责任编辑　刘湘琦
封面设计　文翰誉诚
上架编码　Java-00041

面向对象与Java基础实战

主编　黄风华　张燕琴

湖南大学出版社

ISBN 978-7-5667-4207-0
9 787566 742070 >
定价：59.80元

新时代高等教育创新型精品教材
“互联网+”教育改革立体化教材

# 面向对象与Java基础实战

主编　黄风华　张燕琴

湖南大学出版社

图书在版编目（CIP）数据

面向对象与 Java 基础实战 / 黄风华，张燕琴主编 .
长沙 : 湖南大学出版社，2025. 6. -- ISBN 978-7-5667-4207-0

Ⅰ. TP312.8

中国国家版本馆 CIP 数据核字第 2025LY9243 号

**面向对象与 Java 基础实战**

MIANXIANG DUIXIANG YU JAVA JICHU SHIZHAN

主　　编：黄风华　张燕琴
责任编辑：刘湘琦
印　　装：崇阳文昌印务股份有限公司
开　　本：787 mm×1092 mm　1/16　　印张：20　　字数：438 千字
版　　次：2025 年 6 月第 1 版　　印次：2025 年 6 月第 1 次印刷
书　　号：ISBN 978-7-5667-4207-0
定　　价：59.80 元

出 版 人：李文邦
出版发行：湖南大学出版社
社　　址：湖南·长沙·岳麓山　　邮　　编：410082
电　　话：0731-88822559（营销部），88821327（编辑室），88821006（出版部）
传　　真：0731-88822264（总编室）
网　　址：http：//press. hnu. edu. cn
电子邮箱：395405867@ qq. com

# 目 录

# 第5章 继承与多态

## 5.1 情景导入

继承性是面向对象的三大特性之一，即在采用原有类所有非私有属性和方法的基础上，创建包含新属性或新方法的新类过程，例如可以在原有房屋建造图纸的基础上做部分改进和添加（如增加房顶旗帜、方形侧门和圆形窗房等），进而使新房屋具有更好的外形和更多的功能，如图 5-1 所示。这个已有的类可以是系统类库提供的类，也可以是自定义类。此外，虽然 Java 类只能有一个直接父类，但可通过使用接口的方式继承或实现其他类中的抽象方法，从而扩充自身的功能以实现多态性。

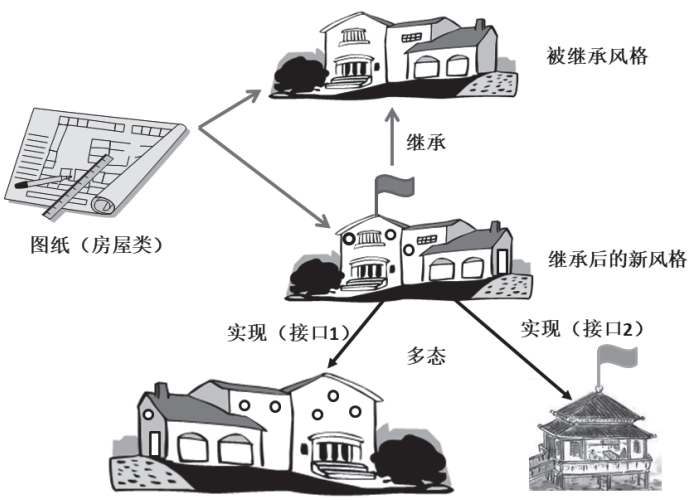


图 5-1　情景导入图

## 5.2 知识点概述

### 5.2.1 类的继承

继承就是根据已有的类创建新类的过程。应用继承的方式，可以创建一个父类，在父类中定义共有的特性。该类设计好后可以被具体的类继承，称为子类。每个子类中都可以增加部分自己特有的属性和方法，例如已经设计了一个球类，而现在又要设计一个具体的足球类，由于足球类具有球类的特征，因此足球类就可以定义为球类的子类，这样足球类就不需要再重复定义球类的属性，而直接声明足球类特有的属性即可。

1. 继承的规则

在 Java 中，类的继承通过 extends 关键字实现。被继承的类可以是自定义类，也可以是 Java 标准库提供的类。自定义类的继承格式如下：

```
class A {
    // 属性和方法
}

class B extends A {
    // B 特有的属性和方法
}
```

例 5-1　实现 B 类继承 A 类。

```
class Test {
    public static void main(String[] args) {
    A a = new A();
    B b = new B();
    a.i = 10;
    b.j = 20;
    System.out.println(" 显示 B 类的属性：");
    b.displayB();
    b.i = 7;
    b.j = 8;
    System.out.println(" 显示 B 类的属性：");
    b.displayA();
    b.displayB();
    b.sum();
    }
}
class A {
    int i;
```

```
    void displayA() {
        System.out.println("i 的值为：" + i);
    }
}

class B extends A {
    int j;

    void displayB() {
        System.out.println("j: " + j);
    }

    void sum() {
        System.out.println("i + j: " + (i + j));
    }
}
```

运行结果如图 5–2 所示。

```
显示B类的属性：
j: 20
i的值为:  7
j: 8
i + j: 15
```

图 5–2　程序运行结果

在例 5–1 中，子类 B 继承了父类 A 的所有成员。i 是 A 类的属性，而 B 类则定义了属性 j。当 B 类继承了 A 类后，B 类对象自然拥有父类 A 的属性和方法，然而父类 A 并不包含 B 类特有的属性和方法。

### 2. 继承结构概述

子类的定义结构如下：

```
class 子类名 extends 父类名 {
    // ...
}
```

子类包含两部分内容，即从父类继承的成员和子类自身定义的成员。子类拥有父类的属性，而父类不具备子类的特性。在创建子类对象时，系统首先会创建父类的部分，然后再分配子类特有的空间。

例 5–2　子类与父类的关系。

```
public class Test {
    public static void main(String[] args) {
        B b1 = new B();  // b1 对象包含 b1 属性和方法
```

```
            A a1 = new A(); // a1 对象包含 a1 属性和方法
        }
    }
    class A {
        int x;

        void fn() {
            System.out.println(this.x);
        }
    }

    class B extends A {
        int m;

        void show() {
            int y = 10;
            System.out.println(" 子类属性：" + x + " " + m + " " + y);
        }
    }
```

在例 5–2 中，子类 B 具有父类 A 的所有属性和方法。对于 B 类中的方法 show()：

y 是局部变量，不能使用 this.y 或 super.y。

m 是 B 类的属性，可以直接用 m 或 this.m 来引用，不能用 super.m。

x 是从父类 A 继承的属性，可以用 x、this.x 或 super.x 访问。

这样的继承结构使得代码更加模块化和具有可重用性，减少了重复代码。通过继承，子类不仅能利用父类已有的功能，还能根据自身特性扩展新的功能。

### 5.2.2 包与访问权限

#### 1. 包的创建与引用

包是 Java 采用的树结构文件系统的组织方式，可把包含类代码的文件组织起来，使其便于查找和使用。包不仅能包含类和接口，还能包含其他包，形成多层次的包空间。包有助于避免命名冲突，形成层次命名空间，从而缩小了名称冲突的范围，易于管理名称。

（1）包的创建。

在 Java 程序中，package 语句必须是程序的第一个非注释、非空白行、行首无空格的语句，它用来说明类和接口所属的包。创建包的一般语法格式为：

package com.user;

通过声明包信息，表示该 Java 源文件的路径是“\com\user\”，即项目文件夹下的“com”文件夹中的“user”文件夹内。

Java 中使用 package 关键字来打包，每一个源文件只能声明一个包，并且

package 语句必须作为源文件中的第一条语句。在这个文件中定义的所有类都属于这个包，具体格式为：

package 包名 ;

习惯上包名用小写字母定义。包也可以嵌套使用，同文件夹的嵌套一样，只要用“.”把嵌套的包名分开。注意用“;”号结束。

在 E 盘新建一个项目 MyPackage，在 MyPackage 项目中建立文件夹 test，再新建一个 java 类 MyTest。在 MyTest.java 中们将看到以下代码：

```
package test;
public class MyTest{
    public static void main(String args[]){
        System.out.println("Hello World!");
    }
}
```

Java 中常用的包有以下几个：

java.lang　java 语言包，该包自动被导入

java.awt　java 窗口界面设计包

java.awt.event　java 事件处理包

java.swing　java 组件包

java.io　java 输入输出包

java.util　java 常用容器包

java.sql　java 数据库访问包

java.net　java 网络操作包

（2）包的引用。

创建包的目的是调用时能方便地找到所需要的信息。包的导入目的是将所需要的各个类导入回来，让编译器能够找到所需要的类。导入的语法如下。

导入包中所有的类：

import 包名 .*;

导入包中子包中所有的类：

import 包名 . 子包名 .*;

导入包中的某个类：

import 包名 . 类名 ;

如果需要导入该包下的所有类可以使用 import 包名 .*; 但是 import 包名 .* 并不能导入子包里的类。要调用的类和被调用的类如果在同一包下，则不需要导入，直接使用就可以。

### 2. 访问权限修饰符

子类继承父类所有的成员，但是在父类中声明为 private 的成员在子类中是不能

被访问的。

例 5-3　找出下列程序中的错误。

```
class Test {
        public static void main(String args[]) {
                A a = new A();
                B b = new B();
                b.showB();
        }
}
class A {
                private int i1=1;
                protected int i2=2;
                public int i3=3;
                int i4=4;
}
class B extends A {
        void showB() {
                System.out.println("i1: " + i1);
                System.out.println("i2: " + i2);
                System.out.println("i3: " + i3);
                System.out.println("i4: " + i4);
        }
}
```

该程序中 B 类 showB() 方法使用 i1 语句将出错。原因是 i1 是 A 类中的 private 修饰，子类不能访问父类中私有的成员。

访问权限修饰符如下。

类成员的访问权限修饰符：public、default、protected、private。

类的访问权限修饰符：public、default。

访问权限见表 5-1。

表 5-1　访问权限

| | private | default | protected | public |
|---|---|---|---|---|
| 同一类 | 可访问 | 可访问 | 可访问 | 可访问 |
| 同一包中的类 | | 可访问 | 可访问 | 可访问 |
| 不同包中的子类 | | | 可访问 | 可访问 |
| 其他包中的类 | | | | 可访问 |

### 5.2.3 super 关键字

super 关键字与第 4 章介绍的 this 关键字作用相似，主要是用来引用父类中被屏

蔽的属性和方法。this 关键字指本类的属性或者方法。super 关键字是用在子类中，作用是在子类中访问父类中的属性和方法。

super 关键字有两种常用形式：一种是在子类中访问父类的成员，另一种是调用父类的构造方法。

### 1. super 访问父类成员

**例 5–4**　super 访问父类成员。

```
class Test {
    public static void main(String args[]) {
        B b=new B(10,20);
        b.show();
    }
}
class A {
    int x;
}
class B extends A {
    int x;
    B(int a, int b) {
        super.x = a;
        x = b;
    }
    void show() {
        System.out.println(" 父类中的 x " + super.x);
        System.out.println(" 子类中的 x " + x);
    }
}
```

该程序运行结果如图 5–3 所示。

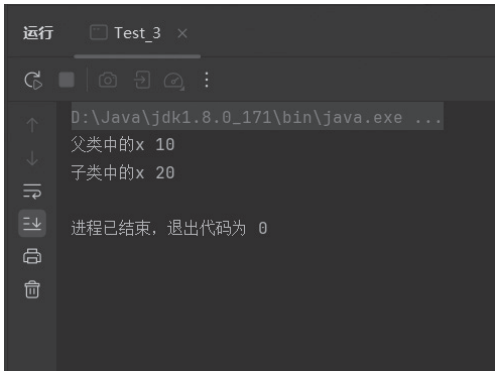


**图 5–3　程序运行结果**

由于子类 B 继承了 A 类，A 类有 x 属性，B 类有同名的 x 属性，这种情况称为属性的隐藏。那么在子类 B 中使用 x 时，如果不加特殊说明则表示是 B 类的属性 x，

此时如果需要使用 A 类的 x，需要使用 super 关键字来说明，如 super.x。

### 2. super 关键字调用父类的构造函数

在子类中初始化父类的数据，一般使用父类的构造方法进行初始化，并有以下两个类：

```
class A {
    private int x;
    private int y;
}
class B extends A{
    int z;
}
```

在 A 类中定义了属性 x，y 为 private，当 B 类继承 A 类。B 类对象对 A 类私有成员变量 x，y 没有访问权限，在 B 类中无法对 A 类的 x，y 初始化。如果需要初始化 A 类中 x，y，必须在 A 类中完成初始化操作。在 A 类中加入构造方法完成初始化。

```
class A {
    private int x;
    private int y;
    A(){x=0;y=0;}
    A(int x,int y){
        this.x=x;
        this.y=y;
    }
}
```

此时在 B 类构造方法中可以通过 super() 调用 A 类的构造方法来对 A 类 x，y 赋值。

```
class B extends A{
    int z;
    B(){
        super();
        z=0;
    }
    B(int x,int y,int z){
        super(x,y) ;
        this.z=z ;
    }
}
```

在主方法中创建 B 类对象。

```
B b1=new B();
B b2=new B(1,2,3);
```

b1 中 x=0，y=0，z=0

b2 中 x=1，y=2，z=3

在每一个子类的构造方法中，都会调用父类的构造方法。调用父类的构造方法分为显式的调用和隐式的调用。显式的调用是指在子类构造方法的第一条语句处使用 super() 调用相应的父类构造方法。需要注意的是 super() 的参数必须与父类中某一个参数列表匹配，找不到匹配的父类构造方法，将报错处理。

例 5–5　找出该程序的错误。

```
class A {
        private int x;
        private int y;
        A(){x=0;y=0;}
        A(int x,int y){
                this.x=x;
                this.y=y;
        }
}
class B extends A{
        int z;
        B(){
                z=0;
        }
        B(int x,int y,int z){
                super(x,y) ;
                this.z=z ;
        }
        B(int x,int z){
                super(x) ;
                this.z=z ;
        }
}
```

B 类第三个构造方法 B(int x,int z) 中第一条语句 super(x) ; 将报错。原因是父类中没有与之匹配的构造方式。注意 super() 方法必须放置在子类构造方法的第一条语句处。

在例 5–5 中，B(){ z=0;} 构造方法中没有显示调用构造方法，则系统为该构造方法提供一个无参的父类构造方法，即 super()。这就是隐式的调用，与下面构造方法等价：

```
B(){
                super() ;
                z=0;
        }
```

无论是显式还是隐式调用，一个子类构造方法中只能有一个 super()。

总结：

（1）在子类构造方法中要调用父类的构造方法，用“super( 参数列表 )”的方式调用，而且必须出现在子类构造中的第一行。

（2）当父类中成员变量被子类隐藏时或者在子类中访问父类的成员变量，可以使用“super. 父类成员变量名”。

（3）当父类中成员方法被子类方法覆盖时，可以使用“super. 方法名 ( 参数列表 )”。

### 5.2.4 类的多态性

#### 1. 方法覆盖

方法的覆盖是出现在继承过程中父子类之间的方法，例如创建一个大学生类别，大学生有学习的方法 study()。

```
public void study(){ // 在大学生类中定义 study() 方法
// 大学生所学习的课程。
    ……
}
```

有一个研究生类，研究生是特殊的大学生，研究生类继承大学生类，研究生也需要有学习的方法 study()。但是由于研究生类所学习的课程与大学生类所学习的课程不一样，因此即使大学生类有学习的方法，研究生类也必须重新定义 studay() 方法。

```
public void study(){ // 在研究生类中定义 study() 方法
// 研究生所学习的课程。
    ……
}
```

此时在子类中定义的方法，方法名、形参列表与父类中某个方法的名称、参数列表完全一致，可以说子类的方法覆盖了父类的方法。

满足方法覆盖的特征如下。

（1）子类某个方法的名称、参数列表及返回值类型必须与父类中某个方法的名称、参数列表和返回类型一致，有以下程序段：

```
public class 大学生 {
    public void study(){
    …
    }
}
public class 研究生 extends 大学生 {
```

```
    public int study() {
        return 0;
    }
}
```

这段代码编译器将编译出错。原因是子类 study() 方法与父类 study() 方法，方法名一样、形参列表一样，编译器试图覆盖父类的方法，但是方法的覆盖要求返回值也一样，而这两个方法中返回值类型不同，不能实现覆盖，所以编译器报错。这段代码必须改正，如下：

```
public class 大学生 {
    public void study(){
    …
    }
}
public class 研究生 extends 大学生 {
    public void study() {
        return 0;
    }
}
```

（2）子类方法不能缩小父类方法的访问权限，例如：

```
public class 大学生 {
    public void study(){
    …
    }
}
public class 研究生 extends 大学生 {
    private int study() {
        return 0;
    }
}
```

上段代码中子类 study () 方法是私有的，而父类的 study 方法是公共的。子类方法的访问权限小于父类方法的访问权限，无法实现覆盖，编译报错，改正如下：

```
public class 大学生 {
    public void study(){
    …
    }
}
public class 研究生 extends 大学生 {
    public int study() {
```

```
            return 0;
        }
}
```

（3）子类方法不能抛出比父类方法更多的异常，异常将在下一章具体介绍。子类方法抛出的异常必须和父类方法抛出的异常相同，或者子类方法抛出的异常类是父类方法抛出的异常类的子类。例如下面代码编译正常：

```
public class 大学生 {
        public void study()throws ExceptionBase {
        …
        }
}
public class 研究生 extends 大学生 {
        public int study()throws ExceptionSub {
            return 0;
        }
}
```

（4）方法覆盖只存在于父子类中。在同一个类中方法只能被重载，不能被覆盖。

（5）父类的静态方法不能被子类覆盖为非静态方法。以下代码将编译出错：

```
public class 大学生 {
        public static void study(){
        …
        }
}
public class 研究生 extends 大学生 {
        public int study() {
            return 0;
        }
}
```

（6）同样父类的非静态方法不能被子类覆盖为静态方法。以下代码是不合法的：

```
public class 大学生 {
        public void study(){
        …
        }
}
public class 研究生 extends 大学生 {
        public static int study() {
```

```
            return 0;
        }
}
```

（7）子类可以定义与父类的静态方法同名的静态方法，以便在子类中隐藏父类的静态方法。

（8）父类的私有方法不能被子类覆盖。

（9）父类的抽象方法可以被子类覆盖为抽象方法或非抽象方法；父类的非抽象方法也可以被子类覆盖为抽象方法。

例 5-6　覆盖的使用。

```
class 大学生 {
        static int n=0;
public void study(){
        System.out.println(" 正在学习大学生的课程！ ");
}
static void countNum(){
        n++;
        System.out.println(" 大学生班级人数 :"+n);
}
}
class 研究生 extends 大学生 {
        static int m=0;
        public void study(){
            System.out.println(" 正在学习研究生的课程！ ");
        }
        static void countNum(){
            m++;
            System.out.println(" 研究生班级人数 :"+m);
         }
}
public class Test{
public static void main(String args[]){
        大学生 s1=new 大学生 ();
        大学生 s2=new 大学生 ();
        研究生 g1=new 研究生 ();
        研究生  g2=new 研究生 ();
            s1.study();
            s1.countNum();
            s2.study();
            s2.countNum();
            g1.study();
            g1.countNum();
            g2.study();
```

```
        g2.countNum();
    }
}
```

程序运行结果如图 5–4 所示。


图 5–4　程序运行结果

### 2. 多态的特征

多态性是面向对象程序设计的一个重要特性。“polymorphism( 多态 )”一词来自希腊语，意为“多种形式”。Java 中多态性的产生主要跟类型有关。在程序设计过程中不需要去考虑父子类对象的类型问题，系统会根据对象的创建类型自动判别执行子类操作还是父类操作。

Java 中的多态性主要分为编译时多态和运行时多态。编译时多态也被称为静态多态性，即在编译程序时就能确定执行哪一个方法，方法的重载就是编译时多态的应用。运行时多态也被称为动态多态性，即在编译时无法确定调用哪一个方法，而是到实际执行时才能确定调用哪一个方法。方法覆盖就是运行时多态性的应用。

方法覆盖和方法重载有很多相同之处。

方法覆盖和方法重载的区别：

（1）方法覆盖是在父子继承中才会应用，而重载既可以在父子类中，也可以在同一类中。

（2）方法覆盖的两个方法，一个在父类中，一个在子类中，而且要具有相同的方法名、形参列表及返回值的类型。

（3）方法重载的两个方法方法名一样，但是形参列表必须不同。形参列表的不同指形参个数和形参类型有区别，与形参参数名无关。

例 5–7　方法重载与方法覆盖的应用。

```
class 大学生 {
    int num;
    String name;
```

```
    大学生 (){
        this.num=0;
        this.name=" 未命名 ";
    }
    大学生 (int num){
        this.num=num;
        this.name=" 未命名 ";
    }
    大学生 (int num,String name){
    this.num=num;
    this.name=name;
    }
    public void study(){
        System.out.println(name+" 正在学习大学生的课程！ ");
    }
    public void study(String km){
        System.out.println(name+" 正在学习大学生的课程 :"+km);
    }
    public void study(String k1,String k2){
        System.out.println(name+" 正在学习大学生的课程 :"+k1+" 和课程 :"+k2);
    }
}
class 研究生 extends 大学生 {
    研究生 (){
        super(2001);
    }
    研究生 (int num){
        super(num);
     }
    研究生 (int num,String name){
        super(num,name);
    }
    public void study(){
        System.out.println(name+" 正在学习研究生的课程！ ");
    }
    public void study(String km){
        System.out.println(name+" 正在学习研究生的课程 :"+km);
    }
     public void study(String k1,String k2){
        System.out.println(name+" 正在学习研究生的课程 :"+k1+" 和课程 :"+k2);
    }
}

public class Test{
    public static void main(String args[]){
```

```
            大学生 s1=new 大学生 ();
            大学生 s2=new 大学生 (1001);
            大学生 s3=new 大学生 (1002,"zhang");
            研究生 g1=new 研究生 ();
                研究生  g2=new 研究生 (2002,"li");
                s1.study();
                s2.study("C 语言 ") ;
                s3.study("C 语言 ","Java");
                g1.study(" 数据结构 ");
                g2.study(" 数据结构 ","Java 高级程序设计 ");
        }
    }
```

程序运行结果如图 5–5 所示。

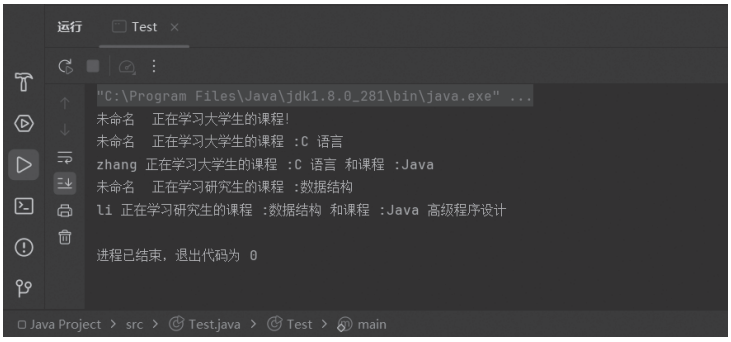


图 5–5　程序运行结果

该程序中大学生的构造方法体现了方法重载，研究生的构造方法也体现了方法重载。同时，大学生类中三个 study() 方法由于形参列表不同是方法重载；研究生类中三个 study() 方法由于形参列表不同也是方法重载。而大学生类中 study() 无参方法和子类研究生中 study() 无参方法方法名、形参列表、返回值类型都一样，实现的是方法覆盖。同理大学生类中的另两个 study 方法与研究生类中对应的 study 方法实现的也是方法覆盖。

### 5.2.5 构建方法的继承与重载

在 Java 中，构建方法（也称为构造方法）是一种特殊的方法，用于在创建对象时初始化对象。构建方法不能被继承，但子类可以通过调用父类的构造方法来初始化从父类继承的字段。同时，构建方法也可以被重载，以提供多种创建对象的方式。

#### 1. 方法的继承

方法的继承是类继承的一部分。当子类继承父类时，子类会自动获得父类中所有非私有的方法，即 public、protected 和默认访问权限的方法。子类既可以调用这

些继承来的方法，也可以重写（覆盖）这些方法以提供子类特有的行为。

例 5-8　假设有一个 Animal 类，它有一个 makeSound() 方法，然后创建一个 Dog 类继承自 Animal 类。

```
class Animal {
    void makeSound() {
        System.out.println("Some generic sound");
    }
}

class Dog extends Animal {
    // Dog 类继承了 Animal 类的 makeSound() 方法
    // 但在这里，并没有重写它，所以 Dog 的实例调用 makeSound() 将输出 "Some
      generic sound"
}

public class TestMethodInheritance {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.makeSound(); // 输出：Some generic sound
    }
}
```

在例 5-8 中，Dog 类没有定义自己的 makeSound() 方法，因此它继承了 Animal 类的 makeSound() 方法。

### 2. 方法的重载

方法的重载允许在同一个类中定义多个同名的方法，但这些方法的参数列表必须不同。参数列表的不同可以体现在参数的类型、顺序或数量上。

例 5-9　考虑一个 Calculator 类，它可能包含多个版本的 add() 方法来执行不同的加法运算。

```
class Calculator {
    // 两个整数相加
    int add(int a, int b) {
        return a + b;
    }

    // 三个整数相加
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // 两个浮点数相加
    double add(double a, double b) {
```

```
            return a + b;
        }
    }

    public class TestMethodOverloading {
        public static void main(String[] args) {
            Calculator calc = new Calculator();
            System.out.println(calc.add(1, 2)); // 输出：3
            System.out.println(calc.add(1, 2, 3)); // 输出：6
            System.out.println(calc.add(1.5, 2.5)); // 输出：4.0
        }
    }
```

在例 5–9 中，Calculator 类有三个 add() 方法，它们的方法名相同但参数列表不同，可以根据参数的类型和数量来调用不同版本的 add() 方法。

继承与多态是面向对象编程中的核心概念。继承允许用户根据已有的类（父类）创建新类（子类），子类可以继承父类的属性和方法，并可以添加或覆盖自己的属性和方法。Java 中使用 extends 关键字实现类的继承。多态性则表现为同一操作作用于不同的对象可以有不同的行为，这主要通过方法的覆盖和重载来实现。方法的覆盖发生在父子类之间，要求方法名、参数列表及返回类型相同，子类方法不能缩小父类方法的访问权限，也不能抛出更多的异常。多态性增强了程序的灵活性和可扩展性，使得程序更加易于维护和修改。通过继承与多态，用户可以构建出更加复杂且功能丰富的软件系统。

### 5.2.6 引用数据类型的转换

内部类是定义在另一个类内部的类。内部类提供了一种封装代码的方式，使得代码更加模块化，并可以在同一个作用域内访问外部类的成员（包括私有成员）。内部类可以分为成员内部类、静态内部类、局部内部类和匿名内部类。

#### 1. 成员内部类

成员内部类是非静态的，它依赖于外部类的实例。在成员内部类中，可以访问外部类的所有成员（包括私有成员），但是外部类不能直接访问内部类的成员（除非创建内部类对象）。

例 5–10　成员内部类。

```
public class OuterClass {
    private int outerField = 100;

    class InnerClass {
        public void display() {
            System.out.println("Access outerField: " + outerField);
```

```
        }
    }

    public void testInnerClass() {
        InnerClass inner = new InnerClass();
        inner.display();
    }

    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        outer.testInnerClass(); // 输出 : Access outerField: 100
    }
}
```

在例 5–10 中，InnerClass 是 OuterClass 的成员内部类。InnerClass 能够访问 OuterClass 的私有成员 outerField。

### 2. 静态内部类

静态内部类使用 static 关键字修饰，它不依赖于外部类的实例，因此可以独立于外部类存在。静态内部类不能访问外部类的非静态成员，但可以通过外部类名来访问其静态成员。

例 5–11　静态内部类。

```
public class OuterClass {
    private static int staticOuterField = 200;

    static class StaticInnerClass {
        public void display() {
            System.out.println("Access staticOuterField: " + staticOuterField);
        }
    }

    public static void main(String[] args) {
        StaticInnerClass inner = new StaticInnerClass();
        inner.display(); // 输出 : Access staticOuterField: 200
    }
}
```

在例 5–11 中，StaticInnerClass 是 OuterClass 的静态内部类，它能够访问 OuterClass 的静态成员 staticOuterField。

### 3. 匿名内部类

匿名内部类是在声明和实例化类时直接创建类的对象的语法，通常用于实现简单的接口或继承简单的类。

例 5–12　匿名内部类。

```
interface HelloWorld {
    void greet();
}

public class Demo {
    public static void main(String[] args) {
        HelloWorld hello = new HelloWorld() {
            @Override
            public void greet() {
                System.out.println("Hello, World!");
            }
        };

        hello.greet(); // 输出 : Hello, World!
    }
}
```

在例 5–12 中，没有显式地定义 HelloWorld 接口的实现类，而是直接通过匿名内部类的方式实现了该接口，并创建了 HelloWorld 接口的对象。

内部类是 Java 中一个重要的概念，它提供了更高的代码封装性和灵活性。通过合理地使用内部类，可以设计出更加清晰、易于维护的 Java 程序。

### 5.2.7 对象类型转换

Java 语言允许把引用类型转换为子类类型，称为向下类型转换。也允许把子类型对象转换为父类型对象，称为向上类型转换。这种转换只适合于父子类对象中。

1. 向下转型

例 5–13　将父类对象转换为子类对象。

```
class 大学生 {
    int num;
    String name;
    大学生 (int num,String name){
        this.num=num;
        this.name=name;
    }
    public void study(){
        System.out.println(name+" 正在学习大学生的课程！ ");
    }
}
class 研究生 extends 大学生 {
    研究生 (int num,String name){
```

```
        super(num,name);
    }
    public void study(){
        System.out.println(name+" 正在学习研究生的课程 ");
    }
}
public class Test{
    public static void main(String args[]){
        大学生 s1=new 大学生 (1001,"zhang");
        研究生 s2=new 研究生 (2001,"wang");
        大学生 s3=new 研究生 (2002,"li");
        s1.study();
        s2.study();
        s3.study();
        s1=( 大学生 )s2;
        s1.study();
    }
}
```

程序运行结果如图 5-6 所示。

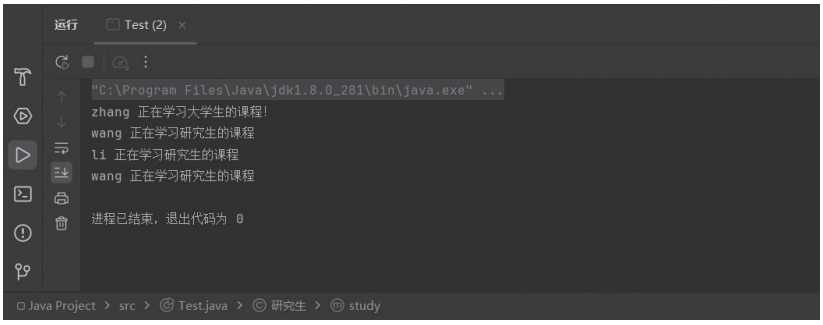


图 5-6　程序运行结果

### 2. 向上转型

例 5-14　父类对象转换为子类对象。

```
class 大学生 {
    int num;
    String name;
    大学生 (int num,String name){
        this.num=num;
        this.name=name;
    }
    public void study(){
        System.out.println(name+" 正在学习大学生的课程！ ");
    }
```

```
}
class 研究生 extends 大学生 {
    研究生 (int num,String name){
        super(num,name);
    }
    public void study(){
        System.out.println(name+" 正在学习研究生的课程 ");
    }
}
public class Test{
    public static void main(String args[]){
        大学生 s1=new 大学生 (1001,"zhang");
        研究生 s2=new 研究生 (2001,"wang");
        s1.study();
        s2.study();
        s2=( 研究生 )s1;
        s2.study();
    }
}
```

程序运行结果如图 5-7 所示。

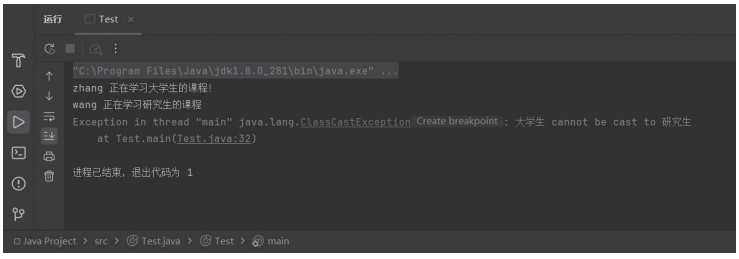


图 5-7　程序运行结果

当父类对象转换为子类对象时首先需要强制类型的转换，但是在实际运行调用方法时仍报错。

### 5.2.8 final 修饰符

final 关键字在继承中的主要作用有以下两个方面。

#### 1. 使用 final 修饰方法

用 final 关键字修饰方法，该方法将不能被覆盖。

例 5-15　找出下列程序中的错误。

```
class A {
    final void show() {
        System.out.println("A 类方法 ");
```

```
        }
    }
    class B extends A {
        void show() { // 此处将报错，因为 A 类中 show 方法声明为 final
            System.out.println("B 类方法 ");
        }
    }
```

show( ) 被声明成 final，所以它不能被 B 类方法覆盖。

### 2. 使用 final 修饰类

使用 final 修饰的类不能被继承。

**例 5–16**　找出下列程序中的错误。

```
final class A {
    void show() {
        System.out.println("A 类方法 ");
    }
}
class B extends A {// 此处将报错，因为 A 类声明为 final
    void show() {
        System.out.println("B 类方法 ");
    }
}
```

A 类用 final 修饰，A 类不能被继承，所以 B 类继承 A 类时将报错。

### 3. Object 类

在 Java 中，有一个特殊的 Object 类，它是所有类的直接或间接子类，所有其他的类都是 Object 的子类。子类继承父类时，子类将拥有父类的属性和方法，所以在 Java 中任何一个类都具有 Object 类的属性和操作。表 5–2 是 Object 类方法介绍。

表 5–2　Object 类方法及用途

| 方　法 | 用　途 |
|---|---|
| Object clone( ) | 创建一个和被复制的对象完全一样的新对象 |
| boolean equals(Object object) | 判定对象是否相等 |
| void finalize( ) | 在一个不常用的对象被使用前调用 |
| Class getClass( ) | 获取运行时一个对象的类 |
| int hashCode( ) | 返回调用对象有关的散列值 |
| void notify( ) | 恢复一个等待调用对象线程的执行 |
| void notifyAll( ) | 恢复所有等待调用对象线程的执行 |

（续表）

| 方　法 | 用　途 |
| --- | --- |
| String toString( )<br>void wait( )<br>void wait(long milliseconds)<br>void wait(long milliseconds,<br>int nanoseconds) | 返回描述对象的一个字符串<br>等待另一个线程的执行 |

### 5.2.9 抽象类

#### 1. 什么是抽象类

在面向对象的概念中，一切皆为对象，而所有的对象都是通过类来创建的。但并不是所有类的设计都是为了创建对象。有些类的设计是为了描述一些共性问题，那么只满足这些共性问题的具体的对象是不存在的，或者说是没有实际意义的。例如，设计学生信息管理系统时，要记录每一类学生的信息，如小学生、中学生、大学生等，这些学生有一些共性的信息，而每类学生又有自己的特征，所以在进行类的设计时，往往把所有学生的共性特点提取出来形成一个类别，该类别的设计只是为了其他类继承。用该类创建的对象无实际意义，所以该类不会创建具体的对象，这样的类称为抽象类。

抽象类就是从多个事物中将共性的、本质的内容抽取出来。多个对象都具备相同的功能，但是功能具体内容有所不同，那么在抽取过程中，只抽取了功能定义，并未抽取功能主体，那么只有功能声明，没有功能主体的方法称为抽象方法。例如狼和狗都有吼叫，可是吼叫内容是不一样的，所以抽象出来的犬科虽然有吼叫功能，但是并不明确吼叫的细节。

抽象类的特点：

（1）抽象方法只能定义在抽象类中，抽象的方法和抽象的类必须由关键字 abstract 修饰。

（2）抽象类值只定义方法声明，不定义功能主题，即方法的实现。

（3）抽象类不可以被创建对象。

（4）抽象类只有子类继承了父类中的方法，并且对其中的所有抽象方法进行了重写，该子类才不是抽象类，只要不是重写当中的所有抽象方法，那么这个子类还是抽象类。

抽象类是使用 abstract 关键字来修饰的。abstract 关键字既可以修饰类，也可以修饰方法。由 abstract 修饰的类就是抽象类，由 abstract 修饰的方法就是抽象方法。抽象类的定义格式如下：

```
public abstract class 抽象类名 {
        属性
        方法
}
```

例 5–17　设计学生类为抽象类。

```
abstract class 学生 {
        int num;
        String name;
                学生 (int num,String name){
                this.num=num;
                this.name=name;
                }
public void study(){
                ;
}
}
class 小学生 extends 学生 {
        小学生 (int num,String name){
                super(num,name);
        }
public void study(){
        System.out.println(name+" 正在学习小学的课程 ");
}
}
class 中学生 extends 学生 {
        中学生 (int num,String name){
                super(num,name);
        }
public void study(){
        System.out.println(name+" 正在学习中学的课程 ");
}
}
class 大学生 extends 学生 {
        大学生 (int num,String name){
                super(num,name);
        }
public void study(){
        System.out.println(name+" 正在学习大学的课程 ");
}
}
public class Test{
public static void main(String args[]){
        小学生 s1=new 小学生 (1001,"zhang");
```

```
        中学生 s2=new 中学生 (2001,"wang");
        大学生 s3=new 大学生 (3001,"li");
        s1.study();
        s2.study();
        s3.study();
    }
    }
```

程序运行结果如图 5–8 所示。


图 5–8　程序运行结果

该程序中学生类被设计为抽象类，学生类的存在就是为了继承。在学生类的设计中有 void study() 方法，但是该方法体中没有任何语句，所以该方法可以设计为抽象方法。

### 2. 抽象方法

当抽象类中某些方法的实现没有实际意义时，可以将该方法定义为抽象方法。抽象方法是只有方法的声明，而没有方法体的方法。

例 5–18　定义抽象方法。

```
abstract class 学生 {
        int num;
        String name;
            学生 (int num,String name){
            this.num=num;
            this.name=name;
            }
abstract void study();
}
class 小学生 extends 学生 {
        小学生 (int num,String name){
            super(num,name);
        }
public void study(){
        System.out.println(name+" 正在学习小学的课程 ");
}
}
```

```
class 中学生 extends 学生 {
    中学生 (int num,String name){
        super(num,name);
    }
public void study(){
    System.out.println(name+" 正在学习中学的课程 ");
}
}
class 大学生 extends 学生 {
    大学生 (int num,String name){
        super(num,name);
    }
public void study(){
    System.out.println(name+" 正在学习大学的课程 ");
}
}
public class Test{
public static void main(String args[]){
    小学生 s1=new 小学生 (1001,"zhang");
    中学生 s2=new 中学生 (2001,"wang");
    大学生 s3=new 大学生 (3001,"li");
    s1.study();
    s2.study();
    s3.study();
}
}
```

运行结果同上。

关于抽象方法的几点说明：

（1）抽象类中可以有抽象方法，也可以没有，但是有抽象方法的类一定是抽象类。

（2）在抽象类中，抽象方法前必须加 abstract 关键字修饰。

```
abstract class 学生 {
    int num;
    String name;
        学生 (int num,String name){
        this.num=num;
        this.name=name;
        }
abstract void study();
}
```

该类中 abstract void study(); 方法只有方法的声明，没有方法的实现为抽象方法，

如果不加 abstract 将报错。

（3）如果抽象类的子类是一个具体类，就必须实现抽象类的所有抽象方法。

```
class 大学生 extends 学生 {
        大学生 (int num,String name){
              super(num,name);
        }
public void study(){
        System.out.println(name+" 正在学习大学的课程 ");
        }
}
```

大学生类没有用 abstract 修饰，所以该类不是抽象类，而是一个具体类，该类必须实现父类的所有抽象方法。

（4）private、static 关键字不能用于修饰抽象方法。

### 5.2.10 接口

Java 只支持单重继承，即一个类只能有一个直接父类，不能同时是多个父类的子类，例如已知沙发类描述沙发的属性和操作，又已知床类描述床的属性和操作。现要设计沙发床，具有沙发和床的共同特点，但是由于 Java 是单重继承，因此设计沙发床不能同时继承沙发和床。Java 使用接口的设计来解决该类问题，接口是 Java 程序设计中最重要的概念之一。

#### 1. 接口的定义

Java 中的接口是特殊的抽象类，是一些抽象方法和常量的集合，其主要作用是使得处于不同层次上并且互不相干的类能够执行相同的操作和引用相同的值，而且可以同时实现来自不同类的方法。

接口与普通的抽象类的不同之处在于接口的数据成员必须被初始化，接口中的方法必须全部都声明为抽象方法。

接口的一般定义格式为：

```
[public] interface 接口名 {
[public][static][final] 类型  常量 = 常量值;
// 数据成员必须被初始化
[public][abstract] 方法类型  方法名 ([ 参数列表 ]);
// 方法必须声明为抽象方法
}
```

其中，interface 是接口的保留字，接口名是 Java 标识符。如果缺少 public 修饰符，则该接口只能被与它在同一个包中的类实现。常量名是 Java 标识符，通常用大写字母标识，常量值必须与声明的类型相一致；方法名是 Java 标识符，方法类型是

指该方法的返回值类型。在 Java 程序中接口中的 final 和 abstract 可以省略。

例 5–19　接口的定义。

```
interface 沙发 {
public static final int rs=5;
public void WatchTV();
}
```

接口是特殊的抽象类，接口中所有的属性都是常量值，所有的方法都是抽象方法。在接口中每一个方法只能有方法的声明，不能有方法体，但是可以省略 abstract 修饰。同时接口没有构造方法，更不能被实例化。

接口和接口之间的关系仍然是继承的关系，但是在接口继承接口时可以继承多个接口，与类之间的单继承有区别。

```
interface A{
      …
}
interface B{
      …
}
interface C extends A,B{
      …
}
```

### 2. 接口的使用

接口的使用与类的使用不同，类可以直接创建对象，而接口不能，接口中所有的方法都是抽象方法。接口在使用的时候要实例化相应的实现类。使用接口时首先需要创建类实现接口，而且一个类可以实现多个接口。

class 类名 implements 多个接口 {
　实现方法
}

例 5–20　接口的实现。

```
public class Test
{
      public static void main(String args[])
      {
            沙发床 s=new 沙发床 ();
            s.setz(1);
            if(s.getz()==1)
                  s.watchTV();
            else
```

```
                s.sleeping();
        }
}
interface 沙发
{
        void watchTV();
}
interface 床
{
        void sleeping();
}
class 沙发床  implements 沙发 , 床
{   int bz=0;
        void setz(int bz)
        {
                this.bz=bz;
        }
        int getz()
        {
                return bz;
        }
        @Override

        public void sleeping() {
                // TODO Auto-generated method stub
                System.out.println(" 正在睡觉 ");
        }

        @Override
        public void watchTV() {
                // TODO Auto-generated method stub
                System.out.println(" 正在看电视 ");
        }
}
```

程序运行结果如图 5-9 所示。

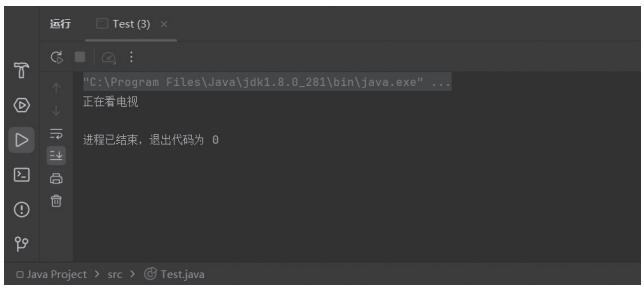


图 5-9　程序运行结果

需要注意的是，如果一个类不能实现接口中所有的接口，该类一定为抽象类。

### 5.2.11 类间关系的 UML 表示

在面向对象编程和软件设计中，UML（unified modeling language，统一建模语言）提供了一种强大的方式来描述系统中的类、接口以及它们之间的复杂关系。在 UML 中，类图是最基本也是最重要的图之一，它展示了系统中类的静态结构和它们之间的关系。

#### 1. 类图的基本元素

在 UML 类图中，主要包含以下几种基本元素。

（1）类（Class）：使用带有三个分隔区的矩形表示，第一行是类名，第二行是属性（可选默认值），第三行是方法。可以使用 +、–、# 分别表示公开（public）、私有（private）、保护（protected）成员。

（2）接口（Interface）：使用带有 <<interface>> 标记的矩形表示，只列出接口方法，不展示属性和实现细节。

#### 2. 类间的关系

（1）泛化（Generalization）。

泛化表达了“is a”的关系，主要有继承（Inheritance）和实现（Implementation）两种形式。

①继承使用实线加空心三角箭头表示，箭头指向父类，例如鸟类继承动物类，如图 5–10 所示。

②实现使用虚线加空心三角箭头表示，箭头指向接口，例如大雁实现了飞翔接口，如图 5–11 所示。

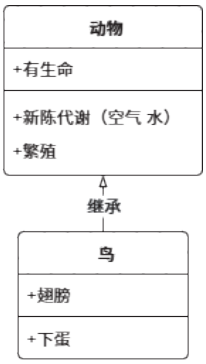


图 5–10　继承关系

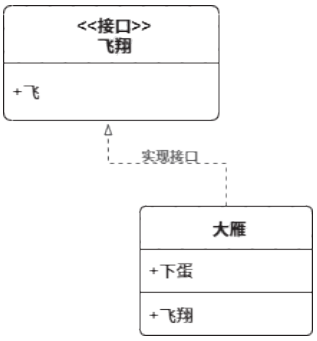


图 5–11　实现关系

（2）关联（Association）。

关联表达了“has a”的关系，主要有三种具体形式：聚合（Aggregation）、组合（Composition）、普通关联（Association）。

①聚合使用实线加空心菱形箭头表示，整体拥有部分但关联较弱，部分可以独立

于整体存在，例如大雁聚集成雁群，但大雁可以离开雁群独立存在，如图 5–12 所示。

②组合使用实线加实心菱形箭头表示，整体与部分紧密相连，部分不能独立于整体存在，例如翅膀是鸟的一部分，不能脱离鸟而单独存在，如图 5–13 所示。

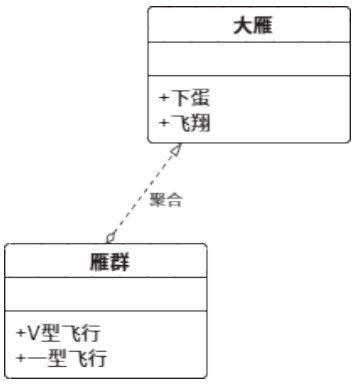


图 5–12　聚合关系

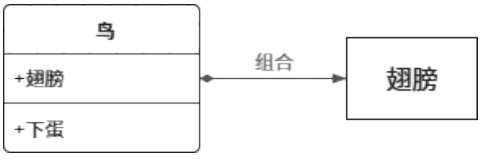


图 5–13　组合关系

③普通关联使用实线箭头表示，用于描述一般性的“has a”关系，可以是双向的，例如企鹅与气候之间的相互影响关系，如图 5–14 所示。

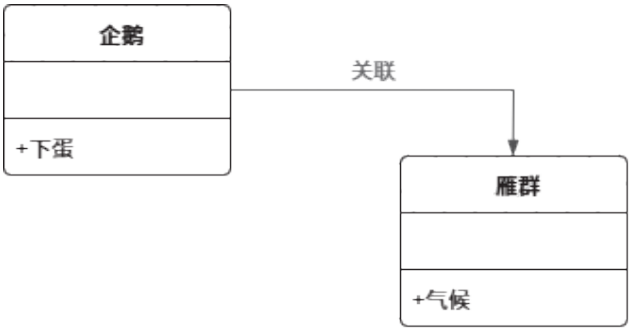


图 5–14　关联关系

（3）依赖（Dependency）。

依赖表达了“use a”的关系，表示一个类在功能实现上依赖于另一个类。依赖关系使用虚线箭头表示，从使用者指向被使用者，例如动物的新陈代谢功能依赖于空气和水，如图 5–15 所示。

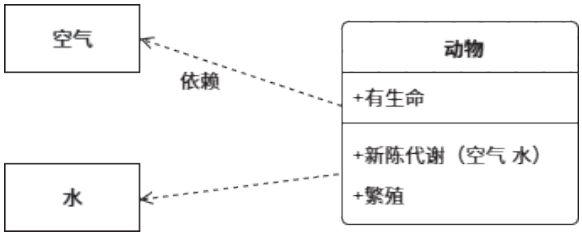


图 5–15　依赖关系

## 5.3 项目实践

设计一个员工薪资增长系统，根据员工的类型和工作年限进行薪资调整。要求实现一个名为 Person 的类和它的子类 Employee，Manager 是 Employee 的子类。

（1）Person 类中的属性有姓名（name，String），地址（address，String），定义该类的构造方法；

（2）Employee 类中的属性有工号（ID，String），工资（wage，double），工龄（workingyears，int），级别（level，String），定义该类的构造方法；

（3）Manager 类中的属性有部门（department，String），定义该类的构造方法；

（4）要求使用方法的重写（人、员工、经理类信息的输出方法）和实现类的构造方法（super 关键字）；

（5）编写一个测试类，产生一个员工和一个经理对象，输出员工和经理的基本信息；

（6）设计一个 Add 类用于涨工资，普通员工一次能涨 10%，经理能涨 20%；

（7）员工工龄超过 3 年的涨一次工资，部门经理工龄超过 1 年的涨一次工资。

程序运行结果如图 5-16 所示。


图 5-16　员工薪资增长系统运行结果

程序源码如下：

```
Person.java
public class Person {
    protected String name;
    protected String address;

    public Person(String name, String address) {
        this.name = name;
        this.address = address;
    }

    // 重写 toString 方法输出基本信息
```

```
        @Override
        public String toString() {
            return "Person{" +
                    "name='" + name + '\'' +
                    ", address='" + address + '\'' +
                    '}';
        }
    }
    Employee.java
    public class Employee extends Person {
        private String ID;
        private double wage;
        private int workingyears;
        private String level;

        public Employee(String name, String address, String ID, double wage, int workingyears,
String level) {
            super(name, address);
            this.ID = ID;
            this.wage = wage;
            this.workingyears = workingyears;
            this.level = level;
        }

        // Getters and Setters for ID
        public String getID() {
            return ID;
        }

        public void setID(String ID) {
            this.ID = ID;
        }

        // Getters and Setters for wage
        public double getWage() {
            return wage;
        }

        public void setWage(double wage) {
            this.wage = wage;
        }

      // Getters and Setters for workingyears
        public int getWorkingyears() {
            return workingyears;
```

```
        }

        public void setWorkingyears(int workingyears) {
            this.workingyears = workingyears;
        }

        // Getters and Setters for level
        public String getLevel() {
            return level;
        }

        public void setLevel(String level) {
            this.level = level;
        }

        @Override
        public String toString() {
            return "Employee{" +
                    super.toString() +
                    "ID='" + ID + '\'' +
                    ", wage=" + wage +
                    ", workingyears=" + workingyears +
                    ", level='" + level + '\'' +
                    '}';
        }
    }
    Manager.java
    public class Manager extends Employee {
        private String department;

        public Manager(String name, String address, String ID, double wage, int workingyears,
String level, String department) {
            super(name, address, ID, wage, workingyears, level);
            this.department = department;
        }

        // Getter for department
        public String getDepartment() {
            return department;
        }

        // Setter for department
        public void setDepartment(String department) {
            this.department = department;
        }
```

```
        @Override
        public String toString() {
            return "Manager{" +
                    super.toString() +
                    ", department='" + department + '\'' +
                    '}';
        }
    }
    add.java
    class Add {
        public static void increaseWage(Employee employee) {
            if (employee instanceof Manager) {
                if (employee.getWorkingyears() > 1) {
                    employee.setWage(employee.getWage()*1.20);
                }
            } else {
                if (employee.getWorkingyears() > 3) {
                    employee.setWage(employee.getWage()*1.10);
                }
            }
        }
    }

    Test.java
    public class Test {
        public static void main(String[] args) {
            Employee employee = new Employee(" 张 三 ", " 北 京 市 ", "E001", 5000, 4, "
初级 ");
            Manager manager = new Manager(" 李四 ", " 上海市 ", "M001", 8000, 2, " 高级 ",
" 研发部 ");

            System.out.println(" 涨工资前 :");
            System.out.println(employee);
            System.out.println(manager);

            Add.increaseWage(employee);
            Add.increaseWage(manager);

            System.out.println(" 涨工资后 :");
            System.out.println(employee);
            System.out.println(manager);
        }
    }
```

## 5.4 实作强化

使用 Java 的继承、接口和抽象类设计一个交通工具类（飞机、轮船、公交车、自行车），检测交通工具是否具备 GPS 或者自动驾驶功能。

具体要求：

（1）所有交通工具都具有启动和中止功能；

（2）所有交通工具的行驶方式都不一样，例如飞机在空中飞、轮船在水上飘、公交车在公路上跑、自行车应在自行车道上跑；

（3）部分交通工具的控制方式不一样，例如飞机、轮船具有自动驾驶功能，飞机、轮船、公交车具有 GPS 功能，自行车需要人力控制驾驶，无 GPS 功能。

```
ControlledTransport.java
public abstract class ControlledTransport implements Transport {
    private boolean hasGPS;
    private boolean hasAutoPilot;

    public ControlledTransport(boolean hasGPS, boolean hasAutoPilot) {
        this.hasGPS = hasGPS;
        this.hasAutoPilot = hasAutoPilot;
    }

    public boolean hasGPS() {
        return hasGPS;
    }

    public void setGPS(boolean hasGPS) {
        this.hasGPS = hasGPS;
    }

    public boolean hasAutoPilot() {
        return hasAutoPilot;
    }

    // 自动驾驶功能（可选实现，因为并非所有子类都需要）
    public void autoPilot() {
        if (hasAutoPilot) {
            System.out.println(" 有自动驾驶功能 ");
    // 具体的自动驾驶逻辑
        } else {
            System.out.println(" 没有自动驾驶功能 ");
        }
    }

    // 抽象方法需要子类实现
```

```
    @Override
    public abstract String move();
}
Transport.java
public interface Transport {
    void start();
    void stop();
    String move(); // 描述行驶方式
}
TransportKind.java
class Plane extends ControlledTransport {
    public Plane() {
        super(true, true); // 飞机有 GPS 和自动驾驶功能
    }

    @Override
    public void start() {
        System.out.println(" 飞机起飞 ");
    }

    @Override
    public void stop() {
        System.out.println(" 飞机降落 ");
    }

    @Override
    public String move() {
        return " 飞机在天上飞 ";
    }
}

    class Ship extends ControlledTransport {
    public Ship() {
        super(true, true); // 轮船有 GPS 和自动驾驶功能
    }

    @Override
    public void start() {
        System.out.println(" 轮船开动 ");
    }

    @Override
    public void stop() {
        System.out.println(" 轮船停船 ");
    }
```

```
    @Override
    public String move() {
        return " 轮船在水上走 ";
    }
}

    class Bus extends ControlledTransport {
    public Bus() {
        super(true, false); // 公交车有 GPS，但没有自动驾驶功能
    }

    @Override
    public void start() {
        System.out.println(" 巴士开动 ");
    }

    @Override
    public void stop() {
        System.out.println(" 巴士停车 ");
    }

    @Override
    public String move() {
        return " 巴士在公路上走 ";
    }
}

    class Bicycle implements Transport {
    @Override
    public void start() {
        System.out.println(" 骑自行车 ");
    }

    @Override
    public void stop() {
        System.out.println(" 自行车停下 ");
    }

    @Override
    public String move() {
        return " 骑自行车 ";
    }

}
```

TransportTest.java

```
public class TransportTest {
    public static void main(String[] args) {
    // 创建交通工具实例
        Plane plane = new Plane();
        Ship ship = new Ship();
        Bus bus = new Bus();
        Bicycle bicycle = new Bicycle();

    // 测试飞机
        testTransport(plane, " 飞机 ");

    // 测试轮船
        testTransport(ship, " 轮船 ");

    // 测试公交车
        testTransport(bus, " 巴士 ");

    // 测试自行车
        testTransport(bicycle, " 自行车 ");
    }

    private static void testTransport(Transport transport, String name) {
        System.out.println(" 测试 " + name + ":");
        transport.start();
        System.out.println(transport.move());
        transport.stop();

    // 如果交通工具是 ControlledTransport 的实例，则测试 GPS 和自动驾驶功能
    if (transport instanceof ControlledTransport) {
        ControlledTransport controlledTransport = (ControlledTransport) transport;
        System.out.println(name + " 有 GPS: " + controlledTransport.hasGPS());

    // 测试自动驾驶功能（仅当该交通工具有自动驾驶时）
            if (controlledTransport.hasAutoPilot()) {
                controlledTransport.autoPilot();
            } else {
                System.out.println(name + ", 该交通工具没有自动驾驶能力 ");
            }
        } else {
            System.out.println(name + ", 该交通工具的驾驶方式没有 GPS 或者自动驾驶 ");
        }
        System.out.println(); // 空行分隔不同交通工具的测试输出
    }
}
```

程序运行结果如图 5–17 所示。

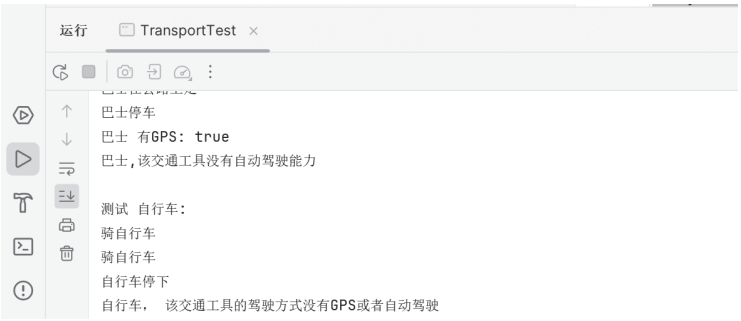


图 5–17　交通系统运行结果

## 5.5 精选练习

### 一、选择题

1. 继承是面向对象编程的一个重要特征，它可降低程序的复杂性并使代码（　　）。

A. 可读性好　　B. 可重用　　C. 可跨包访问　　D. 运行更安全

2. 下列关于父类、子类的关系说法不正确的是（　　）。

A. 子类可以共享父类的公共域和方法

B. 子类和父类一定会存在某些差异，否则就应该是同一个类

C. 子类中的类变量可以隐藏父类中的实例变量

D. 子类可以从父类中继承域和方法，但是不可以对这些域和方法进行重定义及扩充新的内容

3. 下列关于抽象类说法错误的是（　　）。

A. 抽象类不能被初始化

B. 抽象类的声明是在类说明中使用 abstract 修饰符

C. 抽象类是一种完整类

D. 抽象类是指没有具体对象的一种概念类

4. Java 对方法和成员变量提供了（　　）个修饰符号用于权限控制。

A. 2　　B. 3　　C. 4　　D. 5

5. 下列说法正确的是（　　）。

A. 在面向过程的程序设计中，各函数可以重名

B. 在面向对象的程序设计中，各函数不可以重名

C. Object 类中的成员都是方法

D. Object 类不是用户自定义的所有类的父类

## 二、程序阅读题

1. 阅读以下程序，写出输出结果。

```
class  Animal {
      Animal() {
            System.out.print ("Animal  "); }
}
public  class  Dog  extends  Animal {
      Dog() {
            System.out.print ("Dog "); }
      public static void main(String[] args) {
            Dog  snoppy= new  Dog(); }
}
```

2. 阅读以下程序，写出输出结果。

```
abstract class Shape {
      abstract void display();
}
class Circle extends Shape {
      void display() {
            System.out.println("Circle");
      }
}
class Rectangle extends Shape {
      void display() {
            System.out.println("Rectangle");
      }
}
class Triangle extends Shape {
      void display() {
            System.out.println("Triangle");
      }
}

public class AbstractClassDemo{
      public static void main(String args[]){
            (new Circle()).display();
            (new Rectangle()).display();
            (new Triangle()).display();
      }
}
```

## 三、编程题

1. 利用多态性编程，创建一个 Square 类，实现求三角形、正方形和圆形的面积。方法：抽象出一个共享父类，定义一个函数为求面积的公共界面，再重新定义各形