第5章 继 承

继承是面向对象程序设计(Object Oriented Programming, OOP)的三大特征之一,描述了类不同抽象级别之间的关系:“is a”的关系,即“特殊与一般”的关系。换句话说,一般(父类)是特殊(子类)更高级别的抽象。子类可以继承父类所有的非 private 类型的属性和方法,也可以具有自己独有的属性和方法。通过类的继承关系,使公共的特性能够共享,提高了软件的重用性。但在 Java 中只允许单继承。

5.1 继承的语法

在 Java 中描述两个类之间的继承关系时,使用关键字 extends,格式如下:

```
class SubClass extends SuperClass{
    …
}
```

其中,SubClass 为子类,SuperClass 为父类(或超类)。

在第 4 章中定义了 Person 类:

```
class Person {
    private String name;
    private char sex='M';
    Person(String name){
        this.name=name;
    }
    Person(String name,char sex) {
        this.name=name;
        this.sex=sex;
    }
    public void show() {
        String str="下面展示姓名和性别";
        System.out.println(str);
        System.out.println("姓名: "+name+" 性别: "+sex);
    }
}
```

现在要定义一个学生类(Stu),由于“学生是人”,所以学生类和 Person 类之间是“is a”的关系,即“继承”关系,那么就可以按例 5.1 的方法定义 Stu 类。

例 5.1 Stu 类的定义(ch05\Stu.java)。

```
class Stu extends Person {
    long id;
```

```
    private String name;          //仅为演示用,实际编程中无须声明该变量
    private char sex='M';
    public Stu (String name, long id, char sex){
        super(name,sex);
        this.id=id;
    }
}
```

前面讲过子类可以继承父类的非 private 类型的属性和方法,在这个例子中可以看到:虽然在 Person 类中定义了 name、sex 属性,但它们是 private 类型的数据,如果 Stu 也想拥有这些属性,就必须重新定义,不能继承于父类,也可以定义这些属性为 protected;但 show 方法在 Person 类中是以 public 的身份定义的,所以 Stu 类虽然没有显式地定义该方法,却拥有该方法,因为它继承了父类的 show()方法,编写测试类如下:

```
class UseStu {
    public static void main(String[] args) {
        Stu s=new Stu("王强",20094140213L,'M');
        s.show();
    }
}
```

运行结果:

```
下面展示姓名和性别
姓名: 王强 性别: M
```

另外,还可以在子类中定义子类独有的属性和方法,例如本例中的 id。

在这里需要说明以下几点。

(1) 父类的构造函数不能被子类继承。

(2) 子类不能继承或访问父类中的 private 属性和方法。

(3) 父类中的 friendly(包访问权限)的属性和方法只有在父类和子类在同一包中时才能被子类继承和访问。

(4) 父类中由 protected 或 public 修饰的属性和方法,都可以被子类继承访问(无论子类是否与父类在同一包中)。

5.2 成员变量的隐藏和方法的覆盖

当子类和父类中定义的成员变量的名字相同时,子类可以隐藏父类的成员变量。同样,子类也可以通过方法重写(或称为覆盖,Overriding)来隐藏从父类继承的方法。方法覆盖是指子类中定义的方法的头部(方法的名字、返回类型、参数个数和类型)与父类的方法完全相同,而方法体可以不同。但在进行方法覆盖时要注意:在覆盖时访问权限只能放大或相同,不能缩小;覆盖方法不能抛出新的异常(关于异常,将在第 8 章介绍)。

例 5.2 成员变量的隐藏和方法的覆盖(ch05\OverriddingTest.java)。

```
class Father {
```

```
    String s="Father";
    int i=1;
    public void f() {
        System.out.println("Father s="+s);
        System.out.println("Father i="+i);
    }
}
class Child extends Father {
    String s="Child";                    //隐藏了父类的成员变量 s
    public void f(){                     //覆盖了父类的 f()方法,但访问权限只能是 public
        System.out.println("Child s="+s);
        System.out.println("Child i="+i);
    }
}
class OverriddingTest {
    public static void main(String[] args) {
        Father f=new Father();
        Child c=new Child();
        f.f();
        c.f();
    }
}
```

运行结果：

```
Father s=Father
Father i=1
Child s=Child
Child i=1
```

方法覆盖与方法重载的区别如下：方法覆盖发生在父类和子类之间，即子类重写了父类的某个方法，子类中定义的方法的头部(方法的名字、返回类型、参数个数和类型)与父类的方法完全相同，而方法体可以不同；重载是在同一类中出现的现象，是指一个类中可以有多个方法具有相同的名字，但这些方法的参数不同。

5.3 super

如果想在子类中使用父类的非 private 类型的变量和方法(特别是被隐藏的变量和方法)，可以使用 super 关键字。例如，要在例 5.2 中访问父类的变量 s，就要使用 super.s，试验在子类 Child 的 f()方法中加入如下的代码，查看输出结果。

```
System.out.println("Father s="+super.s);
super.f();
```

如果要在子类的构造方法中访问父类的构造方法，也要使用 super 关键字，例如例 5.1 中的 super(name,sex)；但要注意，该调用语句必须出现在子类构造方法非注释语句的第

一行。

注意：如果在子类的构造方法中没有使用 super 调用父类的构造方法，编译器将自动添加

```
super();
```

即调用父类不带参数的构造方法，此时就应保证父类中有不带参数的构造方法（当父类未定义任何构造方法时，系统会自动合成；一旦父类定义了一个或多个构造方法，系统将不再提供默认的构造方法，必须手工定义），否则就会产生错误。如例 5.1 中，由于父类 Person 未定义不带参数的构造方法，所以必须用 super(name,sex)显式地调用父类中某个已定义的构造方法。

5.4 final 和 sealed

1. final 关键字

final 关键字可以用来修饰类、方法、变量（包括成员变量和局部变量及方法中的参数）。

（1）当 final 修饰类时，意味着该类不能被继承，即该类不能有 String 类等子类。

例 5.3 final 修饰类（ch05\FinClass .java）。

```
final class FinClass {                                    //最终类
    int i;
    FinClass() {
        System.out.println("This is a final class.");
    }
}
class SubFinClass extends FinClass {                      //错误,不能从最终类继承
}
```

编译时会出现下面的出错信息：

```
FinClass.java:7: 无法从最终类继承
class SubFinClass extends FinClass {
```

（2）当 final 修饰方法时，代表该方法不能被重写。

（3）当 final 修饰成员变量时，该变量可以理解为常量，必须赋以初值（可在声明时赋值，或在类的构造方法中赋值），并且该变量的值不能再改变；当 final 修饰局部变量时，该局部变量只能被赋一次值；当 final 修饰方法中的参数时，该参数的值不能被改变。

例 5.4 final 修饰方法和变量（ch05\UseFinal.java）。

```
class UseFinal {
    final int i=1;
    final int j;          //最终变量若不在声明时赋值,就要在其所属的类的构造方法中赋值
    int k;
    UseFinal() {
        j=2;
```

```
    }
    final void f(){     //最终方法,在子类中不能被覆盖
        System.out.println("This is a final method.");
    }
    void g() {
        //i++;错误,不能重新指定最终变量的值
        //j++;错误,不能重新指定最终变量的值
        k++;
        final String s="Hello ";
        //s="Hi";错误,当 final 修饰局部变量时,该变量只能被赋一次值
        final String str;
        str="Java";
        System.out.println(s+str+"  i="+i+"  j="+j+"  k="+k);
    }
    void h(final int a) {
        //a++;错误,不能指定最终参数
        System.out.println("a="+a);
    }
    public static void main(String[] args){
        UseFinal uf=new UseFinal();
        uf.f();
        uf.g();
        uf.h(100);
    }
}
```

2. sealed 类

通过 final 关键字,使得一些类不能被继承,这在一定程度上对继承关系进行了限制,优化了代码重用。但在有些时候,可能只要求某些被指定的子类,才能从一个类进行继承,即部分地限制类的继承,这就要用到 sealed 关键字。被 sealed 修饰的类称为密封类,它只允许被指定的类继承。

可以在 class 和 interface 之前使用 sealed 修饰符。由 sealed 定义的密封类必须定义子类。子类可以用 permits 显式指出;也可以不指定,而与父类定义在同一个 Java 源文件中。sealed 类的子类必须由 sealed、non-sealed、final 中的一个关键字修饰,即 sealed 类的子类可以是密封类、非密封类或最终类。

如下为定义在同一个文件中的例子,通过这种方式可定义哪些类可以被继承:

```
public sealed class SealedClassDemo {
    final class ASon extends SealedClassDemo{}
}
final class BSon extends SealedClassDemo{}
```

如下用 permits 显式定义可以继承的子类,这时可以不用写在同一个文件中:

```
sealed class SealedClassDemo2 permits CSon,DSon{}
non-sealed class CSon extends SealedClassDemo2{}
```

```
final class DSon extends SealedClassDemo2{}
```

5.5 多　　态

多态是 OOP 的三大特征之一，此处结合 5.4 节讲述的覆盖来理解多态的含义。当一个类（如 Instrument 类）有多个子类（Wind、Percussion、Stringed），并且这些类都重写了父类中的某个方法（void play()方法）时，如图 5.1 所示，根据前面所讲的内容，下面的代码很容易理解：

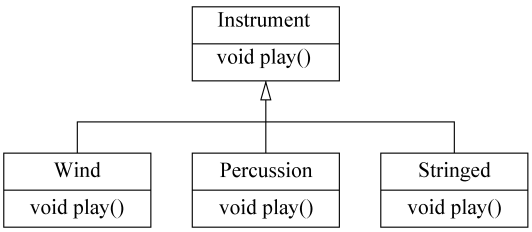

图 5.1　Instrument 及其子类

```
Wind w=new Wind();                          //产生 Wind 类的对象
w.play();                                   //调用 Wind 类中的 play 方法
Percussion p=new Percussion();              //产生 Percussion 类的对象
p.play();                                   //调用 Percussion 类中的 play 方法
Stringed s=new Stringed();                  //产生 Stringed 类的对象
s.play();                                   //调用 Stringed 类中的 play 方法
```

那么下面的代码又如何理解呢？

```
Instrument insw=new Wind();
insw.play();
```

把 Wind 类的对象赋值给 Instrument 类型的变量（insw）对吗？insw 调用的是 Instrument 类中的 play 方法还是 Wind 类中的 play 方法？

子类和父类之间的关系是“is a”的关系，即“特殊与一般”的关系，可以说管乐器（Wind）是乐器（Instrumen），打击乐器（Percussion）是乐器（Instrument），弦乐器（Stringed）是乐器（Instrument），所以下面的代码是正确的：

```
Instrument insw=new Wind();
Instrument insp=new Percussion();
Instrument inss=new Stringed();
```

这就是常说的向上转型（upcasting）。向上转型后的对象（简称上转型对象），如 insw、insp、inss。例如 play()，在调用方法时，其实调用的仍是子类中所重写的 play 方法，而不是父类的 play 方法。这是因为 Java 对 Override 方法调用采用的是运行时绑定，也就是按照对象的实际类型来决定调用的方法，不是按照对象的声明类型来决定调用的方法。但 Overload 方法则相反，在编译时已经进行了方法绑定，按照对象的声明类型决定调用的方法。

例 5.5　多态示例（ch05\Music.java）。

```
class Instrument {
    public void play() {
        System.out.println("Instrument.play()");
    }
}
class Wind extends Instrument {
    public void play() {
        System.out.println("Wind.play()");
    }
}
class Percussion extends Instrument {
    public void play() {
        System.out.println("Percussion.play()");
    }
}
class Stringed extends Instrument {
    public void play() {
        System.out.println("Stringed.play()");
    }
}
public class Music {
    static void tune(Instrument i) {
        i.play();
    }
    public static void main(String[] args) {
        Instrument[] ins=new Instrument[3];
        int i=0;
        ins[i++]=new Wind();
        ins[i++]=new Percussion();
        ins[i++]=new Stringed();
        for (i=0; i<ins.length; i++)
            tune(ins[i]);
    }
}
```

运行结果：

```
Wind.play()
Percussion.play()
Stringed.play()
```

那么能否将父类的对象赋值给子类类型的变量呢？答案是否定的，除非进行强制类型转换。例如：

```
Wind w=new Instrument();                    //错误
Wind w=(Wind) new Instrument();             //正确
```

1. 类型判断操作符

要进行强制类型转换，往往需要先知道变量的具体类型。instanceof 是 Java 中用来判断变量类型的操作符，经常用于判断对象类型。代码如下：

```
class Animal{}
class Mammal extends Animal{}
class Fish extends Animal{}
public class Zoo {
    public static void main(String args[]) {
        Animal a=new Mammal();
        if (a instanceof Animal)
            System.out.println("Animal");
        if (a instanceof Mammal)
            System.out.println("Mammal");
        if (a instanceof Fish)
            System.out.println("Fish");
    }
}
```

编译、运行上述代码，会根据对象 a 的类型，进行输出：

```
Animal
Mammal
```

由结果可以看出，对象 a 既属于 Animal 类型，也属于 Mammal 类型。

在 JDK 16 中，增强了 instanceof 操作符的功能，在使用 instanceof 进行类型判断的同时，生成对应类型的模式匹配变量，将类型判断、类型转换和变量声明写到一条语句，精简了代码。如下代码将 num 类型判断、类型转换、变量 a 的声明和赋值简化为 if 括号中的一条语句，在 if 语句块中，可以正常使用 Integer 变量 a。

```
Object num=15;
if (num instanceof Integer a){
    System.out.println("a=" +(++a));
}
```

运行上述代码片段会输出：

```
a=16
```

结合例 5.5 中的 Instrument 和 Wind 类，可以使用 instanceof 操作符进行类型判断和模式变量声明，代码如下：

```
Instrument inst=new Wind();
inst.play();
if (inst instanceof Wind wind){
    wind.play();
}
```

上述代码中，instanceof 语句相当于进行了强制类型转换，即将父类类型又转换成子类类型。运行上述代码片段会输出：

```
Wind.play()
Wind.play()
```

Java 是面向对象的语言，一般而言，子类会对父类进行扩展。将子类对象赋值给父类对象变量，其原有的对象成员依然会存在；反之，会存在一定问题。

2. 模式变量的可见范围

instanceof 模式变量的有效范围仅为 instanceof 判断为 true 的范围，instanceof 判断为 false 的范围内模式变量不可见。例如以下代码：

```
1    public static void test2() {
2        Object num=15;
3        if (!(num instanceof String s)) {
4            return;
5        }
6        System.out.println(s);
7    }
8    public static void test3() {
9        Object num=15;
10       if (!(num instanceof String s)) {
11           System.out.println("not String");
12       }
13       System.out.println(s);
14   }
```

编译时 test2 方法中无错误，test3 方法中提示第 13 行 s 不可解析。

test2 方法，在 if 语句块中，instanceof 判断为 false，不能使用模式变量 s。在第 6 行，因为在第 4 行 instanceof 判断为 false 时已经是 return，所以第 6 行的代码只有在 instanceof 判断为 true 时才能执行，因此可以访问模式变量 s。

test3 方法和 test2 方法非常类似，区别仅在第 4 行和第 11 行，第 4 行是在 instanceof 判断为 false 时返回，第 11 行为显示信息。接下来，不管 instanceof 判断结果为 true 还是 false，都会执行第 13 行的代码，而模式变量仅在 instanceof 判断为 true 时可见，所以第 13 行是不能访问模式变量 s 的，编译错误。

利用可见范围可以写出简洁的代码，如下为判断 num 是否大于 2 的 Integer：

```
if (num instanceof Integer a&&a>2) {
    System.out.println("a=" +a);
}
```

注意，仅从语法上讲，以上代码中的“&&”不能换成“||”。对于 num instanceof Integer a||a>2，由于“||”表示或，所以当 instanceof 判断为 false 时，模式变量 a 在后一个条件不可见。

如果仅需要判断结果，可以写为如下形式：

```
return num instanceof Integer a&&a>2;
```

5.6 继承与组合

通过使用继承，提高了类的可重用性，减少了代码的重复书写，提高了效率。除了继承这种方式外，还可以通过组合的方式来重复使用类。所谓组合，就是在一个新类中创建已有类的对象，即新类由已有类的对象组成。例如，下面的学生成绩管理程序，Score 类中使用了已有类（Student 类和 Course 类）中的对象，所以这种重复使用类的方式就是组合。

例 5.6 组合示例，学生成绩管理程序（ch05\StuApp.java）。

```
package app;
//学生类
class Student {
    String name;
    long id;
    public Student() {
    }
    public Student(String name,long id){
        this.name=name;
        this.id=id;
    }
}
//课程类
class Course {
    long id;
    String name;
    Course(long id, String name){
        this.id=id;
        this.name=name;
    }
}
//成绩类
class Score {
    Student stu;
    Course course;
    double grade;
    Score(Student stu,Course course,double grade){
        this.stu=stu;
        this.course=course;
        this.grade=grade;
    }
}
//应用类
class StuApp {
    private static Course[] courses=new Course[5];
```