第 3 章

Java 类的继承与多态

本章介绍 Java 类的继承，abstract 和 final 等类，this、super 关键字的应用，以及引用 Java 对象成员的多态特性。

3.1 Java 类的继承

在面向对象程序设计中，类的继承(inheritance)是其最重要的特色之一。通过继承可以创建分等级层次的类，使得对对象的描述更加清晰；通过继承可以实现代码的复用，实现减少编写程序工作量的目的；通过继承可以实现重写类中的变量或方法，修改和完善类定义；通过继承可以实现在无源代码的情况下修改被继承的类；通过继承可以实现使用 Java 类库提供的类。

3.1.1 概念和语法

在面向对象程序设计体系中，类的继承是其重要的一种机制，被定义为对已有的类加以利用，并在此基础上为其添加新功能的一种方式。类的继承与人类遗传因子的继承类似，与之不同的是，Java 类的继承是单一的，而且继承是“完全”继承的，单一继承避免了多重继承出现的二义性等问题造成的继承混乱。

类的继承是通过 Java 关键字 extends 实现的，被继承的类称为父类，继承父类的类称为子类，类继承语句的语法格式为：

```
[modifer] class SubClassName extends SuperClassName{
  修饰符          子类名        继承      父类名
…;                                        //在父类的基础上新添加的功能
}
```

关键字 extends 是用来指示定义的类所继承的父类，由于 Java 体系规定 Java 类的继承是单一的，所以，extends 关键字后面只能指定一个父类。继承有时也被称为派生，子类是由父类派生而来的。

在 Java 语言体系中，所有的类都是有父类的，当定义的类没有关键字 extends 指出父

类，则该类默认的父类是 Object 类。

【示例 3-1】 定义 Sample 类代码。

```
public class Sample {                          //没有显现指出父类
  public static void main(String[ ] args) {
    System.out.println("Hello World!");
  }
}
```

Sample 类的父类是 Object 类，其等效的实际代码为：

```
public class Sample extends Object{            //显现指出父类
  public static void main(String[ ] args) {
    System.out.println("Hello World!");
  }
}
```

Object 类是 java.lang 基础包中的一个类，它被称为“根类”——所有类的根，Java 体系中所有的类最终的父类就是 Object 类，即 Java 所有的类都是 Object 类的子类，换句话说，Java 所有的类都是从 Object 类派生出来的，而 java.lang 包是 Java 语言体系提供编程的基础包。因此，Java 的编程是从一个已存在的类开始的，在此基础上再添加一些新的属于子类的代码，从而实现程序新功能的要求。

Java 的继承是完全的继承，即一个类继承了父类的所有状态（成员变量）和行为（成员方法）。例如，Sample 类继承了 Object 类的所有成员，Object 类中成员有 clone、equals、finalize、getClass、hashCode、notify、notifyAll、toString、wait 等方法，在用 Sample 类创建的类对象中，这些方法都可以被调用。因此，类的继承体现了 Java 代码的复用性。

Java 的继承是一个单一“链”状态的，继承不只是对其直接父类的继承，父类包括所有直接或间接被继承的类，继承是一直延续到 Object 类。

【示例 3-2】 继承演示示例。下述程序代码为父类代码：

```
class SuperSample {                            //其父类为 Object 类
  public void DisplaySuper(){                  //SuperSample 类中的方法
    System.out.println("This is a super - class.");
  }
}
```

SubSample 类继承 SuperSample 类，SubSample 是 SuperSample 类的子类，其代码为：

```
class SubSample extends SuperSample {          //继承 SuperSample 类
  public void DisplaySub(){                    //SubSample 类中的方法
    System.out.println("This is a sub - class.");
  }
}
```

一个类对象的创建是从其根类到父类对象开始的，创建一个子类对象时，最先创建的是

根类对象，然后依次创建父类对象，最后创建子类对象，但是，所有父类对象成员的引用都是通过子类对象实现的，就好像所有父类中的成员变量和方法全都包含在子类中一样。

【示例 3-3】 下面的 Sample 程序是当使用 SubSample 类创建一个对象 subSample 时，父类中 DisplaySuper()方法和子类中 DisplaySub()方法都包含在 subSample 对象中，同时根类 Object 中的方法也包含在 subSample 对象中，因为 SuperSample 类继承了 Object 类。使用 SubSample 类创建一个对象 subSample 以及应用对象成员的程序代码为：

```
public class Sample {
  public static void main(String[] args) {
    SubSample subSample = new SubSample();          //创建 SubSample 类对象
    subSample.DisplaySub();                         //调用 SubSample 类中方法
    subSample.DisplaySuper();                       //调用 SuperSample 父类中方法
    System.out.println("This is a " +
      subSample.getClass().getName());              //调用 Object 根类中的方法
  }
}
```

类的继承允许子类使用父类的变量和方法，如同这些变量和方法是属于子类本身的一样，但是，类的继承是有权限的，子类可以继承父类中访问权限设定为 public，protected 的成员变量和方法，不能继承访问权限为 private 的成员变量和方法。

【示例 3-4】 Java 类的继承实际上是子类对父类功能的扩展。例如一个描述平面直角坐标系的类为 PlaneCoordinate，其中有 x 和 y 两个方向的坐标量。其程序代码为：

```
public class PlaneCoordinate {                      //定义平面直角坐标系类
  int x;                                            //声明 x 坐标变量
  int y;                                            //声明 y 坐标变量
  public PlaneCoordinate() {                        //设置坐标原点处坐标为(0,0)
    x = 0;
    y = 0;
  }
}
```

当需要定义一个空间直角坐标系类 SpaceCoordinate 时，只需要在继承了平面直角坐标系 PlaneCoordinate 类的子类中添加一个 z 方向的坐标量，其程序代码为：

```
public class SpaceCoordinate extends PlaneCoordinate { //定义空间直角坐标系类
    int z;                                          //声明 z 坐标变量
    public SpaceCoordinate() {                      //设置坐标 z 原点处的值
      z = 0;
    }
}
```

上述程序是在已有平面直角坐标系 PlaneCoordinate 类的基础上扩展为 SpaceCoordinate 类，只添加一个坐标量 z 就可以实现描述空间直角坐标系的功能。

另外，当一个父类被验证没有错误时，例如 Java 语言提供的类库，通过子类继承后，只

是在子类中添加新代码，新代码对父类没有任何影响。因此，如果子类创建的对象在运行时发生错误，则其错误将被限定在子类代码中，与父类代码无关。

3.1.2　Java 类继承关系的测试

Java 类都是有继承关系的，每个类至少有一个父类，还可以有多个父类，Java 运行系统提供了动态测试一个对象是属于哪些类（含父类）的实例的功能。Java 关键字 instanceof 就是用于实现该功能的，instanceof 是一个双目对象运算符，使用 instanceof 运算符构成表达式的语法格式为：

```
[ boolean b = ] oneObj instanceof ClassName;              //布尔表达式
[   布尔变量 = ] 对象   关键字        类名;
```

由 instanceof 运算符构成的表达式功能是判断 oneObj 对象是否为 ClassName 类（或者 ClassName 类的父类）的一个实例，当 oneObj 对象是 ClassName 类的一个实例时，该表达式的值为 true，否则为 false。instanceof 运算符的功能是可以明确子类继承了哪些父类，当该表达式的值为 true 时，说明创建 oneObj 对象的类与 ClassName 类是一个类，或者有父子关系，当该表达式的值为 false 时，说明两个类没有关系。

【示例 3-5】 使用 instanceof 运算符的 Java 应用程序。

```
class SuperSample {                              //父类
}
class SubSample extends SuperSample {           //子类
}
public class Sample {
  public static void main(String[ ] args) {
    SubSample subSample = new SubSample();  //创建 SubSample 类对象
    boolean b = subSample instanceof SubSample;
                                                 //测试 subSample 对象是否为 SubSample 类的实例
    System.out.println(b);                       //输出显示判断结果
    System.out.println(subSample instanceof SuperSample);
                                                 //测试 subSample 对象是否为 SuperSample 类的实例
    System.out.println(subSample instanceof Object);
                                                 //测试 subSample 对象是否为 Object 类的实例
  }
}
```

该程序执行结果为显示 3 个 true，因为 subSample 对象是所有被测试类的实例。

3.1.3　隐藏、覆盖和重载

在面向对象程序设计中，由于 Java 的类都具有继承关系，并且所有的类都有父类，因此，在继承的过程中将会发生隐藏（hidden）、覆盖（override）和重载（overload）现象。隐藏、覆盖和重载都是针对父类中的非 private 成员变量和方法而言的，访问权限为 private 的变

量和方法不会被继承到子类中，因此也就不存在隐藏、覆盖和重载的现象。

1. 隐藏

隐藏现象发生于子类与父类之间，隐藏是针对父类中成员变量和静态方法而言的。当在子类中声明与父类中成员变量具有相同变量名的变量时，则实现了对父类中成员变量的隐藏。

【示例 3-6】 成员变量隐藏的演示程序。

```
class SuperSample {                          //父类
  int x = 10;                                //在父类中声明的变量 x
}
class SubSample extends SuperSample {       //继承 SuperSample 类的子类
  int x = 20;                                //在子类中声明的变量 x
}
public class Sample {                        //应用子类创建对象
  public static void main(String[] args) {
    SubSample subSample = new SubSample();
    int z = subSample.x;                     //访问的是子类对象中的 x 变量
    System.out.println(z);
  }
}
```

当创建一个子类 SubSample 的对象时，该对象的成员变量只有子类中声明的变量 x，在父类中声明的变量 x 被隐藏了，通过子类对象是访问不到父类中的 x 变量的。

当在子类中声明与父类中静态成员方法具有相同的方法名并具有相同的输入参数列表和相同的返回类型的方法，即子类与父类中的方法完全相同时，则实现了对父类中静态成员方法的隐藏。

【示例 3-7】 成员方法隐藏的演示程序。

```
class SuperSample {                          //父类
  public static void Display(){              //父类中的方法
    System.out.println("This is a super-class.");
  }
}
class SubSample extends SuperSample {       //继承 SuperSample 类的子类
  public static void Display(){              //与父类完全相同的方法
    System.out.println("This is a sub-class.");
  }
}
public class Sample {
  public static void main(String[] args) {
    SubSample subSample = new SubSample();
    subSample.Display();                     //引用的是子类中的 Display()方法
  }
}
```

在父类中的 Display()方法永远不能通过子类创建的对象实现引用，因为 Java 运行环境不可能为两个完全相同的静态方法分配不同的存储空间，因此，就父类的静态方法而言，实现对该方法的隐藏。

2．覆盖

覆盖也称为重写，覆盖现象发生于子类与父类之间，是指在子类中声明一个与父类具有相同的方法名、输入参数列表、返回值、访问权限等的方法，不同之处只有方法体，即在子类中重新编写方法实现的功能。覆盖常用于替换父类相同的方法，实现功能的更新。

【示例 3-8】 在子类中实现对父类中方法覆盖的 Java 演示程序。

```
class SuperSample {                          //父类
  public void Display(){                     //父类中的 Display()方法
    System.out.println("This is a super-class.");
  }
}
class SubSample extends SuperSample {        //继承 SuperSample 类的子类
  public void Display(){                     //覆盖父类中的 Display()方法
    System.out.println("This is a sub-class.");
  }
}
```

覆盖不同于静态方法的隐藏，父类中被隐藏的方法在子类中是完全不可用的，而父类中被覆盖的方法在子类中是可以通过其他方式被引用的。另外，覆盖常被用于对接口中声明的方法的实现。

3．重载

重载现象可以发生在子类与父类之间，也可以发生在同一个类中。重载是指在子类与父类之间或在同一类中定义多个具有相同的方法名、访问权限等的方法，这些方法的区别在于它们可以有不同的返回类型，或者有不同的输入参数列表，在参数列表中参数类型、参数个数、参数顺序等至少有一个是不相同。重载也可以看作定义不同的方法。

【示例 3-9】 发生在子类与父类之间的重载方法的 Java 演示程序。

```
class SuperSample {                                  //父类
  public void Display(){                             //父类中的 Display()方法
    System.out.println("This is a super-class.");
  }
}
class SubSample extends SuperSample {                //继承 SuperSample 类的子类
  public void Display(String newS){                  //重载父类中的 Display()方法
    System.out.println(newS);
  }
}
public class Sample {
  public static void main(String[] args) {
    SubSample subSample = new SubSample();
```

```
    subSample.Display();                          //调用父类的 Display()方法
    subSample.Display("This is a sub-class.");    //调用子类的 Display()方法
  }
}
```

【示例 3-10】 发生在同一类中重载方法的 Java 演示程序。

```
class OverloadSample {
  public void Display(){                          //类中的 Display()方法
    System.out.println("This is a method of Display.");
  }
  public void Display(String newS){               //重载类中的 Display()方法
    System.out.println(newS);
  }
}
public class Sample {
  public static void main(String[] args) {
    OverloadSample os = new OverloadSample();
    os.Display();                                 //调用无输入参数的 Display()方法
    os.Display("This is another method of Display.");
                                                  //调用另一个有输入参数的 Display 方法
  }
}
```

3.1.4 构造方法的重载

在创建对象的语句格式中,关键字 new 后指示的是调用类的构造方法,当一个类没有定义构造方法时,其创建类对象时调用无任何操作的默认构造方法,一个类默认的构造方法是指无形式参数列表的构造方法,在创建类对象的同时调用父类和根类的无形式参数列表的(默认)构造方法创建子类的对象。

【示例 3-11】 默认父类构造方法创建子类对象的 Java 演示程序。

```
class SuperSample {                               //父类
  public SuperSample(){                           //父类的默认构造方法
    System.out.println("This is a super-constructor.");
  }
}
class SubSample extends SuperSample {             //子类
}
public class Sample {
  public static void main(String[] args) {
    SubSample subSample = new SubSample();        //创建对象
  }
}
```

上述程序在 main()方法中创建 subSample 对象时,其输出显示为:

```
This is a super - constructor.                    //父类构造方法显示输出
```

该输出为 SubSample 类的父类被创建时调用其构造方法得到的输出显示。

当子类有自身的无形式参数列表的构造方法时，其创建对象时先创建父类对象，再创建子类对象。

【示例 3-12】 创建子类对象的同时也创建父类对象的 Java 演示程序。

```
class SuperSample {                               //父类
  public SuperSample(){                           //父类的默认构造方法
    System.out.println("This is a super - constructor.");
  }
}
class SubSample extends SuperSample {             //子类
  public SubSample(){                             //子类的默认构造方法
    System.out.println("This is a sub - constructor.");
  }
}
public class Sample {
  public static void main(String[] args) {
    SubSample subSample = new SubSample();        //创建对象
  }
}
```

上述程序在 main()方法中创建 subSample 对象时，其输出显示为：

```
This is a super - constructor.
This is a sub - constructor.
```

其输出显示说明了创建对象的顺序。

当一个类有多个构造方法时，则发生构造方法的重载现象，创建类对象时需要指明使用哪个构造方法实现对象的创建。

【示例 3-13】 多个构造方法创建类对象的 Java 演示程序。

```
public class Sample {
  public Sample(){                                //无形参的默认构造方法
    System.out.println("no parameterList - constructor");
  }
  public Sample(String newS){                     //有形参的构造方法
    System.out.println(newS);
  }
  public static void main(String[] args) {
    Sample sample1 = new Sample();                //用默认的构造方法创建一个对象
    Sample sample2 = new Sample("parameterList - constructor");
                                                  //用非默认的构造方法创建另一个对象
  }
}
```

上述程序在 main()方法中分别使用不同的构造方法创建两个对象 sample1 和 sample2。

当一个父类的构造方法出现重载现象时，即有多于一个构造方法时，并在父类中有默认的构造方法，应用子类创建类对象时，先创建的父类对象是使用父类中默认构造方法创建的，当父类没有默认构造方法，只有非默认构造方法时，在继承该父类的子类中需要显式调用父类的某一个非默认的构造方法来实现父类对象的创建。

【示例 3-14】 显式调用父类构造方法的 Java 演示程序。

```
class SuperSample {                                    //无默认构造方法的父类
  public SuperSample(String newS){                     //父类中非默认的构造方法 1
    System.out.println(newS);
  }
  public SuperSample(int newX){                        //父类中非默认的构造方法 2
    System.out.println(newX);
  }
}
class SubSample extends SuperSample {                  //子类
  public SubSample(){
    super("super - constructor.");                     //显式调用父类的一个非默认的构造方法
  }
}
public class Sample {
  public static void main(String[] args) {
    SubSample subSample = new SubSample();
  }
}
```

关键字 super 相当于指向父类构造方法的指针，通过它指定创建父类对象时所需要使用的父类中的一个非默认的构造方法。

3.2 abstract 和 final 修饰符

abstract(抽象的)和 final(终态的)是 Java 的两个修饰符，都可以修饰类。被 abstract 修饰的类称为抽象类，抽象类是不可使用的，Java 体系不允许创建抽象类的类对象，抽象类是作为父类应用的，需要由其他类继承后方可使用。被 final 修饰的类称为终态类，终态类是不能作为父类使用的，是不允许被其他的类继承的。另外，abstract 和 final 修饰符还可以修饰类中的方法，final 修饰符可以修饰类中的变量。

3.2.1 abstract 修饰符

abstract 修饰符可以修饰类，也可以修饰方法，但是，只能修饰抽象类中的方法。

顾名思义，抽象类在概念上描述的就是抽象世界。例如，动物相对于具体的狗、猫等就是一个抽象的概念；数字相对于具体的整数、浮点数等也是一个抽象的概念。抽象类是一

种特殊的类，抽象类的定义方式是在类定义的前面加上修饰符 abstract，抽象方法的定义也是类似的。其语法格式为：

```
abstract class abstractClass {                          //声明抽象类
  …;                                                    //定义类的普通成员
  abstract ReturnType methodName(parameterList);        //声明抽象方法
    抽象的      返回类型      方法名      形参列表
}
```

与普通类不同的是在抽象类中允许声明类中方法为 abstract 抽象的，但是抽象方法应为没有方法体的空方法，因为“抽象”，没有具体内容，所以抽象方法为空方法，只是在抽象类中声明有这个方法而已，当使用 abstract 声明方法时，不能与关键字 static、private 或 final 同时使用。

因为抽象类不能创建其类对象，只能作为父类被应用，因此，需要非抽象的子类来继承抽象类，同时在子类中还需要覆盖抽象类中的抽象方法，为该方法填写具体内容（方法体）。

另外，当一个类的定义完全表示抽象概念时，就不应该被实例化为一个对象。例如，Java 基础类库中的 Number（数字）类就是一个抽象类，只表示数字这一抽象概念，只有作为整数类 Integer 或实数类 Float 等的父类时它才具有其意义。

【示例 3-15】 描述动物的抽象类的 Java 演示程序代码。

```
abstract class Animal {                                 //定义动物抽象类
  private String s_Name;                                //声明动物名称变量
  private String s_Kind;                                //声明动物类别变量
  public String getS_Name() {                           //声明方法
    return s_Name;
  }
  public void setS_Name(String new_Name) {
    s_Name = new_Name;
  }
  public String getS_Kind() {
    return s_Kind;
  }
  public void setS_Kind(String new_Kind) {
    s_Kind = new_Kind;
  }
  public abstract void Action();                        //声明动物行为的抽象方法
}
```

在抽象类 Animal 中的普通方法是针对所有动物的，例如操作动物名称、类别等，而 Action()方法是描述动物行为的，其行为方式是不确定的，需要视具体的动物而定，所以被修饰为 abstract 抽象的。

下面的程序是一个继承 Animal 类的描述“狗”的类，其程序代码为：

```
public class Dog extends Animal {                       //定义"狗"的具体类
  private boolean wildness;                             //声明子类特有的属性
```

```
    public void setWildness(boolean new_Wildness){
      wildness = new_Wildness;
    }
    public boolean isWildness(){                          //声明子类特有的方法
      return wildness;
      }
    public void Action(){                                 //覆盖抽象类中的抽象方法
      System.out.println("吃、喝、运动型等");              //编写方法实现的具体内容
    }
    public static void main(String[] args) {
      Dog dog = new Dog();                                //创建一个"狗"的对象
      dog.setS_Name("狗");                                //调用 dog 对象的 setS_Name()方法
      dog.setS_Kind("犬科");                              //调用抽象类中声明的方法
      dog.setWildness(false);                             //调用子类中声明的方法
      System.out.println(dog.getS_Name());
      System.out.println(dog.getS_Kind());
      System.out.println(dog.isWildness());
      dog.Action();                                       //调用 dog 对象的 Action()方法
    }
}
```

当抽象类中声明了抽象方法时，则在继承抽象类的子类中一定要覆盖抽象方法，并实现方法体的内容，例如上述程序在抽象类 Animal 中定义的 Action()方法，其内容是在继承了 Animal 类的 Dog 类中实现的，因为 Dog 类是描述具体的“狗”对象，因此有其行为方式，抽象方法被覆盖后，其调用方式与普通方法一样。

抽象类描述的是事物所具有的共性，共同拥有的特征和行为，为其他需要继承的类提供一个基础类，当子类继承抽象类将共性完全继承时，还可以定义子类固有的与其他事物有别的特征和行为，例如在上述 Dog 程序中定义的 wildness(野生)属性等。

3.2.2 final 修饰符

Java 语言体系出于安全性或面向对象设计上的考虑，有时希望一些类不能被继承，一些类、方法或变量不能被改变；或者一个类定义得非常完美，不需要再进行修改或扩展；或者该类、方法或变量对编译器和解释器的正常运行非常重要，不能轻易动态改变的；或者确保类、方法或变量的唯一性等，使用 final 修饰符可以达到上述目的。另外，final 修饰符与 abstract 修饰符是不能同时使用的。

关键字 final 是用于类、方法或变量的修饰符。当 final 出现在类的声明中时，表示这个类不能有子类或被继承，是一个终态类；当 final 关键字出现在方法声明中时，表示该方法不能被覆盖，使用关键字 static 或 private 声明的方法隐含有 final 修饰功能；当 final 出现在变量的声明中时，表示该变量为常量，常量需要有初始化，即要被赋予一个固定的值，赋值操作可以在声明常量时进行，也可以在类的构造方法中进行。final 修饰符的使用语法格式为：

```
final class finalClass {                                    //声明终态类
  final variableName [ = initializationValue ];             //声明常量
  final ReturnType methodName(parameterList);               //声明终态方法
  终态的  返回类型     方法名     形参列表
}
```

例如，在 Java 的基础类库中，String(字符串)类就被声明为 final 型的，它保证了 String 数据类型的唯一性。

【示例 3-16】 使用 final 修饰符修饰的类、方法和变量的 Java 演示程序。

```
final class FinalClass{                                     //final 修饰的类
    final String str = "final Data";                        //final 修饰的变量
    public String str1 = "non final data";
    final public void print(){                              //final 修饰的方法
      System.out.println("Final Method.");
    }
    public void what(){
      System.out.println(str + "\n" + str1);
    }
}
public class FinalSample {                                  //使用 FinalClass 类,但不能继承
  public static void main(String[] args){
    FinalClass f = new FinalClass();                        //创建 FinalClass 类对象 f
    f.what();                                               //调用 f 对象中的方法 what()
    f.print();                                              //调用 f 对象中的方法 print()
  }
}
```

因为当一个类被修饰为 final 时，它无法被任何类继承，继承“链”也就到了终点(节点)，因此 final 类也被称为“叶子类”，在叶子类中定义的方法是属于叶子类的，自然也就成为 final 型的。因此，在叶子类中的方法被修饰为 final 型是没有意义的，但是，在其他非叶子类中定义 final 型方法是有意义的，即该方法在被继承的子类中是不能被覆盖的。

3.3　this 和 super 变量

面向对象程序设计体系的继承性会出现变量和方法的隐藏和覆盖等现象，为避免访问变量和调用方法出现二义性，在 Java 语言体系中设计了 this 和 super 两个变量。

关键字 this 和 super 是作为两个变量被应用的，更确切地说是两个“指针”变量，this 指向当前类对象，super 指向父类对象。当需要访问或调用当前类对象的变量或方法时，使用 this 变量，当需要访问或调用当前类的父类对象的变量或方法时，使用 super 变量。

3.3.1　this 变量

this 变量是针对访问或调用当前类对象的变量或方法而言的，this 变量可作为当前类

对象使用，其访问变量和调用方法的作用域为整个类。this 变量使用的语法格式为：

```
this.variableName [methodName([parameterList])][ expression ];
        变量名   或   方法名      形式参数          表达式
```

【示例 3-17】 通过 this 变量访问类中变量和调用类中方法的 Java 演示程序。

```
class AccessVandM{                                  //定义类
  private int x;                                    //声明变量 x,作用域为整个类
  public void OneMethod() {                         //定义一个方法
    this.x = 20;                                    //访问变量 x
    this.AnOtherMethod();                           //调用 AccessVandM 类的 AnOtherMethod()方法
  }
  public void AnOtherMethod() {                     //定义另一个方法
    System.out.println(this.x);
  }
}
public class Sample {
  public static void main(String[] args) {
    AccessVandM avm = new AccessVandM();            //创建 AccessVandM 类对象 avm
    avm.OneMethod();                                //调用 avm 对象的 OneMethod()方法
  }
}
```

this 变量是一个特殊的实例值，它用来在类中一个成员方法内部指向当前的对象。在 this 使用的语法格式中，this 访问的变量和调用的方法实际上是表达式中的一个元素，由 this 构成的语句实现的是一种操作，因此，应该在方法体中完成，但是 this 变量不能使用在静态方法体中，因为 this 不是静态变量。例如，不能在静态的 main()方法中使用 this 变量。

当在类中声明的变量与在方法中声明的变量具有相同的名字时，由于两个同名变量的作用域是不同的，因此，在方法体中使用 this 可以访问整个类作用域的变量。

【示例 3-18】 this 作用域演示的 Java 程序。

```
class AccessVariable {                              //定义类
  private int x;                                    //声明变量 x,作用域为整个类
  public void OneMethod() {                         //定义方法
    int x = 10;                                     //声明方法内变量 x,作用域为方法体内
    this.x = 20;                                    //访问作用域为整个类的 x 变量
    System.out.println(x);
    System.out.println(this.x);
  }
}
public class Sample {
  public static void main(String[] args) {
    AccessVariable av = new AccessVariable();
    av.OneMethod();                                 //调用 AccessVariable 类对象的方法
  }
```

```
}
```

当在类中声明的变量与在方法的形式参数列表中声明的变量具有相同的名字时，在方法体中使用 this 将访问变量指向类声明的变量。

【示例 3-19】 使用 this 避免二义性的 Java 演示程序。

```
class AccessVariable {
  private int x;                                  //声明变量 x,作用域为整个类
  public void OneMethod(int x) {                  //在方法的形式参数列表中声明变量 x
    this.x = x;                                   //为类中变量 x 赋值
                                                  //如果不用 this,则 x = x;将产生二义性
    System.out.println(this.x);                   //输出显示类中变量 x
  }
}
public class Sample {
  public static void main(String[] args) {
    AccessVariable av = new AccessVariable();
    av.OneMethod(20);                             //调用 AccessVariable 类对象的方法
  }
}
```

this 的另一个用途是调用当前类的构造方法，其语法格式为：

```
this([parameterList]);
        形式参数
```

该应用形式只针对类的构造方法，其使用也在构造方法体内，并且是构造方法体内的第一行语句，this 可调用当前类的带输入参数和无输入参数的构造方法。

【示例 3-20】 使用 this 调用当前类构造方法的 Java 演示程序。

```
public class Sample {                             //定义类
  private int x;                                  //声明变量
  private int y;
  private String s;
  public Sample(int newX){                        //定义一个带输入参数的构造方法
    this.x = newX;                                //为变量 x 赋初始值
  }
  public Sample() {                               //定义默认的构造方法,this 为第一条语句
    this(0);                                      //调用本类中带输入参数的构造方法
    this.y = 0;                                   //为变量 y 赋值
  }
  public Sample(String newS) {                    //在此构造方法中 this()是第一条语句
    this();                                       //调用本类的默认构造方法
    this.s = newS;
  }
}
```

上述程序的原意是无论在什么情况下都需要在构造方法中为 x 变量赋予初始值，带输入参数的构造方法并非默认的构造方法，当该类需要有一个默认构造方法(用于继承等)，同时又要求为 x 变量赋予初始值时，则在默认构造方法中使用 this 指定调用一个带输入参数的构造方法；当使用其他构造方法创建对象并为 x 和 y 变量赋予初始值时，则可通过 this 调用默认的构造方法。

3.3.2 super 变量

super 变量是在当前类中访问或调用其父类对象的变量或方法的。super 变量可作为当前类的父类对象使用，super 访问的是父类非 private 变量和调用的是父类非 private 方法。super 变量使用的语法格式为：

```
super.variableName [methodName([parameterList])] [ expression ];
          变量名   或   方法名      形式参数            表达式
```

【示例 3-21】 通过 super 变量访问直接父类中变量和调用直接父类中方法的 Java 演示程序。

```
class SuperSample {                              //定义父类
  private int x;                                 //声明变量 x
  int y;                                         //声明变量 y,非 private 型
  public int getX() {                            //定义读变量 x 的方法
    return x;
  }
  public void setX(int x) {                      //定义写变量 x 的方法
    this.x = x;                                  //为类中变量 x 赋值
  }
}
class SubSample extends SuperSample {            //定义继承 SuperSample 类的子类
  public void AccessVandM(){                     //定义类中方法
    super.setX(20);                              //调用父类 setX()方法
    super.y = 10;                                //访问父类中的 y 变量
  }
}
```

super 变量也是一个实例值，用来在子类中一个成员方法内部指向当前类的父类对象，在 super 使用的语法格式中，super 访问的变量和调用的方法实际上是表达式中的一个元素，由其构成的语句实现的是一种操作，因此，super 操作应该在子类的方法体中实现。

有继承关系的子类和父类都存在着变量、方法的隐藏和覆盖等现象，使用 super 和 this 可分别操作隐藏和被隐藏、覆盖和被覆盖的变量和方法，super 实现了访问在父类中被隐藏的变量和调用被覆盖的方法。

【示例 3-22】 通过 super 访问在父类中被隐藏变量和调用被覆盖方法的 Java 演示程序。

```
class SuperSample {                              //定义父类
    String s = "Super Variable";                 //声明变量 s,在父类中,继承后被隐藏
    public void printMethod() {                  //定义父类中方法 printMethod(),被覆盖
      System.out.println("Super Method");
    }
}
class SubSample extends SuperSample {           //定义子类
    String s = "this Variable";                  //声明变量 s,在子类中,隐藏了父类的 s
    public String getS(){                        //定义子类中方法
      return this.s;
    }
    public void printMethod() {                  //定义子类中与父类同名的方法,覆盖父类方法
      System.out.println("this Method");
    }
    public void AccessSuper(){                   //定义子类中方法
      System.out.println(super.s);               //访问父类中被隐藏的 s 变量
      super.printMethod();                       //调用父类被覆盖的 printMethod()方法
    }
    public static void main(String[] args){      //定义 main 方法
      SubSample ss = new SubSample();            //创建子类对象 ss
      System.out.println(ss.getS());             //输出显示子类变量 s
      ss.printMethod();                          //调用 ss 对象的 printMethod()方法
      ss.AccessSuper();                          //调用 ss 对象的 AccessSuper()方法
    }
}
```

在子类中,通过 AccessSuper()方法访问了父类被隐藏的 s 变量和调用了父类被覆盖的 printMethod()方法。

super 的另外一个用途是在子类的构造方法中调用直接父类的构造方法,其语法格式为:

```
super([parameterList]);
        形式参数
```

super 的应用形式也是只针对类的构造方法的,其使用在子类的构造方法体内,并且是构造方法体内的第一行语句,super 可调用直接父类的带输入参数和无输入参数的构造方法,例如示例 3-23。

【示例 3-23】 通过 super 调用直接父类构造方法的 Java 演示程序。

```
class SuperSample {                              //定义父类
  private int x;                                 //声明变量
  private int y;
  public SuperSample() {                         //定义默认构造方法
  }
  public SuperSample(int newX,int newY) {        //定义带输入参数的构造方法
    this.x = newX;                               //为变量赋予初始值
```

```
    this.y = newY;
  }
}
class SubSample extends SuperSample {          //定义继承 SuperSample 的子类
  String s ;                                   //声明变量
  public SubSample(){                          //定义子类默认构造方法
    super();                                   //在第一行调用父类默认构造方法
    this.s = "Sub - Class";
  }
  public SubSample(int newX,int newY){         //定义子类构造方法
    super(newX,newY);                          //调用父类的其他构造方法
  }
}
```

在一般情况下,当父类定义了构造方法时,在继承的子类构造方法中应先调用父类的构造方法实现父类对象的创建和初始化。

3.4 Java 的多态性

多态性是面向对象程序设计的一大特征,多态体现了程序的可扩展性,在应用程序运行时多态又体现了程序代码的重复使用特性。

3.4.1 多态的概念

在面向对象程序设计中,描述一个对象时,多态指的是一个对象的行为方式可以有多种操作形态,根据对象的不同进行不同的操作,因此多态是与对象关联的,这种关联被称为绑定(binding)。绑定分为静态绑定和动态绑定,静态绑定是在编译时完成的,动态绑定则是在程序运行时完成的。绑定与类的继承相结合使对象呈现出多态性,达到一次编写代码多次使用的目的。

Java 类中方法的重载呈现出多态的特性,它属于静态绑定。例如,实现两个数的加法运算有两个整数相加、两个长整数相加、两个浮点数相加等。下面的程序是定义了实现两个数的加法运算的类和使用该类的应用程序。

【示例 3-24】 应用加法运算类的 Java 演示程序。

```
public class AddSample {                       //定义两个数 x 和 y 相加类
  public int Add(int x,int y){                 //两个整数相加
    return x + y;
  }
  public long Add(long x,long y){              //两个长整数相加
    return x + y;
  }
  public double Add(double x,double y){        //两个浮点数相加
    return x + y;
```

```
  }
}
public class Sample {                          //定义应用类
  public static void main(String[ ] args){
    int i_x = 10;                              //声明数据变量
    int i_y = 20;
    long l_x = 123456789;
    long l_y = 987654321;
    double d_x = 10e-12;
    double d_y = 20e-15;
    AddSample as = new AddSample();            //创建 AddSample 对象 as
    int i_z = as.Add(i_x,i_y);                 //调用整数相加方法
    long l_z = as.Add(l_x,l_y);                //调用长整数相加方法
    double d_z = as.Add(d_x,d_y);              //调用浮点数相加方法
  }
}
```

虽然名为 Add 的方法都是实现相加操作，完成一种操作，但是每个 Add()方法都是针对不同的数据类型的，由返回类型和输入参数决定调用哪个 Add()方法，上述程序为 Add()方法的重载，体现了面向对象程序设计的多态特性。另外，当在子类中覆盖父类中同名的方法时，也是多态特性的一种表现，因为在子类中可以通过 super 变量访问父类中被覆盖的方法。

绑定与类的继承相结合则可体现出动态绑定的多态特性，当方法的覆盖存在于父、子类之间时，Java 的多态特性体现在运行 Java 程序时动态方法调用上。多态特性发生在程序运行期间，并非在程序编译指定调用重载方法期间。下面的示例程序是在运行期间发生的多态特性。

【示例 3-25】 多态特性的 Java 演示程序。

```
abstract class SuperSample {                   //定义抽象父类
  public void Display(){                       //SuperSample 类中的方法
    System.out.println("This is a super-class.");
  }
}
public class SubSample_A extends SuperSample{ //定义 SubSample_A 子类
  public void Display(){                       //覆盖 SuperSample 类中的方法
    System.out.println("This is a subSample_A-class.");
  }
}
public class SubSample_B extends SuperSample{ //定义另一个 SubSample_B 子类
  public void Display(){                       //覆盖 SuperSample 类中的方法
    System.out.println("This is a subSample_B-class.");
  }
}
public class SubSample_C extends SuperSample{ //定义另一个 SubSample_C 子类
```

```
  public void DisplaySubSample_C(){            //定义子类自身的方法,非覆盖
    System.out.println("This is a subSample_C - class.");
  }
}
public class Sample {                          //定义应用父、子类的 Sample 类
  public static void main(String[] args){
    SuperSample f;                             //声明父类变量 f
    SubSample_A a = new SubSample_A();         //创建 SubSample_A 类对象 a
    SubSample_B b = new SubSample_B();         //创建 SubSample_B 类对象 b
    SubSample_C c = new SubSample_C();         //创建 SubSample_C 类对象 c
    f = a;                                     //将对象 a 赋给类变量 f
        //它相当于 SuperSample f = new SubSample_A();
    f.Display();                               //调用 f 对象中的 Display()方法
    f = b;                                     //将对象 b 赋给类变量 f
    f.Display();                               //调用 f 对象中的 Display()方法
    f = c;                                     //将对象 c 赋给类变量 f
    f.Display();                               //调用 f 对象中的 Display()方法
  }
}
```

运行 Sample 程序,其输出结果为:

```
This is a subSample_A - class.
This is a subSample_B - class.
This is a super - class.
```

当父类对象 f 被赋予子类对象 a 时,其调用的方法为子类 SubSample_A 中定义的覆盖父类的 Display()方法,当重新为父类对象 f 赋予为 b 对象时,其调用的方法改为子类 SubSample_B 中定义的覆盖父类的 Display()方法,当父类对象 f 被赋予子类对象 c 时,其调用的方法为父类定义的 Display()方法,因为在 SubSample_C 类中并没有定义 Display()方法,没有发生覆盖现象。该程序显示了在运行时动态调用不同的方法,实现了 Java 程序运行时的多态特性。

上述示例将子类对象类型作为创建父类对象的基本对象类型的处理过程被称为“上溯造型,Upcasting”或“回溯造型”,子类对象是已经创建好的现成对象模型,是继承了父类的,因此其父类对象可以由子类对象构造生成,实现“上溯,Up”或“回溯”的“造型,Casting”。

另外,“上溯造型”功能也是充分体现了 Java 在动态时的多态特性。子类对象是在父类的指定范围内重新动态构造父类对象的类型模型,例如下面的示例程序是当声明一个父类变量并为其赋予子类对象时,其属性和行为被限定在父类范围内。

【示例 3-26】 父类限定子类的 Java 演示程序。

```
class SuperSample {                            //定义父类
  public void Display(){                       //SuperSample 类中的方法
    System.out.println("This is a super - class.");
  }
```

```
}
public class SubSample extends SuperSample{     //定义一个 SubSample 子类
  public void Display(){                         //覆盖 SuperSample 类中的方法
    System.out.println("Override Method in subSample - class.");
  }
  public void Display(String newS){              //定义子类自身的方法,重载 Display()方法
    System.out.println(newS);
  }
}
public class Sample {                            //定义应用父、子类的 Sample 类
  public static void main(String[ ] args){
    SuperSample f;                               //声明父类变量 f
    SubSample s = new SubSample();               //创建 SubSample 类对象 s
    f = s;                                       //将子类对象 s 赋给父类变量 f
    f.Display();                                 //调用子类中的 Display()方法,多态性
//f.Display("Overload Method in subSample - class.");   //不正确调用
//不能调用子类中带输入参数的重载的 Display()方法,因为 f 对象的属性和行为被限定在父类范围
  }
}
```

3.4.2　多态的应用

【示例 3-27】 在面向对象程序设计中,多态主要是通过方法的重载和覆盖体现的,其过程是通过向不同的对象发送相同的信息,根据对象的不同完成不同的工作。例如,定义一个 People()类,为相互沟通,所有人共有行为是说话,在 People 类中则可定义 Speak()方法,但是,各个区域的人是说不同语言的,中国人(Chinese)说中文、美国人(American)说英文、日本人(Japanese)说日文等,假设在 People 类中重载 Speak()方法表示不同区域人的说话行为的程序代码为:

```
public class People {                            //定义 "人"类
  public void Speak(){                           //定义人的说话行为
    System.out.println("The People say language");
  }
  public void Speak(String s){                   //定义人的区域说话行为
    if(s == "Chinese")
      System.out.println("The Chinese say Chinese");
    if(s == "American")
      System.out.println("The American say English");
    if(s == "Japanese")
      System.out.println("The Japanese say Japanese");
  }
}
```

上述程序通过 Speak()方法的输入参数指定不同区域人的说话行为,该程序设计虽然体现了多态特性,但是它并不符合面向对象程序设计理念,对象没有细分,层次不清晰,

People 类应该是各个区域人类的父类，因此，上述程序可以修正为：

```
public abstract class People {                          //定义抽象的"人"类
  public abstract void Speak();                         //定义人的抽象说话行为
}
class Chinese extends People {                          //定义属于中国人的类
  public void Speak(){                                  //定义中国人的说话行为
    System.out.println("The Chinese say Chinese");      //覆盖父类方法
  }
}
class American extends People {                         //定义属于美国人的类
  public void Speak(){                                  //定义美国人的说话行为
    System.out.println("The American say English");     //覆盖父类方法
  }
}
class Japanese extends People {                         //定义属于日本人的类
  public void Speak(){                                  //定义日本人的说话行为
    System.out.println("The Japanese say Japanese");    //覆盖父类方法
  }
}
public class AnyPeople extends People{                  //定义任何人的类
  People p;
  public AnyPeople(People p) {                          //重新指定具体的人
    this.p = p;
  }
  public void Speak(){                                  //定义人的说话行为
    p.Speak();
  }
}
public class Sample {                                   //定义应用类
  public static void main(String[] args){
    People people;                                      //声明"人"类变量
    Chinese chinese = new Chinese();                    //创建中国人对象
    American american = new American();                 //创建美国人对象
    Japanese japanese = new Japanese();                 //创建日本人对象
    people = chinese;                                   //当该人为中国人时
    people.Speak();                                     //语言为中文
    people = american;                                  //当该人为美国人时
    people.Speak();                                     //语言为英文
    people = japanese;                                  //当该人为日本人时
    people.Speak();                                     //语言为日文
    AnyPeople ap = new AnyPeople(chinese);              //上溯造型重新确定具体的人
    ap.Speak();
  }
}
```

运行 Sample 程序时其输出结果为：

```
The Chinese say Chinese
The American say English
The Japanese say Japanese
The Chinese say Chinese
```

上述程序在运行时根据 people 是哪个区域的“人”决定 people. Speak()的实际执行代码，该程序设计是符合面向对象程序设计理念的，因为不论是哪个区域的“人”都是属于“人”类的，而每个区域的“人”又有自己的说话方式，因此当一个“人”属于不同区域时有不同的说话方式，在应用中其行为与对象是动态绑定的，即实现上溯造型，它也是体现了面向对象程序设计中行为方式的多态特性。

3.4.3　构造方法与多态

一个类可以定义多个构造方法，即构造方法的重载，当使用同一个类的不同构造方法创建多个类对象时，其对象也会有所不同，呈现出多种对象的形态，该现象也是面向对象程序设计的多态特性的一种体现。例如，用一个类描述坐标系对象，坐标系有绝对和相对坐标系之分，定义绝对坐标系的原点为(0,0)时，相对坐标系的原点为(x0,y0)，当 x0 和 y0 的值为 0 时，绝对坐标系和相对坐标系重合。

【示例 3-28】 描述绝对坐标系和相对坐标系的 Java 演示程序。

```
class Coordinate {                                    //定义直角坐标系类
  private int x0;                                     //声明 x0 坐标原点变量
  private int y0;                                     //声明 y0 坐标原点变量
  public Coordinate() {                               //设置绝对坐标系原点处坐标值
    x0 = 0;
    y0 = 0;
  }
  public Coordinate(int newX0,int newY0) {            //设置相对坐标系原点处坐标值
    x0 = newX0;
    y0 = newY0;
  }
}
public class Sample {
  public static void main(String[] args) {
    Coordinate absoluteCoordinate =
      new Coordinate();                               //创建绝对坐标系对象
    Coordinate oppositeCoordinate =
      new Coordinate(10,10);                          //创建相对坐标系对象
  }
}
```

创建的两个不同坐标系对象的关系如图 3-1 所示，相对坐标系在绝对坐标系中，相对坐标系的原点坐标在绝对坐标系中的(10,10)处。

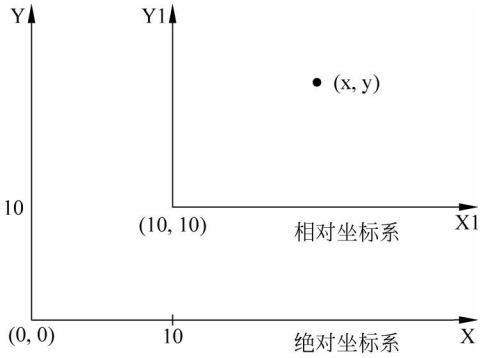

图 3-1　绝对坐标系和相对坐标系

3.5　小结

(1) Java 类的继承是完全继承，子类全部继承父类的所有内容。

(2) 只要有继承，在子类和父类中的变量和方法之间就存在隐藏、覆盖、重载现象。

(3) 隐藏是隐藏父类中的成员变量和静态方法；覆盖是替换父类中相同的操作方法；重载可发生在子类与父类之间，也可发生在同一个类中，重载用不同的操作方法实现相同(一种)类型的操作功能。

(4) Java 的重载体现了静态的多态特性，动态绑定或“上溯造型”技术则体现了动态的多态特性。

(5) abstract(抽象的)修饰的类不可直接使用，需要派生，修饰的方法为空方法；final 修饰的类不可被继承，final 修饰的量为常量。

(6) this 变量指向当前类，super 变量指向父类。

3.6　习题

(1) Java 类的继承有什么特点？

(2) 在一个 Java 类中可同时定义许多同名的方法，这些方法的形式参数的个数、类型、顺序各不相同，返回值也可以不相同，这种面向对象程序设计的特性被称为：

A. 隐藏　　B. 覆盖　　C. 重载　　D. Java 不支持此特性

(3) 下面是关于类及其修饰符的一些描述，哪些是正确的？

A. abstract 类只能当父类使用，不能用来创建 abstract 类的对象。

B. final 类不但可以当父类使用，也可以用来创建 final 类的对象。

C. abstract 不能与 final 同时修饰一个类。

D. abstract()方法在 abstract 类中声明，但 abstract 类中可以没有 abstract()方法。

E. 通过 super 变量在子类中可以引用 abstract 父类中声明的 abstract()方法。

F. this 变量不可以引用 final 类的父类中声明的变量和操作方法。

(4) 编写一个在学校的人员 SchoolPeople 类，SchoolPeople 类的属性和行为如下。

属性：id　　编号　　int 类型
　　name　　姓名　　String 类型
　　age　　年龄　　byte 类型
　　sex　　性别　　boolean 类型，true 表示男，false 表示女
　　phone　　电话　　String 类型

行为：独立返回 5 个属性值的 5 个方法
　　统一设置 5 个属性值的方法

通过继承 SchoolPeople 类编写一个学生 Student 类，Student 类特有的属性和行为如下。

属性：student_Id　　学号　　String 类型

行为：独立返回学号属性值的方法
　　设置学号和 SchoolPeople 类定义的 5 个属性值的方法

通过继承 SchoolPeople 类编写一个教师 Teacher 类，Teacher 类特有的属性和行为如下。

属性：id_number　　身份证号　String 类型

行为：独立返回身份证号属性值的方法
　　设置身份证号和 SchoolPeople 类定义的 5 个属性值的方法

编写 Java 应用程序，创建一个教师对象 s_people1 和一个学生对象 s_people2，设置它们的属性，并输出显示教师和学生的所有信息。

(5) 在 J2SDK 环境中编译、调试、运行下述几个程序，查看和分析程序运行后输出的显示结果。

```
/* ============ 程序 1 ============ */
class A {
}
public class B extends A {
  public static void main(String[] args){
      A a = null; B b = null;
      if(a instanceof A)
        System.out.println("a belong to A");
      else
        System.out.println("a NOT belong to A");
      if(b instanceof B)
        System.out.println("b belong to B");
      else
        System.out.println("b NOT belong to B");
      a = new B();
      if(a instanceof A)
```

```
            System.out.println("a belong to A");
        else
            System.out.println("a NOT belong to A");
        if(a instanceof B)
            System.out.println("a belong to B");
        else
            System.out.println("b NOT belong to B");
        b = new B();
        if(b instanceof A)
            System.out.println("b belong to A");
        else
            System.out.println("b NOT belong to A");
        if(b instanceof B)
            System.out.println("b belong to B");
        else
            System.out.println("b NOT belong to B");
    }
}
/* ============ 程序 2 ============ */
class SuperClass{
    int a,b;
    SuperClass(int newA,int newB){
        a = newA; b = newB;
    }
    void show(){
        System.out.println("a = " + a + "\nb = " + b);
    }
}
class SubClass extends SuperClass{
    int c;
    SubClass(int newA,int newB,int newC){
        super(newA,newB);
        c = newC;
    }
}
class SubSubClass extends SubClass{
    int a;
    SubSubClass(int newA,int newB,int newC){
        super(newA,newB,newC);
        a = newA + newB + newC;
    }
    void show(){
        System.out.println("a = " + a + "\nb = " + b + "\nc = " + c);
    }
}
public class Sample {
    public static void main(String[] args){
```

```
    SubSubClass x = new SubSubClass(10,20,30);
    x.show();
  }
}
/* ========== 程序 3 ========== */
class Base{
  String var = "base var";
  static String staticVar = "static var";
  void method(){
      System.out.println("base method");
  }
  static void staticMethod(){
      System.out.println("base static method");
  }
}
class Sub extends Base{
  String var = "Sub var";
  static String staticVar = "static Sub Var";
  void method(){
      System.out.println("sub method");
  }
  static void staticMethod(){
      System.out.println("sub static method");
  }
}
public class Sample {
  public static void main(String[] args){
    Base base = new Sub();
    System.out.println(base.var);
    System.out.println(base.staticVar);
    base.method();
    base.staticMethod();
  }
}
/* ========== 程序 4 ========== */
class SuperClass{
  float x; int n;
  SuperClass(float x,int n){
    this.x = x; this.n = n;
  }
}
class SubClass extends SuperClass{
  SubClass(float x,int n){
    super(x,n);
  }
  float exp(){
    float s = 1;
```

```
    for(int i = 1; i <= n; i++)
      s = s * x;
    return s;
  }
}
public class Sample {
  public static void main(String[] args){
    SubClass a = new SubClass(8,4);
    System.out.println(a.exp());
  }
}
```