CHAPTER 第5章
面向对象(二)

本章学习重点：

- 理解继承的概念。
- 掌握 super 关键字。
- 掌握 final 关键字。
- 掌握方法的重写与重载的区别。
- 熟练掌握抽象类和接口。
- 理解多态的概念。

5.1 类的继承

继承是面向对象的另一大特征，它用于描述类的所属关系，多个类通过继承形成一个关系体系。继承是在原有类的基础上扩展新的功能，实现了代码的复用。

5.1.1 继承的概念

继承是面向对象的核心特征之一，是从已有类创建新类的一种机制。利用继承机制。可以先创建一个具有共性的一般类，从一般类再派生出具有特殊性的新类，新类继承一般类的属性和方法，并根据需要增加它自己的新属性和新方法。类的继承机制是面向对象程序设计中实现软件重写的重要手段。

继承机制使得软件开发人员可以充分利用已有的类来创建自己的类，例如，利用 Java 类库中丰富而且有用的类，可根据自己的需要进行扩展。Java 通过继承机制很好地实现了类的可重用性和扩展性。

类的继承也称类的派生。通常，将被继承的类称为父类或超类，派生出来的类称为子类，从一个父类可以派生出多个子类，子类还可以派生出新的子类，这样就形成了类的层次关系。在 Java 中一个类只能继承一个父类，称为单继承。但一个父类却可以派生出多个子类，每个子类作为父类又可以派生出多个子类，从而形成具有树状结构的类的层次体系。

1. 子类继承父类

Java 中，子类继承父类的语法格式如下：

```
[修饰符] class 子类名 extends 父类名{
    属性定义;
```

```
    方法定义;
}
```

Java 使用 extends 关键字指明两个类之间的继承关系。子类继承了父类中的属性和方法,也可以添加新的属性和方法。

【例 5-1】 子类继承父类实例。

```
class Parent {
    String name;
    String sex;
    public void saySex() {
        System.out.println(name+"的性别:"+sex);
    }
}
class Child extends Parent{
        int age;
        public void sayAge() {
        System.out.println(name+"的年龄:"+age);
    }
}
public class TestExtends {
    public static void main(String[] args) {
        Child c = new Child();
        c.name = "王鹏";
        c.sex = "男";
        c.age = 20;
        c.saySex();
        c.sayAge();
    }
}
```

从程序运行结果可以看出,Child 类通过 extends 关键字继承了 Parent 类,Child 类便是 Parent 的子类。Child 类虽然没有定义 name、sex 成员变量和 saySex()方法,但能访问这些成员,说明子类可以继承父类的成员。另外,在 Child 类中还定义了一个 sayAge()方法,说明子类可以扩展父类功能。

Java 语言只支持单继承,不允许多重继承,即一个子类只能继承一个父类,否则会引起编译错误,具体代码如下:

```
class A{}
class B{}
class C extends A,B{}          //编译错误
```

Java 语言虽然不支持多重继承,但它支持多层继承,即一个类的父类可以继承另外的父类。因此,Java 类可以有无限多个间接父类,具体代码如下:

```
class A{}
class B{} extends A{}
class C{} extends B{}
```

2. 父类成员的访问权限

子类可以继承父类中的属性成员和除构造方法以外的方法成员，但不能继承父类的构造方法。而且，并不是对父类的所有属性成员和方法成员都具有访问权限，即并不是在子类声明的方法中能够访问父类中的所有属性成员和方法成员。Java 中子类访问父类成员的权限如下。

（1）子类对父类的 private 成员没有访问权限。既不能直接引用父类中的 private 属性成员，也不能调用父类中的 private 方法成员，如果需要访问父类的成员，可以通过父类中的非 private 成员方法来引用父类的成员。

（2）子类对父类的 public 或 protected 成员具有访问权限。

（3）子类对父类的默认权限成员的访问分为两种情况：一是对同一包中父类的默认权限成员具有访问权限；二是对其他包中父类的默认权限成员没有访问权限。

类层次结构中，这种访问权限的设定体现了类封装的信息隐藏原则，即如果类中成员仅限于该类使用，则声明为 private；如果类中的成员允许子类使用，则声明为 protected；如果类中成员允许所有类使用，则声明为 public。

5.1.2 成员变量的隐藏

类的继承使得子类从父类中既继承了有用的属性成员，也会继承一些不需要或不恰当的属性成员。当父类中的属性不适合子类需要时，子类可以把从父类继承来的属性成员进行重新定义。由于在子类中也定义了与父类中名字相同的成员变量，因此父类中的成员变量在子类中就不可见了，这就是成员变量的隐藏。这时在子类中若想引用父类中的同名变量，可以用关键词 super 作为前缀来引用父类中的同名变量，即

```
super.属性名
```

例如，由类 B 继承类 A：

```
public class A{
    int a;
    void pa(){}
}
public class B extends A{
    int a;
    void pb(){
        a=8;
    }
}
```

父类 A 和子类 B 中都包含变量 a，根据成员变量的隐藏原则，在类 B 中的 pb 方法中引

用的 a 是类 B 的变量,而不是父类 A 的变量。若要引用类 A 的变量,则需要使用 super.a 访问。

5.1.3　方法的重写

在继承关系中,子类从父类中继承了可访问的方法,但有时从父类继承下来的方法不能完全满足子类需要,如在例 5-1 中,如果要求父类与子类中的 saySex()方法输出不同内容,就需要在子类方法中修改父类方法,即子类重新定义从父类中继承的成员方法,这个过程称为方法重写或覆盖。在方法重写时必须考虑权限,即被子类重写的方法不能拥有比父类方法更加严格的访问权限。

子类方法覆盖父类的方法时,方法头要与父类一样,即两个方法要具有完全相同的方法名、返回类型、参数表。方法体要根据子类的需要重写,从而满足子类功能的要求。与类中使用父类被隐藏的属性成员类似,如果子类中需要调用被覆盖的父类中的同名方法,通过 super 关键字作前缀来调用,即

```
super.方法名()
```

【例 5-2】 方法覆盖实例。

```
class Parent {
    protected void say() {
        System.out.println("父辈");
    }
}
class Child extends Parent {
    public void say() {
        System.out.println("子女");
    }
}
public class OverrideDemo{
    public static void main(String[] args) {
        Child c = new Child();
        c.say();
    }
}
```

在例 5-2 中,Child 类继承了 Parent 类的 say()方法,但在子类 Child 中重新定义的 say()方法,对父类的 say()进行了重写。从程序运行结果可以发现,在调用 Child 类对象的 say()方法时,只会调用子类重写的方法,并不会调用父类的 say()方法。

另外,需要注意方法重载与方法重写的如下区别。

(1) 方法重载是在同一个类中,方法重写是在子类与父类中。

(2) 方法重载要求方法名相同,参数个数或参数类型不同。

(3) 方法重写要求子类与父类的方法名、返回值类型和参数列表相同。

5.1.4 super 关键字

在类中可以直接通过变量名来使用类中的变量，直接通过方法名调用类中的方法。类的成员也可以通过 this 作为前缀来引用，关键字 this 代表对象本身。

与 this 相似，Java 用关键字 super 表示父类对象，因此在子类中使用 super 作为前缀，可以引用被子类隐藏的父类的变量和被子类覆盖的父类的方法。

其语法格式如下：

```
super.成员变量
super.成员方法
```

当子类中没有声明与父类同名的成员变量时，引用父类的成员变量可以不使用 super。当子类中没有声明与父类中同名的成员方法时，调用父类的成员方法也可以不使用 super。

【例 5-3】 super 实例。

```
class Parent {
    String name="Parent";
    public void say() {
        System.out.println("父辈");
    }
}
class Child extends Parent {
    String name="child";
    public void say() {
        super.say();
        System.out.println("姓名:" +super.name);
        System.out.println("姓名:" +name);
    }
}
public class SuperDemo{
    public static void main(String[] args) {
        Child c = new Child();
        c.say();
    }
}
```

在例 5-3 中，Child 类继承 Parent 类，并重写 say()方法。在子类 Child 的 say()方法中，使用 super.name 调用了父类的成员变量，使用 super.say()调用了父类被重写的成员方法。从程序运行结果可以看出，通过 super 关键字可以在子类中访问被隐藏的父类成员。

在继承中，实例化子类对象时，首先调用父类的构造方法，再调用子类的构造方法，这与实际生活中先有父母再有孩子类似。子类继承父类时，并没有继承父类的构造方法，但子类构造方法可以调用父类的构造方法。在一个构造方法中调用另一个重载的构造方法使用 this 关键字，在子类构造方法中调用父类的构造方法时应使用 super 关键字，其语法格式

如下：

```
super([参数列表])
```

【例 5-4】 super 调用构造方法。

```
class Parent {
    String name;
    public Parent(String name) {
        this.name = name;
        System.out.println("parent");
    }
}
class Child extends Parent {
    public Child() {
        super("Parent");
    }
}
public class SuperRefConstructorDemo{
    public static void main(String[] args) {
        Child c = new Child();
    }
}
```

从程序运行结果中可以发现，实例化 Child 类对象时调用了父类的有参构造方法。super 关键字调用构造方法和 this 关键字调用构造方法类似，语句必须位于子类构造方法的第一行，否则会编译出错。而且，子类构造方法中的 this()和 super()只能调用其中的一个，不能同时使用。

另外，子类中如果没有显式地调用父类的构造方法，将自动调用父类中不带参数的构造方法。

【例 5-5】 自动调用父类无参数构造方法 1。

```
class Parent {
    String name;
    public Parent(String name) {
        this.name = name;
        System.out.println("Parent");
    }
}
class Child extends Parent {
    public Child() {
        System.out.println("Child");
    }
}
```

```
public class AutoCallConstructorDemo{
    public static void main(String[] args) {
        //创建 Child 对象
        Child c = new Child();
    }
}
```

从程序运行结果可以看出，程序编译结果报错。原因是在 Child 类的构造方法中没有显式地调用父类构造方法，便会默认调用父类无参数构造方法，而父类 Parent 中显式定义了有参构造方法，此时编译器将不会自动生成默认构造方法。因此，程序找不到无参数构造方法而报错。

为了解决上述程序的错误，可以在子类显式地调用父类中定义的构造方法，也可以在父类中显式定义无参数构造方法。

【例 5-6】 自动调用父类无参数构造方法 2。

```
class Parent {
    String name;
    public Parent() {
        System.out.println("Parent");
    }
    public Parent(String name) {
        this.name = name;
        System.out.println("Parent");
    }
}
class Child extends Parent {
    public Child() {
        System.out.println("Child");
    }
}
public class ConstructorOrderDemo{
    public static void main(String[] args) {
        Child c = new Child();
    }
}
```

从程序运行结果可以看出，子类在实例化时默认调用父类的无参数构造方法，并且父类的构造方法在子类构造方法之前执行。

5.1.5 final 关键字

在 Java 中，为了考虑安全因素，要求某些类不允许被继承或不允许被子类修改，这时可以用 final 关键字修饰。它可用于修饰类、方法和变量，表示“最终”的意思，即用它修饰的类、方法和变量不可改变，具体特点如下。

(1) final 修饰的类不能被继承。

(2) final 修饰的方法不能被子类重写。

(3) final 修饰的变量是常量,初始化后不能再修改。

1. final 关键字修饰类

使用 final 关键字修饰的类,称为最终类,表示不能再被其他的类继承,具体示例如下。

【例 5-7】 final 类实例。

```
final class Parent {
}
class Child extends Parent {
}
```

从程序运行结果可以看出,使用 final 关键字修饰了 Parent 类。因此,Child 类继承 Parent 类时,程序编译结果报错并提示 The type Child cannot subclass the final class Parent。由此可见,被 final 修饰的类为最终类,不能再被继承。

2. final 关键字修饰方法

使用 final 关键字修饰的方法,称为最终方法,表示子类不能重写此方法,具体示例如下。

【例 5-8】 final 修饰方法实例。

```
class Parent {
    public final void say() {
        System.out.println("final 修饰 say()方法");
    }
}
class Child extends Parent {
    public void say() {
        System.out.println("重写父类 say()方法");
    }
}
public class FinalFunctionDemo {
    public static void main(String[] args) {
        Child c = new Child();
        c.say();
    }
}
```

从程序运行结果可以看出,Parent 类中使用 final 关键字修饰了成员方法 say(),Child 类继承 Parent 类并重写了 say()方法。程序编译结果报错并提示 Cannot override the final method from Parent。由此可见,被 final 修饰的成员方法为最终方法,不能再被子类重写。

3. final 关键字修饰变量

使用 final 关键字修饰的变量,称为常量,只能被赋值一次。如果再次对该变量进行赋值,则程序在编译时会报错,具体示例如下。

【例 5-9】 final 修饰常量实例。

```
public class FinalLocalVarDemo {
    public static void main(String[] args) {
        final double PI = 3.14;
        PI = 3.141592653;
    }
}
```

从程序运行结果可以看出，使用 final 修饰变量 PI，再次对其进行赋值，程序编译结果报错并提示“无法为最终变量 PI 分配值”。由此可见，final 修饰的变量为常量，只能初始化一次，初始化后不能再修改。

使用 final 修饰的是局部变量，接下来使用 final 修饰成员变量，具体示例如下。

【例 5-10】 final 修饰成员变量实例。

```
class Parent  {
    //使用 final 修饰成员变量
    final double PI;
    public void say() {
        System.out.println(this.PI);
    }
}
class Child extends Parent {
}
public class TestFinalMemberVar {
    public static void main(String[] args) {
        //创建 Child 对象
        Child c = new Child();
        c.say();
    }
}
```

从程序运行结果可以看出，在 Parent 类中使用 final 修饰了成员变量 PI，程序编译结果报错并提示 The blank final field PI may not have been initialized。由此可见，Java 虚拟机不会为 final 修饰的变量默认初始化。因此，使用 final 修饰成员变量时，需要在声明时立即初始化，或者在构造方法中进行初始化。

下面使用构造方法初始化 final 修饰的成员变量，在 Parent 类中添加代码，具体代码如下：

```
public Parent(){
    PI=3.14;
}
```

此外，final 关键字还可以修饰引用变量，表示该变量只能始终引用一个对象，但可以改

变对象的内容。

5.2　抽象类和接口

对于面向对象编程来说,抽象是它的主要特征之一。在 Java 中,可以通过两种形式来体现面向对象的抽象:抽象类和接口。这两者有许多相似的地方,又有许多不同的地方。初学者往往会以为它们可以随意互换使用,但实际不然。下面分别讲解抽象类和接口。

5.2.1　抽象类

在面向对象概念中,所有对象都是通过类来描述的,但是反过来却不是这样,并不是所有的类都能描绘对象。如果一个类中没有包含足够的信息来描述一个具体的对象,这样的类可以定义为抽象类。

抽象类往往用来表征在对问题领域进行分析、设计中得到的抽象概念,是对一系列看上去不同,但本质上相同的具体概念的抽象。例如,如果进行一个图形软件的开发,就会发现问题领域存在着圆形、菱形等一些具体的概念,它们是不同的,但是都属于形状这一概念。形状是一个抽象的概念,所以用来表征抽象概念的类是不能实例化的。

Java 中提供了方法声明和方法实现分离的机制,可以定义不含方法体的方法,方法的方法体由该类的子类根据实际需求去实现,这样的方法称为抽象方法(abstract method),包含抽象方法的类称为抽象类(abstract class)。

Java 用 abstract 关键字表示抽象的意思,用 abstract 修饰的方法,称为抽象方法,是一个不完整的方法,只有方法的声明,没有方法体。用 abstract 修饰的类,称为抽象类,语法如下:

```
[权限修饰符] abstract class 类名{
    [权限修饰符] abstract 返回值类型 方法名(参数列表);
}
```

说明:

(1) 抽象方法声明只需给出方法头,不需要方法体,直接以";"结束。

(2) 构造方法不能声明为抽象方法。

(3) 在抽象类中,可以包含抽象方法,也可以不包含抽象方法,但类中如果有抽象方法,此类必须声明为抽象类。

(4) 抽象类不能被实例化,即使抽象类中没有抽象方法,也不能被实例化。

(5) 子类必须实现抽象父类中所有的抽象方法,否则子类必须要声明为抽象类。

【例 5-11】 子类实现抽象方法实例。

```
abstract class Animal {
    public abstract void eat();
}
```