

第5章

类之间的关系



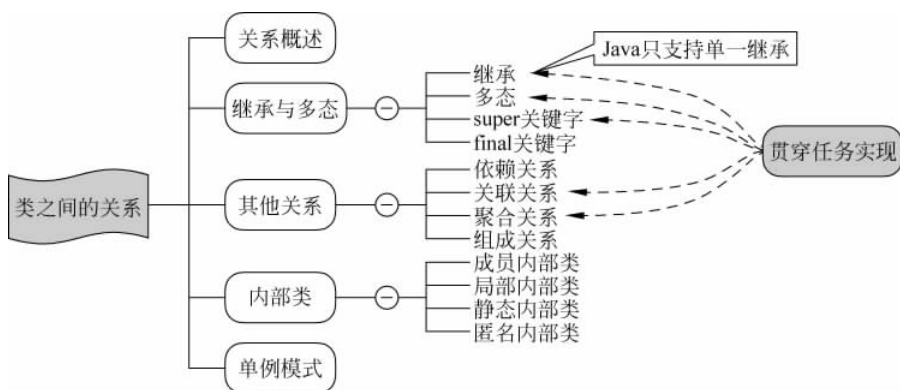
任务驱动

本章任务是使用继承重构“Q-DMS 数据挖掘”系统的数据采集的实体类并测试：

- 【任务 5-1】 编写基础信息实体类。
- 【任务 5-2】 使用继承重构日志、物流实体类，并测试运行。
- 【任务 5-3】 编写日志数据匹配类，对日志实体类数据进行匹配。
- 【任务 5-4】 编写物流数据匹配类，对物流实体类数据进行匹配。



学习路线



本章目标

知 识 点	Listen(听)	Know(懂)	Do(做)	Revise(复习)	Master(精通)
关系概述	★	★			
继承	★	★	★	★	★
多态	★	★	★		
其他关系	★	★	★		
内部类	★	★			
单例模式	★	★			

5.1 关系概述

在面向对象的程序设计中,通常不会存在一个孤立的类,类和类之间总是存在一定的关系,通过这些关系,才能实现软件的既定功能。类与类之间的关系对于深入理解面向对象概念具有非常重要的作用,也有利于程序员从专业的、合理的角度面向对象分析问题和解决问题。

根据 UML(Unified Modeling Language,统一建模语言)规范,类与类之间存在六种关系。

- 继承: 一个类可以继承另外一个类,并在此基础上添加自己的特有功能。继承也称为泛化,表现的是一种共性与特性的关系。
- 实现: 一个类实现接口中声明的方法,其中接口对方法进行声明,而类完成方法的定义,即实现具体功能。实现是类与接口之间常用的关系,一个类可以实现一个或多个接口中的方法。
- 依赖: 在一个类的方法中操作另外一个类的对象,这种情况称为第一个类依赖于第二个类。
- 关联: 在一个类中使用另外一个类的对象作为该类的成员变量,这种情况称为关联关系。关联关系体现的是两个类之间语义级别的一种强依赖关系。
- 聚合: 聚合关系是关联关系的一种特例,体现的是整体与部分的关系,即 has-a 的关系。通常表现为一个类(整体)由多个其他类的对象(部分)作为该类的成员变量,此时整体与部分之间是可以分离的,整体和部分都可以具有各自的生命周期,部分可以属于多个整体对象,也可以为多个整体对象共享。
- 组成: 组成关系也是关联关系的一种特例,与聚合关系一样也是体现整体与部分的关系,但组成关系中的整体与部分是不可分离的,即 contains-a 的关系,这种关系比聚合更强,也称为强聚合,当整体的生命周期结束后,部分的生命周期也随之结束。

类与类之间的这六种关系中,继承和实现体现了类与类之间的一种纵向的关系,而其余四种则体现了类与类之间的横向关系。其中,关联、聚合、组成这三种关系更多体现的是一种语义上的区别,而在代码上则是无法区分的。

 **注意** 本章只对继承、依赖、关联、聚合以及组成这五种关系进行介绍,而实现关系涉及接口的知识,因此在本书的第 6 章再做介绍。UML 是一种流行的面向对象分析与设计技术,对 UML 的详细介绍已超出本书的范畴,读者可以参阅其他相关资料进行了解和学习。

5.2 继承与多态

继承与多态是面向对象的特征之一,也是实现软件复用的重要手段。



视频讲解



视频讲解

5.2.1 继承

继承是类之间的一个非常重要的关系,是面向对象编程的基石及核心。

在 Java 程序中,一个类可以继承另外一个类,被继承的类通常称为“父类”(parent)

class)、“超类”(super class)或者“基类”(base class),继承者通常称为“子类”(child class 或 subclass)或者“派生类”(derived class)。子类与父类的继承关系如图 5-1 所示。

Java 中的继承具有单一继承的特点,即每个子类只有一个直接父类,但一个父类可以有多个子类。通常可以将父类创建成一个通用的类,定义子类共有的通用特性;而子类则是父类的一个专门用途的版本,子类不仅继承了父类的通用特性,并且在此基础之上增加了自己特有的属性和方法,以满足新的需求。

在图 5-1(b)中,学生子类的成员属性除了拥有学号、班号、年级、校名和学习课程外,还拥有父类“人”的成员属性身份证号和姓名等信息;学生子类的成员方法除了拥有出勤、上课、参加考试和查询成绩之外,还拥有父类“人”的成员方法吃饭、睡觉和娱乐等。

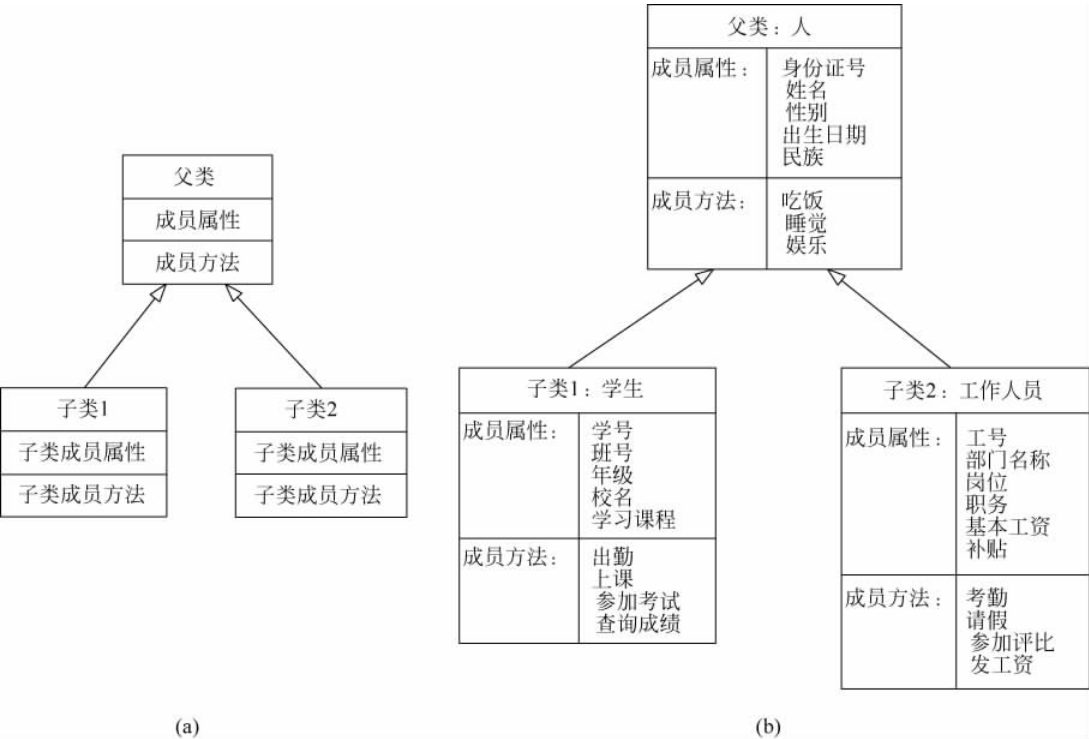


图 5-1 继承关系

Java 中的继承通过使用 extends 关键字来实现,其语法格式如下:

【语法】

```
[访问符][修饰符] class 子类 extends 父类{
...
}
```

【示例】 子类继承父类

```
public class SubClass extends SuperClass{
...
}
```

上述代码通过使用 extends 关键字使子类 SubClass 继承父类 SuperClass。如果定义一个类没有使用 extends 继承任何父类,则自动继承 java.lang.Object 类。Object 类是所有类的顶级父类,在 Java 中,所有类都是直接或间接地继承 Object 类。

Java 只支持单一继承,因此下面代码是错误的:

```
Class C extends A,B{} //Error,一个类不能继承多个类
```



注意 虽然 Java 不能像 C++ 一样支持多继承,但可以通过实现多个接口来弥补。

在继承过程中,子类拥有父类所定义的所有属性和方法,但父类可以通过“封装”思想隐藏某些数据,只对子类提供可访问的属性和方法。实例化一个子类对象时,会先调用父类构造方法进行初始化,再调用子类自身的构造方法进行初始化,即构造方法的执行次序是:父类→子类。

下述系列代码演示在继承关系中父类和子类构造方法的调用次序。

【代码 5-1】 SuperClass.java

```
package com.qst.chapter05;

//父类
public class SuperClass {
    public int a;

    public SuperClass() {
        System.out.println("调用父类构造方法...");
        a = 10;
    }
}
```

上述代码定义了一个父类 SuperClass,该类提供一个公共属性 a 和一个不带参数的构造方法,并在构造方法中使用输出语句进行提示和初始化属性 a 的值为 10。

【代码 5-2】 SuperClass.java

```
package com.qst.chapter05;

//子类
public class SubClass extends SuperClass {

    int b;

    public SubClass() {
        System.out.println("调用子类构造方法...");
        b = 20;
    }

    public static void main(String[] args) {
        //实例化一个子类对象
        SubClass obj = new SubClass();
    }
}
```

```
//输出子类中的属性值
System.out.println("a = " + obj.a + ",b = " + obj.b);
}

}
```

上述代码定义了一个子类 SubClass 并继承父类 SuperClass,除了继承父类的属性 a,该类还定义了一个自己的属性 b,并在构造方法中使用输出语句进行提示和初始化属性 b 的值为 20。在 main()方法中直接示例化一个子类对象,并输出属性 a 和 b 的值。运行结果如下所示:

```
调用父类构造方法...
调用子类构造方法...
a = 10,b = 20
```

通过运行结果可以发现:在构造一个子类对象时,会首先调用父类的构造方法进行初始化,而后再调用子类的构造方法进行初始化。与前面提出的理论是一致的。

下述内容通过一个具体的案例来讲解如何使用继承来分析和解决问题。

在一个商品信息管理系统中,需要存储打印机和手机这两种商品的信息。其中,打印机和手机的信息如表 5-1 所示。

表 5-1 打印机和手机的信息

打印机(Printer)		手机(Mobile)	
打印机型号	type	手机型号	type
生产厂商	manufacture	生产厂商	manufacture
价格	price	价格	price
打印色彩	color	屏幕大小	screenSize
打印纸张大小	paperSize	处理器	cpu

现采用面向对象思想分析得到:打印机和手机都具有型号、生产厂商以及价格这三个属性,采用继承的设计思想,可以将打印机和手机的共同属性抽取出来形成父类 Product 类,然后让 Printer 和 Mobile 这两个子类继承父类,并分别在子类中添加差异属性。Product、Printer 和 Mobile 三者之间的继承关系模型如图 5-2 所示。

下述代码分别创建 Product、Printer 和 Mobile 类,并使 Printer 和 Mobile 类继承 Product 类。

【代码 5-3】 Product.java

```
package com.qst.chapter05;

//父类,商品类
public class Product {
    //属性,成员变量
    private String type;
    private String manufacture;
```

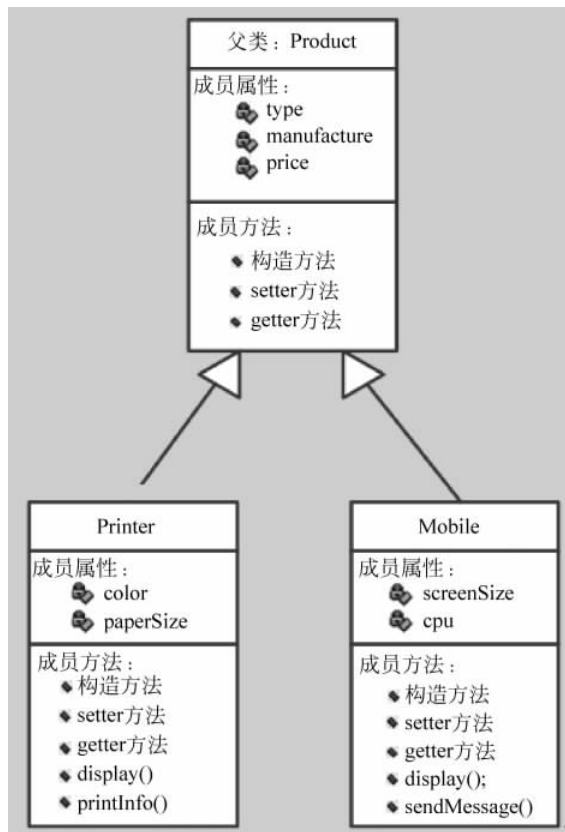


图 5-2 Printer 和 Mobile 继承 Product 类

```

private double price;
//构造方法
public Product() {

}

public Product(String type, String manufacture, double price) {
    this.type = type;
    this.manufacture = manufacture;
    this.price = price;
}
//setter 和 getter 方法
public String getType() {
    return type;
}

public void setType(String type) {
    this.type = type;
}
    
```

```
public String getManufacture() {  
    return manufacture;  
}  
  
public void setManufacture(String manufacture) {  
    this.manufacture = manufacture;  
}  
  
public double getPrice() {  
    return price;  
}  
  
public void setPrice(double price) {  
    this.price = price;  
}  
  
}
```

上述代码创建一个商品父类 Product, 该类提供三个基础属性: type、manufacture 和 price。

【代码 5-4】 Printer.java

```
package com.qst.chapter05;  
  
public class Printer extends Product {  
    //Product 的子类 Printer  
    //属性, 成员变量  
    private String color;  
    private String paperSize;  
    //构造方法  
    public Printer() {  
    }  
  
    public Printer(String type, String manufacture, double price, String color,  
        String paperSize) {  
        //给父类中的基础属性赋值  
        this.setType(type);  
        this.setManufacture(manufacture);  
        this.setPrice(price);  
        //给子类自己的属性赋值  
        this.color = color;  
        this.paperSize = paperSize;  
    }  
    //setter 和 getter 方法  
    public String getColor() {  
        return color;  
    }  
  
    public void setColor(String color) {
```

```

        this.color = color;
    }

    public String getPaperSize() {
        return paperSize;
    }

    public void setPaperSize(String paperSize) {
        this.paperSize = paperSize;
    }

    public void display() { //显示 Printer 的基本信息
        System.out.println("类型: " + this.getType());
        System.out.println("厂商: " + this.getManufacture());
        System.out.println("价格: " + this.getPrice());
        System.out.println("打印色彩: " + this.getColor());
        System.out.println("打印纸张大小: " + this.getPaperSize());
    }
    public void printInfo(String msg){//打印机打印 msg 信息
        System.out.println(this.getManufacture() + "打印机正在打印'" + msg + "'文件");
    }
}

```

上述代码创建一个子类 Printer 继承 Product,该类提供一个带参数的构造方法对属性进行初始化。

【代码 5-5】 Mobile.java

```

package com.qst.chapter05;

public class Mobile extends Product { //Product 的子类 Mobile
    /* 属性,成员变量 */
    private String screenSize;
    private String cpu;
    //构造方法
    public Mobile() {
    }

    public Mobile(String type, String manufacture, double price,
        String screenSize, String cpu) {
        this.setType(type);
        this.setManufacture(manufacture);
        this.setPrice(price);
        this.screenSize = screenSize;
        this.cpu = cpu;
    }
    //setter 和 getter 方法
    public String getScreenSize() {
        return screenSize;
    }
}

```

```

    public void setScreenSize(String screenSize) {
        this.screenSize = screenSize;
    }

    public String getCpu() {
        return cpu;
    }

    public void setCpu(String cpu) {
        this.cpu = cpu;
    }

    public void display() { //显示 Mobile 的基本信息
        System.out.println("类型:" + this.getType());
        System.out.println("厂商:" + this.getManufacture());
        System.out.println("价格:" + this.getPrice());
        System.out.println("屏幕:" + this.getScreenSize());
        System.out.println("cpu:" + this.getCpu());
    }
    public void sendMessage(String msg){ //手机发送信息 msg
        System.out.println(this.getManufacture() + "手机正在发送'" + msg + "'信息");
    }
}

```

上述代码创建一个子类 Mobile 继承 Product, 该类提供一个带参数的构造方法对属性进行初始化。

【代码 5-6】 ClassExtendsDemo.java

```

package com.qst.chapter05;

public class ClassExtendsDemo {

    public static void main(String[] args) {
        //创建一个 Printer 子类实例
        Printer p = new Printer("喷墨打印机", "惠普", 600, "6 色真彩", "A4 纸");
        p.display();
        p.printInfo("青软实训");
        System.out.println("-----");
        //创建一个 Mobile 子类实例
        Mobile m = new Mobile("iPhone 6", "苹果", 5288, "4.7 英寸", "双核");
        m.display();
        m.sendMessage("青软实训");
    }
}

```

上述代码创建一个测试类, 在 main() 方法中分别实例化 Printer 和 Mobile 两个子类对象, 并调用 display() 方法显示信息。运行结果如下所示:

类型:喷墨打印机
厂商:惠普
价格:600.0
打印色彩:6 色真彩
打印纸张大小:A4 纸
惠普打印机正在打印'青软实训'文件

类型:iPhone 6

厂商:苹果

价格:5288.0

屏幕:4.7 英寸

cpu:双核

苹果手机正在发送'青软实训'信息

```
public void printInfo(String msg){//打印机打印 msg 信息
    System.out.println(this.getManufacture() + "打印机正在打印'" + msg + "'文件");
```

通过上述系列代码,能够发现继承减少了代码的冗余,使得维护、更新代码更加容易。例如,临时需要给 Printer 和 Mobile 类都添加“入库时间”,则只需在 Product 类中添加一个“入库时间”属性即可,无须在两个子类中分别添加,通过继承,Printer 和 Mobile 类就会拥有该属性。

5.2.2 多态

Java 引用变量有两个类型:一个是编译时类型,另一个是运行时类型。编译时类型由声明该变量所使用的类型决定;运行时类型则由实际赋给该变量的对象决定。如果编译时类型与运行时类型不一致,则称为“多态”。

多态通常体现在具有继承关系的类之间,一个父类具有多个子类,可以将子类对象直接赋值给一个父类引用变量,无须任何类型转换,如图 5-3 所示。



视频讲解

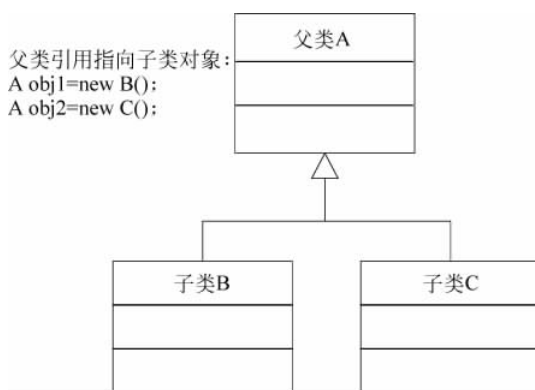


图 5-3 多态

子类重写父类的方法也是 Java 多态性的一种体现。当子类继承父类时,如果父类的方法无法满足子类的需求,子类可以对父类中的方法进行改造,这种情况称为“方法的重写”(override)。

下述代码演示方法的重写。

【代码 5-7】 OverrideDemo.java

```

package com.qst.chapter05;
//父类
class Base {
    public void print() {
        System.out.println("父类...");
    }
}

//子类
class Son extends Base {
    //重写父类的 print()方法
    public void print() {
        System.out.println("子类...");
    }
}

public class OverrideDemo {

    public static void main(String[] args) {
        //多态
        Base obj = new Son();
        obj.print();
    }
}

```

上述代码中,子类 Son 重写了父类 Base 的 print()方法;在 OverrideDemo 类的 main()方法中,使用父类引用构造了 Son 的实例对象,并调用 print()方法输出信息。运行结果如下所示:

```
子类...
```

从执行结果可以看到,由于重写了父类 Base 的 print()方法,所以在调用 Son 实例的 print()方法时使用的是其重写后的方法。

方法重写需要遵循以下几点:

- 方法名以及参数列表必须完全相同。
- 子类方法的返回值类型与父类保持一致,或是父类方法返回值类型的子类。
- 子类方法声明的异常与父类保持一致,或是父类方法声明的异常的子类。
- 父类中的私有方法不能被子类重写,如果在子类中定义了与父类重名的私有方法,则该方法只是子类的一个新方法,与父类中的私有方法无关。
- 子类方法的可访问性必须与父类方法的可访问性保持一致,或是更加公开。例如,父类方法可访问性为 protected,则子类方法可以为 public、protected,但不能为 private 和默认值。
- 不能重写静态方法。

在 Java 中,父类引用可以指向子类的对象,这种机制也符合客观世界的实际情况,子类

对象是父类对象的一个特殊情况,例如一个“学生对象”一定是一个“人对象”,因此父类对象的引用(人对象)可以指向(表示)子类对象(学生对象)。这可以解决在运行期对重写方法的调用,系统会动态地根据当前被引用对象的类型来决定执行哪个方法,而不是由引用变量的类型来决定。因此,如果父类包含一个被子类重写的方法,则通过父类引用不同子类对象类型时,就会执行该重写方法的不同版本,这也是接口规范得以应用的主要原因。

下述代码在原来 OverrideDemo 代码的基础上添加一个 Son1 类,并对 main()方法加以修改。

【代码 5-8】 OverrideDemo.java

```
package com.qst.chapter05;
//父类
class Base {
    public void print() {
        System.out.println("父类...");
    }
}
//子类
class Son extends Base {
    //重写父类的 print()方法
    public void print() {
        System.out.println("子类...");
    }
}

class Son2 extends Base {
    public void print() {
        System.out.println("子类 2...");
    }
}

public class OverrideDemo {

    public static void main(String[] args) {
        //多态
        //obj 指向自己
        Base obj = new Base();
        obj.print();
        //obj 指向子类 Son 对象
        obj = new Son();
        obj.print();
        //obj 指向子类 Son2 对象
        obj = new Son2();
        obj.print();
    }
}
```

运行结果如下所示：

```
父类...
子类...
子类 2...
```

上述代码在调用 print() 方法的过程中, JVM 能够调用正确的实例对象所对应的方法。在编译期, 编译器无从得知 print(), 而在运行时则能够根据对象的类型绑定具体的方法, 这称为动态绑定, 这种动态绑定体现了 Java 的多态应用。

5.2.3 super 关键字

super 关键字代表父类对象, 其主要用途如下:

- 在子类的构造方法中调用父类的构造方法;
- 在子类方法中访问父类的属性或方法。



视频讲解

1. 调用父类构造方法

在 Java 中, 父类和子类属性的初始化过程是各自完成的, 虽然构造方法不能够继承, 但通过使用 super 关键字, 在子类构造方法中可以调用父类的构造方法, 以便完成父类的初始化工作。子类中利用 super 调用父类构造方法时, 该 super 语句必须是子类构造方法的第一条非注释语句。

以 Product、Printer 和 Mobile 类为例, 修改 Printer 和 Mobile 这两个子类的构造方法, 直接使用 super() 初始化父类中的属性值, 代码如下所示。

【代码 5-9】 Printer.java

```
public class Printer extends Product {
    private String color;
    private String paperSize;

    public Printer() {

    }

    public Printer(String type, String manufacture, double price, String color,
        String paperSize) {
//        //给父类中的基础属性赋值
//        this.setType(type);
//        this.setManufacture(manufacture);
//        this.setPrice(price);
//        //调用父类的构造方法初始化父类中的基础属性, 必须是第一条非注释语句
        super(type, manufacture, price);
//        //给子类自己的属性赋值
        this.color = color;
        this.paperSize = paperSize;
    }
    ...
}
```

【代码 5-10】 Product.java

```
package com.qst.chapter05;

public class Mobile extends Product {
    private String screenSize;
    private String cpu;

    public Mobile() {

    }

    public Mobile(String type, String manufacture, double price,
                  String screenSize, String cpu) {
//        this.setType(type);
//        this.setManufacture(manufacture);
//        this.setPrice(price);
//调用父类的构造方法初始化父类中的基础属性,必须是第一条非注释语句
        super(type, manufacture, price);
        this.screenSize = screenSize;
        this.cpu = cpu;
    }
    ...
}
```

在上述代码的 Printer 和 Mobile 两个类的构造方法中,原来都是使用三行赋值语句给父类中的基础属性进行初始化,现都修改成直接调用 super(type, manufacture, price)构造方法进行初始化,简化了代码。

若在子类的构造方法没有明确写明调用父类构造方法,则系统会自动调用父类不带参数的构造方法,即执行 super()方法。现对 Product 类进行修改,去掉 Product 类的默认构造方法,代码如下所示。

【代码 5-11】 Product.java

```
package com.qst.chapter05;

//父类,商品类
public class Product {
    private String type;
    private String manufacture;
    private double price;
//注释去掉默认构造方法,子类 Printer 和 Mobile 将产生错误
//    public Product() {
//    }

    public Product(String type, String manufacture, double price) {
        this.type = type;
        this.manufacture = manufacture;
    }
}
```

```

        this.price = price;
    }
    ...
}

```

修改后的 Product 类没有无参数的默认构造方法,但在子类 Printer 和 Mobile 的默认构造方法中会自动调用 super(),而父类 Product 没有提供不带参数的构造方法,因此会产生错误,编译失败。

 **注意** 为了便于将来程序的扩展,Java 中的类通常都需要提供一个不带参数的默认构造方法。

2. 访问父类的属性和方法

当子类的属性与父类的属性同名时,可以使用“super. 属性名”来引用父类的属性。当子类重写了父类的方法时,可以使用“super. 方法名()”的方式来访问父类的方法。

下述代码修改 Product 类,增加一个 display()方法;然后在 Printer 和 Mobile 两个类中重写 display()方法,并通过“super. display()”调用父类的输出方法。代码分别如下所示。

【代码 5-12】 Product.java

```

package com.qst.chapter05;

//父类,商品类
public class Product {
    private String type;
    private String manufacture;
    private double price;

    public Product() {

    }

    public Product(String type, String manufacture, double price) {
        this.type = type;
        this.manufacture = manufacture;
        this.price = price;
    }

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }
}

```

```

    public String getManufacture() {
        return manufacture;
    }

    public void setManufacture(String manufacture) {
        this.manufacture = manufacture;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public void display() {
        System.out.println("类型: " + this.getType());
        System.out.println("厂商: " + this.getManufacture());
        System.out.println("价格: " + this.getPrice());
    }
}

```

【代码 5-13】 Printer.java

```

package com.qst.chapter05;

public class Printer extends Product {
    private String color;
    private String paperSize;

    public Printer() {

    }

    public Printer(String type, String manufacture, double price, String color,
        String paperSize) {
        // 给父类中的基础属性赋值
        // this.setType(type);
        // this.setManufacture(manufacture);
        // this.setPrice(price);
        //调用父类的构造方法初始化父类中的基础属性
        super(type, manufacture, price);
        //给子类自己的属性赋值
        this.color = color;
        this.paperSize = paperSize;
    }
}

```

```

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public String getPaperSize() {
        return paperSize;
    }

    public void setPaperSize(String paperSize) {
        this.paperSize = paperSize;
    }
    //重写父类的方法
    public void display() {
//        System.out.println("类型: " + this.getType());
//        System.out.println("厂商: " + this.getManufacture());
//        System.out.println("价格: " + this.getPrice());
        //调用父类的 display() 输出前三个属性值
        super.display();
        System.out.println("打印色彩: " + this.getColor());
        System.out.println("打印纸张大小: " + this.getPaperSize());
    }
}

```

【代码 5-14】 Mobile.java

```

package com.qst.chapter05;

public class Mobile extends Product {
    private String screenSize;
    private String cpu;

    public Mobile() {

    }

    public Mobile(String type, String manufacture, double price,
        String screenSize, String cpu) {
//        this.setType(type);
//        this.setManufacture(manufacture);
//        this.setPrice(price);
        //调用父类的构造方法初始化父类中的基础属性
        super(type, manufacture, price);
        this.screenSize = screenSize;
        this.cpu = cpu;
    }
}

```

```

    public String getScreenSize() {
        return screenSize;
    }

    public void setScreenSize(String screenSize) {
        this.screenSize = screenSize;
    }

    public String getCpu() {
        return cpu;
    }

    public void setCpu(String cpu) {
        this.cpu = cpu;
    }
    //重写父类的方法
    public void display() {
//        System.out.println("类型: " + this.getType());
//        System.out.println("厂商: " + this.getManufacture());
//        System.out.println("价格: " + this.getPrice());
        //调用父类的 display()输出前三个属性值
        super.display();
        System.out.println("屏幕: " + this.getScreenSize());
        System.out.println("cpu: " + this.getCpu());
    }
}

```

编写一个测试类对上述代码进行测试,代码如下所示:

【代码 5-15】 DynamicDemo.java

```

package com.qst.chapter05;

//多态测试
public class DynamicDemo {

    public static void main(String[] args) {
        //创建一个商品对象 p
        Product p = new Product("商品 A", "厂家 A", 100);
        p.display();
        System.out.println("-----");
        //p 指向一个 Printer 子类实例
        p = new Printer("喷墨打印机", "惠普", 600, "6 色真彩", "A4 纸");
        p.display();
        System.out.println("-----");
        //p 指向一个 Mobile 子类实例
        p = new Mobile("iPhone 6", "苹果", 5288, "4.7 英寸", "双核");
        p.display();
    }
}

```

运行结果如下所示：

```

类型：商品 A
厂商：厂家 A
价格：100.0
-----
类型：喷墨打印机
厂商：惠普
价格：600.0
打印色彩：6 色真彩
打印纸张大小：A4 纸
-----
类型：iPhone 6
厂商：苹果
价格：5288.0
屏幕：4.7 英寸
cpu：双核

```

从结果可以看到，在 Printer 和 Mobile 这两个子类的 display() 方法中，通过 super.display() 调用了父类的 display() 方法，从而输出了 type、manufacture 和 price 这三个属性的值。



注意 在子类中可以添加与父类中属性重名的属性，但这不是一种良好的设计。

5.2.4 final 关键字



视频讲解

final 关键字表示“不可改变的、最终的”的意思，用于修饰变量、方法和类。

- 当 final 关键字修饰变量时，表示该变量是不可改变的量，即常量；
- 当 final 关键字修饰方法时，表示该方法不可被子类重写，即最终方法；
- 当 final 关键字修饰类时，表示该类不可被子类继承，即最终类。

本书第 2 章已经使用过 final 关键字定义常量，此处不再重复，下面分别介绍使用 final 关键字定义最终方法和最终类。

1. 最终方法

使用 final 修饰的方法不能被子类重写。如果某些方法完成关键性的、基础性的功能，不需要或不允许被子类改变，则可以将这些方法声明为 final，示例如下：

【示例】 最终方法不能被重写

```

public class Base {
    //使用 final 定义最终方法
    public final void method(){
    }
}

```

```

    }
    class Son extends Base {
        //错误!无法重写父类的 final 方法
        public void method(){
        }
    }
}

```

2. 最终类

使用 final 修饰的类不能被继承,示例如下:

【示例】 最终类不能被继承

```

public final class Base {
    ...                //省略
}
class Son extends Base {    //错误!无法继承 final 类
}

```

一个 final 类中的所有方法都被默认为 final,因此 final 类中的方法不必显式地声明为 final。其实,Java 基础类库中的 String、Integer 等类都是 final 类,都无法扩展子类,例如下列代码是错误的:

```

public class MyClass extends Integer {    //错误!Integer 无法被继承
}

```

5.3 其他关系



视频讲解

除了继承和实现外,依赖、关联、聚合、组成也是类之间的重要关系类型。

5.3.1 依赖关系

依赖关系是最常见的一种类间关系,如果在一个类的方法中操作另外一个类的对象,则称其依赖于第二个类。例如,方法的参数是某种对象类型,或者方法中有某种对象类型的局部变量,或者方法中调用了另一个类的静态方法,这些都是依赖关系。

以 Person(人)和 Car(车)这两个类为例,其依赖关系如图 5-4 所示。



图 5-4 Person 和 Car 的依赖关系

下述代码体现 Person 和 Car 这两个类之间的依赖关系。

【代码 5-16】 DependentDemo.java

```
package com.qst.chapter05;

//依赖关系,Person 依赖 Car
class Car {
    void run(String city) {
        System.out.println("汽车开到" + city);
    }
}

class Person {
    //Car 类的对象作为方法的参数
    void travel(Car car) {
        car.run("青岛");
    }
}

public class DependentDemo {

    public static void main(String[] args) {

        Car car = new Car();
        Person p = new Person();
        p.travel(car);
    }

}
```

上述代码中,Person 类的 travel()方法需要 Car 类的对象作为参数,并且在该方法中调用了 Car 的方法,因此 Person 类依赖于 Car 类。依赖关系通常是单向的,Person 依赖 Car,但 Car 并不依赖 Person。该案例实际意义想体现一个人旅游依赖于一辆车,人和车的依赖关系可以理解为:人旅游需要一辆车,并不关心这辆车是如何得到的,只要保证旅游时(即调用 travel()方法时)有一辆车即可,旅游完毕后,这辆车的去向人也不再关心。

运行结果如下所示:

```
汽车开到青岛
```

5.3.2 关联关系

关联关系比依赖关系更紧密,通常体现为一个类中使用另一个类的对象作为该类的成员变量。继续以人驾车旅游为例,改为如图 5-5 所示的关联关系。

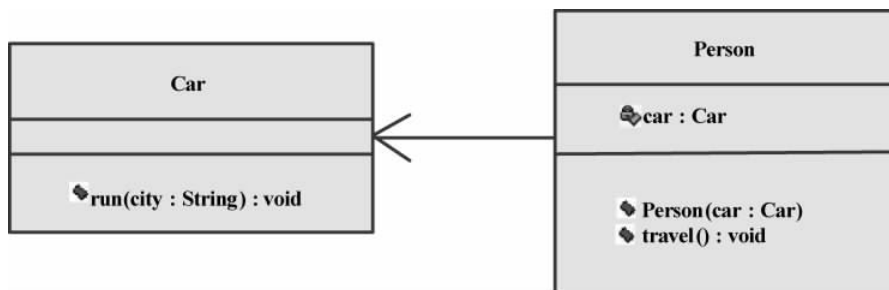


图 5-5 Person 和 Car 的关联关系

【代码 5-17】 AssociationDemo.java

```

package com.qst.chapter05.association;
//关联关系,Person 关联 Car
class Car {
    void run(String city) {
        System.out.println("汽车开到" + city);
    }
}

class Person {
    //Car 对象作为成员变量
    Car car;

    Person(Car car) {
        this.car = car;
    }

    void travel() {
        car.run("青岛");
    }
}

public class AssociationDemo {

    public static void main(String[] args) {
        Car car = new Car();
        Person p = new Person(car);
        p.travel();
    }
}
    
```

上述代码中,Person 类中存在 Car 类型的成员变量,并在构造方法和 travel()方法中都使用该成员变量,因此 Person 和 Car 具有关联关系。人和车的关联关系可以理解为:人拥有一辆车,旅游时可以用这辆车,做别的事情时也可以用。但是关联关系并不要求是独占的,以人车关联为例,即车也可以被别的人拥有。

5.3.3 聚合关系

聚合关系是关联关系的一种特例,体现的是整体与部分的关系,通常表现为一个类(整体)由多个其他类的对象(部分)作为该类的成员变量,此时整体与部分之间是可以分离的,整体和部分都可以具有各自的生命周期。例如,一个部门由多个员工组成,部门和员工是整体与部分的关系,即聚合关系。

部门和员工的聚合关系如图 5-6 所示。

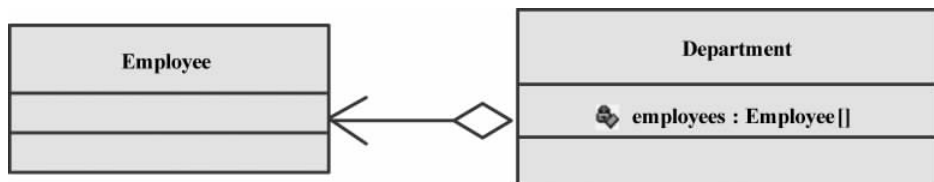


图 5-6 部门和员工的聚合关系

【代码 5-18】 AggregationDemo.java

```

package com.qst.chapter05.aggregation;

//聚合关系,Department 由 Employee 聚合而成
class Employee {
    String name;

    Employee(String name) {
        this.name = name;
    }
}

class Department {
    Employee[] emps;

    Department(Employee[] emps) {
        this.emps = emps;
    }

    void show() {
        //循环遍历
        for (Employee emp : emps) {
            System.out.println(emp.name);
        }
    }
}

public class AggregationDemo {

    public static void main(String[] args) {

        Employee[] emps = { new Employee("张三"),
                            new Employee("李四"),
  
```

```

        new Employee("王五"),
        new Employee("马六"));

    Department dept = new Department(emps);
    dept.show();

}

}

```

上述代码中,部门类 Department 中的 Employee 数组代表此部门的员工。部门和员工的聚合关系可以理解为:部门由员工组成,同一个员工也可能属于多个部门,并且部门解散后,员工依然是存在的,并不会随之消亡。

运行结果如下所示:

```

张三
李四
王五
马六

```

5.3.4 组成关系

组成关系是比聚合关系要求更高的一种关联关系,体现的也是整体与部分的关系,但组成关系中的整体与部分是不可分离的,整体的生命周期结束后,部分的生命周期也随之结束。例如,汽车是由发动机、底盘、车身和电路设备等组成,是整体与部分的关系,如果汽车消亡后,这些设备也将不复存在,因此属于一种组成关系。

汽车和设备之间的组成关系如图 5-7 所示。

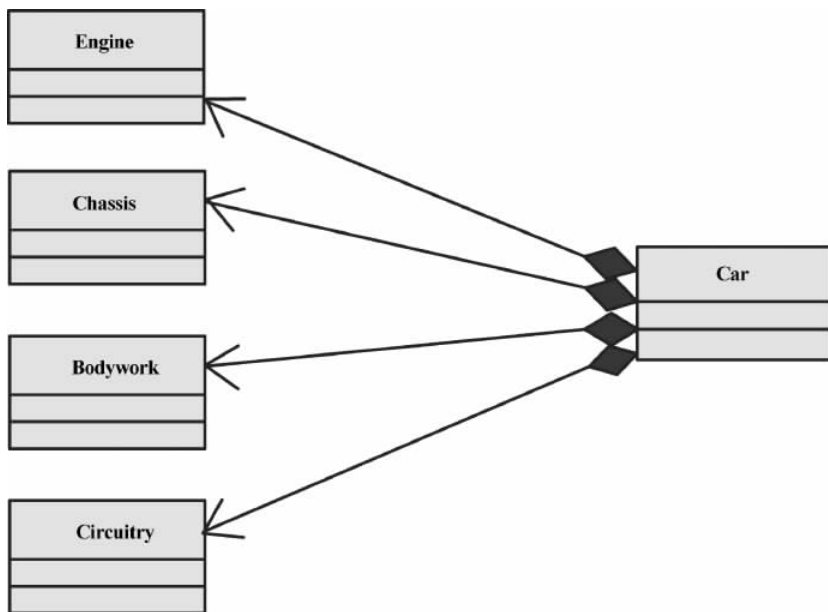


图 5-7 汽车和设备的组成关系

【代码 5-19】 CompositionDemo.java

```
package com.qst.chapter05.composition;

//组成关系,汽车由各设备组成
//发动机
class Engine {
}

//底盘
class Chassis {
}

//车身
class Bodywork {
}

//电路设备
class Circuitry {
}

//汽车
class Car {
    Engine engine;
    Chassis chassis;
    Bodywork bodywork;
    Circuitry circuitry;

    Car(Engine engine, Chassis chassis, Bodywork bodywork,
        Circuitry circuitry) {
        this.engine = engine;
        this.chassis = chassis;
        this.bodywork = bodywork;
        this.circuitry = circuitry;
    }
}

public class CompositionDemo {

    public static void main(String[] args) {
        Engine engine = new Engine();
        Chassis chassis = new Chassis();
        Bodywork bodywork = new Bodywork();
        Circuitry circuitry = new Circuitry();
        Car car = new Car(engine, chassis, bodywork, circuitry);
    }
}
```

上述代码中定义了五个类,其中 Engine、Chassis、Bodywork 和 Circuitry 都是 Car 的成员变量,它们之间构成了一种组成关系。

5.4 内部类



视频讲解

Java 允许在一个类的类体之内再定义一个类,该情况下外面的类称为“外部类”,里面的类称为“内部类”。内部类是外部类的一个成员,并且依附于外部类而存在。

引入内部类的原因主要有以下几个方面:

- 内部类能够隐藏起来,不为同一包的其他类访问;
- 内部类可以访问其所在的外部类的所有属性;
- 在回调方法处理中,匿名内部类尤为便捷,特别是事件处理时经常使用。

Java 内部类主要有成员内部类、局部内部类、静态内部类、匿名内部类四种。

5.4.1 成员内部类

成员内部类的定义结构很简单,就是在“外部类”的内部定义一个类。

下述代码用于演示成员内部类的定义及使用。

【代码 5-20】 Cow.java

```
package com.qst.chapter05.innerclass;

public class Cow {
    private double weight;

    //外部类的两个重载的构造器
    public Cow() {
    }

    public Cow(double weight) {
        this.weight = weight;
    }

    //定义一个成员内部类
    private class CowLeg {
        //成员内部类的两个实例变量
        private double length;
        private String color;

        //成员内部类的两个重载的构造方法
        public CowLeg() {
        }

        public CowLeg(double length, String color) {
            this.length = length;
            this.color = color;
        }
    }
}
```

```

//下面省略 length,color 的 setter 和 getter 方法
...

//成员内部类的方法
public void info() {
    System.out.println("当前牛腿颜色是: " + color + ", 高: " + length);
    //直接访问外部类的 private 修饰的成员变量
    System.out.println("本牛腿所在奶牛重: " + weight);
}

public void test() {
    CowLeg cl = new CowLeg(1.12, "黑白相间");
    cl.info();
}

public static void main(String[] args) {
    Cow cow = new Cow(378.9);
    cow.test();
}
}

```

上述代码中,在 Cow 类中定义了一个 CowLeg 成员内部类,并在 CowLeg 类的方法中直接访问 Cow 的私有成员。运行结果如下所示:

```

当前牛腿颜色是: 黑白相间,高: 1.12
本牛腿所在奶牛重: 378.9

```

上述代码编译后,会生成两个 class 文件:一个是外部类的 class 文件 Cow.class,另一个是内部类的 class 文件 Cow\$CowLeg.class。内部类的 class 文件形式都是“外部类名\$内部类名.class”。

内部类可以很方便地访问外部类的私有成员属性,并且外部类可以通过内部类对象来访问内部类的私有成员属性。在外部类方法中可以访问成员内部类的成员,同时也可以在外部类的外部直接实例化内部类的对象,内部类对象实例化语法格式如下:

【语法】

```

外部类.内部类 对象名 = new 外部类对象.new 内部类构造方法;

```

下述代码用于演示内部类对象实例化。

【代码 5-21】 InnerDemo.java

```

package com.qst.chapter05.innerclass;
class Outer { //定义外部类
    private String outMSG = "你好,外部类!";
    class Inner { //定义成员内部类
        private String innMSG = "你好,内部类!";
    }
}

```

```

        public void printInner() {
            System.out.println(this.innMSG);
            //内部类方法访问外部类私有成员属性
            System.out.println(Outer.this.outMSG);
        }
    }

    public void printOuter() {
        Inner in = new Inner();
        System.out.println(in.innMSG);          //①外部类方法访问内部类对象的私有成员属性
    }
}

public class InnerDemo {
    public static void main(String[] args) {
        Outer out = new Outer();
        Outer.Inner inn = out.new Inner();    //②内部类对象实例化
        inn.printInner();
    }
}

```

上述代码中,定义了外部类 Outer 和成员内部类 Inner,并且私有化了它们的成员属性,在代码①处访问了内部类的私有成员属性,在代码②处实现了内部类的对象实例化。需要强调的是,外部类的访问控制修饰符只能是 public 或者默认,但成员内部类还可以利用 protected 和 private 进行修饰,如果利用 private 修饰了内部类,则不能在外部实例化内部类对象。

执行结果如下:

```

你好,内部类!
你好,外部类!

```

5.4.2 局部内部类

在方法中定义的内部类称为局部内部类。与局部变量类似,局部内部类不能用 public 或 private 访问修饰符进行声明。它的作用域被限定在声明该类的方法块中。局部内部类的优势在于:它可以对外界完全隐藏起来,除了所在的方法之外,对其他方法而言是不透明的。此外与其他内部类比较,局部内部类不仅可以访问包含它的外部类的成员,还可以访问局部变量,但这些局部变量必须被声明为 final。

下述代码演示一个局部内部类的定义及使用。

【代码 5-22】 LocalInnerClass.java

```

package com.qst.chapter05.innerclass;

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

```

```

//定义局部内部类
class InnerBase {
    int a;
}
//定义局部内部类的子类
class InnerSub extends InnerBase {
    int b;
}
//创建局部内部类的对象
InnerSub is = new InnerSub();
is.a = 5;
is.b = 8;
System.out.println("InnerSub 对象的 a 和 b 实例变量是: " + is.a + "," + is.b);
}
}

```

上述代码在 main() 方法中定义了两个局部内部类 InnerBase 和 InnerSub, 编译后会生成三个 class 文件: LocalInnerClass.class、LocalInnerClass \$1InnerBase.class、LocalInnerClass \$1InnerSub.class。局部内部类的 class 文件形式都是“外部类名, \$N 内部类名.class”, 需要注意, \$ 符号后面多了一个数字, 这是因为有可能有两个以上的同名局部类(处于不同方法中), 所以使用一个数字进行区分。

运行结果如下所示:

```
InnerSub 对象的 a 和 b 实例变量是: 5,8
```

 **注意** 在实际开发中很少使用局部内部类, 这是因为局部内部类的作用域很小, 只能在当前方法中使用。

5.4.3 静态内部类

如果使用 static 关键字修饰一个内部类, 则该内部类称为“静态内部类”。静态内部类属于外部类的本身, 而不属于外部类的某个对象, 即 static 关键字将内部类变成与外部类相关, 而不是与外部类的实例相关。

下述代码用于演示静态内部类的定义及使用。

【代码 5-23】 StaticInnerClassDemo.java

```

package com.qst.chapter05.innerclass;

public class StaticInnerClassDemo {
    private int prop1 = 5;
    private static int prop2 = 9;

    static class StaticInnerClass {
        //静态内部类里可以包含静态成员
        private static int age;

        public void accessOuterProp() {

```

```
//下面代码出现错误:
//静态内部类无法访问外部类的实例变量
System.out.println(prop1);
//下面代码正常
System.out.println(prop2);
    }
}
}
```

静态内部类可以包含静态成员。根据静态成员不能访问非静态成员的规则,静态内部类不能访问外部类的实例成员,只能访问外部类的静态成员,即类成员。



注意 静态内部类可以用 public,protected,private 修饰,只能访问外部类的静态成员。

5.4.4 匿名内部类

匿名内部类就是没有名字的内部类。匿名内部类适合只需要使用一次的类,当创建一个匿名类时会立即创建该类的一个实例,该类的定义会立即消失,不能重复使用。

【示例】 匿名内部类

```
public class AnonymousClassDemo {

    public static void main(String[] args) {

        System.out.println(new Object() {
            public String toString() {
                return "匿名类对象";
            }
        });
    }
}
```

上述代码的含义是:创建一个匿名类的对象,该匿名类重写 Object 父类的 toString()方法。在使用匿名内部类时,要注意遵循以下几个原则:

- 匿名内部类不能有构造方法;
- 匿名内部类不能定义任何静态成员、方法和类,但非静态的方法、属性、内部类是可以定义的;
- 只能创建匿名内部类的一个实例;
- 一个匿名内部类一定跟在 new 的后面,创建其实现的接口或父类的对象。

5.5 单例模式

在实际应用中,可能需要整个系统中生成某种类型的对象有且只有一个,此时可以使用单例模式(Singleton 设计模式)来实现。如 Windows 操作系统中的 Recycle Bin(回收站)就是很典型的单例模式,在整个操作系统



视频讲解

运行过程中,回收站一直维护着仅有的一个实例;操作系统的文件系统,也是应用单例模式实现的具体例子,一个操作系统只能有一个文件系统。

一般地,单例模式有如下实现方式:

- 构造方法私有;
- 用一个私有的静态变量引用实例;
- 提供一个公有的静态方法获取实例。

【代码 5-24】 SingletonDemo.java

```
package com.qst.chapter05;

//单例模式
class Singleton {
    private static Singleton instance = null;
    private Singleton(){}
    public static Singleton getInstance() {
        //在第一次使用时生成实例,提高了效率
        if (instance == null)
            instance = new Singleton();
        return instance;
    }
}

public class SingletonDemo {

    public static void main(String[] args) {
        Singleton s1 = Singleton.getInstance();
        Singleton s2 = Singleton.getInstance();
        if (s1 == s2) {
            System.out.println("s1 和 s2 是同一个对象");
        }
    }
}
```

运行结果如下所示:

```
s1 和 s2 是同一个对象
```

上述代码在 main 方法中声明了两个变量 s1 和 s2,利用 getInstance 方法获取对象,通过测试比较,每次返回的都是同一个对象,这样不必每次都创建新对象,从而提高了效率。

5.6 贯穿任务实现

5.6.1 实现【任务 5-1】

下述代码实现 Q-DMS 贯穿项目中的【任务 5-1】编写基础信息实体类。

【任务 5-1】 DataBase.java

```

/**
 * @公司    青软实训 QST
 * @作者    zhaokl
 */
package com.qst.dms.entity;

import java.util.Date;

//数据基础类

public class DataBase {
    //ID 标识
    private int id;
    //时间
    private Date time;
    //地点
    private String address;
    //状态
    private int type;
    //状态常量
    public static final int GATHER = 1;           //"采集"
    public static final int MATHCH = 2;           //"匹配";
    public static final int RECORD = 3;           //"记录";
    public static final int SEND = 4;             //"发送";
    public static final int RECIVE = 5;           //"接收";
    public static final int WRITE = 6;            //"归档";
    public static final int SAVE = 7;             //"保存";

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public Date getTime() {
        return time;
    }

    public void setTime(Date time) {
        this.time = time;
    }
}

```

```

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public int getType() {
        return type;
    }

    public void setType(int type) {
        this.type = type;
    }

    public DataBase() {
    }

    public DataBase(int id, Date time, String address, int type) {
        this.id = id;
        this.time = time;
        this.address = address;
        this.type = type;
    }

    public String toString() {
        return id + "," + time + "," + address + "," + type;
    }
}

```

上述代码创建了一个基础信息实体类,该类将 Q-DMS 中数据采集的共同特性抽取出来组成一个基类,并提供七个静态常量标志数据状态,其他信息类都继承该基类。

5.6.2 实现【任务 5-2】

下述代码实现 Q-DMS 贯穿项目中的【任务 5-2】使用继承重构日志、物流实体类,并测试运行。

【任务 5-2】 LogRec.java

```

/**
 * @公司    青软实训 QST
 * @作者    zhaokl
 * /

```

```
package com.qst.dms.entity;

import java.util.Date;

//用户登录日志记录
public class LogRec extends DataBase {
    /**
     * 登录用户名
     */
    private String user;
    /**
     * 登录用户主机 IP 地址
     */
    private String ip;
    /**
     * 登录状态：登录、登出
     */
    private int logType;
    /**
     * 登录常量 LOG_IN、登出常量 LOG_OUT
     */
    public static final int LOG_IN = 1;
    public static final int LOG_OUT = 0;

    public String getUser() {
        return user;
    }

    public void setUser(String user) {
        this.user = user;
    }

    public String getIp() {
        return ip;
    }

    public void setIp(String ip) {
        this.ip = ip;
    }

    public int getLogType() {
        return logType;
    }

    public void setLogType(int logType) {
        this.logType = logType;
    }

    public LogRec() {
```

```

    }

    public LogRec(int id, Date time, String address, int type, String user,
                  String ip, int logType) {
        super(id, time, address, type);
        this.user = user;
        this.ip = ip;
        this.logType = logType;
    }

    public String toString() {
        return this.getId() + "," + this.getTime() + "," + this.getAddress() + ","
            + this.getType() + "," + user + "," + ip + "," + logType;
    }
}

```

上述代码重构 LogRec 日志信息类,该类继承 DataBase 类,并在 DataBase 类的基础上增加自己的特有属性。

【任务 5-2】 Transport.java

```

/**
 * @公司    青软实训 QST
 * @作者 zhaokl
 */
package com.qst.dms.entity;

import java.util.Date;

//货运物流信息
public class Transport extends DataBase {
    /**
     * 经手人
     */
    private String handler;
    /**
     * 收货人
     */
    private String reciver;
    /**
     * 物流状态
     */
    private int transportType;
    /**
     * 物流状态常量:发货中, 送货中, 已签收
     */
    public static final int SENDDING = 1;           //发货中
    public static final int TRANSPORTING = 2;       //送货中
    public static final int RECIEVED = 3;           //已签收
}

```

```

    public String getHandler() {
        return handler;
    }

    public void setHandler(String handler) {
        this.handler = handler;
    }

    public String getReciver() {
        return reciver;
    }

    public void setReciver(String reciver) {
        this.reciver = reciver;
    }

    public int getTransportType() {
        return transportType;
    }

    public void setTransportType(int transportType) {
        this.transportType = transportType;
    }

    public Transport() {

    }

    public Transport(int id, Date time, String address, int type,
        String handler, String reciver, int transportType) {
        super(id, time, address, type);
        this.handler = handler;
        this.reciver = reciver;
        this.transportType = transportType;
    }

    public String toString() {
        return this.getId() + "," + this.getTime() + "," + this.getAddress()
            + "," + this.getType() + "," + handler + "," + transportType;
    }
}

```

上述代码重构 Transport 物流信息类,该类继承 DataBase 类,并在 DataBase 类的基础上增加自己的特有属性。

【任务 5-2】 EntityDataDemo.java

```

package com.qst.dms.dos;

import com.qst.dms.entity.LogRec;

```

```

import com.qst.dms.entity.Transport;
import com.qst.dms.service.LogRecService;
import com.qst.dms.service.TransportService;

public class EntityDataDemo {
    public static void main(String[] args) {
        //创建一个日志业务类
        LogRecService logService = new LogRecService();
        //创建一个日志对象数组,用于存放采集的三个日志信息
        LogRec[] logs = new LogRec[3];
        for (int i = 0; i < logs.length; i++) {
            System.out.println("第" + (i + 1) + "个日志数据采集:");
            logs[i] = logService.inputLog();
        }
        //输出采集的日志信息
        logService.showLog(logs);

        //创建一个物流业务类
        TransportService tranService = new TransportService();
        //创建一个物流对象数组,用于存放采集的两个物流信息
        Transport[] transports = new Transport[2];
        for (int i = 0; i < transports.length; i++) {
            System.out.println("第" + (i + 1) + "个物流数据采集:");
            transports[i] = tranService.inputTransport();
        }
        //输出采集的物流信息
        tranService.showTransport(transports);
    }
}

```

上述代码对重构后的 LogRec 和 Transport 进行测试运行,使用对应的业务类分别循环采集了三个日志信息和两个物流信息,并将采集的数据进行显示。

运行结果如下所示:

```

第 1 个日志数据采集:
请输入 ID 标识:
1001
请输入地址:
青岛
请输入 登录用户名:
zhaokl
请输入 主机 IP:
192.168.1.1
请输入登录状态:1 是登录,0 是登出
1
第 2 个日志数据采集:
请输入 ID 标识:
1002
请输入地址:

```

```

北京
请输入 登录用户名:
zhangsan
请输入 主机 IP:
192.168.1.2
请输入登录状态:1 是登录,0 是登出
0
第 3 个日志数据采集:
请输入 ID 标识:
1003
请输入地址:
上海
请输入 登录用户名:
zhaokl
请输入 主机 IP:
192.168.1.1
请输入登录状态:1 是登录,0 是登出
0
1001,Tue Nov 04 09:41:11 CST 2014,青岛,1,zhaokl,192.168.1.1,1
1002,Tue Nov 04 09:41:38 CST 2014,北京,1,zhangsan,192.168.1.2,0
1003,Tue Nov 04 09:42:06 CST 2014,上海,1,zhaokl,192.168.1.1,0
第 1 个物流数据采集:
请输入 ID 标识:
2001
请输入地址:
济南
请输入货物经手人:
wangwu
请输入 收货人:
zhaokl
请输入物流状态: 1 发货中,2 送货中,3 已签收
1
第 2 个物流数据采集:
请输入 ID 标识:
2001
请输入地址:
烟台
请输入货物经手人:
maliu
请输入 收货人:
zhaokl
请输入物流状态: 1 发货中,2 送货中,3 已签收
2
2001,Tue Nov 04 09:42:34 CST 2014,济南,1,wangwu,1
2001,Tue Nov 04 09:43:20 CST 2014,烟台,1,maliu,2

```

5.6.3 实现【任务 5-3】

下述代码实现 Q-DMS 贯穿项目中的【任务 5-3】编写日志数据匹配类,对日志实体类数

据进行匹配。

【任务 5-3】 MatchedLogRec.java

```
/**
 * @公司    青软实训 QST
 * @作者    zhaokl
 */
package com.qst.dms.entity;

import java.util.Date;

//匹配日志记录,"登录登出对" 类型

public class MatchedLogRec {

    private LogRec login;
    private LogRec logout;

    //user 用户登录名
    public String getUser() {
        return login.getUser();
    }

    //登入时刻
    public Date getLogInTime() {
        return login.getTime();
    }

    //登出时刻
    public Date getLogoutTime() {
        return logout.getTime();
    }

    //登入记录
    public LogRec getLogin() {
        return login;
    }

    //登出记录
    public LogRec getLogout() {
        return logout;
    }

    public MatchedLogRec() {
    }

    public MatchedLogRec(LogRec login, LogRec logout) {
        if (login.getLogType() != LogRec.LOG_IN) {
```

```

        throw new RuntimeException("不是登录记录!");
    }
    if (logout.getLogType() != LogRec.LOG_OUT) {
        throw new RuntimeException("不是登出记录");
    }
    if (!login.getUser().equals(logout.getUser())) {
        throw new RuntimeException("登录登出必须是同一个用户!");
    }
    if (!login.getIp().equals(logout.getIp())) {
        throw new RuntimeException("登录登出必须是同一个 IP 地址!");
    }
    this.login = login;
    this.logout = logout;
}

public String toString() {
    return login.toString() + " | " + logout.toString();
}
}

```

上述代码定义一个 MatchedLogRec 日志匹配类,该类中有两个 LogRec 类型的成员变量,分别存放匹配成功的登录日志和登出日志信息。

修改 LogRecService 日志业务类,在该类中增加显示匹配日志信息的 showMatchLog()方法,代码如下所示。

【任务 5-3】 LogRecService.java

```

package com.qst.dms.service;

import java.util.Date;
import java.util.Scanner;

import com.qst.dms.entity.DataBase;
import com.qst.dms.entity.LogRec;
import com.qst.dms.entity.MatchedLogRec;

//日志业务类
public class LogRecService {
    //日志数据采集
    public LogRec inputLog() {
        //建立一个从键盘接收数据的扫描器
        Scanner scanner = new Scanner(System.in);
        //提示用户输入 ID 标识
        System.out.println("请输入 ID 标识: ");
        //接收键盘输入的整数
        int id = scanner.nextInt();
        //获取当前系统时间
        Date nowDate = new Date();
    }
}

```

```

//提示用户输入地址
System.out.println("请输入地址:");
//接收键盘输入的字符串信息
String address = scanner.next();
//数据状态是"采集"
int type = DataBase.GATHER;

//提示用户输入登录用户名
System.out.println("请输入登录用户名:");
//接收键盘输入的字符串信息
String user = scanner.next();
//提示用户输入主机 IP
System.out.println("请输入主机 IP:");
//接收键盘输入的字符串信息
String ip = scanner.next();
//提示用户输入登录状态、登出状态
System.out.println("请输入登录状态:1 是登录,0 是登出");
int logType = scanner.nextInt();
//创建日志对象
LogRec log = new LogRec(id, nowDate, address, type, user, ip, logType);
//返回日志对象
return log;
}

//日志信息输出
public void showLog(LogRec... logRecs) {
    for (LogRec e : logRecs) {
        if (e != null) {
            System.out.println(e.toString());
        }
    }
}

//匹配日志信息输出
public void showMatchLog(MatchedLogRec... matchLogs) {
    for (MatchedLogRec e : matchLogs) {
        if (e != null) {
            System.out.println(e.toString());
        }
    }
}
}

```

5.6.4 实现【任务 5-4】

下述代码实现 Q-DMS 贯穿项目中的【任务 5-4】编写物流数据匹配类,对物流实体类数

据进行匹配。

【任务 5-4】 MatchedTransport.java

```
/**
 * @公司 青软实训 QST
 * @作者 zhaokl
 */
package com.qst.dms.entity;

public class MatchedTransport {
    private Transport send;
    private Transport trans;
    private Transport receive;

    public Transport getSend() {
        return send;
    }

    public void setSend(Transport send) {
        this.send = send;
    }

    public Transport getTrans() {
        return trans;
    }

    public void setTrans(Transport trans) {
        this.trans = trans;
    }

    public Transport getReceive() {
        return receive;
    }

    public void setReceive(Transport receive) {
        this.receive = receive;
    }

    public MatchedTransport() {
    }

    public MatchedTransport(Transport send, Transport trans, Transport receive) {
        if (send.getTransportType() != Transport.SENDDING) {
            throw new RuntimeException("不是发货记录!");
        }
        if (trans.getTransportType() != Transport.TRANSPORTING) {
            throw new RuntimeException("不是送货记录!");
        }
        if (receive.getTransportType() != Transport.RECIEVED) {
```

```

        throw new RuntimeException("不是签收记录!");
    }
    this.send = send;
    this.trans = trans;
    this.receive = receive;
}

public String toString() {
    //TODO Auto-generated method stub
    return send.toString() + "|" + trans.toString() + "|" + receive;
}
}

```

上述代码定义一个 MatchedTransport 物流匹配类,该类中有三个 Transport 类型的成员变量,分别存放匹配成功的发货、送货和签收这三条物流记录信息。

修改 TransportService 物流业务类,在该类中增加显示匹配物流信息的 showMatchTransport() 方法,代码如下所示。

【任务 5-4】 TransportService.java

```

package com.qst.dms.service;

import java.util.Date;
import java.util.Scanner;

import com.qst.dms.entity.DataBase;
import com.qst.dms.entity.MatchedTransport;
import com.qst.dms.entity.Transport;

public class TransportService {
    //物流数据采集
    public Transport inputTransport() {
        //建立一个从键盘接收数据的扫描器
        Scanner scanner = new Scanner(System.in);
        //提示用户输入 ID 标识
        System.out.println("请输入 ID 标识:");
        //接收键盘输入的整数
        int id = scanner.nextInt();
        //获取当前系统时间
        Date nowDate = new Date();
        //提示用户输入地址
        System.out.println("请输入地址:");
        //接收键盘输入的字符串信息
        String address = scanner.next();
        //数据状态是"采集"
        int type = DataBase.GATHER;

        //提示用户输入登录用户名
        System.out.println("请输入货物经手人:");
        //接收键盘输入的字符串信息
        String handler = scanner.next();
        //提示用户输入主机 IP
    }
}

```

```

        System.out.println("请输入收货人:");
        //接收键盘输入的字符串信息
        String reciver = scanner.next();
        //提示用于输入物流状态
        System.out.println("请输入物流状态: 1 发货中, 2 送货中, 3 已签收");
        //接收物流状态
        int transportType = scanner.nextInt();
        //创建物流信息对象
        Transport trans = new Transport(id, nowDate, address, type, handler,
            reciver, transportType);
        //返回物流对象
        return trans;
    }

    //物流信息输出
    public void showTransport(Transport... transports) {
        for (Transport e : transports) {
            if (e != null) {
                System.out.println(e.toString());
            }
        }
    }

    //匹配的物流信息输出
    public void showMatchTransport(MatchedTransport... matchTrans) {
        for (MatchedTransport e : matchTrans) {
            if (e != null) {
                System.out.println(e.toString());
            }
        }
    }
}

```

本章总结

小结

- 类与类之间存在六种关系：继承、实现、依赖、关联、聚合和组成
- 一个类可以继承另外一个类，并在此基础上添加自己的特有功能
- 在一个类的方法中操作另外一个类的对象，这种情况称为第一个类依赖于第二个类
- 在一个类中使用另外一个类的对象作为该类的成员变量，这种情况称为关联关系
- 聚合关系是关联关系的一种特例，体现的是整体与部分的关系，即 has-a 的关系
- 组成关系也是关联关系的一种特例，与聚合关系一样也是体现整体与部分的关系，但组成关系中的整体与部分是不可分离的，即 contains-a 的关系
- super 关键字代表父类对象
- final 关键字表示“不可改变的、最终的”的意思，用于修饰变量、方法和类
- Java 内部类主要有成员内部类、局部内部类、静态内部类、匿名内部类四种

- 单例模式(Singleton 设计模式)是系统中生成某种类型的对象有且只有一个

Q&A

1. 问题：成员内部类与静态内部类的区别。

回答：成员内部类对象必须寄生在外部类对象里，而静态内部类是属于类的，不依赖于外部类的对象。

2. 问题：Java 中的多态体现在哪几个方面？

回答：多态通常体现在具有继承关系的类之间，一个父类具有多个子类，可以将子类对象直接赋值给一个父类引用变量，无须任何类型转换；子类重写父类的方法也是 Java 多态性的一种体现。

章节练习

习题

- 对于 Java 语言，在下面关于类的描述中，正确的是_____。
 - 一个子类可以有多个父类
 - 一个父类可以有多个子类
 - 子类可以使用父类的所有
 - 子类一定比父类有更多的成员方法
- 下列_____关键字修饰类后不允许有子类。
 - abstract
 - static
 - protected
 - final
- 假设 Child 类为 Base 类的子类，则下面_____创建对象是错误的。
 - Base base = new Child();
 - Base base = new Base();
 - Child child = new Child();
 - Child child = new Base();
- 下列关于关键字 super 和 this 的说法中不正确的是_____。
 - super(..)方法可以放在 this(..)方法前面使用
 - this(..)方法可以放在 super(..)方法前面使用
 - 可以使用 super(..)来调用父类中的构造方法
 - 可以使用 this(..)调用本类的其他构造方法
- 给定如下 Java 代码，关于 super 的用法，以下_____描述是正确的。

```
class Student extends Person{
    public Student (){
        super();
    }
}
```

- 用来调用 Person 类中定义的 super()方法
- 用来调用 Student 类中定义的 super()方法

- C. 用来调用 Person 类的无参构造方法
D. 用来调用 Person 类的第一个出现的构造方法
6. 下列关于内部类的说法中,错误的是_____。
- A. 内部类能够隐藏起来,不为同一包的其他类访问
B. 内部类是外部类的一个成员,并且依附于外部类而存在
C. Java 内部类主要有成员内部类、局部内部类、静态内部类、匿名内部类
D. 局部内部类可以用 public 或 private 访问修饰符进行声明
7. 下列关于继承的说法中,不正确的是_____。
- A. 在继承过程中,子类拥有父类所定义的所有属性和方法
B. 在构造一个子类对象时,会首先调用自身的构造方法进行初始化,而后再调用父类的构造方法进行初始化
C. Java 只支持单一继承
D. 使用 extends 关键字使子类继承父类
8. 下列关于方法重写的说法中,错误的_____。
- A. 父类中的私有方法不能被子类重写
B. 父类的构造方法不能被子类重写
C. 方法名以及参数列表必须完全相同,返回类型可以不一致
D. 父类的静态方法不能被子类重写

上机

1. 训练目标: 继承的使用。

培养能力	继承与多态的使用		
掌握程度	★★★★★	难度	难
代码行数	300	实施方式	编码强化
结束条件	独立编写,不出错		

参考训练内容

- (1) 使用继承编写人类、教师、学生类的实体类。
(2) 编写测试类,实例化教师和学生类对象并显示

2. 训练目标: 继承的使用。

培养能力	继承与多态的使用		
掌握程度	★★★★★	难度	中
代码行数	200	实施方式	编码强化
结束条件	独立编写,不出错		

参考训练内容

有一个水果箱(Box),箱子里装有水果(Fruit),每一种水果都有不同的重量和颜色,水果有苹果、梨、橘子。每个苹果(Apple)都有不同的重量和颜色,每个橘子(Orange)都有不同的重量和颜色,每个梨(Pear)都有不同的重量和颜色。可以向水果箱(Box)里添加水果(addFruit),也可以取出水果(getFruit),还可以显示水果的重量和颜色。编写代码实现上述功能