第5单元　类的继承

面向对象程序设计的核心是定义类。前面介绍的类是直接定义的，除此之外，还可以在已经定义类的基础上定义新的类，由新的类继承已经定义类的部分代码实现部分代码的重用。这一单元通过学生和研究生之间存在的继承(inheritance)，也称泛化(generalization)关系，讨论一般继承(泛化)关系的 Java 描述以及所引出的有关问题。

5.1　学生类-研究生类层次结构

5.1.1　由 Student 类派生 GradStudent 类

1. 从两个独立的类说起

学生类和研究生类可以分别定义为 Student 类和 GradStudent 类。

【代码 5-1】 Student 类和 GradStudent 类的独立定义。

```
class Student {
    private String studentName;                                          // 学生姓名
    private int studentID;                                               // 学号

    public Student(String studentName, int studentID){                   // 构造器
        this.studentName = studentName;
        this.studentID = studentID;
    }
    public void print(){                                                 // 输出方法
        System.out.println("学生姓名: " + studentName + ",学号: " + studentID);
    }
}

class GradStudent {
    private String studentName;                                          // 研究生姓名
    private int studentID;                                               // 学号
    private String tutorName;                                            // 导师姓名

    public GradStudent(String studentName, int studentID, String tutorName){    // 构造器
        this.studentName = studentName;
        this.studentID = studentID;
        this.tutorName = tutorName e;
    }
    public void print(){                                                 // 输出方法
```

```
        System.out.println("研究生姓名: " + studentName + ",学号: "
                           + studentID + "导师姓名: " + tutorName);
    }
}
```

2. 由学生类派生研究生类形成的类层次结构

研究生也是学生,即研究生是学生的一部分,学生是研究生的抽象。这种关系在面向对象的程序设计中用继承(inheritance)(也称泛化(generalization),有时也称派生(derived))表示。对于本例,可以说是 Student 类派生出 GradStudent 类,也可以说 GradStudent 类继承了 Student 类。

【代码 5-2】 GradStudent 类继承 Student 类的 Java 代码。

```
class Student {
    protected String studentName;                              // 使用了 protected
    protected int studentID;                                   // 使用了 protected

    public Student(String studentName, int studentID){         // 构造器
        this.studentName = studentName;
        this.studentID = studentID;
    }

    public void print(){                                       // 输出方法
        System.out.println("学生姓名: " + studentName + ",学号: " + studentID);
    }
}

class GradStudent extends Student {
    private String tutorName;                                  // 导师姓名

    public GradStudent(String studentName, int studentID, String tutorName){     // 构造器
        supert(studentName, studentID);
        this.tutorName = tutorName;
    }
    public void print(){                                       // 输出方法
        System.out.println("研究生姓名: " + studentName
                           + ",学号: " + studentID + "导师姓名: " + tutorName);
    }

public class ExtendsDemo0502{
    public static void main(String[] args){
        Student s = new Student("王舞", 123456);
        s.print();
        GradStudent g = new GradStudent("李司", 654321, "张伞");
        g.print();
    }
}
```

程序运行结果如下：

```
学生姓名：王舞,学号：123456
研究生姓名：李司,学号：654321,导师姓名：张伞
```

说明：

(1) 关键词 extends 表示扩展或派生，即以一个类为基础派生出一个新类。这个新生成的类称为派生类(derived class)或直接子类(direct subclass)；原始的类作为派生类形成的基础存在，称为基类(base class)，也称为派生类的超类(super class)或父类(parent class)。在本例中，class GradStudent extends Student 表明 GradStudent 类是以 Student 为基类扩展而成的派生类。派生类也可以继续扩展成新的派生类。

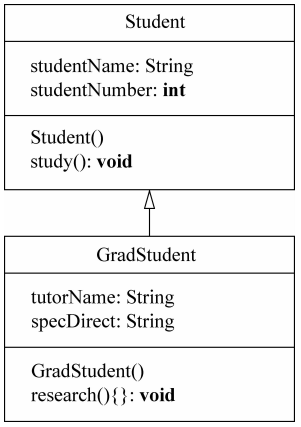

图 5.1　类的继承关系

从另一方面看，extends 关键词使派生类继承(inherit)了基类的属性和方法，因此派生类无法脱离基类而存在。图 5.1 是本例中派生关系的 UML 描述。

(2) 继承有两个方面的意义，一是派生类继承了基类的一些成员，如本例中的 name 和 studID。更重要的是，继承表明了两个类之间的父子关系，这是面向对象程序设计中很重要的一点。通过后面的介绍读者将会认识到有这种关系和没有这种关系在程序操作中的方便程度是不同的。

(3) 注意在类 Student 中成员 studentName 和 studentAge 改用 protected 修饰，而不是用 private 修饰。因为 private 将所修饰的成员的访问权限限制在本类中，而 protected 允许将所修饰成员的访问权限扩展到派生类中。这样，这两个成员才能被 GradStudent 类中的构造器和 print()方法访问。

3. Java 继承规则

Java 语言的继承有如下特征。

(1) 每个子类只能有一个直接父类，但一个父类可以有多个子类。

(2) 派生具有传递性。如果类 A 派生了类 B，类 B 又派生了类 C，则 C 不仅继承了 B，也继承了 A。

(3) 不可循环派生。若 A 派生了类 B，类 B 又派生了类 C，则类 C 不可派生 A。

5.1.2　super 关键字

super 是 Java 的一个关键字，它有两种用法。

1. 用 super 调用基类(对象)的可见成员

【代码 5-3】 使用 super 调用的 GradStudent 类 print()方法的 Java 代码。

```
public void print(){                                        // 输出方法
    System.out.println("研究生姓名: " + super.studentName
                       + ",学号: " + super.studentID + "导师姓名: " + tutorName);
}
```

2. 用 super()代表父类构造器

与 this()一样,它必须放在调用函数中的第 1 行,即当调用派生类的构造器实例化时首先要调用基类的构造器对从基类继承的成员进行实例化,再对本类新增成员进行实例化。

【代码 5-4】 使用 super()的 GradStudent 类构造器。

```
public GradStudent(String studentName, int studentID, String tutorName){
    super(studentName, studentID);                          // 调用父类构造器
    this.tutorName = tutorName;
}
```

注意:与 this()不同,使用 super()必须为所调用成员的访问权限允许,否则无法调用。

【代码 5-5】 在 Student-GradStudent 类层次中的 this()和 super()应用。

```
public class Student{
    private String studentName;
    private int studentNumber;
    public Student(){                                       // 默认构造器
        System.out.println("将 Student 成员变量初始化为类型默认值!");
    }
    public Student(String studentName, int studentNumber){  // 有参构造器
        this()                                              // 调用另一个构造器
        this.studentName = studentName ;
        this.studentNumber = studentNumber;
    }
    public void study(){}
}
public class GradStudent extends Student{
    private String tutorName;
    private String specDirect;
    public GradStudent(String name,int studNum){            // 构造器
        super(name,studNum);                                // 调用父类构造器
    }
    public GradStudent(String name,int studNum, String tutName, String spcDir){
        this(name,studNum);                                 // 调用另一构造器
        tutorName = tutName;
        specDirect = spcDir;
    }
    public void research(){}
}
```


说明:

(1) 代码 5-5 用于说明构造器 GradStudent 的执行过程。①表示开始执行这个构造器;

②～⑦表明逐步向上层(基类)调用的过程;⑧～⑮表示由上层逐步构建的过程。虚线表示参数传递的情况。

(2) 注意,当派生类中有多个构造方法,并且它们之间要使用 this( )互相调用时,至少要留有一个调用父类构造方法,否则会导致编译错误。

5.1.3 final 关键字

关键字 final 具有“终极”“不可改变”的含义。在声明中,final 不仅可以用来修饰属性(变量),还可以用来修饰方法和类。

1. final 变量

用 final 修饰一个具有初始值的变量就会使该变量一直保持这个值不再改变,成为一个符号常量。

注意:

(1) final 变量在使用前必须进行初始化。通常在声明的同时初始化或在构造器以及初始化段中进行初始化。例如,代码 4-5 中的 final 变量 DEKE_SIZE 是在声明的同时初始化的,因为假定玩的是一副扑克,其数量一定是 54。而 final 变量 shuffleTime 是在构造器中初始化的,因为每次玩扑克游戏之前玩家可以商定洗几次牌。

(2) final 变量只能初始化一次。

2. final 方法

用 final 修饰方法,则该方法为最终方法,即其在子类中不可被重定义。

3. final 类

用 final 修饰类,则该类为最终类,不可再派生子类。例如 in、out 和 err 不仅是 System 类的静态成员,而且是 System 类的 3 个常量,它们的定义和说明如下。

(1) public static final java.io.InputStream in: 标准输入流对象,此对象可以通过 read()方法接收从键盘输入的内容。

(2) public static final java.io.printStream out: 标准输出流对象,此对象可以通过 println()方法或 print()方法输出内容到显示器。

(3) public static final java.io.printStream err: 标准错误输出流对象,用于显示错误消息,或者显示那些应该立刻引起用户注意的其他信息。

5.2 Java 的访问权限控制

访问权限指一个程序元素(类、属性和方法)被某个类和该类成员访问的权利。

5.2.1 类成员的访问权限控制

按照信息隐蔽的原则,Java 将类成员的访问权限分为表 5.1 所示的 4 个等级。

表 5.1　Java 类成员的 4 种访问权限(√:可以,×:不可)

访问权限	关键字	作用元素	作　用　域			
级别			同一类	同一包	不同包的子类	所有类(全局)
私密	private	类成员	√	×	×	×
默认	无	类,成员	√	√	×	×
保护	protected	类,成员	√	√	√	×
公开	public	类、接口、类成员	√	√	√	√

(1) 私密级,用 private 修饰,表明该成员仅可被本类的其他成员访问。

(2) 默认级,不用任何访问权限修饰,表明该成员仅被同包的其他类成员访问。

(3) 保护级,用 protected 修饰,表明该成员被同包的类以及派生类访问。

(4) 公开级,用 public 修饰,表明该成员无任何访问限制。

5.2.2　类的访问权限控制

类只有 public 和默认两种权限,权限的内容包括访问、使用和继承。

一个类被修饰为 public,表示该类为公共类,可以被任何类访问、使用和继承。

一个类没有权限修饰,表示该类为包中类,只能被同一包中的其他类访问、使用和继承。

注意:如果一个类声明为 public,则文件名必须与该类的名称一致,即一个文件中只能有一个被声明为 public 的类。

5.2.3　private 构造器

构造器的访问级别也可以是 public、protected、默认和 private,不过当构造器被 private 修饰时会发生如下一些特殊情况。

(1) 构造器为 private,意味着它只能在当前类中被访问,具体如下。

- 在当前类的其他构造器中可以用 this 调用它;
- 在当前类的其他方法中用 new 调用它。

(2) 当一个类的构造器都为 private 时,这个类将无法被继承,因为子类构造器无法调用该类的构造器。

(3) 当一个类的构造器都为 private 时,将不允许程序的其他类通过 new 创建这个类的实例,只能向程序的其他部分提供获得自身实例的静态方法,并且这种类的实例只能有一个,所以广泛应用于只为一个类创建一个实例的情况,这种应用称为单例模式。

【代码 5-6】 单例模式示例 1。

```
class Singleton {
    private static Singleton instance = new Singleton();  // 将 instance 定义为静态的,即类中唯一的
    private Singleton(){}                                  // 私密构造器
    static Singleton getInstance() {                       // 向程序的其他部分提供这个实例
        return instance;
    }
}
```

这种方法是不管三七二十一,一上来就创建,像饥饿多日的人看见食品一样,所以称饥

汉方式。

【代码 5-7】 单例模式示例 2。

```
class Singleton {
    private static Singleton instance = null;                    // 仅建立一个空的引用
    private Singleton(){}                                        // 私密构造器
    static Singleton getInstance() {                             // 提供外界时创建
        if (instance == null)
                instance = new Singleton();
        return instance;
    }
}
```

这种单例模式不像饥汉方式那样，不管需要不需要、有没有都要创建一个单例，而是在外界需要并调用方法 getInstance()，并且当实例的引用还不存在时才会创建这个实例。就像一个懒汉一样，能不干就不干，所以称之为懒汉方式。

5.3 类层次中的类型转换

5.3.1 类层次中的赋值兼容规则

一个类层次结构有许多特性，其中一个重要的特性称为赋值兼容性（assignment compability），指在需要基类对象的任何地方都可以使用公有派生类对象来替代。具体地说是可以将派生类对象赋值给基类对象，或者说可以用派生类对象初始化基类的引用，而无须进行强制类型转换。

【代码 5-8】 对代码 5-2 中的类进行赋值兼容性验证的主方法。

```
public class ExtendsDemo0508 {
    public static void main(String[] args){
        Student s = new Student("王舞", 123456);
        s.print();
        GradStudent g = new GradStudent("李司", 654321, "张伞");
        g.print();
        s = g;                                          // 将派生类对象赋值给基类引用
        s.print();                                      // 指向派生类的 Student 引用调用
    }
}
```

测试结果如下：

```
学生姓名：王舞,学号：123456
研究生姓名：李司,学号：654321,导师姓名：张伞
研究生姓名：李司,学号：654321,导师姓名：张伞
```

讨论：

（1）从测试结果可以看出，在使用基类对象的地方用派生类对象替代后系统仍然可以编译运行，语法关系符合赋值兼容规则。

(2) 赋值兼容规则是单向的,即不可以将基类对象赋值给派生类对象。

5.3.2 里氏代换原则

里氏代换原则(Liskov substitution principle,LSP)是由 2008 年的图灵奖得主、美国第一位计算机科学女博士 Barbara Liskov 教授和卡内基·梅隆大学的教授 Jeannette Wing 于 1994 年提出的。其原始表达是:如果对类型 T1 的任何一个对象 ob1 都可以有一个类型 T2 的对象 ob2,使得在 T1 定义的程序 P 中,当所有的对象 ob1 都代换为 ob2 时,程序 P 的行为没有变化,那么类型 T2 是类型 T1 的子类型。里氏代换原则可以通俗地表述为在程序中能够使用父类对象的地方必须能透明地使用其子类的对象。

一般来说,继承的优越性在于子类通过继承重用了父类的代码,这称为继承重用。但是,这个重用是建立在派生类可以替换掉基类对象的基础上的,所以说里氏代换原则是继承重用的一个基础。因为只有当软件单位的功能不会受到影响时基类才能真正被重用,而派生类也才能够在基类的基础上增加新的行为。反过来代换是不成立的。

里氏代换原则是赋值兼容规则的另一种描述,这个原则已经被编译器采纳。在程序编译期间,编译器会检查其是否符合里氏代换原则。这是一种无关实现的、纯语法意义上的检查。里氏代换原则要求子类方法的访问权限不能小于父类对应方法的访问权限。例如,当“狗”是“动物”的派生类时,在程序段

```
动物 d = new 狗();
d.吃();
```

中,若“动物”类中的成员方法“吃()”的访问权限为 public,而“狗”类中的成员方法“吃()”的访问权限为 protected 或 private,此时是不能编译的。

关于里氏代换原则的意义,读者通过后面几个单元的学习将会进一步理解。

5.3.3 类型转换与类型测试

1. 对象的向上造型与向下造型

对象类型在子类与父类之间的转换(cast)也称造型或转型。造型(或转型)按照转换的方向分为向上造型(upcasting,也称向上转换)和向下造型(downcasting,也称向下转换)。

向上造型就是把子类对象作为父类对象使用,这总是安全的,其转换是可行的。因为子类对象总可以当作父类的实例。从类的组成角度看,向上造型无非是去掉子类中比父类多定义的一些成员而已。

至于向下造型,则往往是不自然的、不安全的。例如在代码 5-5 中,若使用语句

```
GradStudent g = new Student();
```

就会出现如下类型错误。

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
        Type miematch: cannot convert from Student1 to GradStudent

        at Student.main(Student.java:33)
```

因为要把一个普通大学生当作研究生会缺少研究生应当具备的一些信息，例如导师姓名、研究方向等。

2. 强制类型转换与类型测试

当需要进行向下造型时必须进行强制类型转换，例如：

```
Student stu = new Student();
GradStudent grad = (GradStudent)stu;                    // 向下造型,强制转换
```

其中用圆括号括起的类名就是一种强制造型操作。

为了保证程序的安全，在进行向下造型时应当先用 instanceof 测试父类能不能作为子类的实例。例如：

```
if (stud instanceof GradStudent)
    grad = (GradStudent)stud;
```

instanceof 是 Java 的一个与==、>、<操作性质相同的二元操作符。由于它由字母组成，所以也是 Java 的保留关键字。它的作用是测试其左边的表达式是否与右边的类名赋值兼容，如果是，则返回 boolean 类型。

5.4 方法覆盖与隐藏

继承机制使子类可以继承父类的所有属性和方法。但是，派生类对于基类的成员除了继承以外还有另外两种处理，即覆盖(override)与隐藏(hidden)。这一节介绍方法的覆盖与隐藏。对于属性只有隐藏，但不推荐使用，因为隐藏属性会使代码难以阅读。

5.4.1 派生类实例方法覆盖基类中签名相同的实例方法

1. 方法覆盖的基本概念

方法签名(也称特征标，signature)是指方法的名字、参数个数和每个参数的类型。方法签名和返回类型相同就是函数头中的所有内容都要相同。在一个类层次结构中，当派生类定义了一个与基类具有相同原型的方法时将会覆盖基类那个方法，即派生类对象无法直接调用到基类那个方法覆盖的方法。如在前面的代码中，类 Student 和 GradStudent 中都定义了一个 print()方法，它们的签名和返回类型都相同，因此类 GradStudent 的对象引用无论如何调用不到 Student 类中的 print()方法。代码 5-2 的执行结果可以得出这个结论。

方法覆盖可以带来一个动态多态性的好处，即一个指向基类的引用，若用基类对象初始化，就可以用其调用基类中的方法；若用派生类对象初始化，就可以用其调用派生类中的那种覆盖方法，这样就大大提高了程序设计的灵活性。这种多态性是在程序运行中由 JVM 实现的，所以称为动态多态性。

2. 方法覆盖的条件

派生类实例方法与成员变量不同，覆盖基类同名方法必须满足下面一些约束：

(1) 方法覆盖只能存在于派生类和基类(包括直接基类和间接基类)之间，不能在同一类中。在同一类中同名方法所形成的关系是重载。

(2) 覆盖方法的返回类型和签名必须与被覆盖方法保持一致。

(3) 不能覆盖已经用 final 或 static 修饰的方法，但被覆盖方法的参数可以是 final 的。

(4) 覆盖方法的 throws 子句列出的类型可以少于被覆盖方法的 throws 子句列出的类型，或更加具体，或二者皆有之。

(5) 覆盖方法的访问权限不能比被覆盖方法的访问权限小，只能比被覆盖方法的访问权限大。例如，被覆盖方法为 public，则覆盖方法必须是 public 的，否则无法编译。

(6) 被覆盖的方法不能为 private，否则在其子类中只是新定义了一个方法，并没有对其进行覆盖。

(7) 在派生类中不可用空方法覆盖其类中的方法。

3. 方法覆盖与方法重载的区别

方法覆盖与方法重载都给程序提供了一个名字多种实现的灵活性，但它们也有许多不同。表 5.2 列出了方法覆盖与重载的区别。

表 5.2　方法覆盖与重载的区别

	位置关系	方法名	参数列表	返回类型	访问权限	抛　出	数　量	绑定实施及时间
重载	同一类中(包括从父类继承的)	必须相同	必须不同	无要求	无要求	无限制	可以多个	编译器编译时
覆盖	派生类与基类	必须相同	必须相同	必须相同	派生类方法不可更严格。不能覆盖 private 方法	要求一致	只能有一次	JVM 运行中

5.4.2　用@Override 标注覆盖

1. 标注的概念

标注(annotation)是 Java 5 提供的新特性，这里将其译成“标注”，与之相近的术语是注释(comment)。在程序中二者的基本区别在于：注释是供阅读者理解程序而加入的，仅在源代码中存在；标注虽然也可以起到供阅读者理解的作用，但更主要的是向有关软件(编译器、解释器、JVM)提供一些说明或向程序传递一些参数，且不同的标注有不同的使用位置和保存范围。

从作用上看，标注相当于对程序元素(包、类型、构造器、方法、成员变量、参数、本地变量等)的额外修饰并应用于声明中，但是与一般关键字声明修饰符有如下不同：

(1) 标注都以符号@开头，例如@Override、@Deprecated 和@SuppressWarning 等。

(2) 一般声明修饰符不可带参数，而标注可以带参数，例如@SuppressWarning。这些参数可以向编译器提供附加信息，也可以用来向程序传递数据。