属性就可以了。

当准备编写一个类的时候，发现某个类有我们所需要的成员变量和方法，如果想复用这个类中的成员变量和方法，即在所编写的类中不用声明成员变量就相当有了这个成员变量，不用定义方法就相当有了这个方法，那么可以将编写的类声明为这个类的子类，子类通过继承可以不必一切“从头做起”。

3.1　子类与父类

继承是一种由已有的类定义出新类的机制。利用继承，可以先定义一个共有属性的一般类，根据该一般类再定义具有特殊属性的子类，子类继承一般类的属性和行为，并根据需要增加它自己的新的属性和行为。

由继承而得到的类称为子类，被继承的类称为父类（超类）。需要读者特别注意的是 Java 不支持多重继承，即子类只能有一个父类（和 C++不同）。人们习惯地称子类与父类的关系是 is-a 关系。

在类的声明中，通过使用关键字 extends 定义一个类的子类，格式如下：

```
class 子类名 extends 父类名 {
⋮
}
```

例如：

```
class Student extends People {
⋮
}
```

把 Student 类定义为 People 类的子类，People 类是 Student 类的父类（超类）。人们习惯地称子类与父类的关系是 is-a 关系（“学生是一个人”是正确的说法）。

如果一个类是另一个类的子类，那么 UML 通过使用一个实线连接两个类的 UML 图来表示二者之间的继承关系，实线的起始端是子类的 UML 图，终点端是父类的 UML 图，但终点端使用一个空心的三角形表示实线的结束。图 3.1 是 Student 类和 People 类之间的继承关系的 UML 图。

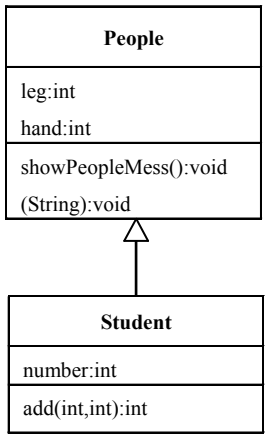

图 3.1　继承关系的 UML 图

如果 C 是 B 的子类，B 又是 A 的子类，习惯上称 C 是 A 的子孙类。Java 的类按继承关系形成树形结构（将类看做树上的结点），在这个树形结构中，根结点是 Object 类（Object 是 java.lang 包中的类），即 Object 是所有类的祖先类。任何类都是 Object 类的子孙类，每个类（除了 Object 类）有且仅有一个父类，一个类可以有多个或零个子类。如果一个类（除了 Object 类）的声明中没有使用 extends 关键字，这个类被系统默认为是 Object 的子类，即类声明 class A 与 class A extends Object 是等同的。

3.2 子类的继承性

1．继承性

类可以有两种重要的成员：成员变量和方法。子类的成员中有一部分是子类自己声明、定义的，另一部分是从它的父类继承的。那么，什么叫继承呢？所谓子类继承父类的成员变量作为自己的一个成员变量，就好像它们是在子类中直接声明一样，可以被子类中自己定义的任何实例方法操作，也就是说，一个子类继承的成员应当是这个类的完全意义的成员，如果子类中定义的实例方法不能操作父类的某个成员变量，该成员变量就没有被子类继承；所谓子类继承父类的方法作为子类中的一个方法，就好像它们是在子类中直接定义了一样，可以被子类中自己定义的任何实例方法调用。

如果子类和父类在同一个包中，那么子类自然地继承了其父类中不是 private 的成员变量作为自己的成员变量，并且也自然地继承了父类中不是 private 的方法作为自己的方法，继承的成员变量或方法的访问权限保持不变。

当子类和父类不在同一个包中时，父类中的 private 和友好访问权限的成员变量不会被子类继承，也就是说，子类只继承父类中的 protected 和 public 访问权限的成员变量作为子类的成员变量；同样，子类只继承父类中的 protected 和 public 访问权限的方法作为子类的方法。

2．耦合关系

类与类之间一旦确定是父子关系，那么这种关系就是永久的，不会再发生变化（就像生活中儿子与父亲的关系是永久的、不可改变的），而且父类对方法的修改都会影响到子类，这也是称子类与父类的关系是 is-a 关系的原因。因此，面向对象程序将子类与父类的关系看做强耦合关系（组合关系是弱耦合关系，见后续的第 4 章）。

3.3 关于 protected 的进一步说明

一个类 A 中的 protected 成员变量和方法可以被它的子孙类继承，比如 B 是 A 的子类，C 是 B 的子类，D 又是 C 的子类，那么 B、C 和 D 类都继承了 A 类的 protected 成员变量和方法。在没有讲述子类之前，曾对访问修饰符 protected 进行了讲解，现在需要对 protected 总结的更全面些。如果用 D 类在 D 本身中创建了一个对象，那么该对象总是可以通过.运算符访问继承的或自己定义的 protected 变量和 protected 方法的，但是如果在另外一个类中，比如在 Other 类中用 D 类创建了一个对象 object，该对象通过.运算符访问 protected 变量和

protected 方法的权限如下所述。

（a）对于子类 D 自己声明的 protected 成员变量和方法，只要 Other 类和 D 类在同一个包中，object 就可以访问这些 protected 成员变量和方法。

（b）对于子类 D 从父类继承的 protected 成员变量或 protected 方法，需要追溯到这些 protected 成员变量或方法所在的“祖先”类，比如可能是 A 类，只要 Other 类和 A 类在同一个包中，object 对象能访问继承的 protected 变量和 protected 方法。

3.4 子类对象的特点

当用子类的构造方法创建一个子类的对象时，不仅子类中声明的成员变量被分配了内存，而且父类的成员变量也都分配了内存空间（技术细节见后面的 3.5 节），但只将其中一部分，即子类继承的那部分成员变量，作为分配给子类对象的变量。也就是说，父类中的 private 成员变量尽管分配了内存空间，也不作为子类对象的变量，即子类不继承父类的私有成员变量。同样，如果子类和父类不在同一包中，尽管父类的友好成员变量分配了内存空间，但也不作为子类对象的变量，即如果子类和父类不在同一包中，子类不继承父类的友好成员变量。

通过上面的讨论，我们有这样的感觉：子类创建对象时似乎浪费了一些内存，因为当用子类创建对象时，父类的成员变量也都分配了内存空间，但只将其中一部分作为分配给子类对象的变量，比如，父类中的 private 成员变量尽管分配了内存空间，也不作为子类对象的变量，当然它们也不是父类某个对象的变量，因为我们根本就没有使用父类创建任何对象。这部分内存似乎成了垃圾一样。但是，实际情况并非如此，我们需注意到，子类中还有一部分方法是从父类继承的，这部分方法却可以操作这部分未继承的变量。

在下面的代码中，子类 ChinaPeople 的对象调用继承的方法操作未被子类继承却分配了内存空间的变量。程序运行效果如图 3.2 所示。

People.java

```
public class People {
   private int averHeight=166;
   public int getAverHeight() {
      return averHeight;
   }
}
```

```
子类对象未继承的averageHeight的值是:166
子类对象的实例变量height的值是:178
```

图 3.2 对象调用继承的方法

ChinaPeople.java

```
public class ChinaPeople extends People {
    int height;
    public void setHeight(int h) {
        height=h;
    }
    public int getHeight() {
       return height;
```

```
    }
}
```

Application.java

```
public class Application {
  public static void main(String args[]) {
      ChinaPeople zhangSan = new ChinaPeople();
      System.out.println("子类对象未继承的averageHeight的值是:"+
      zhangSan.getAverHeight());
      zhangSan.setHeight(178);
      System.out.println("子类对象的实例变量height的值是:"+
      zhangSan.getHeight());
  }
}
```

3.5 隐藏继承的成员

定义子类时，仍然可以声明成员变量，一种特殊的情况就是，所声明的成员变量的名字和从父类继承来的成员变量的名字相同（声明的类型可以不同），在这种情况下，子类就会隐藏掉所继承的成员变量。

子类隐藏继承的成员变量的特点如下：

（1）子类对象以及子类自己定义的方法操作与父类同名的成员变量是指子类重新声明的这个成员变量。

（2）子类对象仍然可以调用从父类继承的方法操作被子类隐藏的成员变量，也就是说，子类继承的方法所操作的成员变量一定是被子类继承或隐藏的成员变量。

（3）子类继承的方法只能操作子类继承和隐藏的成员变量。子类新定义的方法可以操作子类继承和子类新声明的成员变量，但无法操作子类隐藏的成员变量。

下面的代码演示货物价格的计算。父类在按重量计算货物的价格时，重量的计算精度是 double 型，对客户的优惠程度较小。子类在按重量计算货物的价格时，重量的计算精度是 int 型，对客户的优惠程度较大。代码中，父类 Goods 有一个名字为 weight 的 double 型成员变量，子类 CheapGoods 本来可以继承这个成员变量，但是子类 CheapGoods 又重新声明了一个 int 型的名字为 weight 的成员变量，这样就隐藏了继承的 double 型的名字为 weight 的成员变量。但是，子类对象可以调用从父类继承的方法操作隐藏的 double 型成员变量，按照 double 型重量计算价格，子类对象可以调用子类新定义的方法操作新声明的 int 型成员变量，按照 int 型重量计算价格。程序运行效果如图 3.3 所示。

```
对象cheapGoods的weight的值是:198
cheapGoods用子类新增的优惠方法计算价格：1980.0
cheapGoods使用继承的方法（无优惠）计算价格：1989.87
```

图 3.3 隐藏成员变量

Goods.java

```
public class Goods {
   public double weight;
   public void setDoubleWeight(double w) {
```

```
        weight=w;
    }
    public double getPrice() {
        double price=weight*10;
        return price;
    }
}
```

CheapGoods.java

```
public class CheapGoods extends Goods {
  public int weight;
    public void setIntWeight(int w) {
       weight=w;
    }
    public double getCheapPrice() {
        double price=weight*10;
        return price;
    }
}
```

Application.java

```
public class Application {
  public static void main(String args[]) {
    CheapGoods cheapGoods=new CheapGoods();
     cheapGoods.setIntWeight(198);
     System.out.println("对象cheapGoods的weight的值是:"+
     cheapGoods.weight);
     System.out.println("cheapGoods用子类新增的优惠方法计算价格: "+
                  cheapGoods.getCheapPrice());
     cheapGoods.setDoubleWeight(198.987);
                        //调用继承的方法操作隐藏的double型变量weight
     System.out.println("cheapGoods使用继承的方法（无优惠）计算价格: "+
                  cheapGoods.getPrice());
  }
}
```

3.6 通过重写实现多态

子类通过重写可以隐藏已继承的方法（方法重写称为方法覆盖（method overriding））。

1．重写的语法规则

如果子类可以继承父类的某个方法，那么子类就有权利重写这个方法。方法重写是指子类中定义一个方法，这个方法的类型和父类的方法的类型一致或者是父类的方法的类型的子类型（所谓子类型，是指如果父类的方法的类型是“类”，那么允许子类的重写方法的

类型是“子类”），并且这个方法的名字、参数个数、参数的类型和父类的方法完全相同。子类如此定义的方法称做子类重写的方法（不属于新增的方法）。

2．重写与多态

多态性就是指父类的某个方法被其子类重写时，可以各自产生自己的行为功能。子类通过方法的重写可以隐藏继承的方法，子类通过方法的重写可以把父类的状态和行为改变为自身的状态和行为。如果父类的方法 f()可以被子类继承，子类就有权利重写 f()，一旦子类重写了父类的方法 f()，就隐藏了继承的方法 f()，那么子类对象调用方法 f()一定调用的是重写方法 f()；如果子类没有重写，而是继承了父类的方法 f()，那么子类创建的对象当然可以调用 f()方法，只不过方法 f()产生的行为和父类的相同而已。

重写方法既可以操作继承的成员变量、调用继承的方法，也可以操作子类新声明的成员变量、调用新定义的其他方法，但无法操作被子类隐藏的成员变量和方法。

3.7　上转型对象体现多态

人们经常说“老虎是哺乳动物”、“狗是哺乳动物”等。若哺乳类是老虎类的父类，这样说当然正确，因为人们习惯地称子类与父类的关系是 is-a 关系。但需要注意的是，当说老虎是哺乳动物时，老虎将失掉老虎独有的属性和功能。从人的思维方式上看，说“老虎是哺乳动物”属于上溯思维方式。

1．上转型对象

假设 A 类是 B 类的父类，当用子类创建一个对象，并把这个对象的引用放到父类的对象中时，例如：

```
A a;
a=new B();
```

或

```
A a;
B b=new B();
a=b;
```

这时，称对象 a 是对象 b 的上转型对象（好比说“老虎是哺乳动物”）。

对象的上转型对象的实体是子类负责创建的，但上转型对象会失去原对象的一些属性和功能（上转型对象相当于子类对象的一个“简化”对象）。上转型对象具有如下特点（如图 3.4 所示）：

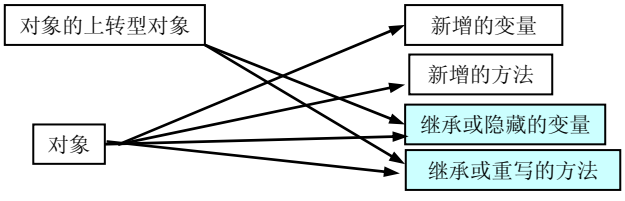

图 3.4　上转型对象示意图

（1）上转型对象不能操作子类新增的成员变量（失掉了这部分属性），不能调用子类新增的方法（失掉了一些行为）。

（2）上转型对象可以访问子类继承或隐藏的成员变量，也可以调用子类继承的方法或子类重写的实例方法。上转型对象操作子类继承的方法或子类重写的实例方法，其作用等价于子类对象去调用这些方法。因此，如果子类重写了父类的某个实例方法后，当对象的上转型对象调用这个实例方法时一定是调用了子类重写的实例方法。

（3）如果子类重写了父类的静态方法，那么子类对象的上转型对象不能调用子类重写的静态方法，只能调用父类的静态方法。

2．用上转型对象体现多态

当一个类有很多子类时，并且这些子类都重写了父类中的某个实例方法，那么当把子类创建的对象的引用放到一个父类的对象中时，就得到了该对象的一个上转型对象，那么这个上转型对象在调用这个方法时就可能具有多种形态，因为不同的子类在重写父类的方法时可能产生不同的行为。

人们经常说：“哺乳动物有很多种叫声”，比如，“吼”、“嚎”、“汪汪”、“喵喵”等，这就是叫声的多态。比如，狗类的上转型对象调用“叫声”方法时产生的行为是“汪汪”，而猫类的上转型对象调用“叫声”方法时，产生的行为是“喵喵”，等等。

下面的代码使用上转型对象体现多态。程序运行效果如图 3.5 所示。

Animal.java

```
public class Animal {
   void cry() {
   }
}
```

```
汪汪.....
喵喵.....
```

图 3.5　多态

Dog.java

```
public class Dog extends Animal {
   void cry() {
      System.out.println("汪汪.....");
   }
}
```

Cat.java

```
public class Cat extends Animal  {
   void cry() {
      System.out.println("喵喵.....");
   }
}
```

Application.java

```
public class Application {
   public static void main(String args[]) {
      Animal animal;
```

```
        animal=new Dog();
        animal.cry();
        animal=new Cat();
        animal.cry();
    }
}
```

3.8 通过 final 禁止多态

可以使用 final 将类声明为 final 类。final 类不能被继承，即不能有子类。例如：

```
final class A {
⋮
}
```

A 就是一个 final 类，将不允许任何类声明成 A 的子类。有时候是出于安全性的考虑，将一些类修饰为 final 类。例如，Java 在 java.lang 包中提供的 String 类对于编译器和解释器的正常运行有很重要的作用，Java 不允许用户程序扩展 String 类，为此 Java 将它修饰为 final 类。

如果用 final 修饰父类中的一个方法，那么这个方法不允许子类重写，也就是说，不允许子类隐藏可以继承的 final 方法（老老实实继承，不许做任何篡改）。

3.9 通过 super 解决多态带来的问题

子类一旦隐藏了继承的成员变量，那么子类创建的对象就不再拥有该变量，该变量将归关键字 super 所拥有，同样，子类一旦隐藏了继承的方法，那么子类创建的对象就不能调用被隐藏的方法，该方法的调用由关键字 super 负责。因此，如果在子类中想使用被子类隐藏的成员变量或方法，就需要使用关键字 super。比如，super.x、super.play()就是访问和调用被子类隐藏的成员变量 *x* 和方法 play()。

在下面的代码中，父类 WaterUser 有 double waterMoney(int amount)方法，该方法根据参数 amount 的值，即根据用水量（吨）返回水费。水费按每吨 2 元计算。

但 WaterUser 的子类 BeijingWaterUser 决定重写 double waterMoney(int amount)方法，重写的方法按参数 amount 的值，即根据用水量（吨）返回水费。重写规则是：对于小于或等于 6 吨的水量，按父类 WaterUser 类的 double waterMoney(int amount)方法计算水费；对于大于 6 吨的水量，按每吨 3 元计算。这样一来，BeijingWaterUser 类就必须使用 super 调用被隐藏的 double waterMoney(int amount)方法。程序运行效果如图 3.6 所示。

WaterUser.java

```
public class WaterUser {
   double unitPrice;
   WaterUser() {
      unitPrice=2;
   }
```

```
水量:6吨,水费:12.000000元
水量:11吨,水费:27.000000元
```

图 3.6　使用 super

```
   public double waterMoney(int amount) {
      double money=amount*unitPrice;  //每吨2元
      if(money>0)
         return money;
      else
         return 0;
   }
}
```

BeijingWaterUser.java

```
public class BeijingWaterUser extends WaterUser {
   double unitPrice;
   BeijingWaterUser() {
      unitPrice=3;
   }
   public double waterMoney(int amount) {
      double money=0;
      if(amount<=6) {
          money= super.waterMoney(amount);//使用super调用隐藏的waterMoney方法
      }
      else if(amount>6){
          money=(super.waterMoney(6)+(amount-6)*3);
                                          //使用super调用隐藏的waterMoney方法
      }
      return money;
   }
}
```

Application.java

```
public class Application {
    public static void main(String args[]) {
       BeijingWaterUser user=new  BeijingWaterUser();
       int waterAmount=6;
       System.out.printf("水量:%d吨,水费:%f元\n",
                     waterAmount,user.waterMoney(waterAmount));
       waterAmount=11;
       System.out.printf("水量:%d吨,水费:%f元\n",
                     waterAmount,user.waterMoney(waterAmount));
    }
}
```

3.10 接　　口

1．定义接口与实现接口

1）定义接口

使用关键字 interface 来定义一个接口。接口的定义和类的定义很相似，分为接口的声

明和接口体。例如：

```
interface Printable {
  public final static int MAX=100;
   public abstract void add();
   public abstract float sum(float x ,float y);
}
```

接口使用关键字 interface 来声明自己是一个接口，格式如下：

```
interface 接口的名字
```

接口体中包含常量的声明（没有变量）和抽象方法两部分。接口体中只有抽象方法，没有普通的方法，而且接口体中所有的常量的访问权限一定都是 public，而且是 static 常量（允许省略 public、final 和 static 修饰符，因此，接口中是声明不出变量的）。所有的抽象方法的访问权限一定都是 public（允许省略 public、abstract 修饰符）。例如：

```
interface Printable {
    int MAX=100;     //等价于public final static int MAX=100;
    void add();      //等价于public abstract void add();
    float sum(float x,float y);
                     //等价于public abstract float sum(float x,float y);
}
```

2）实现接口

接口由类来实现，即由类重写接口中的方法。一个类可以在类声明中使用关键字 implements 声明实现一个或多个接口。如果类实现多个接口，用逗号隔开接口名，如 A 类实现 Printable 和 Addable 接口：

```
class A implements Printable,Addable
```

如果一个非抽象类实现了某个接口，那么这个类必须重写这个接口中的所有方法。需要注意的是，由于接口中的方法一定是 public abstract 方法，所以类在重写接口方法时不仅要去掉 abstract 修饰符、给出方法体，而且方法的访问权限一定要明显地用 public 修饰（否则就降低了访问权限，这是不允许的）。

3）耦合关系

类与接口之间一旦确定是实现关系，那么这种关系是强耦合关系，接口方法的修改都会影响到实现接口的类，因此，面向对象将类与其实现的接口之间的关系看做强耦合关系（组合关系是弱耦合关系，见后续的第 4 章）。

2．接口的 UML 图

表示接口的 UML 图和表示类的 UML 图类似，使用一个长方形描述一个接口的主要构成，将长方形垂直地分为 3 层。

顶部第一层是名字层，接口的名字必须是斜体字形，而且需要用<<interface>>修饰名字，并且该修饰和名字分列在两行。

第二层是常量层，列出接口中的常量及类型，格式是“常量名字：类型”。

第三层是方法层，也称操作层，列出接口中的方法及返回类型，格式是“方法名字（参数列表）：类型”。

图 3.7 是接口 Computable 的 UML 图。

如果一个类实现了一个接口，那么类和接口的关系是实现关系，称类实现接口。UML 通过使用虚线连接类和它所实现的接口，虚线起始端是类，虚线的终点端是它实现的接口，但终点端使用一个空心的三角形表示虚线的结束。

图 3.8 是 China 和 Japan 类实现 Computable 接口的 UML 图。

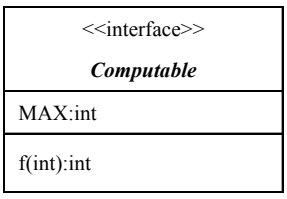

图 3.7　接口 UML 图

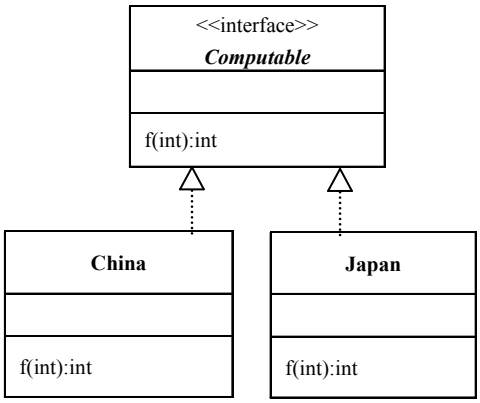

图 3.8　实现关系的 UML 图

3.11　接口回调体现的多态

1．接口回调

和类一样，接口也是 Java 中的一种重要数据类型，用接口声明的变量称做接口变量。那么接口变量中可以存放怎样的数据呢？

接口属于引用型变量，接口变量中可以存放实现该接口的类的实例的引用，即存放对象的引用。比如，假设 Com 是一个接口，那么就可以用 Com 声明一个变量：

```
Com com;
```

内存模型如图 3.9 所示。称此时的 com 是一个空接口，因为 com 变量中还没有存放实现该接口的类的实例（对象）的引用。

假设 ImpleCom 类是实现 Com 接口的类，用 ImpleCom 创建名字为 object 的对象：

```
ImpleCom object=new ImpleCom();
```

那么 object 对象不仅可以调用 ImpleCom 类中原有的方法，而且可以调用 ImpleCom 类实现的接口方法，如图 3.10 所示。

接口回调是指可以把实现某一接口的类创建的对象的引用赋给该接口声明的接口变量，那么该接口变量就可以调用被类实现的接口方法。实际上，当接口变量调用被类实现的接口方法时，就是通知相应的对象调用这个方法。

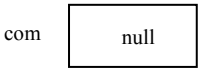

图 3.9　空接口

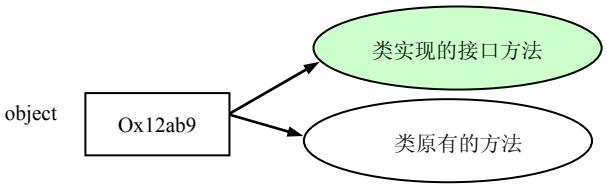

图 3.10　对象调用方法的内存模型

比如，将上述 object 的对象的引用赋值给 com 接口：

```
com=object;
```

那么内存模型如图 3.11 所示，箭头示意接口 com 变量可以调用类实现的接口方法（这一过程被称为接口回调）。

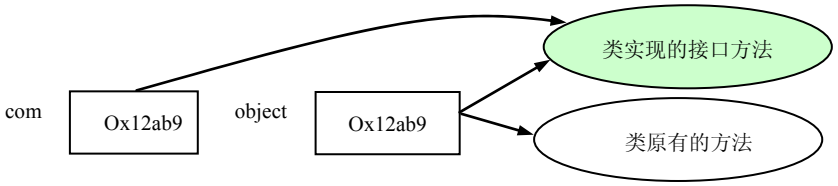

图 3.11　接口回调的内存模型

接口回调非常类似于上转型对象调用子类重写的方法。

2．回调体现的多态性

把实现接口的类的实例的引用赋值给接口变量后，该接口变量就可以回调类重写的接口方法。由接口产生的多态就是指不同的类在实现同一个接口时可能具有不同的实现方式，那么接口变量在回调接口方法时就可能具有多种形态。

例如，对于两个正数 a 和 b，有的人使用算术平均公式：

$$(a+b)/2$$

计算（算术）平均值，而有的人使用几何平均公式：

$$\sqrt{a\times b}$$

计算（几何）平均值。

在下面的代码中，A 类和 B 类都实现了 ComputerAverage 接口，但实现的方式不同。程序运行效果如图 3.12 所示。

```
11.23和22.78的算术平均值:17.01
11.23和22.78的几何平均值:15.99
```

图 3.12　接口与多态

ComputerAverage.java

```
public interface ComputerAverage {
   public double average(double a,double b);
}
```

A.java

```
public class A implements ComputerAverage {
   public double average(double a,double b) {
      double aver=0;
      aver=(a+b)/2;
```

```
      return aver;
   }
}
```


B.java

```
public class B implements ComputerAverage {
   public double average(double a,double b) {
      double aver=0;
      aver=Math.sqrt(a*b);
      return aver;
   }
}
```

Application.java

```
public class Application {
   public static void main(String args[]) {
      ComputerAverage computer;
      double a=11.23,b=22.78;
      computer=new A();
      double result=computer.average(a,b);
      System.out.printf("%5.2f和%5.2f的算术平均值:%5.2f\n",a,b,result);
      computer=new B();
      result=computer.average(a,b);
      System.out.printf("%5.2f和%5.2f的几何平均值:%5.2f",a,b,result);
   }
}
```

3.12 重载体现的多态

通过子类或接口体现多态是最重要的多态体现形式，但 Java 也提供另一种所谓的重载多态（Overload）。重载多态和子类进行重写所体现的多态不同，重载体现的多态是指一个类可以有多个方法有相同的名字，但参数必须不同，而重写体现的多态是指一个类的多个子类可以对父类某个方法采取不同的重写方式。

重载体现的多态习惯称为行为多态，重写体现的多态习惯称为继承多态。例如，让一个人执行“求面积”操作时，他可能会问你求什么面积，在这里“求面积”操作是一个行为多态。因此，必须向“求面积”操作传递所需要的消息，以便让对象根据相应的消息来产生相应的行为。对象的行为通过类中的方法来体现，那么行为的多态性就是方法的重载。

方法重载的意思是：一个类中可以有多个方法具有相同的名字，但这些方法的参数必须不同。两个方法的参数不同是指满足下列条件之一：

（1）参数的个数不同。

（2）参数个数相同，但参数列表中对应的某个参数的类型不同。

下面的代码中 Student 类中的 computerArea 方法是重载方法。程序运行效果如图 3.13

所示。

```
zhang计算圆的面积：
121699.48226600002
zhang计算梯形的面积：
108.0
```

图 3.13　computerArea 方法是重载方法

Circle.java

```
public class Circle {
    double radius,area;
    void setRadius(double r) {
        radius=r;
    }
    double getArea(){
        area=3.14*radius*radius;
        return area;
    }
}
```

Tixing.java

```
public class Tixing {
   double above,bottom,height;
   Tixing(double a,double b,double h) {
      above=a;
      bottom=b;
      height=h;
   }
   double getArea() {
      return (above+bottom)*height/2;
   }
}
```

Student.java

```
public class Student {
   double computerArea(Circle c) {      //是重载方法
      double area=c.getArea();
      return area;
   }
   double computerArea(Tixing t) {      //是重载方法
      double area=t.getArea();
      return area;
   }
}
```

Application.java

```
public classApplication{
  public static void main(String args[]) {
     Circle circle=new Circle();
     circle.setRadius(196.87);
```

```
        Tixing lader=new Tixing(3,21,9);
        Student zhang=new Student();
        System.out.println("zhang计算圆的面积: ");
        double result=zhang.computerArea(circle);
        System.out.println(result);
        System.out.println("zhang计算梯形的面积: ");
        result=zhang.computerArea(lader);
        System.out.println(result);
    }
}
```