第 5 章　子类与继承

主要内容

- 子类与父类
- 子类的继承性
- 子类与对象
- 成员变量的隐藏和方法重写
- super 关键字
- final 关键字
- 对象的上转型对象
- 继承与多态
- abstract 类与 abstract 方法
- 面向抽象编程
- 开-闭原则

视频讲解

在第 4 章学习了怎样从抽象得到类，体现了面向对象最重要的一个方面——数据的封装。本章将讲述面向对象另外两个方面的重要内容，即继承与多态。

视频讲解

5.1　子类与父类

求职者在介绍自己的基本情况时不必“从头说起”，例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。

当我们准备编写一个类的时候，发现某个类有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于有了这个成员变量，不用定义方法就相当于有了这个方法，那么可以将编写的类定义为这个类的子类，子类可以让我们不必一切“从头做起”。

继承是一种由已有的类创建新类的机制。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(尤其是学习过 C++的读者)Java 不支持多重继承，即子类只能有一个父类。人们习惯地称子类与父类的关系是“is-a”关系。

▶ 5.1.1　子类

在类的声明中，通过使用关键字 extends 来定义一个类的子类，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)。

▶ 5.1.2 类的树形结构

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(将类看作树上的结点)，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 Object 的子类，即类声明“class A”与“class A extends Object”是等同的。

5.2 子类的继承性

类可以有两种重要的成员，即成员变量和方法。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。

▶ 5.2.1 子类和父类在同一包中的继承性


视频讲解

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。

下面的例子 1 中有 4 个类，即 People、Student、UniverStudent 和 Example5_1，这些类都没有包名(需要分别打开文本编辑器编写、保存这些类的源文件，例如保存到“C:\ch5”目录中)，其中 UniverStudent 类是 Student 的子类，Student 是 People 的子类。程序运行效果如图 5.1 所示。

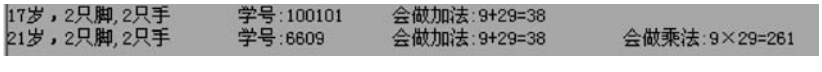

图 5.1 子类的继承性

例子 1

People. java

```
public class People {
    int age,leg = 2,hand = 2;
    protected void showPeopleMess() {
        System.out.printf("%d岁,%d只脚,%d只手\t",age,leg,hand);
    }
}
```

Student. java

```
public class Student extends People {
    int number;
    void tellNumber() {
        System.out.printf("学号:%d\t",number);
    }
    int add(int x,int y) {
        return x+y;
    }
}
```

UniverStudent. java

```
public class UniverStudent extends Student {
    int multi(int x,int y) {
        return x*y;
    }
}
```

Example5_1. java

```
public class Example5_1 {
    public static void main(String args[]) {
        Student zhang = new Student();
        zhang.age = 17;                                    //访问继承的成员变量
        zhang.number=100101;
        zhang.showPeopleMess();                            //调用继承的方法
        zhang.tellNumber();
        int x=9,y=29;
        System.out.print("会做加法:");
        int result=zhang.add(x,y);
        System.out.printf("%d+%d=%d\n",x,y,result);
        UniverStudent geng = new UniverStudent();
        geng.age = 21;                                     //访问继承的成员变量
        geng.number=6609;                                  //访问继承的成员变量
        geng.showPeopleMess();                             //调用继承的方法
        geng.tellNumber();                                 //调用继承的方法
        System.out.print("会做加法:");
        result=geng.add(x,y);                              //调用继承的方法
        System.out.printf("%d+%d=%d\t",x,y,result);
        System.out.print("会做乘法:");
```

```
        result = geng.multi(x,y);
        System.out.printf("%d×%d=%d\n",x,y,result);
    }
}
```

5.2.2 子类和父类不在同一包中的继承性

视频讲解

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。

5.2.3 继承关系的 UML 图

视频讲解

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 图来表示两者之间的继承关系，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 5.2 是例子 1 中 Student 类和 People 类之间的继承关系的 UML 图。

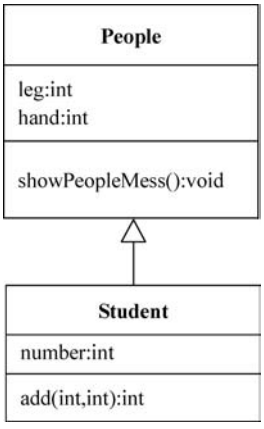

图 5.2 继承关系的 UML 图

5.2.4 protected 的进一步说明

视频讲解

一个类 A 中的 protected 成员变量和方法可以被它的子孙类继承，例如 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 总结得更全面一些。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，但是如果在另外一个类中，例如在 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。

(1) 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 对象就可以访问这些 protected 成员变量和方法。

(2) 对于子类 D 从父类继承的 protected 成员变量或方法，需要追溯到这些 protected 成员变量或方法所在的“祖先”类，例如可能是 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。

5.3 子类与对象

视频讲解

5.3.1 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空间，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 5.5 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，读者有这样的感觉：子类创建对象时似乎浪费了一些内存空间，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。例如，父类中的 private 成员变量尽管分配了内存空间，但不作为子类对象的变量，当然它们也不是父类的某个对象的变量，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，注意子类中还有一部分方法是从父类继承的，这部分方法可以操作这部分未继承的变量。

注：子类继承的方法只能操作子类继承的成员变量或未继承的成员变量，不可能操作子类新声明的变量。

在下面的例子 2 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 5.3 所示。

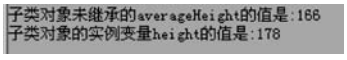

图 5.3　子类对象调用方法

例子 2

Example5_2.java

```
class People {
    private int averHeight = 166;
    public int getAverHeight() {
        return averHeight;
    }
}
class ChinaPeople extends People {
    int height;
    public void setHeight(int h) {
        //height = h + averHeight; //非法，子类没有继承 averHeight
        height = h;
    }
    public int getHeight() {
        return height;
    }
}
public class Example5_2 {
    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());
    }
}
```

▶ 5.3.2　关于 instanceof 运算符

在第 2 章曾简单提到 instanceof 运算符，但未做任何讨论，因为掌握该运算符需要类和子类的知识。instanceof 运算符是 Java 独有的双目运算符，其左面的操作元素是对象，右面的操作元素是类(接口，见第 6 章)，当左面的操作元素是右面的类或其子类所创建的对象时(实现

接口的类所创建的对象时),instanceof 运算的结果是 true,否则是 false。例如,对于例子 1 中的 People、Student 和 UniverStudent 类,如果 zhang 和 geng 分别是 Student 和 UniverStudent 创建的对象,那么 zhang instanceof Student、zhang instanceof People、geng instanceof People 和 geng instanceof UniverStudent 这 4 个表达式的结果都是 true,而 zhang instanceof UniverStudent 表达式的结果是 false(zhang 不是大学生)。

5.4 成员变量的隐藏和方法重写


视频讲解

▶ 5.4.1 成员变量的隐藏

在编写子类时仍然可以声明成员变量,一种特殊的情况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。

下面的例子 3 演示货物价格的计算。假设一般货物按重量计算价格,但重量计算的精度是 double 型,对客户的优惠程度较小;打折货物的重量不计小数,按整数值计算价格,以给用户更多的优惠。在例子 3 中,Goods 类有一个名字为 weight 的 double 型成员变量,本来子类 CheapGoods 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,子类新定义的方法将按 int 型重量计算价格。程序运行效果如图 5.4 所示。

```
int型的weight=198
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
double型的weight=198.987
cheapGoods使用继承的方法(无优惠)计算价格:1989.87
```

图 5.4 子类隐藏继承的成员变量

例子 3

Goods.java

```
public class Goods {
    public double weight;
    public void oldSetWeight(double w) {
        weight = w;
        System.out.println("double 型的 weight = " + weight);
    }
    public double oldGetPrice() {
        double price = weight * 10;
        return price;
    }
}
```

CheapGoods.java

```
public class CheapGoods extends Goods {
    public int weight;
```

```
    public void newSetWeight(int w) {
        weight = w;
        System.out.println("int 型的 weight = " + weight);
    }
    public double newGetPrice() {
        double price = weight * 10;
        return price;
    }
}
```

Example5_3.java

```
public class Example5_3 {
    public static void main(String args[]) {
        CheapGoods cheapGoods = new CheapGoods();
        //cheapGoods.weight = 198.98; 是非法的,因为子类对象的 weight 变量已经是 int 型
        cheapGoods.newSetWeight(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.newGetPrice());
        cheapGoods.oldSetWeight(198.987);      /*子类对象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.oldGetPrice());
    }
}
```

注：子类继承的方法可以操作子类继承和隐藏的成员变量，不可以操作子类新声明的成员变量。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(需使用 super 关键字操作子类隐藏的成员变量，见后面的 5.5 节)。


视频讲解

▶ 5.4.2 方法重写

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))。

❶ 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(不包括 final 方法，见 5.6 节)。所谓方法重写，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作子类重写的方法。

❷ 重写的目的

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，子类就有权重写 f()(不包括 final 方法，见 5.6 节)，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()调用的一定是重写方法 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。如果子类想使用被隐藏的成员变量或方法，必须使用关键字 super(5.5 节讲述 super 的用法)。

注：如果子类隐藏了继承的成员变量 m，那么子类继承的方法中操作的成员变量 m 是被子类隐藏的 m，而子类新增或重写的方法中操作的 m 一定是子类新声明的成员变量 m。

高考入学考试课程为 3 门，每门满分为 100。在高考招生时，大学录取规则为录取最低分数线是 180 分，而重点大学重写录取规则为录取最低分数线是 220 分。

在下面的例子 4 中，ImportantUniversity 是 University 类的子类，子类重写了父类的 enterRule()方法，运行效果如图 5.5 所示。

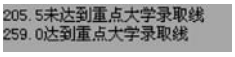

图 5.5 重写录取规则

例子 4

University.java

```
public class University {
    void enterRule(double math,double english,double chinese) {
        double total = math + english + chinese;
        if(total >= 180)
           System.out.println(total + "达到大学录取线");
        else
           System.out.println(total + "未达到大学录取线");
    }
}
```

ImportantUniversity.java

```
public class ImportantUniversity extends University{
   void enterRule(double math,double english,double chinese) {
      double total = math + english + chinese;
      if(total >= 220)
         System.out.println(total + "达到重点大学录取线");
      else
         System.out.println(total + "未达到重点大学录取线");
   }
}
```

Example5_4.java

```
public class Example5_4 {
   public static void main(String args[]) {
      double math = 62,english = 76.5,chinese = 67;
      ImportantUniversity univer = new ImportantUniversity();
      univer.enterRule(math,english,chinese);        //调用重写的方法
      math = 91;
      english = 82;
      chinese = 86;
      univer.enterRule(math,english,chinese);        //调用重写的方法
   }
}
```

下面再看一个简单的重写的例子，并就该例子讨论一些重写的注意事项。在下面的例子5中，子类B重写了父类的computer()方法，运行效果如图5.6所示。

图5.6 方法重写

例子5

Example5_5.java

```
class A {
   float computer(float x,float y) {
      return x + y;
   }
   public int g(int x,int y) {
      return x + y;
   }
}
class B extends A {
   float computer(float x,float y) {
      return x * y;
   }
}
public class Example5_5 {
   public static void main(String args[]) {
      B b = new B();
      double result = b.computer(8,9);                    //b调用重写的方法
      System.out.println(result);
      int m = b.g(12,8);                                  //b调用继承的方法
      System.out.println(m);
   }
}
```

在上面的例子5中，如果子类如下定义computer()方法，将产生编译错误：

```
double computer(float x,float y) {
   return x * y;
}
```

其原因是，父类的方法computer()的类型是float，子类定义的方法computer()没有和父类的方法computer()保持类型一致，如此定义的computer()方法不是重写(覆盖)继承的computer()方法，这样子类就无法隐藏继承的方法(没有覆盖继承的computer方法)，导致子类出现两个方法的名字相同(名字都是computer)，并且参数也相同，这是不允许的(见4.8节，方法重载overload()的语法规则)。

请读者思考，如果子类如下定义computer()方法，是否属于重写继承的computer()方法呢？编译可以通过吗？运行结果怎样？

```
float computer(float x,float y,double z) {
   return x - y;
}
```

答案是不属于重写computer()方法，编译无错误(子类没有覆盖继承的computer()方法，使得子类出现了方法重载，有两个方法的名字都是computer，但二者的参数不同)，运行结果是：

```
17.0
20
```

子类在重写可以继承的方法时，可以完全按照自己的意图编写新的方法体，以便体现重写方法的独特的行为（学习了后面的 5.7 节之后，读者会更深刻地理解重写方法在面向对象程序设计中的意义）。重写方法的类型可以是父类方法类型的子类型，即不必完全一致（JDK 5 版本之前要求必须一致），例如父类的方法的类型是 People（People 是类，类是面向对象语言中最重要的一种数据类型，类声明的变量称作对象，见 4.2 节和 4.3 节），重写方法的类型可以是 Student（假设 Student 是 People 的子类）。

在下面的例子 6 中，父类的方法是 Object 类型，子类重写方法的类型是 Integer 类型（Object 类是所有类的祖先类，见 5.1.2 节）。

例子 6

Example5_6.java

```
class A {
   Object get() {
      return null;                          //返回一个空对象
   }
}
class B extends A {
   Integer get() {                          //Integer 是 Object 的子类
      return new Integer(100);              //返回一个 Integer 对象
   }
}
public class Example5_6 {
   public static void main(String args[]) {
      B b = new B();
      Integer t = b.get();
      System.out.println(t.intValue());
   }
}
```

❸ 重写的注意事项

在重写父类的方法时，不允许降低方法的访问权限，但可以提高访问权限（访问限制修饰符按访问权限从高到低的排列顺序是 public→protected→友好的→private）。在下面的代码中，子类重写父类的方法 f()，该方法在父类中的访问权限是 protected 级别，子类重写时不允许级别低于 protected：

```
class A {
   protected float f(float x,float y) {
       return x-y;
   }
}
class B extends A {
   float f(float x,float y) {                          //非法，因为降低了访问权限
        return x+y;
   }
}
```

```
class C extends A {
    public float f(float x,float y) {                    //合法,提高了访问权限
        return x * y;
    }
}
```

5.5 super 关键字


视频讲解

▶ 5.5.1 用 super 操作被隐藏的成员变量和方法

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,如果想在子类中使用被子类隐藏的成员变量或方法,就需要使用关键字 super。例如 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()。

在下面的例子 7 中,子类使用 super 访问和调用被子类隐藏的成员变量和方法,运行效果如图 5.7 所示。

```
resultOne=50.5
resultTwo=2525.0
```

图 5.7 用 super 调用隐藏的方法

例子 7

Example5_7.java

```
class Sum {
    int n;
    float f() {
        float sum = 0;
        for(int i = 1;i <= n;i++)
            sum = sum + i;
        return sum;
    }
}
class Average extends Sum {
    int n;
    float f() {
        float c;
        super.n = n;
        c = super.f();
        return c/n;
    }
    float g() {
        float c;
        c = super.f();
        return c/2;
    }
}
public class Example5_7 {
    public static void main(String args[]) {
        Average aver = new Average();
```

```
        aver.n = 100;
        float resultOne = aver.f();
        float resultTwo = aver.g();
        System.out.println("resultOne = " + resultOne);
        System.out.println("resultTwo = " + resultTwo);
    }
}
```

请读者思考，如果将例子 7 的 Example5_7 类中的代码

```
float resultOne = aver.f();
float resultTwo = aver.g();
```

颠倒次序，即更改为：

```
float resultTwo = aver.g();
float resultOne = aver.f();
```

程序的输出结果是什么？答案是：

```
resultOne = 50.5
resultTwo = 0.0
```

注：当用 super 调用被隐藏的方法时，该方法中出现的成员变量是被子类隐藏的成员变量或继承的成员变量。

▶ 5.5.2 使用 super 调用父类的构造方法


视频讲解

当用子类的构造方法创建一个子类的对象时，子类的构造方法总是先调用父类的某个构造方法。也就是说，如果子类的构造方法没有明显地指明使用父类的哪个构造方法，子类就调用父类的不带参数的构造方法。

由于子类不继承父类的构造方法，所以子类在其构造方法中需使用 super 来调用父类的构造方法，而且 super 必须是子类构造方法中的头一个语句，即如果在子类的构造方法中没有显式地写出 super 关键字来调用父类的某个构造方法，那么默认有：

```
super();
```

如果在子类的构造方法中显式地写出了 super 关键字来调用父类的某个构造方法，那么编译器不再提供默认的 super 语句。

在下面的例子 8 中，UniverStudent 是 Student 的子类，UniverStudent 子类在构造方法中使用了 super 关键字，运行效果如图 5.8 所示。

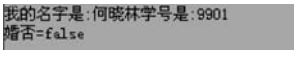

图 5.8 用 super 调用父类构造方法

例子 8

Example5_8.java

```
class Student {
    int number;String name;
    Student() {
    }
```

```
    Student(int number,String name) {
        this.number = number;
        this.name = name;
        System.out.println("我的名字是:" + name +"学号是:" + number);
    }
}
class UniverStudent extends Student {
    boolean 婚否;
    UniverStudent(int number,String name,boolean b) {
        super(number,name);
        婚否 = b;
        System.out.println("婚否 = " + 婚否);
    }
}
public class Example5_8 {
    public static void main(String args[]) {
        UniverStudent zhang = new UniverStudent(9901,"何晓林",false);
    }
}
```

大家已经知道,如果类中定义了一个或多个构造方法,那么 Java 不提供默认的构造方法(不带参数的构造方法),因此当在父类中定义多个构造方法时应当包括一个不带参数的构造方法(如上述例子 8 中的 Student 类),以防子类省略 super 时出现错误。

请读者思考,如果上述例子 8 中的 UniverStudent 子类的构造方法中省略 super,程序的运行效果是怎样的?


视频讲解

5.6 final 关键字

final 关键字可以修饰类、成员变量和方法中的局部变量。

▶ 5.6.1 final 类

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(见第 8 章)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。

▶ 5.6.2 final 方法

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(老老实实地继承,不许做任何篡改)。

▶ 5.6.3 常量

如果成员变量或局部变量被修饰为 final，那么它就是常量。由于常量在运行期间不允许再发生变化，所以常量在声明时没有默认值，这就要求程序在声明常量时必须指定该常量的值。下面的例子 9 使用了 final 关键字。

例子 9

Example5_9.java

```
class A {
  final double PI = 3.1415926;                          //PI 是常量
  public double getArea(final double r) {
     //r = r + 1; //非法,不允许对 final 变量进行更新操作
     return PI * r * r;
  }
  public final void speak() {
     System.out.println("您好,How's everything here?");
  }
}
public class Example5_9 {
   public static void main(String args[]) {
      A a = new A();
      System.out.println("面积: " + a.getArea(100));
      a.speak();
   }
}
```

5.7 对象的上转型对象

视频讲解

人们经常说“老虎是动物”“狗是动物”等。若动物类是老虎类的父类，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。需要注意的是，当说老虎是动物时，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“老虎是动物”属于上溯思维方式，下面讲解和这种思维方式很类似的 Java 语言中的上转型对象。

假设 Animal 类是 Tiger 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，例如：

```
Animal a;
a = new Tiger();
```

或

```
Animal a;
Tiger b = new Tiger();
a = b;
```

这时称对象 a 是对象 b 的上转型对象（好比说“老虎是动物”）。

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 5.9

所示)：

(1) 上转型对象不能访问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，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。

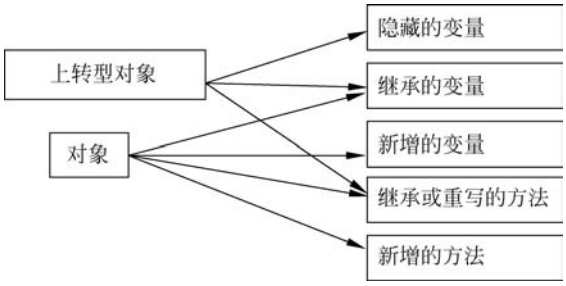

图 5.9　上转型对象示意图

注：

① 不要将父类创建的对象和子类对象的上转型对象混淆。

② 可以将对象的上转型对象再强制转换为一个子类对象，这时该子类对象又具备了子类所有的属性和功能。

③ 不可以将父类创建的对象的引用赋值给子类声明的对象(不能说“人是美国人”)。

④ 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。

在下面的例子 10 中，monkey 是 People 类对象的上转型对象，运行效果如图 5.10 所示。

```
true
true
***I love this game***
100
```

图 5.10　使用上转型对象

例子 10

Example5_10.java

```
class  类人猿 {
   void crySpeak(String s) {
        System.out.println(s);
   }
}
class People extends 类人猿 {
   void computer(int a,int b) {
      int c = a * b;
      System.out.println(c);
   }
   void crySpeak(String s) {
      System.out.println(" *** " + s + " *** ");
   }
}
public class Example5_10 {
   public static void main(String args[]) {
      类人猿 monkey;
      People geng = new People();
      System.out.println(geng instanceof People);      //输出为 true
      monkey = geng;                                   //monkey 是 People 对象 geng 的上转型对象
      System.out.println(monkey instanceof People);    //输出为 true
```

```
        monkey.crySpeak("I love this game");          //等同于 geng.crySpeak("I love this game");
        People people = (People)monkey;               //把上转型对象强制转换为子类的对象
        people.computer(10,10);
    }
}
```

在上述例子 10 中，上转型对象 monkey 调用方法为

```
monkey.crySpeak("I love this game");
```

得到的结果是“ *** I love this game *** ”，而不是“I love this game”。因为 monkey 调用的是子类重写的方法 crySpeak()。需要注意

```
monkey.computer(10,10);
```

是错误的，因为 computer()方法是子类新增的方法。


视频讲解

5.8 继承与多态

人们经常说“哺乳动物有很多种叫声”，例如“吼”“嚎”“汪汪”“喵喵”等，这就是叫声的多态。

若一个类有很多子类，并且这些子类都重写了父类中的某个方法，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，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。例如，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。

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功能行为。

下面的例子 11 展示了多态，运行效果如图 5.11 所示。

图 5.11 多态

例子 11

Example5_11.java

```
class  动物 {
    void cry() {
    }
}
class 狗 extends 动物 {
    void cry() {
        System.out.println("汪汪...");
    }
}
class 猫 extends 动物  {
    void cry() {
        System.out.println("喵喵...");
    }
}
public class Example5_11 {
    public static void main(String args[]) {
```

```
        动物 animal;
        animal = new 狗();
        animal.cry();
        animal = new 猫();
        animal.cry();
    }
}
```


视频讲解

5.9 abstract 类和 abstract 方法

用关键字 abstract 修饰的类称为 abstract 类(抽象类),例如:

```
abstract class A {
…
}
```

用关键字 abstract 修饰的方法称为 abstract 方法(抽象方法),例如:

```
abstract int min(int x,int y);
```

对于 abstract 方法,只允许声明,不允许实现(没有方法体),而且不允许使用 final 和 abstract 同时修饰一个方法或类,也不允许使用 static 和 private 修饰 abstract 方法,即 abstract 方法必须是非 private 的实例方法(访问权限必须高于 private)。

❶ abstract 类中可以有 abstract 方法

和普通类(非 abstract 类)相比,abstract 类中可以有 abstract 方法(非 abstract 类中不可以有 abstract 方法),也可以有非 abstract 方法。

下面的 A 类中的 min()方法是 abstract 方法,max()方法是普通方法(非 abstract 方法)。

```
abstract class A {
   abstract int min(int x,int y);
   int max(int x,int y) {
      return x>y?x:y;
   }
}
```

注:abstract 类中也可以没有 abstract 方法。

❷ abstract 类不能用 new 标识符创建对象

对于 abstract 类,不能使用 new 标识符创建该类的对象。如果一个非抽象类是某个抽象类的子类,那么它必须重写父类的抽象方法,给出方法体,这就是为什么不允许使用 final 和 abstract 同时修饰一个方法或类的原因。

❸ abstract 类的子类

如果一个非 abstract 类是 abstract 类的子类,那么它必须重写父类的 abstract 方法,即去掉 abstract 方法的 abstract 修饰,并给出方法体。如果一个 abstract 类是 abstract 类的子类,那么它可以重写父类的 abstract 方法,也可以继承父类的 abstract 方法。

❹ abstract 类的对象做上转型对象

可以使用 abstract 类声明对象，尽管不能使用 new 标识符创建该对象，但该对象可以成为其子类对象的上转型对象，那么该对象就可以调用子类重写的方法。

❺ 理解 abstract 类

抽象类的语法很容易被理解和掌握，但理解抽象类的意义是更为重要的。理解的关键点如下：

(1) 抽象类可以抽象出重要的行为标准，该行为标准用抽象方法来表示，即抽象类封装了子类必须要有的行为标准。

(2) 抽象类声明的对象可以成为其子类的对象的上转型对象，调用子类重写的方法，即体现子类根据抽象类中的行为标准给出的具体行为。

人们已经习惯给别人介绍数量标准，例如，在介绍人的时候可以说人的身高是 float 型的，头发的个数是 int 型的，但是在学习了类以后也要习惯介绍行为标准。所谓行为标准，仅是方法的名字、方法的类型而已，就像介绍人的头发数量标准是 int 型，但不要说出有多少根头发。例如，人具有 run()行为或 speak()行为，但仅说出行为标准，不要说出 speak()行为的具体体现，即不要说 speak()行为是用英语说话或中文说话，这样的行为标准就是抽象方法(没有方法体的方法)。这样一来，开发者可以把主要精力放在一个应用中需要哪些行为标准(不用关心行为的细节)上，不仅节省时间，而且非常有利于设计出易维护、易扩展的程序(见后面的 5.10 节)。抽象类中的抽象方法可以由子类去实现，即行为标准的实现由子类完成。

一个男孩要找女朋友，他可以提出一些行为标准，例如，女朋友具有 speak()和 cooking()行为，但可以不给出 speak()和 cooking()行为的细节。下面的例子 12 使用 abstract 类封装了男孩对女朋友的行为要求，即封装了他要找的任何具体女朋友都应该具有的行为。程序运行效果如图 5.12 所示。

图 5.12 使用抽象类

例子 12

Example5_12.java

```
abstract class GirlFriend {                          //抽象类,封装了两个行为标准
    abstract void speak();
    abstract void cooking();
}
class ChinaGirlFriend extends GirlFriend {
    void speak(){
        System.out.println("你好");
    }
    void cooking(){
        System.out.println("水煮鱼");
    }
}
class AmericanGirlFriend extends GirlFriend {
    void speak(){
        System.out.println("hello");
    }
    void cooking(){
        System.out.println("roast beef");
    }
}
```

```
class Boy {
    GirlFriend friend;
    void setGirlfriend(GirlFriend f){
         friend = f;
    }
    void showGirlFriend() {
        friend.speak();
        friend.cooking();
    }
}
public class Example5_12 {
    public static void main(String args[]) {
        GirlFriend girl = new ChinaGirlFriend();              //girl 是上转型对象
        Boy boy = new Boy();
        boy.setGirlfriend(girl);
        boy.showGirlFriend();
        girl = new AmericanGirlFriend();                      //girl 是上转型对象
        boy.setGirlfriend(girl);
        boy.showGirlFriend();
    }
}
```

视频讲解

5.10 面向抽象编程

在设计程序时经常会使用 abstract 类，其原因是 abstract 类只关心操作，不关心这些操作具体的实现细节，可以使程序的设计者把主要精力放在程序的设计上，而不必拘泥于细节的实现(将这些细节留给子类的设计者)，即避免设计者把大量的时间和精力花费在具体的算法上。例如，在设计地图时，首先考虑地图最重要的轮廓，不必去考虑诸如城市中的街道牌号等细节，细节应当由抽象类的非抽象子类去实现，这些子类可以给出具体的实例，从而完成程序功能的具体实现。在设计一个程序时，可以通过在 abstract 类中声明若干个 abstract 方法表明这些方法在整个系统设计中的重要性，方法体的内容细节由它的非 abstract 子类去完成。

使用多态进行程序设计的核心技术之一是使用上转型对象，即将 abstract 类声明的对象作为其子类对象的上转型对象，那么这个上转型对象就可以调用子类重写的方法。

所谓面向抽象编程，是指当设计某种重要的类时不让该类面向具体的类，而是面向抽象类，即所设计类中的重要数据是抽象类声明的对象，而不是具体类声明的对象。

以下通过一个简单的问题来说明面向抽象编程的思想。

假如已经有了一个 Circle 类(圆类)，该类创建的对象 circle 调用 getArea()方法可以计算圆的面积。Circle 类的代码如下：

Circle.java

```
public class Circle {
    double r;
    Circle(double r){
        this.r = r;
    }
    public double getArea() {
```

```
        return(3.14 * r * r);
    }
}
```

现在要设计一个 Pillar 类(柱类),该类的对象调用 getVolume()方法可以计算柱体的体积。Pillar 类的代码如下:

Pillar. java

```
public class Pillar {
    Circle bottom;                          //bottom 是用具体类 Circle 声明的对象
    double height;
    Pillar(Circle bottom,double height) {
        this.bottom = bottom;
        this.height = height;
    }
    public double getVolume() {
        return bottom.getArea() * height;
    }
}
```

在上述 Pillar 类中,bottom 是用具体类 Circle 声明的对象,如果不涉及用户需求的变化,上面 Pillar 类的设计没有什么不妥,但是在某个时候用户希望 Pillar 类能创建出底是三角形的柱体。显然上述 Pillar 类无法创建出这样的柱体,即上述设计的 Pillar 类不能应对用户的这种需求(软件设计面临的最大问题是用户需求的变化)。我们发现,用户需要的柱体的底无论是何种图形,有一点是相同的,即要求该图形必须有计算面积的行为,因此可以用一个抽象类封装这个行为标准:在抽象类里定义一个抽象方法 abstract double getArea(),即用抽象类封装许多子类都必有的行为。

现在重新设计 Pillar 类。首先注意到计算柱体体积的关键是计算出底面积,一个柱体在计算底面积时不应该关心它的底是什么形状的具体图形,只应该关心这种图形是否具有计算面积的方法。因此,在设计 Pillar 类时不应该让它的底是某个具体类声明的对象,一旦这样做,Pillar 类就依赖该具体类,缺乏弹性,难以应对需求的变化。

下面将面向抽象重新设计 Pillar 类。首先编写一个抽象类 Geometry,在该抽象类中定义了一个抽象的 getArea()方法。Geometry 类如下:

Geometry. java

```
public abstract class Geometry  {
   public abstract double getArea();
}
```

上述抽象类将所有计算面积的算法抽象为一个标识——getArea(),即抽象方法,不再考虑算法的细节。

现在 Pillar 类的设计者可以面向 Geometry 类编写代码,即 Pillar 类应该把 Geometry 对象作为自己的成员,该成员可以调用 Geometry 的子类重写的 getArea()方法。这样一来,Pillar 类就可以将计算底面积的任务指派给 Geometry 类的子类的实例(用户的各种需求将由不同的子类去负责)。

以下 Pillar 类的设计不再依赖具体类,而是面向 Geometry 类,即 Pillar 类中的 bottom 是

用抽象类 Geometry 声明的对象，而不是具体类声明的对象。重新设计的 Pillar 类的代码如下：

Pillar. java

```
public class Pillar {
   Geometry bottom;                          //bottom 是抽象类 Geometry 声明的变量
   double height;
   Pillar(Geometry bottom,double height) {
      this.bottom = bottom; this.height = height;
   }
   public double getVolume() {
      if(bottom == null) {
         System.out.println("没有底,无法计算体积");
         return -1;
      }
      return bottom.getArea() * height;      //bottom 可以调用子类重写的 getArea()方法
   }
}
```

下列 Circle 和 Rectangle 类都是 Geometry 的子类，两者都必须通过重写 Geometry 类的 getArea()方法来计算各自的面积。

Circle. java

```
public class Circle extends Geometry {
   double r;
   Circle(double r) {
      this.r = r;
   }
   public double getArea() {
      return(3.14 * r * r);
   }
}
```

Rectangle. java

```
public class Rectangle extends Geometry {
   double a,b;
   Rectangle(double a,double b) {
      this.a = a;
      this.b = b;
   }
   public double getArea() {
      return a * b;
   }
}
```

注意，在增加了 Circle 和 Rectangle 类后不必修改 Pillar 类的代码。现在就可以用 Pillar 类创建出具有矩形底或圆形底的柱体了，如下列 Application. java 所示，程序运行效果如图 5.13 所示。

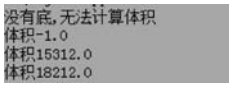

图 5.13　计算柱体的体积

Application.java

```
public class Application{
   public static void main(String args[]){
      Pillar pillar;
      Geometry bottom = null;
      pillar = new Pillar(bottom,100);                    //没有底的柱体
      System.out.println("体积" + pillar.getVolume());
      bottom = new Rectangle(12,22);
      pillar = new Pillar(bottom,58);                     //pillar 是具有矩形底的柱体
      System.out.println("体积" + pillar.getVolume());
      bottom = new Circle(10);
      pillar = new Pillar(bottom,58);                     //pillar 是具有圆形底的柱体
      System.out.println("体积" + pillar.getVolume());
   }
}
```

通过面向抽象来设计 Pillar 类，使得该 Pillar 类不再依赖具体类，因此每当系统增加新的 Geometry 的子类时，例如增加一个 Triangle 子类，用户不需要修改 Pillar 类的任何代码，就可以使用 Pillar 创建出具有三角形底的柱体。

通过前面的讨论可以做出如下总结：

面向抽象编程的目的是应对用户需求的变化，将某个类中经常因需求变化而需要改动的代码从该类中分离出去。面向抽象编程的核心是让类中每种可能的变化对应地交给抽象类的一个子类去负责，从而让该类的设计者不用去关心具体实现，避免所设计的类依赖于具体的实现。面向抽象编程使设计的类容易应对用户需求的变化。

注：如果读者进一步学习设计模式，会更深刻地理解面向抽象的重要性，可参见作者在清华大学出版社出版的《Java 设计模式》一书。

5.11 开-闭原则


视频讲解

所谓“开-闭原则(Open-Closed Principle)”，就是让设计的系统对扩展开放，对修改关闭。那么怎么理解对扩展开放，对修改关闭呢？实际上，这句话的本质是指当系统中增加新的模块时不需要修改现有的模块。在设计系统时，应当首先考虑到用户需求的变化，将应对用户变化的部分设计为对扩展开放，而设计的核心部分是经过精心考虑之后确定下来的基本结构，这部分应当是对修改关闭的，即不能因为用户的需求变化而再发生变化，因为这部分不是用来应对需求变化的。如果系统的设计遵守了“开-闭原则”，那么这个系统一定是易维护的，因为在系统中增加新的模块时不必去修改系统中的核心模块。

以下结合 5.10 节中的类来说明“开-闭原则”，5.10 节给出的 4 个类的 UML 图如图 5.14 所示。

在 5.10 节中，如果再增加一个 Java 源文件(对扩展开放)，该源文件有一个 Geometry 的子类 Triangle(负责计算三角形的面积)，那么 Pillar 类不需要做任何修改(对 Pillar 类的修改关闭)，应用程序就可以使用 Pillar 创建出具有 Geometry 的新子类指定的底的柱体。

如果将 5.10 节中的 Pillar 类、Geometry 类、Circle 类和 Rectangle 类看作一个小的开发

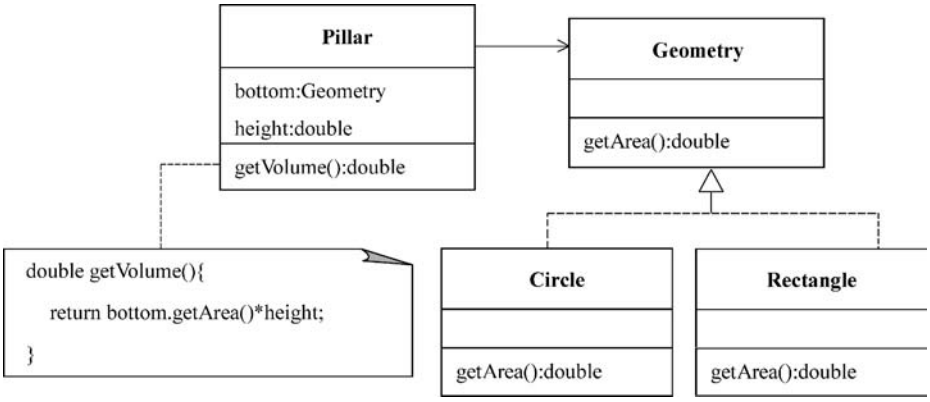

图 5.14　4 个类的 UML 图(1)

框架,将 Application.java 看作使用该框架进行应用开发的用户程序,那么框架满足“开-闭原则”,该框架相对用户的需求就比较容易维护,因为当用户程序需要使用 Pillar 创建出具有三角形底的柱体时,系统只需简单地扩展框架,即在框架中增加一个 Geometry 类的 Triangle 子类,而无须修改框架中的其他类,如图 5.15 所示。

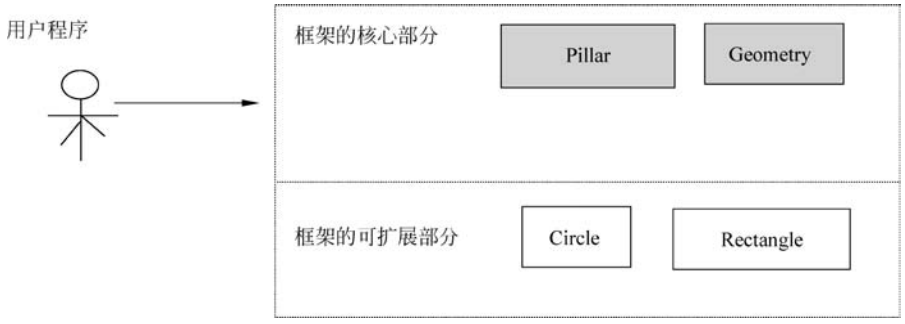

图 5.15　满足“开-闭原则”的框架

通常无法让设计的每个部分都遵守“开-闭原则”,甚至不应该这样去做,应该把主要精力用来集中应对设计中最有可能因需求变化而需要改变的地方,然后想办法应用“开-闭原则”。

视频讲解

5.12　应用举例

本章重点讲解了面向对象的两个特点,即继承与多态,并结合多态给出了面向抽象编程的核心思想。下面结合一个问题巩固本章的主要知识点。

用类封装手机的基本属性和功能,要求手机既可以使用移动公司的 SIM 卡,也可以使用联通公司的 SIM 卡(可以使用任何公司提供的 SIM 卡)。

❶ 问题的分析

如果设计的手机类中用某个具体公司的 SIM 卡,例如移动公司的,声明了对象,那么手机就缺少弹性,无法使用其他公司的 SIM 卡,因为一旦用户需要使用其他公司的 SIM 卡,就需要修改手机类的代码,例如增加用其他公司声明的成员变量。

如果每当用户有新的需求就会导致修改类的某部分代码,那么就应该将这部分代码从该类中分割出去,使它和类中其他稳定的代码之间是松耦合关系(否则系统将缺乏弹性,难以维

护),即将每种可能的变化对应地交给抽象类的子类去负责完成。

❷ 设计抽象类

根据以上对问题的分析,首先设计一个抽象类 SIM,该抽象类有 3 个抽象方法,即 giveNumber()、setNumber()和 giveCorpName(),那么 SIM 的子类必须实现 giveNumber()、setNumber()和 giveCorpName()方法。

❸ 设计手机类

设计 MobileTelephone 类(模拟手机),该类有一个 useSIM(SIM card)方法,该方法的参数是 SIM 类型。显然,参数 card 可以是抽象类 SIM 的任何一个子类对象的上转型对象,即参数 card 可以调用 SIM 的子类重写的 giveNumber()方法显示手机所使用的号码,调用子类重写的 giveCorpName()方法显示该号码所归属的公司。

在例子 13 中除了主类以外,还有 SIM 类及其子类 SIMOfChinaMobile(模拟移动公司提供的卡)、SIMOfChinaUnicom(模拟联通公司提供的卡)和 MobileTelephone 类。

图 5.16 是 MobileTelephone、SIM、SIMOfChinaMobile 和 SIMOfChinaUnicom 类的 UML 图,程序运行效果如图 5.17 所示。

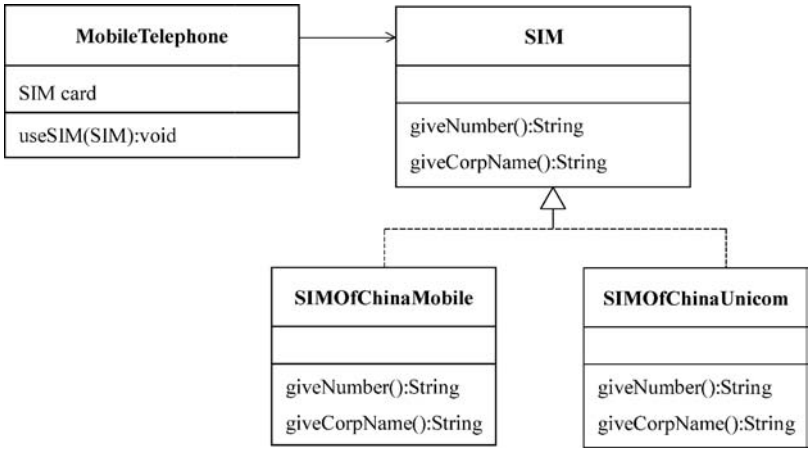

图 5.16 4 个类的 UML 图(2)

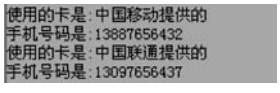

图 5.17 手机使用 SIM 卡

例子 13

SIM.java

```
public abstract class SIM {
    public abstract void setNumber(String n);
    public abstract String giveNumber();
    public abstract String giveCorpName();
}
```

MobileTelephone.java

```
public class MobileTelephone {
   SIM card;
   public void useSIM(SIM card) {
      this.card = card;
   }
```

```
    public void showMess() {
        System.out.println("使用的卡是:" + card.giveCorpName() + "提供的");
        System.out.println("手机号码是:" + card.giveNumber());
    }
}
```

SIMOfChinaMobile.java

```
public class SIMOfChinaMobile extends SIM {
    String number;
    public void setNumber(String n) {
        number = n;
    }
    public String giveNumber() {
        return number;
    }
    public String giveCorpName() {
        return "中国移动";
    }
}
```

SIMOfChinaUnicom.java

```
public class SIMOfChinaUnicom extends SIM {
    String number;
    public void setNumber(String n) {
        number = n;
    }
    public String giveNumber() {
        return number;
    }
    public String giveCorpName() {
        return "中国联通";
    }
}
```

Application.java

```
public class Application {
    public static void main(String args[]) {
        MobileTelephone telephone = new MobileTelephone();
        SIM sim = new SIMOfChinaMobile();
        sim.setNumber("13887656432");
        telephone.useSIM(sim);
        telephone.showMess();
        sim = new SIMOfChinaUnicom();
        sim.setNumber("13097656437");
        telephone.useSIM(sim);
        telephone.showMess();
    }
}
```

例子13中的类满足5.11节提到的“开-闭原则”,如果再增加一个Java源文件(对扩展开

放),该源文件有一个 SIM 的子类,例如 ChinaFeiTong 子类,那么 MobileTelephone 类不需要做任何修改(对 MobileTelephone 类的修改关闭),在应用程序中就可以让 telephone 对象使用 ChinaFeiTong 类提供的 SIM 卡。

5.13 小结

(1) 继承是一种由已有的类创建新类的机制。利用继承,可以先创建一个共有属性的一般类,再根据该一般类创建具有特殊属性的新类。

(2) 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,可以被子类中自己声明的任何实例方法操作。

(3) 所谓子类继承父类的方法作为子类中的一个方法,就像它们是在子类中直接声明一样,可以被子类中自己声明的任何实例方法调用。

(4) 子类继承的方法只能操作子类继承和隐藏的成员变量。

(5) 子类重写或新增的方法能操作子类继承和新声明的成员变量,但不能直接操作隐藏的成员的变量(需使用关键字 super 操作隐藏的成员变量)。

(6) 多态是面向对象编程的又一重要特性。子类可以体现多态,即子类可以根据各自的需要重写父类的某个方法,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。

(7) 在使用多态设计程序时要熟练使用上转型对象以及面向抽象编程的思想,以便体现程序设计所提倡的“开-闭原则”。

5.14 课外读物

课外读物均来自作者的教学辅助微信公众号 java-violin,扫描二维码即可观看、学习。

1. 青山原不老,绿水本无忧

2. 一骑红尘妃子笑,无人知是荔枝来

习 题 5

一、判断题(题目叙述正确的,在括号中打√,否则打×)

1. 子类继承父类的构造方法。 ( )

2. 在子类中想使用被子类隐藏的实例成员变量或实例方法就需要使用关键字 super。 ( )

3. 可以用 final 修饰构造方法。 ( )

4. 如果在子类的构造方法中没有显式地写出 super 关键字来调用父类的某个构造方法,那么编译器默认有“super();”调用父类的无参数的构造方法,如果父类没有这样的构造方法,

代码将出现编译错误。 ( )

5. 可以同时用 final 和 abstract 修饰同一个方法。 ( )

6. 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。 ( )

7. 如果一个类的所有构造方法的访问权限都是 private 的，意味着这个类不能有子类，理由是一个类的 private 方法不能在其他类中被使用，但子类的构造方法中一定会调用父类的某个构造方法。 ( )

8. 子类在进行方法重写时，不可以把父类的实例方法重写为类(static)方法，也不可以把父类的类(static)方法重写为实例方法。 ( )

9. 在 abstract 类中只可以有 abstract 方法。 ( )

10. 子类可以有多个父类。 ( )

二、选择题(单选或多选)

1. 下列叙述正确的是( )。

A. 子类继承父类的构造方法

B. abstract 类的子类必须是非 abstract 类

C. 子类继承的方法只能操作子类继承和隐藏的成员变量

D. 子类重写或新增的方法也能直接操作被子类隐藏的成员变量

2. 下列叙述正确的是( )。

A. final 类可以有子类

B. abstract 类中只可以有 abstract 方法

C. abstract 类中可以有非 abstract 方法，但该方法不可以用 final 修饰

D. 不可以同时用 final 和 abstract 修饰同一个方法

E. 允许使用 static 修饰 abstract 方法

3. 下列程序中，带有注释(A、B、C、D)的代码错误(无法通过编译)的是( )。(多选)

```
class Father {
   private int money = 12;
   float height;
   int seeMoney(){
      return money;                          //A
   }
}
class Son extends Father {
   int height;
   int lookMoney() {
      int m = seeMoney();                              //B
      return m;
   }
}
class E {
   public static void main(String args[]) {
      Son erzi = new Son();
      erzi.money = 300;                                //C
      erzi.height = 1.78F;                             //D
   }
}
```

4. 假设C是B的子类，B是A的子类，cat是C类的一个对象，bird是B类的一个对象，下列叙述错误的是(　　)。

A. cat instanceof B的值是true　　B. bird instanceof A的值是true

C. cat instanceof A的值是true　　D. bird instanceof C的值是true

5. 下列程序中，带有注释(A、B、C、D)的代码错误(无法通过编译)的是(　　)。

```
class A {
    static int m;
    static void f(){
        m = 20;                                    //A
    }
}
class B extends A {
    void f()                                       //B
    {   m = 222;                                   //C
    }
}
class E {
    public static void main(String args[]) {
       A.f();                                      //D
    }
}
```

6. 下列程序中，带有注释(A、B、C、D)的代码错误的是(　　)。

```
abstract class Takecare {
     protected void speakHello() {}                //A
     public abstract static void cry();            //B
     static int f(){return 0;}                     //C
     abstract float g();                           //D
}
```

7. 下列程序中，带有注释(A、B、C、D)的代码错误(无法通过编译)的是(　　)。

```
abstract class A {
    abstract float getFloat();                     //A
    void f()                                       //B
    { }
}
public class B extends A {
    private float m = 1.0f;                        //C
    private float getFloat()                       //D
    {   return m;
    }
}
```

8. 下列选项中，(　　)放入程序中标注的【代码】处将导致编译错误。

A. public float getNum(){return 4.0f;}

B. public void getNum(){ }

C. public void getNum(double d){ }

D. public double getNum(float d){return 4.0d;}

```
class A {
    public float getNum() {
        return 3.0f;
    }
}
public class B extends A {
      【代码】
}
```

9. 对于以下代码，下列叙述正确的是(　　)。

A. 程序提示编译错误(原因是 A 类没有不带参数的构造方法)

B. 编译无错误，【代码】的输出结果是 0

C. 编译无错误，【代码】的输出结果是 1

D. 编译无错误，【代码】的输出结果是 2

```
class A {
    public int i = 0;
    A(int m) {
      i = 1;
    }
}
public class B extends A {
     B(int m) {
        i = 2;
     }
     public static void main(String args[]){
        B b = new B(100);
        System.out.println(b.i); //【代码】
     }
}
```

三、挑错题(A、B、C、D 注释标注的哪行代码有错误?)

1.

```
abstract class AAA {
     final static void speakHello(){}                    //A
     final abstract void cry();                          //B
     static final int f(){return 0;}                     //C
     abstract float g();                                 //D
}
```

2.

```
abstract class Animal {
     int m = 100;
}
class Dog extends Animal{
     double m;
}
public class E {
     public static void main(String args[]){
```

```
        Animal animal = null;                    //A
        Dog dog = new Dog();
        animal = dog;                            //B
        dog.m = 3.14;                            //C
        animal.m = 3.14;                         //D
    }
}
```

四、阅读程序题

1. 请说出 E 类中【代码 1】和【代码 2】的输出结果。

```
class A {
   double f(double x,double y) {
     return x + y;
   }
}
class B extends A {
   double f(int x,int y) {
     return x * y;
   }
}
public class E {
   public static void main(String args[]) {
      B b = new B();
      System.out.println(b.f(3,5));              //【代码 1】
      System.out.println(b.f(3.0,5.0));          //【代码 2】
  }
}
```

2. 请说出 B 类中【代码 1】和【代码 2】的输出结果。

```
class A {
   public int getNumber(int a) {
      return a + 1;
   }
}
class B extends A {
   public int getNumber(int a) {
      return a + 100;
   }
   public static void main(String args[])  {
      A a = new A();
      System.out.println(a.getNumber(10));       //【代码 1】
      a = new B();
      System.out.println(a.getNumber(10));       //【代码 2】
   }
}
```

3. 请说出 E 类中【代码 1】～【代码 4】的输出结果。

```
class A {
double f(double x,double y) {
```

```
        return x + y;
    }
    static int g(int n) {
        return n * n;
    }
}
class B extends A {
    double f(double x,double y) {
        double m = super.f(x,y);
        return m + x * y;
    }
    static int g(int n) {
        int m = A.g(n);
        return m + n;
    }
}
public class E {
    public static void main(String args[]) {
        B b = new B();
        System.out.println(b.f(10.0,8.0));              //【代码 1】
        System.out.println(b.g(3));                     //【代码 2】
        A a = new B();
        System.out.println(a.f(10.0,8.0));              //【代码 3】
        System.out.println(a.g(3));                     //【代码 4】
   }
}
```

4. 请说出 E 类中【代码 1】～【代码 3】的输出结果。

```
class A {
    int m;
    int getM() {
        return m;
    }
    int seeM() {
         return m;
    }
}
class B extends A {
        int m;
        int getM() {
            return m + 100;
        }
}
public class E {
    public static void main(String args[]) {
        B b = new B();
        b.m = 20;
        System.out.println(b.getM());       //【代码 1】
        A a = b;
        a.m = -100;                         //上转型对象访问的是被隐藏的 m
```

```
        System.out.println(a.getM());          //【代码 2】
                                               //上转型对象调用的一定是子类重写的 getM()方法
        System.out.println(b.seeM());          //【代码 3】
                                               //子类继承的 seeM()方法操作的 m 是被子类隐藏的 m
    }
}
```

五、编程题(参考本章例子 13)

设计一个动物声音模拟器,希望模拟器可以模拟许多动物的叫声,要求如下:

(1) 编写抽象类 Animal。Animal 类有两个抽象方法 cry()和 getAnimalName(),即要求各种具体的动物给出自己的叫声和种类名称。

(2) 编写模拟器类 Simulator。该类有一个 playSound(Animal animal)方法,该方法的参数是 Animal 类型,即参数 animal 可以调用 Animal 的子类重写的 cry()方法播放具体动物的声音,调用子类重写的 getAnimalName()方法显示动物种类的名称。

(3) 编写 Animal 类的子类 Dog 和 Cat。图 5.18 是 Simulator、Animal、Dog、Cat 类的 UML 图。

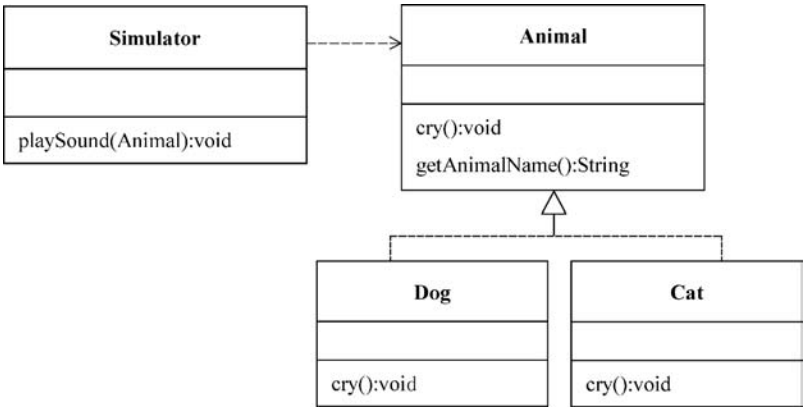

图 5.18 UML 图

(4) 编写主类 Application(用户程序)。在主类 Application 的 main 方法中至少包含如下代码:

```
Simulator simulator = new Simulator();
simulator.playSound(new Dog());
simulator.playSound(new Cat());
```