第5章 类和对象

本章主要介绍面向对象编程(Object Oriented Programming,OOP)的核心元素——类和对象。Java是一种常用的面向对象编程语言,它通过将程序的数据和行为(方法)封装到类中,并通过新建类的实例(对象)来操作这些数据。因此,理解类和对象是学习Java的关键步骤。首先介绍Java语言实现面向对象程序设计的基本概念和相关语法,如何定义类、创建对象,并进一步讲解构造方法及其重载。本章还将深入探讨Java中的两个重要关键字:this和static,对它们在实例化对象和操作数据时的应用进行详细讲解。最后讨论访问权限修饰符如何实现对类、变量、方法的访问控制。同时,本章也设计了具体任务,引导读者将理论知识应用到实际操作中,从而更好地理解并掌握类和对象的相关知识。

【学习目标】

【知识目标】

- ☑ 理解面向对象编程的核心概念,掌握类和对象的定义与关系,并能够使用Java语言创建和操作类和对象。
- ☑ 掌握构造方法的定义和重载机制,能够通过不同的构造方法创建具有不同初始状态的类实例。
- ☑ 深入理解Java中的this和static关键字,能够正确运用它们来引用当前对象成员、实现静态属性和方法,以及理解它们在面向对象编程中的作用。
- ☑ 熟悉访问权限修饰符的使用,能够正确设置类、变量、方法的访问权限,保障数据的安全性和程序的健壮性。
- ☑ 能够通过编程实践,将理论知识应用于实际问题的解决中,提升面向对象编程思维能力和实践能力。

【能力目标】

- ☑ 能够熟练定义和使用Java类,包括类的成员变量和成员方法,能够根据实际业务需求设计合理的类结构,并编写相应的构造方法实现对象的初始化。
- ☑ 熟练掌握访问权限修饰符的使用,能够根据实际需求设置类、变量、方法的访问级别,确保代码的安全性和可维护性。
- ☑ 能够运用面向对象编程思维,设计出结构清晰、易于维护和扩展的Java程序,通过编

程实践将理论知识转化为实际代码，解决具体问题，并具备在团队项目中合理划分类和对象、与团队成员协作完成复杂编程任务的能力。

【素质目标】

- ☑ 深化学习者对面向对象编程思想的理解，通过类和对象的实践应用，培养其运用OOP原理分析和解决问题的能力，形成系统化、结构化的编程思维。
- ☑ 提升学习者的抽象能力，通过类的设计，使学习者能够将实际问题抽象为类和对象的关系，提高问题解决的效率和准确性。
- ☑ 增强学习者的自主学习能力，通过对类和对象相关知识的深入学习和实践，激发学习者的探索精神，鼓励他们主动寻求新知识、新技术，不断提升自己的编程技能。

【思政元素】

面向对象编程强调的封装、继承、多态等理念，可以类比于社会中的分工合作、知识传承和创新发展。通过OOP的学习，可以培养学生的团队协作精神和创新意识，以及适应不断变化的社会环境的能力。

类和对象的概念可以类比于现实生活中的类别和个体。通过学习定义类和创建对象，可以培养学生的抽象思维能力和分类归纳的能力，进而培养他们对社会现象进行深入分析和理解的能力。

构造方法是创建对象时调用的特殊方法，重载构造方法则体现了方法的多样性和灵活性。通过学习构造方法和重载，可以培养学生的创新思维和解决问题的能力，以及适应不同场景和需求的应变能力。

访问权限修饰符如public、private、protected等用于控制类、变量、方法的可见性和访问权限。学习这些修饰符的使用可以培养学生的隐私保护意识和信息安全意识，以及尊重他人知识产权和遵守社会规范的道德观念。

【任务预览】

本章主要任务是编写一个简单的图书管理系统。在该系统中，需要定义至少两个类：图书类和图书管理类。图书类包含标题、作者、出版日期等属性，并能够实现对图书的添加、查询、删除、更新等基本功能。完成本章的任务不仅可以检验读者对本章类和对象的理解程度，更可以实现知识的实际应用，提升编程能力并通过解决具体问题提升读者的问题解决能力。

5.1 面向对象程序设计

面向对象编程(OOP)是一种程序设计范式，将数据和处理数据的功能捆绑在一起形成“对象”，模拟真实世界中各个实体及其互动行为，是编程的重要理念。OOP建立在四个主要概念之上，即封装、继承、抽象和多态。下面就来学习面向对象程序设计中涉及的基本概念：类与对象，抽象与封装，继承与多态。

5.1.1　面向对象的基本概念

1. 类

在日常生活中，有许多事物都属于某一种类别或类型，比如手机，手机可以看作是一种大的类别或类型，因为所有的手机都有一些共同的特征，比如都可以进行通话、发送信息、上网等。然而，手机这个大类别又可以被细分出很多小的类别，如华为手机、苹果手机、小米手机等，它们都是手机，但是各有各的特色。比如华为手机，可能搭载的是鸿蒙操作系统，有超级快充的功能。而苹果手机，则运行 iOS 系统，具有 Face ID 解锁功能。

这个概念在计算机的面向对象编程中也有出现，我们称之为类。类是对现实世界中的事物或者对象的一种抽象。在编程中，可以创建一个类，这就像是在定义一个新的事物类型，用以描述这类事物的共有特性和行为。比如可以创建一个手机的类，这个类定义了手机的基本属性，比如品牌、型号、操作系统、内存、像素等，以及一些基本行为，比如打电话、发短信、上网等。

当根据这个手机类的模板创建出一个具体的实例，也就是一个具体的手机时，这个手机就是一个对象。这个对象拥有手机类中定义的所有属性和行为，比如它的品牌是华为，型号是 Mate 60，操作系统是鸿蒙，内存为 256GB 等，又比如它可以用来打电话、发短信、上网。同样，也可以创建一个苹果手机的对象，这个对象也拥有手机类中定义的所有属性和行为，但因为它是苹果品牌，所以操作系统是 iOS。华为手机或者苹果手机与手机之间的关系就是类和类的成员对象之间的关系。类是具有共同属性与共同行为的对象集合，而单个的对象则是所属类的一个成员，也称作实例。在描述一个类的时候，定义了一组属性和行为，而这些属性和行为可被该类所有的成员所继承，如图 5-1 所示。

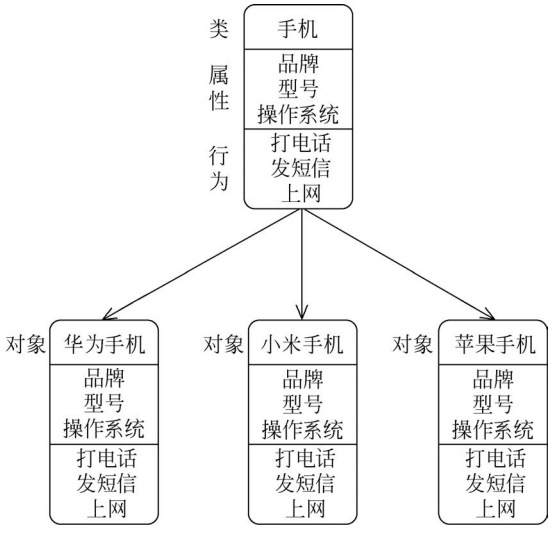

图 5-1　手机类的三个实例

类就是一种创造对象的模板，对象是类的实例。通过定义类，可以让计算机理解现实世界中的事物和行为，实现对现实世界的模拟和操作。

2. 对象

对象是面向对象编程中的核心概念之一。前面已经讨论了类的构成，以及如何根据类来定义不同的实例。现在将进一步探讨类的具体实例，即对象。

在面向对象编程中，把类比作是一种蓝图或模板，描述要创建的对象的基础结构和行为，而对象是根据这个蓝图或模板创建的实例。例如，考虑手机这个类描述了所有手机的公有属性和行为特点。这些可能包括品牌、型号、操作系统、内存、像素等基础属性，以及拨打电话、发短信、上网等行为。然而手机这个类本身并不代表任何特定的手机，它仅仅提供了描述各种具体手机的基础。

在这个基础上，可以创建具体的手机对象。如“华为 Mate60”、“苹果 iPhone15”或“小米 12”都是手机类的一个具体实例。每个手机对象既有手机类中定义的公有属性，也有其独特的个体属性。每个对象由两个主要部分组成：属性和方法。

(1) 属性：属性是用来描述对象状态的变量。例如，在一个华为手机的对象中，其属性可能包括型号(如 Mate60)，颜色(如黑色)，价格(如 5988 元)，重量(如 203 克)等。每个对象的属性值会根据具体的对象有所不同。

(2) 方法：方法是对象行为的表示，即对象可以执行的操作。例如，一部华为手机的可能行为(方法)包括拨打电话、浏览网页、运行程序等。要注意的是，对象的方法也可能会改变对象的内部状态，也就是对象的属性。

在面向对象编程中，通过定义类和创建对象，开发者能够以一种更接近自然描述现实世界事物的方式来建模和编程，这提高了编程任务的灵活性和可理解性。在后续的章节中，我们将更深入地介绍不同对象之间如何互动，以及如何用对象组织和执行更复杂的编程任务。

5.1.2 面向对象的基本特征

面向对象的编程具有 4 个基本的特征：抽象、封装、继承和多态。这 4 个特征为创造和使用复杂的类和对象提供了强大的工具和方法。

1. 抽象与封装

抽象是面向对象编程的核心特性之一。它提供了一种方式，帮助从更高的层次理解问题，忽略不必要的细节，只关注对当前问题具有直接影响的因素，这有助于将复杂的问题简单化。在 Java 中，使用类来实现抽象。类是现实世界中事物的一种抽象，它定义了一种数据类型，描述了这种类型的对象包含哪些状态(属性)和行为(方法)。例如，定义一个汽车类，该类可能包括属性如颜色、型号和速度，以及方法如加速、刹车和倒车等。通过这种方式将现实世界中复杂的汽车抽象成一个简单的汽车类。

封装是面向对象编程的另一个核心特性，它将一个对象的状态和行为绑定在一起，并隐藏了对象内部的实现细节，只通过对象的方法公开有限的接口给外部，这就形成了一种“黑箱”操作的模式。

封装的好处是可以保护对象内部状态的完整和一致。通过拒绝直接访问对象的内部状态，并仅通过其公开的方法来修改状态，可以避免无意间破坏对象状态的危险，从一定程度上提高了安全性。再以汽车类为例，可以将加速和刹车设计为汽车类的方法，但是内部的实

现细节，比如具体如何调整油门和刹车则被隐藏起来。外部只能通过调用加速和刹车方法来改变汽车的状态。

2. 继承与多态

继承是面向对象编程中一种创建新类(子类)的方式，该子类继承了一个或多个已有类(父类)的属性和方法，同时也允许子类添加新的属性和方法或覆盖父类的属性和方法。这种特性提高了代码的复用性，大大提升了开发效率，同时还有助于从已有的类组织构建更复杂的类。例如，可以将汽车看作是一种交通工具，那么汽车类可以继承交通工具类，继承其公用属性如速度、重量等，以及公用行为如开车、停车等。并增加或重写一些特有的属性和行为，比如汽车特有的驾驶模式。

多态是面向对象编程中的一个关键特性，它允许不同的子类对象可以以自己特定的方式实现共同的父类接口或方法。简单来说，多态性允许使用一个父类类型的引用来引用子类对象，从而使得不同的子类对象可以被统一地处理。这意味着，当使用父类类型的引用调用方法时，实际上会调用到子类对象所覆盖的方法，而不是父类的方法。这样做使得代码更加灵活、可扩展，也更符合面向对象编程的思想。例如，汽车类作为交通工具类的子类，当需要一种交通工具时，汽车对象可以自然地替代交通工具对象，因为汽车是交通工具，具备交通工具的所有特性和行为。

5.2 类的定义与对象的创建

5.2.1 类的定义格式

类是 Java 的基本组成单位，通过类可以生成多个对象实例。它封装了数据(即属性)和操作这些数据的方法。类在面向对象编程中是至关重要的，因为它能有效地封装代码，提供创建对象的模板，使代码组织和操作更加整洁和高效。类的作用不可或缺，它能够对对象进行抽象，并对相似对象进行分类。Java 中的类一般分为两大类：系统定义的类和自定义的类。

1. 系统定义的类

顾名思义，系统定义的类是由 Java 系统提供的类库中的类，如 String、System、Scanner 等。这些类在 Java.lang 包中默认被导入，用户可以直接在程序中使用这些类，不需要再次导入。这些类非常丰富，涵盖了从基本数据处理到复杂的网络编程的各种功能。例如，Java.lang.System 类就提供了系统级别的一些常用功能，例如读取环境变量，获取系统当前时间等。

2. 自定义类

自定义的类是由程序员创建的类，这些类根据程序的特定需求，定义其属性和行为。自定义类是 Java 编程的基础，可以打造出完全符合需求的对象模型。在 Java 中，一个类的定义通常包括如下几部分。

(1) 类名称：它可以是任何符合标识符规定的名称，通常情况下，类名的第一个字母要大写，如果类名由多个单词组成，则每个单词的第一个字母均要大写，这种命名方法叫作驼峰命名法。

用户如果想要实现一些特定的需求就要自定义类，以下是定义类的一般形式：

```
[访问修饰符] class 类名 [extends 父类名] [implements 接口名]{
    //类体
}
```

说明：

① 在定义一个自定义类的时候 class 关键字必须存在，[ ]中的内容可以根据实际需要省略或选择。

② 访问修饰符：它决定了其他类对这个类的访问级别。Java 定义了 4 种访问级别，分别是 public、protected、default(默认访问修饰符)和 private。其中，public 表示该类可以从任何地方访问，default 叫作默认访问修饰符，该类或者类中的字段和方法，只能被同一个包内的其他类访问。而 private 则表示这个类只能在其自身中被访问。这几种访问权限修饰符将在 5.4 节中详细讲解。

③ 类名：首字母大写，并遵循驼峰命名法。

④ extends：此关键字用于表达类的继承关系。一个类可以继承其他类的字段和方法。

⑤ implements：此关键字用于表达类实现的接口。一个类可以实现一个或多个接口。

(2) 类的成员变量：在面向对象编程中，类的成员变量是指定义在类内部的变量，也称为字段或属性。类的成员变量描述了对象的状态或特征。成员变量的定义格式如下：

```
[访问修饰符]  数据类型  变量名  [=  初始值];
```

说明：

① 变量的访问修饰符：可能的修饰符同样也有 4 种，分别是：public——任何类都可以访问变量。private——变量只能在其所属的类中访问。protected——变量可以在其所属的类中、任何子类以及同个软件包的类中访问。default——如果未指定访问修饰符，则为包私有。变量可以在其所属的类和同一个包中的其他类中访问。

② 数据类型：这部分为变量指定数据类型。它定义了变量可以容纳的数据的类型。可能是基本类型，如 int、float、boolean、char 等，也可以是引用类型，例如自定义类或者接口。

③ 变量名：这部分是定义的变量名称，用于在代码中引用它。Java 变量命名通常是以简洁且清晰的方式来描述它的作用，变量的命名要符合 Java 相应的命名规范。

④ [=初始值](可选)：可以选择在声明变量的时候给它分配一个初始值。这不是必需的，也可以稍后赋值。对于对象的变量，可以初始化为空，表示该对象还未指向任何对象实例。

(3) 类的成员方法：类的成员方法是定义在类内部的函数，也称为方法或行为。类的成员方法描述了对象的行为或功能，并且每个对象都可以调用类的成员方法来执行特定的操作。成员方法的定义格式如下：

```
[访问修饰符]  返回值类型  方法名(参数列表){
    //方法体
}
```

① [访问修饰符]：访问修饰符定义了其他类对这个方法的访问级别。访问修饰符可以是 public、protected、default。

② 返回值类型：这是方法的返回类型，指明该方法执行后返回的数据类型。如果该方法不返回任何值，则使用关键字 void。

③ 方法名：方法的名称。也要遵循驼峰式命名法进行命名。例如：calculateSum()。

④ (参数列表)：方法的参数列表，包含零个或多个参数。每个参数前面都需要声明其类型，参数之间使用逗号分隔。

⑤ {方法体}：方法的主体部分，即方法的行为定义部分。这里包含了实现方法功能的具体代码。方法体是由一对花括号包围的。

【例 5-1】 设计并编写一个 Rectangle 矩形类，这个类含有 length 和 width 两个私有成员变量，并且实现以下功能：使用公共的 get()和 set()方法，允许外部访问和修改这些私有变量的值，并设计公有成员方法 calculateArea()，用于计算矩形的面积。设计公有成员方法 calculatePerimeter()，用于计算矩形的周长。

【程序实现】

```
package chapter05;
public class Rectangle {
    //成员变量
    private double length;                              //矩形的长
    private double width;                               //矩形的宽

    //构造方法
    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

//设置矩形的长
    public void setLength(double length) {
        this.length = length;
    }

    //获取矩形的长
    public double getLength() {
        return length;
    }

    //设置矩形的宽
    public void setWidth(double width) {
        this.width = width;
    }

    //获取矩形的宽
    public double getWidth() {
        return width;
    }

    //成员方法:计算矩形的面积
    public double calculateArea() {
        return length * width;
    }
```

```
    //成员方法:计算矩形的周长
    public double calculatePerimeter() {
        return 2 * (length + width);
    }
}
```

【例 5-2】 设计并编写一个 Pet 宠物类,这个类包含私有成员变量 name 和 age,分别表示宠物的名字和年龄,含有一个带两个参数的公有构造方法,用于初始化宠物的名字和年龄。并且实现以下功能:使用公共的 get()和 set()方法,允许外部访问和修改这些私有变量的值,并设计 eat(),sleep(),play()三个成员方法分别描述宠物的吃饭行为、睡觉行为和玩耍行为。

【程序实现】

```
package chapter05;
public class Pet {
    //成员变量
    private String name;                                    //宠物的名字
    private int age;                                        //宠物的年龄

    //构造方法
    public Pet(String name, int age) {
        this.name = name;
        this.age = age;
    }

    //成员方法:设置宠物的名字
    public void setName(String name) {
        this.name = name;
    }

    //成员方法:获取宠物的名字
    public String getName() {
        return name;
    }

    //成员方法:设置宠物的年龄
    public void setAge(int age) {
        this.age = age;
    }

    //成员方法:获取宠物的年龄
    public int getAge() {
        return age;
    }

    //成员方法:描述宠物的吃饭行为
    public void eat() {
        System.out.println(name + "正在开心地吃饭!");
    }

    //成员方法:描述宠物的睡觉行为
    public void sleep() {
        System.out.println(name + "正在安稳地睡觉,呼噜噜……");
    }
```

```
    //成员方法:描述宠物的玩耍行为
    public void play() {
        System.out.println(name + " 正在快乐地玩耍!");
    }
}
```

5.2.2 构造方法的定义

构造方法是类中特殊的方法,专门用来创建对象的方法,它在对象被实例化时被调用。尽管它们的调用形式类似于普通的成员方法,但在语义上不同于常规方法。构造方法不返回任何值,其名称与所在类的类名相同,构造方法的主要任务就是初始化对象的状态,而不是提供任何计算或处理结果。

【例 5-3】 定义一个 Book 类表示书籍类,用于显示图书的详细信息,Book 类中有三个成员变量 title、author、publisher 分别表示书籍的名称、作者和出版社,以及一个带三个参数的构造方法,和一个显示图书的详细信息的成员方法。

【程序实现】

```
package chapter05;
public class Book {
    String title;
    String author;
    String publisher;

    //带三个参数的构造方法
    public Book(String title, String author, String publisher) {
        this.title = title;
        this.author = author;
        this.publisher = publisher;
    }

    //显示图书的详细信息的成员方法
    public void displayInfo() {
        System.out.println("书名: " + this.title);
        System.out.println("作者: " + this.author);
        System.out.println("出版社: " + this.publisher);
    }
}
```

定义构造方法时,一定要注意以下几点。

(1) 构造方法的名字必须与它所在的类的名称完全一致,包括大小写。

(2) 构造方法不应该有返回类型,包括 void。如果错误地为构造方法定义了一个返回类型,那么编译器会把它当作一个普通方法,而不是构造方法。

(3) 如果没有定义任何构造方法,那么默认构造方法将被自动添加。但如果定义了任何构造方法,无论是否带参数,都不会自动添加默认构造方法。

(4) 在声明构造方法时,可以决定它是否需要输入参数,也可以根据需要定义一个或多个构造方法,这被称为构造方法的重载。这在当类的对象可以以多种方式初始化时非常有用。例如,我们可能想要以一种方式来创建一个完全默认的对象,又想要以另一种方式来创建一个具有特定初始状态的对象,这时就需要用到重载构造方法,如例 5-4 所示。

【例 5-4】 在例 5-3 中有的时候为了秉持简化的目标，仅需要标题和作者就可以创建一本书，出版社信息可以在需要时再填补。有的时候需要更多的详细信息，想在创建对象时即提供完整信息，请设计相应的代码。

此时就需要根据具体的需求在 Book 书籍类中添加两个重载的构造方法，一个只接收标题和作者，另一个接收标题、作者和出版社。每当创建新的 Book 书籍类对象时，可以选择使用其中的任意一个构造方法。

【程序实现】

```
package chapter05;
public class Book {
    String title;
    String author;
    String publisher;

    //第一个构造方法，只需要标题和作者
    public Book(String title, String author) {
        this.title = title;
        this.author = author;
        this.publisher = null;
    }

    //第二个构造方法，需要标题、作者和出版社
    public Book(String title, String author, String publisher) {
        this.title = title;
        this.author = author;
        this.publisher = publisher;
    }

    //显示图书的详细信息的成员方法
    public void displayInfo() {
        System.out.println("书名：" + this.title);
        System.out.println("作者：" + this.author);
        System.out.println("出版社：" + this.publisher);
    }

    //更新图书的作者的成员方法
    public void updateAuthor(String newAuthor) {
        this.author = newAuthor;
    }
}
```

5.3 对象

5.3.1 对象的创建

在面向对象的编程语言中，经常会听到对象，那么对象究竟是什么？对象实际上是由类实例化而来的一个实体，它是一个具体的，可以看见、感知的事物。类是对象的模板，规定了对象的基本形态和行为方式，对象是基于这些规定的类所实例化出来的实体，具备了该类所有的属性和能力。

在程序设计的过程中，需要一种方式来模拟现实世界的情境和事物。对象就提供了这样一种模拟方式。每个对象可以看作是真实世界中某一事物或者情境的抽象，比如学生、图书、汽车、动物等，都可以建模为对象。通过对象可以在程序中操作这些真实的事物，比如查询一个学生的成绩，或者改变一个汽车的记录信息。对象具有状态(属性)和行为(方法)，状态描述了对象是什么，行为描述了对象能做什么。

对象是现实世界的抽象，通过创建对象，就能在程序中代表或者创建出现实世界的事物。但更重要的是，每个对象都拥有独立的状态，可以按照设定的规则独立地行动，这使得代码更为灵活，复用性也更强。这就是创建对象的根本目的：以更易于管理和操作的方式对复杂的程序世界进行建模。在 Java 等面向对象的编程语言中，通常有两种典型的创建对象的方式，具体如下。

(1) 先声明对象的引用，再实例化对象。示例如下：

```
类名 对象名;
对象名 = new 类名([参数列表]);
Book myBook;
myBook = new Book("《活着》", "余华");
```

在这个例子中先是创建了一个书籍类 Book 的对象引用 myBook，此时，并没有创建真实的 Book 对象，myBook 指向的是 null。当执行“myBook＝new Book("《活着》", "余华");”时，new 关键字被用来实例化一个 Book 类的对象，并且该对象的引用被赋值给了 myBook。

(2) 声明对象的同时实例化对象。示例如下：

```
类名 对象名 = new 类名([参数列表]);
Book myBook = new Book("《活着》", "余华");
```

在这个例子中，在声明 Book 类的对象引用 myBook 的同时就实例化了 Book 类的一个对象。

在创建对象的时候要注意以下几点。

① 方法(1)是分两步进行的，先声明一个对象引用，然后再使用 new 关键字来创建一个新对象并将其引用赋值给引用变量。而方法(2)则是将声明引用变量和创建对象合二为一步。

② 对于方法(1)，如果在声明对象引用和实例化对象之间的代码中，尝试去调用那个尚未实例化的对象，程序将会抛出 NullPointerException，因为那个对象引用并未指向任何对象。

③ 在方法(2)中，因为对象的实例化是与对象引用声明同时进行的，所以不存在声明引用和实例化对象之间的代码，不会引发上述问题。

5.3.2 对象的访问

访问对象也称引用对象或调用对象，它是程序中与对象交互的关键方式之一，它通过直接引用对象的实例，使得程序能够访问和操作对象所包含的数据和行为。通过访问对象，程序可以调用对象的方法执行特定的功能，获取对象的状态信息，并且可以根据需要修改对象的属性值，从而实现对程序的功能逻辑的具体实现。在面向对象的程序设计中，对象是程序

的核心概念，而访问对象则是连接程序与对象之间关系的桥梁，它使得程序能够更加灵活、可扩展，同时也提高了代码的可读性和可维护性。通过合理地访问对象，程序可以实现各种复杂的功能，从而满足用户的需求，达到预期的效果，使得程序具有更强的适用性和实用性。访问对象的格式如下：

```
对象名.方法名([实际参数列表])
对象名.变量名
```

在访问对象时，需要注意以下几点。

（1）对象名必须是已经创建的对象的名称，通过该名称来引用对象的实例。

（2）方法名必须是对象所属类中已经定义的方法名，用于对对象进行操作或获取信息。

（3）实际参数列表是调用方法时传递给方法的具体参数，如果方法不需要参数，则可以为空。

（4）变量名必须是对象所属类中已经定义的成员变量名，用于访问对象的属性或状态。

（5）在访问对象的方法或变量时，需要确保对象已经被正确创建，并且不为 null，否则会出现空指针异常。

（6）在访问对象的方法或变量时，需要确保方法或变量的访问权限符合要求，不能访问私有成员或没有权限的成员。

【例 5-5】 利用例 5-1 中 Rectangle 矩形类，分别计算长为 15，宽为 8 的矩形的面积和周长。

【程序实现】

```
package chapter05;
public class Test5_5 {
    public static void main(String[] args)
        Rectangle rectangle = new Rectangle(15, 8);          //创建一个 Rectangle 对象
        //访问对象的方法并输出结果
        double area = rectangle.calculateArea();
        //调用了对象的 calculateArea()来计算矩形的面积
        double perimeter = rectangle.calculatePerimeter();
        //调用了对象的 calculatePerimeter()方法来计算矩形的周长
        System.out.println("矩形的面积为: " + area);
        System.out.println("矩形的周长为: " + perimeter);
    }
}
```

【运行结果】

```
矩形的面积为: 120.0
矩形的周长为: 46.0
```

5.4 this 关键字

在 Java 编程语言中，this 是一种特殊的关键字，其作用是指代当前对象，即正在执行当前方法的对象实例。在面向对象的程序设计中，this 关键字通常出现在非静态方法中，用于在方法内部引用当前对象，从而方便地访问对象的属性和调用对象的其他方法。通过使用

this 关键字，可以明确指明当前方法所操作的对象，避免与同名参数或局部变量产生歧义，提高代码的清晰度和可读性。此外，this 关键字的使用还能够在方法内部将对象作为参数传递给其他方法，或者在方法内部进行对象的赋值操作。总之，this 关键字在 Java 中扮演着重要的角色，它提供了一种便捷的方式来引用当前对象，使得对象的操作和方法调用更加直观和灵活。

5.4.1 this 关键字的作用

1. this 表示当前对象

在面向对象编程中，对象是一个完整的单元，具有特定的属性(数据)和方法。对象代表现实世界中的独立个体，如人、汽车、狗、图书等。在类中创建方法或属性时，有时需要引用正在调用此方法的对象，或者需要引用包含正在执行此属性的对象。在这种情况下，需要一个关键字来引用这个当前对象。在很多面向对象的编程语言中，this 关键字用于完成这个任务。

this 关键字是一个自引用的变量，它表示调用或包含被操作的属性或方法的对象。可以将 this 视为指向正在动态运行的对象的指针。this 为代码提供了一种内省能力，让代码知道正在操作什么。例如，在下面的 Employee 类中，this 代表一个员工对象。

```
public class Employee {
    private String name;
    public void setName(String n) {
        this.name = n;    //使用 this 关键字指代当前对象,将方法参数赋值给对象的 name 属性
    }
}
```

2. 区分同名变量

当方法的参数名与类的成员变量名相同时，可以使用 this 关键字来区别它们。在 Java 中，方法的参数名和类的成员变量名相同会造成歧义，编译器无法确定到底是访问方法的参数还是访问类的成员变量。因此，使用 this 关键字可以明确指示编译器，当前操作的是类的成员变量而不是方法的参数。通过在成员变量前加上 this 关键字，可以清楚地表明要操作的是对象的成员变量，从而避免产生混淆和错误。这种方式提高了代码的清晰度和可读性，使得程序更易于理解和维护。例如，在下面的 Employee 类中，this.name 表示的是员工对象的 name 成员变量，而 name 表示的是 setName()方法的参数。

```
public class Employee {
    private String name;
    public void setName(String name) {
        this.name = name;      //使用 this 关键字区分方法参数 name 和类的成员变量 name
    }
}
```

3. 用于链式编程

this 关键字经常用于链式编程，即在一个对象上连续调用多个方法，并且每个方法都返

回对象本身，以便可以在一行代码中完成多个操作。通过在方法内部返回 this，可以方便地在多个方法之间传递对象，并且可以在调用方法后立即继续对对象进行其他操作。这种技术在构建具有流畅接口的 API 时非常有用，提高了代码的可读性和简洁性，使得代码更易于理解和维护。例如，在下面的 Employee 类中，每个方法在执行完赋值操作后都返回当前对象，这就允许链接调用："new Employee(). setName("John"). setAge(25);"。

```
public class Employee {
    private String name;
    private int age;

    public Employee setName(String name) {
        this.name = name;
        return this;                    //返回当前对象的引用,以支持链式调用
    }

    public Employee setAge(int age) {
        this.age = age;
        return this;                    //返回当前对象的引用,以支持链式调用
    }
}
```

4. 作为参数传递

在使用 Java 编程解决某些需求的时候，在某个方法中需要访问调用该方法的对象。为了实现这一目的，可以将 this 作为参数传递给该方法。这样做的好处是可以在被调用的方法中明确地操作调用该方法的对象，而不必依赖隐式的方式来访问对象的属性或调用对象的其他方法。通过将 this 作为参数传递给方法，可以增强代码的可读性和可维护性，使得代码的意图更加清晰明了。例如，在下面的 Employee 类有一个 joinDepartment()方法，该方法将当前员工对象(this)作为参数传递给 Department 类中的 addEmployee()方法。addEmployee()方法根据这个参数添加一个员工到部门，并且打印出员工姓名和部门名字。这种使用 this 作为参数传递的方式，使得不同对象之间可以方便地互相传递和操作数据。

```
public class Employee {
    private String name;
    private Department department;

    public Employee(String name) {
        this.name = name;
        //这里的 this 指的是当前 Employee 对象,用来区分参数 name 和类的属性 name
    }

    public void joinDepartment(Department department) {
        department.addEmployee(this);     //这里的 this 指的是当前 Employee 对象

    }

    public String getName() {
        return name;                      //这里没有使用 this,因为没有命名冲突
    }
}
public class Department {
```

```
    private String name;
    private List<Employee> employees;
    public Department(String name) {
        this.name = name;                        //这里的 this 指的是当前 Department 对象
        this.employees = new ArrayList<>();  //这里的 this 也是指的当前 Department 对象

    }

    public void addEmployee(Employee employee) {
        this.employees.add(employee);          //这里的 this 指的是当前 Department 对象
        System.out.println("Employee " + employee.getName() + " has joined the " + this
.name + " department.");
    }
}
```

5. 调用构造方法

this 关键字还可以用于在一个构造器中调用同一类中的另一个构造器。这种情况下，被调用的构造器必须位于同一类中，并且调用语句必须作为构造器中的第一条语句。通过使用 this 关键字来调用另一个构造器，可以在构造器中重用代码，并且能够实现更灵活的构造器调用。这种技术使得代码更具可维护性和可读性，同时也有助于减少重复代码的编写。例如，在下面的 Employee 类中，Employee()构造方法通过“this("Jack", 18);”语句调用了另一个构造方法 Employee(String name, int age)，用于设置默认的员工名称和年龄。这样做的好处是可以避免在多个构造方法中重复相同的初始化代码，提高代码的可维护性。

```
public class Employee {
    private String name;
    private int age;

    //默认构造方法
    public Employee(){
        this("Jack", 18);           //调用另一个构造器
    }

    //带两个参数的构造方法
    public Employee(String name, int age) {
        this.name = name;           //使用 this 关键字指代当前对象，设置对象的 name 属性
        this.age = age;             //使用 this 关键字指代当前对象，设置对象的 age 属性
    }
}
```

5.4.2 使用 this 关键字的注意事项

(1) 不要在静态方法中使用 this 关键字：因为静态方法是属于类的，而不是某个具体的对象。静态方法被所有对象共享，因此在静态方法中使用 this 是不合适的。

(2) 只有在非静态方法或构造函数中才可以使用 this 关键字：因为非静态方法和构造函数属于某个具体的对象。

(3) 谨慎地使用 this 关键字调用其他构造函数：在构造函数中使用 this 调用其他构造函数是可以的，但是需要注意的是，this()必须是构造器中的第一个语句。

(4) 在非静态方法中,如果局部变量和类的属性没有同名冲突,可以选择不用 this,虽然 this 可以明确区分出正在引用类的属性,但也可能会加大代码的阅读难度,特别是在大型项目中。

(5) 在返回当前实例时使用 this,可以在非静态方法或者构造函数中通过 return this 来返回当前对象。这通常在需要链式操作时使用。

(6) 要注意 this 的使用范围和生命周期:this 关键字只能在类内部使用,对类外部是不可见的。this 的生命周期与它所在的对象相同。当对象被创建时,this 就开始存在,当对象被销毁时,this 就会消失。

5.5 static 关键字

在面向对象编程中,static 关键字是一种被广泛应用的特性,它赋予了变量和方法一系列特定的性质。静态变量和静态方法属于类本身,而不是类的实例对象。这意味着即使在未创建任何类的实例对象的情况下,静态变量和方法也可以直接被访问。通过 static 关键字定义的成员,可以在程序的整个生命周期内只创建一次,并且它们的值对于所有实例对象都是共享的。因此,static 关键字在面向对象编程中常用于创建属于类本身的共享变量和方法,从而提供了一种有效管理和访问类级别资源的方式。

5.5.1 静态变量

在 Java 中,静态变量(static variable)也被称为类变量,它是被类的所有实例共享的。静态变量的特性来自于它在 Java 内存中的存储方式。当加载一个 Java 类时,静态变量被存储在 Java 的方法区(Method Area)。方法区是 Java 虚拟机(JVM)的一个组成部分,用于存储加载的类信息、常量、静态变量等数据。属于每个类的静态变量在方法区中只有一份存储。(方法区在 Java 8 之后被称为元空间,或者 Metaspace,但用于存储的类型的信息没有变化。)

与此不同的是,非静态变量(实例变量)被存储在堆(Heap)区域。每次通过 new 关键字创建一个类的实例,就会为所有的实例变量分配新的内存空间。即使这些变量的名称相同,但由于它们所属的对象不同,所以在堆中会存在多份实例变量。

这种内存管理方式引发了静态变量和非静态变量的主要区别:静态变量在类被加载时分配内存,只有一份存储,所有对象共享这个变量。因此,无论创建了多少个对象,静态变量都只有一个副本,改变一份,会影响到其他所有副本。而实例变量对于每个对象都拥有自己的一份副本,更改某个对象的实例变量不会影响其他对象的同名实例变量。

【例 5-6】 编写一个程序,用于统计创建的学生对象的数量,并输出结果。具体要求如下。

(1) 定义一个名为 Student 的类,该类包含一个静态变量 count 和一个非静态变量 name。count 用于记录创建的 Student 对象的数量,初始值为 0,name 用于表示学生的名字。

(2) 实现 Student 类的构造方法,该构造方法接收一个参数 name,并将其赋值给 name

变量，然后将 count 增加 1。

（3）在测试类的 main 方法中，创建三个学生对象，分别为“张三”“李四”和“王五”，并输出创建的学生数量。

【程序实现】

```
package chapter05;
class Student {
    static int count = 0;     //定义一个静态变量 count,初始值为 0.用于记录 Student 对象的数量
    String name;              //定义一个成员变量 name,用于记录学生的名字
    Student(String name){
        this.name = name;
        count++;
    }
}
public class Test5_6 {
    public static void main(String[] args) {
        Student student1 = new Student("张三");
        Student student2 = new Student("李四");
        Student student3 = new Student("王五");
        System.out.println("学生的数量为: " + Student.count);
    }
}
```

【运行结果】

```
学生的数量为: 3
```

在例 5-6 中，定义一个类 Student，该类有一个静态变量 count 和一个非静态变量 name。每当创建一个新的 Student 对象时，构造函数就会被调用，当构造函数被调用时，静态变量 count 的值会增加 1。最后在测试类中创建了三个 Student 对象，因此最后打印出来的学生数量为 3。

5.5.2 静态方法

在 Java 中，静态方法与静态变量一样，在类被加载时已经存在，在 Java 内存模型中位于方法区（Method Area）或者 Java 8 之后的元空间（Metaspace）。特别需要注意的是，静态方法并不归属于任何一个实例对象，它属于类。无论是静态方法还是非静态方法，方法的代码本身只有一份，存储在方法区内。当调用这些方法时，实际上是在栈内存（Stack）中创建了一个栈帧来存储局部变量、操作数栈、常量引用等。

对于非静态方法，需要用一个类的实例去调用它，此时会新建一个栈帧，其中包含了指向方法的实例对象（即 this 引用），以及方法的局部变量等。

而对于静态方法，因为它不依赖任何实例对象，所以在调用时并不需要有 this 引用之类的指向对象的信息。可以直接使用类名调用静态方法来创建对应的栈帧执行。在静态方法调用过程中，不会涉及任何与堆内存中实例对象的交互，除非静态方法中明确引用了某个实例对象。

因此，静态方法的特性使其使用起来更加灵活，无须实例化任何对象就可以调用静态方法，当然，这种灵活性也带来了一些限制，比如静态方法不能直接访问类中的非静态成员。

【例 5-7】 定义一个 Sum 类，在该类中定义一个静态方法，该方法用于计算两个整数的和，并进行测试。

【程序实现】

```
package chapter05;
class Sum {
    //定义一个静态方法,用于计算两个整数的和
    public static int add(int a, int b) {
        return a + b;
    }
}

public class Test5_7 {
    public static void main(String[] args) {
        int result = Sum.add(10, 20);              //调用静态方法,无须创建 Test 类的实例
        System.out.println("计算结果为: " + result);
    }
}
```

【运行结果】

```
计算结果为: 30
```

例 5-7 中，在 Sum 类中定义了一个静态方法 add(int a, int b)，可以通过类名 Test 直接调用它，不必创建 Sum 类的实例。在 main 方法中，调用了这个静态方法计算 10 和 20 的和，并将结果 30 打印出来。

注意：

在 Java 中使用 Static 关键字的时候要注意以下几点。

(1) 静态成员属于类而不是对象，使用 static 关键字声明的成员(变量或方法)属于类而不是类的实例对象。这意味着静态成员可以在没有创建类的实例对象的情况下被访问。

(2) 静态成员共享，所有类的实例对象共享相同的静态成员。因此，当一个实例对象修改了静态变量的值时，其他实例对象也会受到影响。

(3) 静态方法只能访问静态成员：静态方法只能访问静态成员变量和静态方法，不能直接访问非静态成员变量和实例方法。这是因为静态方法在对象创建之前就已经存在，无法访问与对象相关的成员。

(4) 静态成员可以通过类名直接访问，静态成员可以直接通过类名进行访问，不需要通过对象实例来访问。

(5) 静态成员在内存中只有一份副本，静态成员在程序的整个生命周期内只会被创建一次，它们的值对于所有实例对象都是共享的。因此，对静态成员的修改会影响到所有实例对象。

5.5.3 静态代码块

在 Java 中，静态代码块是在类加载的时候自动执行的代码段。它是在类定义中，即在类的成员位置，用 static 关键字修饰的代码块。静态代码块有如下特点。

(1) 如果在同一类中，静态代码块会优先于构造函数执行。类加载的时候静态代码块

就会被执行，用于初始化类的静态属性，无论这个类后续创建多少个对象，静态代码块只执行一次，而构造函数则是在每次创建新的对象时执行，是用来初始化对象的非静态属性的。

（2）如果在同一类中有多个静态代码块，它们按照在代码中的顺序从上到下顺序执行。

（3）在创建子类对象时，会先初始化父类再初始化子类。所以父类的静态代码块会优先于子类的静态代码块执行。如果子类和父类中都有构造函数，那么父类的构造函数则会优先于子类的构造函数执行。

【例 5-8】 静态代码块示例。

【程序实现】

```
package chapter05;
//父类
class Parent {
    //父类中的静态代码块
    static {
        System.out.println("父类静态代码块"); //这个代码块在父类被加载时执行,且仅执行一次
    }
    //父类的构造方法
    public Parent() {
        System.out.println("父类构造方法");   //每次创建父类的对象时都会执行这个构造方法
    }
}

//子类,继承自父类
class Child extends Parent {
    //子类中的第一个静态代码块
    static {
        System.out.println("子类静态代码块 1"); //这个代码块在子类被加载时执行,且仅执行一次
    }
    //子类中的第二个静态代码块
    static {
        System.out.println("子类静态代码块 2"); //这个代码块在子类被加载时执行,且仅执行一次
    }
    //子类的构造方法
    public Child() {
        System.out.println("子类构造方法");  //每次创建子类的对象时都会执行这个构造方法
    }
}

public class Test5_8 {
    //主方法,程序的执行入口
    public static void main(String[] args) {
        Child c1 = new Child();
    //创建第一个 Child 对象时,父类的静态代码块、子类的静态代码块、父类的构造方法、子类的构
    //造方法依次执行
        Child c2 = new Child();
    //创建第二个 Child 对象时,由于静态代码块已经在类加载时执行过了,所以只有父类和子类的
    //构造方法会再次执行
    }
}
```

【运行结果】

```
父类静态代码块
子类静态代码块 1
子类静态代码块 2
父类构造方法
子类构造方法
父类构造方法
子类构造方法
```

在例 5-8 展示了父类和子类的静态代码块、构造方法在类加载和对象创建时的执行顺序。具体执行流程如下：

(1) 首先，当程序执行时，会加载 Child 类。在加载 Child 类的过程中：

执行父类 Parent 的静态代码块，输出“父类静态代码块”。

执行子类 Child 的第一个静态代码块，输出“子类静态代码块 1”。

执行子类 Child 的第二个静态代码块，输出“子类静态代码块 2”。

(2) 接着，创建第一个 Child 对象 c1。在创建对象的过程中：

调用父类 Parent 的构造方法，输出“父类构造方法”。

调用子类 Child 的构造方法，输出“子类构造方法”。

(3) 同样地，创建第二个 Child 对象 c2。由于类已经加载过一次，静态代码块只会在类加载时执行一次，所以在创建第二个对象时：

仅调用父类 Parent 的构造方法，输出“父类构造方法”。

调用子类 Child 的构造方法，输出“子类构造方法”。

5.6 访问权限修饰符

在 Java 中，访问控制修饰符决定了其他类对该类及其字段(成员变量)和方法的可访问性。在 Java 中，有 4 种不同的访问权限修饰符：private、public、default(无修饰符)和 protected。

1. 类成员前的权限修饰符

1) private

当一个成员变量被声明为 private，意味着它只能被所属类的方法访问，而不能被该类的实例对象或其他类直接访问。这种私有性保证了数据的封装性和安全性，防止数据被不相关的类直接修改。如果外部的类需要访问这个私有成员变量的值，通常需要提供一些公共的方法，比如 getter()和 setter()方法。通过 getter()方法，外部类可以获取私有成员变量的值；而通过 setter()方法，外部类可以修改私有成员变量的值。这种间接的访问方式不仅保护了数据的完整性，还使得类的实现细节对外部类是隐藏的，提高了代码的封装性和灵活性。

【例 5-9】 编写一个 Order 订单类，其中包含两个私有成员变量 orderId(订单 ID)和 customerName(客户姓名)，并提供对应的公共方法 setter()和 getter()用于修改和访问 orderId 和 customerName。在 main()方法中创建一个 Order 订单类对象进行测试。

【程序实现】

```
package chapter05;
class Order {
    private String orderId;                    //订单 ID
    private String customerName;               //客户姓名

    //setter 方法,用于修改私有成员变量
    public void setOrderId(String orderId) {
        this.orderId = orderId;
    }
    public void setCustomerName(String customerName) {
        this.customerName = customerName;
    }

    //getter 方法,用于访问私有成员变量
    public String getOrderId() {
        return orderId;
    }
    public String getCustomerName() {
        return customerName;
    }
}
public class Test5_9 {
    //主方法测试
    public static void main(String[] args) {
        Order order = new Order();
        //使用 setter 方法设置 orderId 和 customerName
        order.setOrderId("20240101");
        order.setCustomerName("张三");
        //使用 getter 方法打印 orderId 和 customerName
        System.out.println("订单 ID: " + order.getOrderId());
        System.out.println("客户姓名: " + order.getCustomerName());
    }
}
```

【运行结果】

```
订单 ID: 20240101
客户姓名: 张三
```

在例 5-9 中,orderId 和 customerName 都被标记为 private,表示它们只能在 Order 这个类的内部被访问或修改,其他类,如主类或其他任何类都无法直接访问或修改它们。而使用公共的 getter()和 setter()方法提供对它们的访问或修改,这就是封装。

这样做的好处是可以在这些方法中加入对数据的检查或处理逻辑,比如在设置 customerName 的时候,可以检查输入的名字是否非空,或者在获取 orderId 的时候,可以检查 orderId 是否一直存在等。这就提供了一个控制和保护成员变量的独立接口。

另外当 private 修饰成员方法的时候,该成员方法只能被该类的其他方法调用,而不能被该类的实例或者其他类调用。例如,在类内部,经常会有一些辅助性的功能方法,这些方法只在类内部使用,就很适合声明为 private。被 private 修饰的方法,只能在类的内部使用。

2）public

使用public修饰成员数据时，其主要特点在于其具有广泛的可见性和可访问性。public修饰的成员数据对于任何类都是可见的，无论是在同一个包中还是不同的包中的类都可以直接访问它。这意味着任何类都可以直接访问和修改public修饰的成员数据的值，而不受任何访问限制。然而，这种公开的成员数据不利于类的封装，因为它们暴露了类的内部实现细节，使得类的行为不可控，数据的安全性难以得到保障。因此，在实际开发中，通常不建议直接使用public修饰成员数据，而是通过提供公共的访问方法（如getter()和setter()方法）来间接访问和修改成员数据，以实现类的封装和数据的安全性。

3）default（无修饰符）

成员数据前无修饰符时，这意味着该成员数据对于同一个包中的其他类是可见的，但对于不同包中的类是不可见的。无修饰符的成员数据可以在同一个包中的其他类中直接访问和修改，但对于不同包中的类则无法直接访问。这种访问级别提供了一定程度的封装性，可以限制对成员数据的直接访问，使得类的实现细节对于其他包中的类是隐藏的，提高了类的封装性和安全性。然而，需要注意的是，虽然无修饰符的成员数据对于不同包中的类是不可见的，但在同一个包中的其他类仍然可以直接访问和修改它们，因此在一些情况下可能需要额外的访问控制。

4）protected

protected修饰的成员数据对于同一个包中的其他类和该类的子类是可见的，而对于不同包中的类则是不可见的。这意味着protected修饰的成员数据可以在同一个包中的其他类和该类的子类中直接访问和修改，而对于不同包中的类则只能在该类的子类中被访问，而不能在其他类中直接访问。这种访问级别提供了一定程度的封装性和继承性，可以限制对成员数据的直接访问，并且允许子类继承并修改它们。因此，使用protected修饰的成员数据通常用于需要在子类中访问和修改，但不希望被其他类直接访问的情况。

2. 类前的权限修饰符

在Java中，类前的权限修饰符可以是public或default（无修饰符）两种。

public修饰符，当一个类使用public修饰时，表示该类对于任何其他类都是可见的。这意味着该类可以被同一个包中的其他类和不同包中的类直接访问和实例化。使用public修饰的类通常用于需要在整个程序中都能够被访问和使用的情况。

default（无修饰符），如果一个类没有使用任何权限修饰符，则默认的访问级别为包私有（package-private）。这意味着该类只能在同一个包中的其他类中被访问和实例化，而对于不同包中的类则是不可见的。默认修饰的类通常用于需要在同一个包中内部使用，而不希望被其他包中的类直接访问的情况。

注意：

在实际编程中，选择合适的权限修饰符是至关重要的，因为它直接影响到代码的封装性、可维护性和安全性。一般而言应该遵循最小化暴露原则，即将类、成员变量和方法的访问权限设为最小化，只允许必要的类和方法可以访问它们，以降低代码的耦合度。

通常情况下，成员变量应该被设为私有，通过公共的访问方法来间接访问和修改其值，以保护数据的完整性。如果一个类希望子类能够访问某些成员，但不希望被其他类直接访

问，则可以使用 protected 修饰符。同时，公共接口应该使用 public 修饰符，使得其他类可以直接访问和使用，提高代码的易读性和可维护性。

对于只在同一个包中使用的类和方法，可以使用默认（无修饰符）来限制其可见性，避免不必要的暴露，保持包的内部实现细节对其他包的隐藏。综上所述，选择合适的权限修饰符需要根据具体的设计需求和封装性考虑，以保证代码的安全性、可维护性和可扩展性。表 5-1 总结了 4 种访问权限修饰符的作用范围。

表 5-1 访问权限修饰符的作用范围

权限修饰符	同一个类	同一个包中的类	不同包中的子类	不同包中的非子类
public	√	√	√	√
protected	√	√	√	×
default	√	√	×	×
private	√	×	×	×

5.7 任务实现

本章主要任务是编写一个简单的图书管理系统。在该系统中，需要定义至少两个类：图书类和图书管理类。图书类包含标题、作者、出版日期等属性，并能够实现对图书的添加、查询、删除、更新等基本功能。考查对面向对象编程的理解和应用，特别是如何定义类，创建对象，设置属性和方法，以及如何通过对象访问这些属性和方法。其次，题目考察的是对数组的使用，包括如何初始化，以及如何进行增加、删除、修改和查找操作。此外，对于如何使用 getter()和 setter()方法来享有封装原则的理解也是题目的考查点。同时，题目还涉及了如何重写方法，特别是 toString()方法以便更自然地显示对象信息。最后，题目也涵盖了一些基本编程的构造，如条件语句，循环语句以及常用的打印输出等。总的来说，这是一道涉及面向对象编程基础，数组操作，封装原则，方法重写以及基本编程构造的综合任务。具体的实现思路如下。

【实现思路】

（1）设计 Book 类，将每本书作为一个对象，需要确定一个书的关键属性（书名、作者名、出版日期、出版社）。挑战在于选择正确的数据类型以准确地表示这些属性，使用 Java 的 String 数据类型可以解决。

（2）实现 Book 类的构造函数，需要正确初始化所有属性。这可能需要理解 Java 构造函数的工作方式，通过参数列表和 this 关键字可以解决。

（3）在 Book 类中实现 getter()和 setter()方法，对于每个属性需要一个 getter()方法获取属性值，以及一个 setter()方法修改属性值。关键在于理解 Java 的封装原则和 getter()及 setter()的作用，通过 Java 的方法调用可以解决。

（4）重写 Book 类的 toString()方法，这是为了能更自然地表示图书对象的信息，需要理解 Java 的 Object 类、toString()方法以及如何重写方法，通过字符串拼接（或者使用 StringBuilder/StringBuffer）可以解决。

（5）设计 Library 类，这是在理想的情况下管理所有的图书，挑战在于选定一个合适的

数据结构来存储和处理图书信息，这里用到了数组数据结构来解决。

（6）在 Library 类中实现添加书、移除书、查找书、更新书信息和显示所有书的功能，关键在于需要理解数组操作，特别是插入和删除元素时的索引移动。这需要理解 Java 的循环结构、数组操作以及条件判断，通过对数组索引的操作可以解决。

（7）创建 Task05 类用于测试，需要创建图书和图书馆的对象，然后调用 Library 类的方法进行测试。挑战在于理解如何创建对象，如何调用方法，以及如何进行有效的测试，通过实例化对象，调用方法，以及基本的打印输出可以解决。

【程序实现】

（1）定义一个名为 Book 的 Java 类，包含 4 个私有字段：书名 title、作者名 author、出版日期 publicationDate 和出版社 publisher。Book 类还包含一个构造函数，用于在创建 Book 对象时初始化这些字段的值。此外，该类为每个字段提供了对应的 getter()和 setter()方法，以便于访问和修改这些字段的值。类中还重写了 toString()方法，用于返回包含书对象所有字段信息的字符串表示，以便于打印或展示书的信息。

```
package chapter05;
public class Book {
        //定义了私有的字段:书名,作者名,出版日期,出版社
        private String title;
        private String author;
        private String publicationDate;
        private String publisher;

        //构造函数,创建图书对象时初始化属性值
        public Book(String title, String author, String publicationDate, String publisher) {
            this.title = title;
            this.author = author;
            this.publicationDate = publicationDate;
            this.publisher = publisher;
        }

        //对应的 getter 和 setter 方法
        public String getTitle() {
            return title;
        }

        public void setTitle(String title) {
            this.title = title;
        }

        public String getAuthor() {
            return author;
        }

        public void setAuthor(String author) {
            this.author = author;
        }

        public String getPublicationDate() {
            return publicationDate;
        }
        public void setPublicationDate(String publicationDate) {
```

```
        this.publicationDate = publicationDate;
    }

    public String getPublisher() {
        return publisher;
    }

    public void setPublisher(String publisher) {
        this.publisher = publisher;
    }

    //重写 toString() 方法,用于更自然地表示图书对象的信息

    public String toString() {
        return "Book{" +
            "title='" + title + '\'' +
            ", author='" + author + '\'' +
            ", publicationDate='" + publicationDate + '\'' +
            ", publisher='" + publisher + '\'' +
            '}';
    }
}
```

(2) 定义一个 Library 类来模拟图书馆管理,包括添加、删除、查找、更新图书以及展示所有图书的功能。图书信息由 Book 类表示,包含书名、作者、出版日期和出版社。在 Task05 主类中,使用 Library 类操作图书:创建一个容量为 10 的图书馆,添加三本书籍,显示所有书籍,查找特定书籍,更新一本书的内容,删除一本书,最后再次展示更新后的图书列表。整个程序展示了对图书馆藏书的基本增删查改功能。

```
package chapter05;
class Library {
    private final Book[] books;                    //存储图书信息的数组
    private int size;                              //记录当前数组中存在的图书数量

    //构造函数,创建图书馆对象时初始化图书存储数组的大小
    public Library(int capacity) {
        this.books = new Book[capacity];
        this.size = 0;
    }

    //添加图书的方法,将一个图书对象添加到 books 数组中,并将 size 加 1
    public void addBook(Book book) {
        if (size < books.length) {
            books[size++] = book;
        } else {
            System.out.println("图书馆已满,无法添加新书!");
        }
    }

    //删除图书的方法,根据书名搜索并将找到的书删除,之后的图书向前移动一位,size - 1
    public void removeBook(String title) {
        for (int i = 0; i < size; i++) {
            if (books[i].getTitle().equals(title)) {
                for (int j = i; j < size - 1; j++) {
```

```
                    books[j] = books[j + 1];
                }
                books[--size] = null;
                System.out.println("成功删除书籍:" + title);
                return;
            }
        }

        System.out.println("未找到要删除的书籍:" + title);
    }

    //查找图书的方法,根据书名搜索并返回找到的图书的信息,如果没有找到则返回 null
    public Book findBook(String title) {
        for (int i = 0; i < size; i++) {
            if (books[i].getTitle().equals(title)) {
                return books[i];
            }
        }
        return null;
    }

    //更新图书的方法,根据旧的书名查找 books 数组并用新的图书信息替换它
    public void updateBook(String oldTitle, Book newBook) {
        for (int i = 0; i < size; i++) {
            if (books[i].getTitle().equals(oldTitle)) {
                books[i] = newBook;
                System.out.println("成功更新书籍信息:" + newBook.getTitle());
                return;
            }
        }
        System.out.println("未找到要更新的书籍:" + oldTitle);
    }

    //打印所有图书的方法,遍历 books 数组并打印每本图书的信息
    public void displayAllBooks() {
        if (size == 0) {
            System.out.println("图书馆中暂无书籍!");
            return;
        }

        System.out.println("图书馆中所有书籍的信息:");
        for (int i = 0; i < size; i++) {
            System.out.println(books[i]);
        }
    }
}

public class Task05 {
    public static void main(String[] args) {
        Library library = new Library(10);        //创建一个可以容纳 10 本书的图书馆
        //创建三本图书
        Book book1 = new Book("Java 设计指南", "张三", "2024-01-01", "出版社 A");
        Book book2 = new Book("Python 编程基础", "李四", "2024-02-01", "出版社 B");
        Book book3 = new Book("C++实战指南", "王五", "2024-03-01", "出版社 C");
```

```
            library.addBook(book1);
                library.addBook(book2);
                library.addBook(book3);
                library.displayAllBooks();              //显示图书馆中的所有图书
                System.out.println("书籍查找成功:" + library.findBook("Java设计指南"));
                Book updatedBook = new Book("Java设计指南(更新版)", "张三", "2024-01-01",
    "出版社A");
                library.updateBook("Java设计指南", updatedBook);   //更新一本图书
                library.removeBook("C++实战指南");                  //移除一本图书
                library.displayAllBooks();            //再次显示图书馆中的所有图书
            }
        }
```

对该图书管理系统相关操作测试结果如图 5-2 所示，首先创建了一个图书馆对象，这个图书馆最多可以容纳 10 本书。接着创建了三本不同的图书，每本书都有书名、作者、出版日期和出版社等属性。然后把这三本书添加到图书馆中。添加完图书后，要求图书馆显示当前所有的图书，这样就可以看到图书馆里有哪些书。接下来尝试在图书馆中查找一本名为《Java 设计指南》的书，并打印出查找的结果。之后，决定更新这本《Java 设计指南》的图书信息。创建一个新的图书对象，表示这本书的更新版，并告诉图书馆用新的图书信息替换旧的《Java 设计指南》。更新完图书后，又从图书馆中移除了一本名为《C++实战指南》的书。最后再次要求图书馆显示当前所有的图书，这样可以看到图书馆经过添加、更新和删除操作后的图书列表。

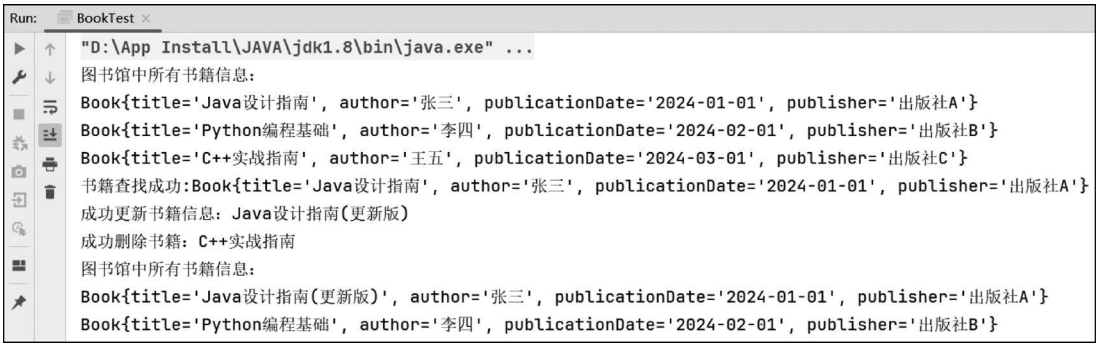

图 5-2　图书管理系统相关操作测试结果

本章小结

本章中，探讨了面向对象编程(OOP)的基础概念，讲述了类和对象的核心概念，围绕着面向对象程序设计的方法进行讲解。明确了面向对象的基本概念和特征，强调面向对象程序设计的封装性。讲解了类的定义与对象的创建。通过具体的示例说明类的定义格式，包括属性和方法的声明等，并介绍了构造方法的定义、作用和使用方式。在讨论对象的部分，重点解释了对象如何创建以及如何访问对象的属性和方法，并解析了对象引用的概念。描述了在 Java 语言中，所有的对象都是通过引用进行操作的。讲解了 this 关键字的用法和含义，它表示当前对象的引用，用来解决局部变量和成员变量之间的命名冲突。详述了 static

关键字在Java语言中的重要角色和用法，包括静态变量、静态方法和静态代码块。强调了静态成员属于整个类，而不是某个具体的对象。详细介绍了访问权限修饰符，解释了public、protected、private和默认访问权限的区别及适用范围。最后，在任务实现中结合了前面讲解的各种概念和理论，通过实际的代码示例，加深学习者对面向对象程序设计的理解和应用。本章全面而深入地介绍了面向对象程序设计的基础知识，为后面的学习打下了坚实的基础。

习题5

一、单项选择题

1. 面向对象编程的三个基本特性是(　　)。

A. 封装、继承和多态　　B. 封装、继承和抽象

C. 封装、抽象和静态　　D. 封装、多态和动态

2. 下列(　　)关键字用于声明类的属性或方法是只属于类，而不属于任何一个特定的对象。

A. final　　B. static　　C. this　　D. super

3. 在Java中可以使用(　　)关键字创建一个新对象。

A. new　　B. this　　C. super　　D. static

4. 在Java语言中，表示当前对象引用的关键字是(　　)。

A. super　　B. this　　C. static　　D. final

5. 在Java中，(　　)访问权限修饰符表示只有同一个类的成员可以访问。

A. public　　B. protected　　C. default　　D. private

二、简答题

1. 什么是面向对象程序设计的基本特征？
2. 类的定义中都包括哪些部分？
3. 在Java中如何创建一个新对象？
4. 在Java中如何通过对象来访问类的属性和方法？
5. Java中的“this”关键字是什么？
6. 在Java中什么是静态变量，它和普通变量有何不同？
7. 静态方法和非静态方法有什么区别？
8. 静态代码块的作用是什么？
9. 什么是访问权限修饰符，Java中有哪些访问权限修饰符？
10. 简述public、protected、private和默认访问权限的区别和作用范围。

三、上机题

1. 设计一个名为Circle的类来表示一个圆形。这个类应该包含一个表示半径的字段，一个计算面积的方法和一个计算周长的方法，并创建测试类进行测试。

2. 编写一个Student类，它具有4个字段：name(名字)、age(年龄)、grade(年级)和score(分数)。Student类应该有一个toString()方法来显示学生的详情。在测试类创建几个Student对象，并显示它们的详情。

3. 编写一个名为 MyMath 的类，该类包含一个静态方法 square()，该方法接收一个 int 类型的参数并返回它的平方，并创建测试类进行测试。

4. 创建一个名为 BankAccount 的类，它包含 accountNumber(账户号)和 balance(余额)两个属性，以及两个方法 deposit()和 withdraw()。这两个方法都应接收一个 double 类型的参数，并相应地改变 balance 的值。然后创建一个测试类，在测试类中创建一个 BankAccount 对象并进行存款和取款操作。