第3章 面向对象的思想

本章重点

- 面向对象的概念。
- 类与对象。
- 构造方法。
- 多态。
- 接口。

面向对象的思想，简称面向对象（Object Oriented，OO），是一种基于对象的编程思想，其中对象指的是数据和方法的结合体。这种编程思想将事物抽象为一个个独立的、有特定功能的对象，并通过定义对象之间的关系和交互来实现复杂的功能。面向对象的思想可以提高代码的可读性、可维护性和可复用性，具有重要的编程思想和实践价值。它在编程语言的设计和开发中被广泛使用，例如 Java、Python、C++等。

面向对象程序设计具有如下特点。

1. 封装性

封装是面向对象的核心思想，它有两层含义：一是指把对象的属性和行为看成一个密不可分的整体，将这两者“密封且装在一起”（即封装在对象中）；二是“信息隐藏”，将不想让外界知道的信息隐藏起来。例如，驾校的学员学开车，只需要知道如何操作汽车，无须知道汽车内部是如何工作的。

2. 继承性

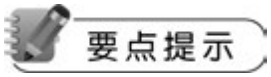
继承性主要描述的是类与类之间的关系。通过继承，可以在无须重新编写原有类的情况下，对原有类的功能进行扩展。例如，有一个汽车类，该类描述了汽车的普遍特性和功能。进一步再产生轿车类，而轿车类中不仅应该包含汽车的特性和功能，还应该增加轿车特有的功能，这时，可以让轿车类继承汽车类，在轿车类中单独添加轿车特性和方法就可以了。继承不仅增强了代码的复用性、提高了开发效率，还降低了程序产生错误的可能性，为程序的维护以及扩展提供了便利。

3. 多态性

多态性指的是在一个类中定义的属性和方法被其他类继承后，它们可以具有不同的数据类型或表现出不同的行为，这使得同一个属性和方法在不同的类中具有不同的语义。例如，当听到 Cut 这个单词时，理发师的行为表现是剪发，演员的行为表现是停止表演，不同的对象所表现的行为是不一样的。多态的特性使程序更抽象、便捷，有助于开发人员设计程序时分组协同开发。

思考：结合其他高级语言，能否阐述面向过程编程与面向对象编程的不同？

3.1　什么是类和对象



类将实现的细节进行封装，通过该类的对象实现使用这个类。

在 Java 中，类(Class)是一种用于封装数据和方法的基本数据结构和编程概念，它是面向对象编程的核心。类是一类事物的统称，是创建对象的模板，通过类来创建的对象具有特定的状态和行为。类是事物的抽象，对象则是事物的实体，可以将类比作生产汽车之前先拟定的草图，将对象比作生产线上生产出来的汽车。

Java 中的类是面向对象的基础，在 Java 中所有的对象在创建前需要先定义类。类是封装对象的属性和行为的载体，在进行类定义时，需要抽象出事物所具有的相同属性和行为，因此，类是面向对象编程的基础和关键。

对象则是抽象的一类事物的具体实体，如通常描述人类时会描述人具备的高矮、性别、年龄等属性以及人类所共有的行为，如说话、行走等。具体的一个人其自身的属性和行为是其自身特有的，也是可以根据人的具体差异来进行对象区分的关键。

在 Java 语言中，类中对象的属性是以成员变量的形式定义的，对象的行为是以方法的形式定义的。对于类和对象的理解将有助于对面向对象思想的学习。

3.2　类的定义

在面向对象的编程思想中最核心的就是对象，为了能够创建对象，需要先定义类，类可以理解成创建对象的模板，类中通常定义有成员变量和成员方法，其中的成员变量指的是对象具有的特性，也叫属性。成员方法代表对象具备的行为简称方法。

▶ 3.2.1　类定义一般格式

在 Java 中类定义基本语法格式如下：

```
[可访问性修饰符] [修饰符]class 类名{
        成员变量;
        成员方法;
}
```

类定义时，需要注意以下几点。

(1) 类名：不能使用 Java 中的关键字、保留字，不能以数字开头。首字母应该大写，后面的每个单词首字母大写。同样，成员变量名、成员方法名也需要满足标识符规则，取名尽量做到见名知义。

(2) 成员变量：定义在类中成员方法外的变量，一般用来描述一类实体具有的共同的属性。定义形式为：

```
[可访问性修饰符] [修饰符] 数据类型 属性名;
```

(3) 成员方法：用来描述实体具有的行为。成员方法的定义一般形式为：

```
[可访问性修饰符] [修饰符] 返回类型 方法名(参数){
    //成员方法实现
}
```

接下来通过一个示例演示如何定义类。

```
class Student{//定义类需要使用关键字 class,在这里定义一个 Student(学生)类
    //定义一个 String 类型的变量 name,表示每个学生都有姓名
    String name;
    //定义一个 int 类型的变量 age,表示每个学生都有年龄
    int age;
    //定义一个关于 eat 的方法
    public void eat(){
        System.out.println("学生爱吃汉堡");
    }
    //定义一个关于学习的方法,可以在方法中访问成员变量 name
    public void study(int hour){
        System.out.println(name + "同学每天需要学习" + hour + "个小时");
    }
}
```

上述代码中定义一个 Student 类,类中定义了两个属性——name 和 age,并且定义了两个方法——eat()和 study()。

小脑袋大问号

学弟:学姐,定义类的都是 class XXX{}吗?

学姐:对的。类的定义格式为"修饰符 class 类名{}"。上面的代码在定义类时只有 class Student{},前面没有修饰符的原因是它使用了默认的修饰符 default,以后还会看见别人定义类时写成 public class XXX{}。

类名之前介绍过,不能使用 Java 中的关键字、保留字、不能以数字开头。首字母应该大写,以后的每个单词首字母大写。当然这里写成 class student{}也不会报错,但是不够规范。

学弟:第 2 行和第 3 行为属性定义,是吗?

学姐:是的。属性表示对象有什么,这里表示每个 student(学生)都有 name(姓名)和 age(年龄)。

第 4 行和第 7 行都是方法的定义,方法可以理解成对象具备的行为。这里代表学生 eat(吃饭)和 study(学习)的行为。

学弟:定义方法时只有 public 这一种修饰符可以选择吗?

学姐:当然不是,后面还会介绍其他修饰符,现在暂且使用 public,表示公共的,它是最宽松的一种访问权限。

学弟:方法名是不是随便定义一个单词就可以?

学姐:虽然方法名定义比较宽泛,但是依旧不能使用 Java 中的保留字、关键字,不能以数字开头,尽量使用一些比较具有代表意义的单词。而且首字母小写,以后的每个单词首字母大写。

注意:类的定义声明、成员变量声明与成员方法声明均存在[可访问性修饰符][修饰符],但是有些关键字只能用在特定声明中。

▶ 3.2.2 成员变量

在 Java 中,对象的属性被称为成员变量,或称为属性。成员变量定义的一般形式为:

[可访问性修饰符][修饰符] 数据类型 属性名;

成员变量定义时可以通过修饰符来限定其访问控制权限，如，若修饰符为 private，则限定该成员变量仅限于在类的内部使用(关于权限修饰符的说明会在 3.9 节中进行介绍)。成员变量的数据类型可以设置为 Java 语言中的合法数据类型，既可以是基本数据类型，也可以是引用数据类型。成员变量可以在定义时设置初始值，也可以不设置初始值，如果不设置初始值，则系统会提供默认值。通常不建议在类中直接给属性赋值。

▶ 3.2.3　成员方法

在 Java 中，成员方法定义的一般形式如下：

```
[可访问性修饰符][修饰符] 返回类型 方法名(参数){
    //成员方法实现
}
```

成员方法通过修饰符来限定成员方法的访问控制权限。成员方法定义时的返回类型表示方法的返回值类型，方法参数表示方法的输入参数。

1. 成员方法返回类型

返回类型用于指明该成员方法执行完返回给调用者结果的数据类型，一般与方法体中 return 关键字配合使用。在 Java 中一个方法可以有以下几种返回类型。

void：表示没有返回类型。

基本数据类型(例如 int、char、double 等)：表示方法返回的是基本数据类型。

引用数据类型(例如 String、自定义类等)：表示方法返回的是对象的引用。

例如，想要封装一个方法得到一个随机的数字，范围是 0～99，代码如下：

```
//修饰符 返回类型 方法名(参数){}
public int getNum(){
    //Math.random():表示系统生成一个随机数,取值范围为[0.0,1.0)
    //(int)(Math.random() * 100)生成的随机数,取值范围为[0,99]
    return (int)(Math.random() * 100);
}
```

上述代码可以获得 0～99 的一个随机数，该方法的返回类型定义为 int 类型，表示方法执行完会返回一个整数类型的随机值。Math.random()相关内容后面章节会进行详细介绍。

2. 成员方法参数

方法参数指的是方法执行的时候需要输入的数据，通俗易懂的解释就是，在执行这个功能时需要调用者提供什么类型的参数。

1) 参数的类型

在 Java 中，方法的参数可以有以下几种类型。

void 或直接为空：方法执行时无须提供任何参数。

基本数据类型参数：方法执行时需要提供基本数据类型的数据。

引用数据类型参数：方法执行时需要提供引用数据类型的数据。

例如，想要封装一个方法用于得到两个数字的和。

```
public void sum(int x,int y){
    System.out.println(x + y);
}
```

上述定义的方法在调用时需要传入两个 int 类型的参数，x 和 y 接收输入的两个 int 类型的值，接着进行求和并打印。

2）形参与实参

在程序设计中，方法的参数分为形参（即形式参数）和实参（即实际参数）。形参是定义在方法声明中的变量，它们用于接收外部传入的数据，是方法中的一种局部变量，仅在方法内部使用，方法执行完局部变量消亡。实参是调用方法时传递给方法的具体参数值，实参可以是常量、变量、表达式，以及其他方法的返回值等。

下面通过一个简单的例子来理解形参和实参。

```
public class Test{
    public static void sum(int x,int y){     //x,y是方法的形参
        System.out.println(x + "+" + y + "=" + (x+y));
    }
    public static void main(String[] args){
        int num1 = 45;
        int num2 = 90;
        sum(num1,num2);                      //num1 和 num2 是 sum()方法调用传入的实参
        String a = "O";
        String b = "K";
        //sum(a,b);        //报错,因为 sum()需要 int 类型的形参,实参传入的是 String 类型
    }
}
```

在上面的例子中，sum()方法的形参是 x 和 y，都是 int 类型。在 main()方法中，定义了两个变量 num1 和 num2，它们实际上就是传递给 sum()方法的实参。在调用 sum()方法时，将 num1 和 num2 作为参数传递给 sum()方法，在 sum()方法内部，x 和 y 就接收到传入的实参，然后进行求和运算。

需要注意的是，实参必须与形参的类型和顺序一一对应，否则编译器会报错。例如，上述代码如果将 a 和 b 定义为字符串类型，并尝试将它们作为参数传递给 sum()方法，编译器会报类型错误。

总之，形参和实参是方法中非常重要的概念，它们用于在方法中接收数据和传递数据，合理使用形参和实参可以提高程序的效率和可扩展性。

3）参数的传递

在 Java 中，参数传递可以通过值传递或引用传递实现。在值传递中，传递的是变量的值，而在引用传递中，传递的是变量的引用或地址。

（1）值传递。

值传递是指将实际参数的值复制一份传递给形参，也就是函数接收的参数。这样，如果形参的值被修改，不会影响实参的值。在 Java 中，所有基本数据类型都是通过值传递进行参数传递的。例如：

```
public static void main(String[] args) {
    int num = 10;
    changeValue(num);
    System.out.println(num);            //输出 10
}
public static void changeValue(int value) {
    value = 20;
}
```

在这个例子中，changeValue()方法接收一个整数类型的参数，并将其值更改为 20。然而，在 main()方法中，变量 num 的值仍然是 10，因为在调用 changeValue()方法时，实参的值被复制并传递给了形参，而不是直接传递实参本身。

下面通过内存图分析值传递的过程，如图 3.1 所示。

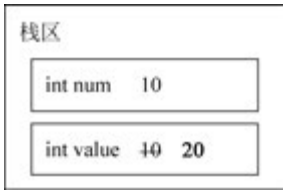


图 3.1　通过内存图分析值传递的过程

程序中变量 num 和变量 value 声明时会在栈区分别分配空间，在发生函数调用 changeValue(num)时，会实现将 num 的值 10 赋值给变量 value，即 value 初始值为 10，在 changeValue()方法中，将 value 值修改为 20，但由于和 num 变量属于不同的空间，值的修改并不会影响到 num，且 value 的内存空间随着函数调用的结束会被释放，因此，输出 num 的值依然会是 10。

(2) 引用传递。

引用传递是指将实参的引用或地址传递给形参。这样，如果形参的值被修改，实参的值也会被修改。在 Java 中，所有引用数据类型都是通过引用传递进行参数传递的。例如：

```
public static void main(String[] args) {
    Student s = new Student("张三",30);
    changeAge(s);
    System.out.println(s.getName() + "" + s.getAge());      //输出:张三 33
}

public static void changeAge(Student stu) {
    // Student stu = s;                                      //隐式地赋值
    stu.age = 33;
}
```

在这个例子中，changeAge()方法接收一个 Student 类型的参数，并将该对象的年龄修改成 33。在 main()方法中，学生对象的年龄被修改成 33，因为在调用 changeAge()方法时，实参 s 的引用被传递给了形参 stu，并在 changeAge()方法中修改了其对象的属性值。

下面通过内存图的形式分析引用传递的过程，如图 3.2 所示。

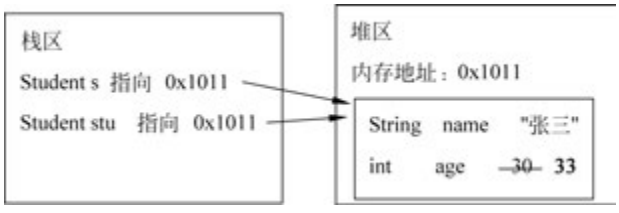


图 3.2　通过内存图的形式分析引用传递的过程

需要注意的是，在 Java 中，虽然对象是通过引用传递进行参数传递的，但对象的引用本身也是通过值传递进行的。也就是说，当一个对象被传递给一个方法时，实际上是将对象的引用复制一份传递给了形参，而不是将对象本身复制一份传递给了形参。因此，如果在方法中修改了对象的属性，这些修改会影响到原始对象。

▶ 3.2.4　可访问性修饰符

在很多情况下，我们需要一定的封装性实现信息隐藏，通过不同的可访问性修饰符指定可访问性，详细说明参照表 3.1。

表 3.1　可访问性修饰符

可访问性修饰符	当前类	同一包内	子孙类(同包)	子孙类(不同包)	其　他
public	√	√	√	√	√
protected	√	√	√	√	×

续表

可访问性修饰符	当前类	同一包内	子孙类(同包)	子孙类(不同包)	其　他
default	√	√	√	×	×
private	√	×	×	×	×

对于访问权限来说,public 具有最大的可被访问性,可以在任何包中访问;private 只能在本类中访问,子类都不行;protected 是在当前类、同一包内和子类中均可以访问;default(可以省略)在同一个包中可以访问,如表 3.2 所示。

表 3.2　可访问性修饰符的作用对象

可访问性修饰符	类	成员变量	成员方法
public	√	√	√
protected	×	√	√
default	√	√	√
private	×	√	√

类只能适用于 public 与 default 关键字,成员变量与成员方法可以使用全部的可访问性修饰符。

▶ 3.2.5　常见的修饰符

1. static 关键字

在 Java 中,static 是一种修饰符,表示“静态的”。它可以用来修饰属性、方法和代码块。它的主要作用是使它们在内存中只存在一份,而不是每次创建对象都会重新分配空间,从而节省内存和提高程序的性能。

1) 静态属性

在类的内部,使用 static 修饰的属性称为静态属性或类变量。它们属于整个类,而不是依赖某个对象,因此只要类被加载,它们就会被初始化,无论是否创建了类的实例。静态属性可以通过类名直接访问,也可以通过对象名访问,但通常建议使用类名来访问。

在类的内部,如果一个属性没有用 static 修饰,那么叫普通属性或者成员变量。成员变量是依赖于对象而存在的,如果想要访问成员变量必须通过对象名访问。

下面通过一段代码演示普通属性和静态属性的调用区别。

```
class Student{
    String name;              //普通属性,表示每个对象都有一份的属性,需要对象.属性
    static String country;    //静态属性,表示整个类型共享一份属性,需要类名.静态属性
}
class TestStatic{
    public static void main(String[] args){
        Student s1 = new Student();
        s1.name = "张三";
        //s1.country = "中国";"对象.静态属性"的调用方式也是可以的,但是尽量使用"类名.静态
属性"
        Student.country = "中国";
        Student s2 = new Student();
        s2.name = "李四";
        System.out.println(s1.name + "  " + Student.country);
        System.out.println(s2.name + "  " + Student.country);
    }
}
```

运行结果：

```
张三　　中国
李四　　中国
```

上述代码中，Student 类中 name 属性没有使用 static 修饰，代表该属性是一个对象都有一份的属性，所以在 TestStatic 类中修改 s1 的 name 属性值和 s2 的 name 属性值没有任何关系，需要通过“对象.属性”才能访问。Student 类中 country 属性使用 static 修饰，代表该属性整个类型所有对象共享一份的属性，整个内存只开辟一块内存空间，不依赖于 s1 或者 s2 中的任何一个对象，需要通过“类名.静态属性”才能访问。

2）静态方法

在类的内部，使用 static 修饰的方法称为静态方法或类方法。它们属于整个类，而不是某个对象，因此可以直接通过类名调用。静态方法不能直接访问非静态变量和非静态方法，因为它们没有隶属于任何对象。通常，静态方法用于执行与类相关的操作，例如计算数学函数、操作静态变量等。

下面通过一段代码演示静态方法和实例方法的区别。

```
class A{
    int x = 34;
    static int y = 90;
    public void test(){
        System.out.println(x);                    //等价于 this.x
        System.out.println(y);
    }
    public static void show(){
        //System.out.println(x);报错,因为可能不存在 A 类对象
        //可以修改成
        //A a = new A();
        //System.out.println(a.x);                //通过
        System.out.println(y);
    }
}
class TestStatic{
    public static void main(String[] args){
        A a = new A();
        a.test();
        A.show();
    }
}
```

运行结果：

```
34
90
90
```

上述代码中，A 类定义一个普通属性 x 和一个静态属性 y。普通属性 x 依赖于对象而存在，必须通过“对象.属性”才能访问，静态属性 y 依赖于类而存在，有没有对象都可以访问。

test()方法由于没有用 static 修饰属于成员方法，因此调用时必须通过“对象.普通方法”才能访问，如果 test()方法被正常调用，则说明内存中已经有 A 类对象了，所以在第一行访问 x 是正确的，这里打印 x 其实等价于 this.x。y 有没有对象都可以访问，所以访问 y 通过。

show()方法由于使用 static 修饰，因此调用时需要通过“类名.静态方法”或者“对象.静态

方法”才能访问。如果 show()方法被正常调用并且代码执行到方法体里面时不能保证内存里面一定有 A 类对象，那么在 show()方法的第一行访问 x 时就会报错，因为 x 需要依赖于 A 类对象才能访问，为了能够在静态方法中访问非静态成员，也可以通过创建 A 类对象去访问，y 变量有没有 A 类对象都可以访问。

3）静态代码块

在类中可以使用静态代码块，使用 static 修饰。它们只在类加载时执行，且只会执行一次，用于执行一些静态初始化操作，例如给静态变量赋初值。

```
class MyClass{
    static int num;
    static{                          //静态代码块
        num = 10;
    }
}
```

在这个示例中，当 MyClass 类被加载时，静态代码块会被执行，num 变量会被赋值为 10。

需要注意的是，静态变量和静态方法在内存中只存在一份，因此它们是共享的，可以被多个对象访问。同时，静态变量和静态方法不依赖于对象的创建，因此它们可以在任何时候被访问，即使没有创建任何对象。因此，静态变量和静态方法可以用来实现一些全局的或与类相关的操作。

2．final 关键字

final 是 Java 中的关键字，表示“最终的”。在 Java 中可以使用 final 修饰符类、方法、变量。它的作用有以下几种。

（1）final 修饰类：叫作最终类，不能有子类，但是可以有父类。

（2）final 修饰方法：叫作最终方法，可以被子类继承，但是不能被子类覆盖。

（3）final 修饰变量：叫作最终变量，也叫常量，一旦赋值之后就不能再修改值。

下面将针对不同的情况进行展开说明。

1）最终类

final 修饰的类叫作最终类，也叫骡子类。最终类只能继承其他的类但是不能当作父类被继承，它的设计通常用于确保类的安全性和稳定性。例如，Java 中的 String 类和 Math 类就是 final 类，它的所有方法和字段都是不能被修改的。

下面通过一段代码展示 final 修饰的类。

```
class A{
    //类体
}
final class B extends A{
    //类体
}
class C extends B{                   //报错
    //类体
}
```

上述代码在 C 继承 B 时出现“无法从最终 B 进行继承”的错误，因为 B 类是最终类。

2）最终方法

final 修饰的方法叫作最终方法。最终方法不能被子类覆盖，即不能在子类中重新定义该方法。这个特性主要用于防止子类修改父类的实现，从而确保父类的功能正确性和安全性。例如下面的代码演示：

```
class A{
    public final void test(){
        System.out.println("父类的最终方法");
    }
}
class B extends A{
    @Override
    public void test(){                          //报错
        System.out.println("子类的最终方法");
    }
}
```

上述代码在子类 B 继承 A 想要覆盖 test()方法时出现“B 中的 test()无法去覆盖父类的 test()方法”的错误,因为被覆盖的方法是用 final 修饰的。

3) 最终变量

final 修饰的变量叫作最终变量,也叫常量。常量在赋值后不能被修改。这个特性通常用于确保变量的值不会被错误地修改,从而保证程序的正确性。在 Java 中,常量通常用大写字母表示,代码中使用常量可以提高代码的可读性和可维护性。

小脑袋大问号

学弟:学姐,我刚刚测试了这一段代码,对于这两个变量一个可以修改而另一个不能修改有点不能理解。

```
public class Test{
    public static void main(String[] args){
    final int x = 45;
    x = 55;                                      //报错
    final Student stu = new Student("张三",23);
    stu.age = 33;                                //通过
    }
}
class Student{
    String name;
    int age;
    public Student(String name,int age){
    this.name = name;
    this.age = age;
    }
}
```

学姐:我们前面的章节已介绍过,Java 中基本数据类型的值指的是数值。Java 中引用数据类型的值指的是地址。所以 final 修饰 x 时,保护的是 x 的数值不变,final 修饰 stu 时,保护的是 stu 地址不变。

学弟:那么是不是改动 stu 指向一个新地址就会报错?

学姐:是的。

3.3　对象的创建和使用

在 Java 中使用关键字 new 来实例化对象,具体格式如下:

```
类名 对象名 = new 类名();
```

例如，创建 Student 类对象：

```
Student stu = new Student();
```

上述代码表示利用 Student 类创建一个实例，这里 Student 表示类名，类就是创建对象的模板。stu 表示对象的引用。在 Java 中，对象的引用指的是一个保存了对象内存地址的变量。Java 通过将对象存储在堆内存中，并使用引用来引用这些对象，实现对对象的操作。

当创建完 Student 类的对象之后，可以通过对象调用类里面的成员变量和成员方法。下面通过一个示例演示如何访问 Student 类中的成员。

```
public class Test{
    public static void main(String[] args){
        Student stu = new Student();            //创建一个 Student 类型对象
        stu.name = "张三";                      //给 stu 对象的 name 属性赋值
        stu.age = 22;                           //给 stu 对象的 age 属性赋值
        System.out.println(stu.name + "今年" + stu.age + "岁");
        stu.eat();                              //调用 eat()方法
        stu.study(4);                           //调用 study()方法
    }
}
```

运行结果：

```
张三今年 22 岁
学生爱吃汉堡
张三同学每天需要学习 4 个小时
```

当对象被初始化之后，可以通过对象的引用来访问该变量的成员属性和成员方法。例如第 4、5 行通过对象的引用访问该对象的成员属性并且进行赋值。调用格式为“对象名.属性＝值;”第 7、8 行通过对象的引用调用该对象的方法。调用格式为“对象名.方法名();”。

在创建对象时，一个对象同时可以有多个引用，但是如果一个引用都没有，会被 GC(Garbage Collector)垃圾回收器视为垃圾，将在下一次回收垃圾时把内存释放。

例如下面的代码展示：

```
public class Test{
    public static void main(String[] args){
        Student stu = new Student();      //创建一个 Student 类对象
        Student x = stu;                  //将该对象的内存地址交给 x 保存一份
        stu.eat();                        //stu 调用 eat()的方法
        stu = null;                       //将对象 stu 置为 null
        x = null;                         //将对象 x 置为 null
    }
}
```

上述代码中创建一个 Student 类对象，并且将 stu 的内存地址交给 x 保存一份，通过内存图分析可以看出堆内存中的 Student 对象有两个不同的引用指向该内存地址，下面通过 stu 调用 eat()方法打印“学生爱吃汉堡”。接着，将 stu 的值赋值为 null，表示 stu 不再指向任何一个对象，同时将 x 赋值为 null，那么 x 也不再指向堆中的对象。堆中的对象失去所有的引用，成为垃圾，GC 会在下一次垃圾回收的时候将内存释放。

下面通过内存图的方式演示内存分布情况，如图 3.3 所示。

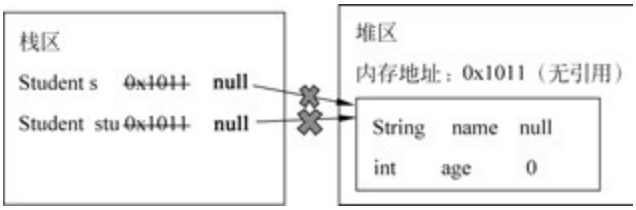
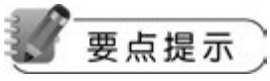


图 3.3　内存分布情况

3.4　类的封装



类的封装是指用 private 关键字修饰类中部分的成员变量与成员方法，从而实现只能在类内部访问这部分成员变量与成员方法。

在 Java 编程中，封装是指将类的实现细节隐藏起来，只暴露必要的接口给外部访问，以确保数据安全性和代码复用性。封装之后的类在修改代码时，不会影响外部调用，从而提高系统的可维护性。

具体来说，封装具有以下要求。

（1）在定义一个类时，将类的成员变量设置为私有的，即使用 private 关键字修改类的属性，私有化的属性只限于在类中被访问，以防止外部随意的访问和修改。

（2）提供每个属性的 getter 和 setter 方法来实现属性的只读或只写的限制。

（3）控制方法的可见性，公共的方法可以在外部被访问，私有的方法只能在类内部调用，用来给核心方法提供服务。

下面通过学生类演示如何封装属性，代码如下：

```
class Student{
    private String name;
    private int age;
    public void setName(String aName){
        name = aName;
    }
    public String getName(){
        return name;
    }
    public void setAge(int aAge){
        age = aAge;
    }
    public int getAge(){
        return age;
    }
    public void study(int hour){        //方法定义形参:int hour
        System.out.println(name + "同学每天需要学习" + hour + "个小时");
    }
}
```

由于属性的封装导致测试时不能再直接通过对象访问属性，例如：stu.name = "张三"将会报错，原因是属性封装后都是 private 修饰符进行修饰，private 修饰的成员只能在本类类体中使用，因此，对于属性的访问和修改只能通过 getter 和 setter 来完成。所以封装后想要访问

成员变量和成员方法应该改成以下方式：

```
public class Test{
    public static void main(String[] args){
        Student stu = new Student();
        stu.setName("张三");              //封装前:stu.name = "张三"
        stu.setAge(22);                   //封装前:stu.age = 22;
        //封装前:stu.name + "今年" + stu.age + "岁"
        System.out.println(stu.getName() + "今年" + stu.getAge() + "岁");
        stu.eat();                        //调用 eat()方法
        stu.study(4);                     //方法调用传入实参:4
    }
}
```

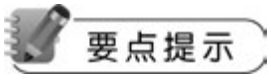
运行结果：

```
张三今年 22 岁
学生爱吃汉堡
张三同学每天需要学习 4 个小时
```

通过封装，类的使用者不需要了解类的内部实现细节，而只需要通过公共接口来操作类。如果类的内部实现发生变化，只需要修改类的私有实现，而不需要修改使用者代码。这能够提高代码的可维护性、可读性和可重用性，因此在 Java 编程中封装是一个很重要的思想和实践。

思考：private 修饰后是否真的无法在外部访问？如何绕过访问控制权限实现访问成员变量、成员方法？

3.5 类的继承



继承支持从已经存在的类中定义新的类。继承使得可以定义一个通用的基类(即父类)，实现继承该类的子类成为满足特定功能的类。

Java 中的继承是一个面向对象编程的概念，它允许创建一个类通过继承拥有另一个类的属性和方法。继承有助于代码重用和减少冗余代码的编写工作，提高代码的可维护性和可扩展性。通过继承，子类可以获取父类的属性和方法，并且可以重写父类的方法来实现某些功能。

▶ 3.5.1 is-a 和 has-a

在 Java 中，我们可以通过使用 extends 关键字来实现继承。常见的继承模式有两种：is-a 和 has-a。

is-a 表示一个对象是另外一个对象的子类型，通过继承来实现，通常用来描述某一个类是另外一个类的子类，属于父子关系。例如下面的示例：

```
class Animal{
    //动物类具有的属性/方法
}
class Cat extends Animal{
    //猫类具有的属性/方法
}
```

在上述代码中描述“猫是一个动物”，即 Cat is a Animal 的关系。

has-a 表示一个对象包含另一个对象，通常用来描述类之间的包含关系，属于合成关系。例如下面的示例：

```
class HeadTeacher{
    //班主任的属性和方法
}
class Student{
    HeadTeacher tea;            //每个学生都有一个班主任对象
}
```

上述代码中描述“学生类包含一个老师对象”，即 Student has a Teacher 的关系。

▶ 3.5.2　拓展子类

在 Java 开发时，还可以通过继承拓展一个类的功能，子类在继承得到父类的属性、方法之后，还可以在子类继续添加新的成员，如果对继承得到父类方法实现不满意，还可以通过方法覆盖进行修改，方法覆盖的内容稍后进行介绍。

下面通过一个示例演示如何使用 extends 关键字实现继承关系。

```
class Person{
    private String name;
    private int age;
    //省略封装的代码
    public void sleep(){
        System.out.println("人每天需要保障 8 小时睡眠");
    }
}
class Teacher extends Person{
    private double salary;
    //省略封装的代码
    public void teach(){
        System.out.println("老师每天需要教课 8 小时");
    }
}
```

上述代码创建一个父类——Person 类，该类具有两个属性和一个方法。子类 Teacher 在继承时可以获得 name、age、sleep()，在此基础上添加新的属性 salary 和方法 teach()。通过继承可以实现代码重用的效果，减少冗余代码。下面演示如何测试。

```
public class Test{
    public static void main(String[] args){
        Person p = new Person();
        p.setName("张三");
        p.setAge(23);
        p.sleep();
        System.out.println(p.getName() + " " + p.getAge());
        Teacher tea = new Teacher();
        tea.setName("Tom");
        tea.setAge(35);
        tea.setSalary(5000.0);
        tea.sleep();
        tea.teach();
        System.out.println(tea.getName() + " " + tea.getAge() + " " + tea.getSalary());
    }
}
```

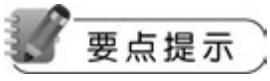
运行结果：

```
人每天需要保障 8 小时睡眠
张三 23
人每天需要保障 8 小时睡眠
老师每天需要教课 8 小时
Tom 35 5000.0
```

因此，Java 中的继承是一个非常重要和有用的概念，它不仅可以简化代码的编写，同时还可以使代码更加可维护和可扩展，是 Java 面向对象编程中必不可少的一部分。

思考：通过父类对象能否访问子类的成员变量与成员方法？如何阻止类被继承？

3.6 构造方法



使用 new 关键字调用构造方法创建对象。

构造方法是 Java 中一种特殊的方法，用于创建和初始化对象。通常在创建对象时通过 new 操作符调用。Java 中的每一个类都有构造方法，即使开发人员没有写构造方法，系统也会提供一个默认的构造方法。

▶ 3.6.1 构造方法的作用

构造方法能够在创建对象的同时初始化数据，在开发时配合构造方法能够给我们的代码带来很多方便。下面通过一段程序演示构造方法的使用。

```
class Person{
    private String name;
    private int age;
    //省去封装的代码
    //public Person(){} 系统默认提供的构造方法:无参空体
    public void sleep(){
        System.out.println("人每天需要保障 8 小时睡眠");
    }
}
class Test{
    public static void main(String[] args){
        Person aa = new Person();        //new 后面调用默认无参构造方法
        aa.setName("Tom");
        aa.setAge(23);
        System.out.println(aa.getName() + " " + aa.getAge());
    }
}
```

运行结果：

```
Tom 23
```

上述代码 Person 类中只定义 name、age 属性和 sleep()方法，在没有提供构造方法的情况下，系统会默认提供一个无参构造方法，所以在 main()方法中测试时使用的 new Person()调用的就是无参构造方法。由于无参构造方法并没有对属性赋值，因此创建完对象之后需要拿着对象引用调用属性进行赋值。

在开发时也可以在创建对象的同时选择给属性赋值，这种情况需要自己写出构造方法。

在 Java 中，如果一个类定义了至少一个构造方法，Java 将不再提供默认的无参构造方法。上述代码可以修改成以下格式：

```
class Person{
    private String name;
    private int age;
    //省去封装的代码
    public Person(String name,int age){
        this.name = name;
        this.age = age;
    }
    public void sleep(){
        System.out.println("人每天需要保障 8 小时睡眠");
    }
}
class Test{
    public static void main(String[] args){
        //一旦 Person 类有构造方法，系统将不再提供无参构造方法
        //Person aa = new Person();            //报错
        //调用有参构造方法直接给所有属性赋值
        Person aa = new Person("Tom",23);
        System.out.println(aa.getName() + " " + aa.getAge());
    }
}
```

运行结果：

```
Tom 23
```

小脑袋大问号

学弟：学姐，看上面的代码我发现构造方法定义很奇怪，感觉和平时写方法不同。

学姐：是不是觉得少了返回类型？

学弟：对，之前我们定义方法都是写返回类型的，构造方法返回类型不用写吗？

学姐：对，构造方法不写返回类型，连 void 都不能写。所以构造方法的定义格式就是“修饰符 方法签名(){...}”。

学弟：之前我记得说过方法名命名规范应该是首字母小写，以后的每个单词首字母大写，这段代码的构造方法名是不是写错了呀？

学姐：构造方法的名字需要和类名一模一样，所以只要类名定义规范，构造方法名字首字母应该大写。

这里的“Person aa=new Person("Tom",23);”会在实例化对象的同时调用有参构造方法，并且将"Tom"赋值给对象的 name 属性，将 23 赋值给 age 属性。那么就能省去拿着对象引用调用属性的赋值，也就是：

```
Person aa = new Person();
aa.setName("Tom");
aa.setAge(23);
```

等价于：

```
Person aa = new Person("Tom",23);
```

这样可以使代码更简化。

学弟：那么在写构造方法的基础上还需要再提供 setter 方法吗？

学姐：当然需要，以前没有写构造方法时，我们创建对象都是调用无参构造方法，再使用setter完成属性的赋值，当写出自己的构造方法之后，可以直接调用构造方法在创建对象的同时初始化属性，但是setter依旧需要写，因为我们可能面临只修改某一个属性值的情况，例如：

```
Person x = new Person("Tom",23);    //调用构造方法完成对象的创建和属性的初始化
x.setAge(24);                       //调用 setter 方法局部对 age 属性进行修改
```

注意：如果类中定义了有参数的构造函数且没有无参数构造函数，则使用时只能使用有参构造函数。子类如何调用父类的构造函数？

▶ 3.6.2 super()和 this()

构造方法是在实例化对象时被Java虚拟机直接调用，所以构造方法不能像普通方法那样调用，但是可以在子类构造方法的首行使用super()、this()方法来调用父类或者本类的其他构造方法完成代码的共享。

super()方法用于调用父类的构造方法，写在某个子类构造方法的首行。如果代码没有显式调用super()方法，那么系统默认找父类的无参构造方法。如果super()方法括号内有传递参数，根据参数的类型来决定调用父类哪一个构造方法。

this()方法用来调用本类其他的构造方法，写在构造方法的首行。this()方法必须显式地调用。具体执行本类的哪一个构造方法看this()方法括号内的参数类型。

下面通过一段代码演示this()和super()方法的使用。

```
class Person{
    private String name;
    private int age;
    //省略封装的代码
    public Person(String name){
        this.name = name;
    }
    public Person(String name,int age){
        //this(String):表示先执行本类传递 String 类型的构造方法
        this(name);
        this.age = age;
    }
}
class Teacher extends Person{
    private double salary;
    //省略封装的代码
    public Teacher(String name,int age,double salary){
        //super(String,int):先执行父类传递 String,int 类型的参数构造方法完成属性的赋值
        super(name,age);
        this.salary = salary;
    }
}
```

上述代码中的Person类有两个属性，并且提供两个不同的构造方法：第一个构造方法创建对象时只给属性name赋值；第二个构造方法调用第一个构造方法完成对name的赋值，并且手动完成age的赋值。

子类Teacher通过继承Person类获得name、age属性，加上Teacher类自己的属性salary，所以Teacher类在创建对象时应该完成三个属性的赋值，其中，name、age的赋值在父类构造方法中

已经实现，所以通过 super(name,age)调用 Person 类传递 String 和 int 类型的构造方法完成 name 和 age 的赋值，并且手动为 salary 赋值。

super()和 this()方法用在构造方法的首行，不能在其他方法中使用，两者不能同时存在，当没有显式调用 super()或者 this()时，默认调用 super()方法。在项目开发时可以使用 super()、this()方法避免大量重复代码的编写。

小脑袋大问号

学弟：学姐，我记得之前是不是说过 this.，它与今天我们学习的 this()有什么区别吗？

学姐：首先 this()只能用在构造方法的首行，不用在普通方法中，而 this.可以出现在构造方法中也可以出现在普通方法中，且没有位置限制。

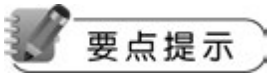
其次 this()表示要执行本构造方法之前先去执行本类其他的构造方法，具体执行哪一个构造方法看参数类型。this.表示当前调用该方法的对象，通常用来区分成员变量和局部变量。例如：

```
public User{
    String name;
    int age;
    String passWord;
    public User(String name,int age){
        System.out.println("实例化对象");
        //this.表示当前创建的对象,用于区分成员变量 name 和参数 name
        this.name = name;
        this.age = age;
    }
    public User(String name,int age,String passWord){
        //this():表示调用本类其他构造方法 必须出现在构造方法的首行
        this(name,age);
        this.passWord = passWord;
    }
    public void show(){
        System.out.println("展示个人信息:");
        //this.表示当前调用 show()方法的对象,this.没有位置限制
        //show()方法内没有局部变量和成员变量 name、age 重名
        //所以这里的 this.name 可以直接换成 name,age 同理
        System.out.println(this.name + "申请的密码是:" + this.passWord);
    }
  }
```

相同理论的还有 super.和 super()。super.表示在子类调用父类的成员属性和成员方法。super()表示在子类调用父类的构造方法完成部分属性的赋值。super()只能调用父类构造方法，而 super.只能调用父类的成员变量或者成员方法。例如：

```
class A{
    int x = 45;
}
class B extends A{
    int x = 90;
    public void test(){
        int x = 122;
        System.out.println(x);              //默认是局部变量 122
        System.out.println(this.x);         //调用本类 x 属性:90
        System.out.println(super.x);        //调用父类 x 属性:45
    }
}
```

3.7 多态



多态是指在父类中定义的属性和方法被子类继承后，被继承的属性与方法具有不同的含义或者实现。对面向对象来说，继承是多态的基础，多态行为的实现包括重载、覆盖、抽象等方法。

Java 中的多态是面向对象中的一个重要特性，它是指同一个类型的对象在不同的情况下有着不同的表现形式和功能。多态性是对象的一种特征，可以表现为父类的引用指向子类的对象（向上转型），即“父类类型 引用变量 = 子类实例”；也可以表现为子类的对象赋值给父类引用变量（向下转型），即“子类类型 引用变量 =（子类）父类对象”。

▶ 3.7.1 向上转型

下面通过一段代码演示向上转型时，即父类引用指向子类对象时的多态特性。

```
class Person{
    private String name;
    private int age;
    //省去封装的代码
    public void sleep(){
        System.out.println("人每天需要保障 8 小时睡眠");
    }
}
class Teacher extends Person{
    private double salary;
    //省去封装的代码
    public void teach(){
        System.out.println("老师每天需要教课 8 小时");
    }
}
class Test{
    public static void main(String[] args){
        //使用多态创建 Teacher 类对象：父类类型 = new 子类对象();
        Person pp = new Teacher();
        pp.setName("Tom");
        pp.setAge(23);
        //pp.setSalary(5000.0);              //报错
        pp.sleep();
        //pp.teach();                        //报错
    }
}
```

在上述代码中使用多态创建对象 pp 时，父类的引用变量可以指向子类的对象，但是这个引用变量只能调用父类中的方法和属性，不能直接调用子类特有的属性或方法，因为在多态的情况下，编译器将 pp 引用变量视为父类类型并在编译时检查它，而在父类中没有定义子类的特有属性或方法，因此编译器在编译时就无法识别这些属性或方法因而报错，所以在使用多态创建对象时一定要谨慎使用，避免代码的崩溃。

▶ 3.7.2 向下转型

多态还可以用在方法的参数上，即方法的参数可以是某个类的父类类型，传入该类的任意

子类对象作为实参，在方法内部使用多态实现方法的重载和覆盖。

举个例子，假设有一个 Person 类作为父类，有 Student 类和 Teacher 类继承自 Person 类，现在编写一个 Job 工作类提供工作的方法，学生的工作是学习，老师工作是教书，那么实现代码如下：

```
class Person{}
class Student extends Person{}
class Teacher extends Person{}
class Police extends Person{}
class Job{
    public void work(Student stu){              //方法重载
        System.out.println("学生学习中");
    }
    public void work(Teacher tea){              //方法重载
        System.out.println("教师授课中");
    }
    public void work(Police pp){                //方法重载
        System.out.println("警察巡逻中");
    }
}

class Test{
    public static void main(String[] args){
        Job worker = new Job();
        Student stu = new Student();
        Teacher tea = new Teacher();
        Police po = new Police();
        worker.work(stu);                       //编译器匹配 work(Student)方法
        worker.work(tea);                       //编译器匹配 work(Teacher)方法
        worker.work(po);                        //编译器匹配 work(Police)方法
    }
}
```

如果有多个类的对象想要传入 work()方法中作为参数，那么该方法需要重载很多次，冗余代码较多，且代码之间的耦合度较高。那么使用多态向下转型可以将方法的参数设置为 Person 类，则可以将 Student、Teacher 或者 Police 对象作为实参传入方法，从而实现代码的灵活性。修改后的代码如下：

```
class Person{
    public void work(){
        System.out.println("工作中");
    }
}
class Student extends Person{
        public void work(){             //重写了父类的方法
        System.out.println("学生学习中");
    }
}
class Teacher extends Person{
        public void work(){             //重写了父类的方法
          System.out.println("教师授课中");
    }
}
class Police extends Person{
        public void work(){             //重写了父类的方法
          System.out.println("警察巡逻中");
    }
}
```

```
class Job{
    public void work(Person p){        //父类作为参数,所有的子类对象都可以传入
        p.work();
    }
}
class Test{
    public static void main(String[] args){
        Job worker = new Job();
        Student stu = new Student();
        Teacher tea = new Teacher();
        Police po = new Police();
        worker.work(stu);              //编译器匹配 work(Person)方法
        worker.work(tea);              //编译器匹配 work(Person)方法
        worker.work(po);               //编译器匹配 work(Person)方法
    }
}
```

在上面的例子中,work()方法的形参是 Person 类型的,可以接收任何 Person 子类的实例作为实参。在调用 work()方法时,可以传入 Student、Teacher 或 Police 对象作为实参,编译器会自动识别出它们的类型。

多态通过提供一个统一的接口,使得程序具有更高的灵活性、可扩展性和可维护性,是面向对象编程中非常重要的一个特性。在实践中,多态可以应用于对象参数、对象返回值以及对象数组等诸多方面。

注意:通常方法定义时父类作为形参,具体调用时传入子类对象。

▶ 3.7.3 重载和覆盖

1. 覆盖属性

子类具有与父类相同的属性名称,但子类属性可以使用不同的可见性修饰符和数据类型。因此,父类和子类的属性彼此独立。

```
public class BaseCls {
    public Integer myAttribute;
}
public class ChildCls extends BaseCls {
    public String myAttribute;
}
public class Main {
    public static void main(String[] args) {
        ChildCls child = new ChildCls();
        child.myAttribute = "myString";
        ((BaseCls) child).myAttribute = 1337;              //向上转为父类对象
        System.out.println(child.myAttribute);             //输出:myString
        System.out.println(((BaseCls)child).myAttribute); //输出:1337
    }
}
```

2. 方法重载

Java 中的方法重载(Overload)是指在同一个类中,定义多个方法名相同、参数列表不同的方法(包括参数的类型、数量或者顺序)。通过方法重载,可以在一个类中定义多个功能类似的方法,但是不同的方法可以接收不同类型或个数的参数,实现的功能也不同。

例如,学信网查询学生成绩时,可以通过身份证号查询成绩,也可以通过学号查询成绩,那么查询的功能应该提供两种,可以用下面的代码实现:

```
public class TestOverload{
    public void getScore(String id){
        System.out.println("通过身份证号往数据库查询");
    }
    public void getScore(long stuId){
        System.out.println("通过学号往数据库查询");
    }
}
```

上面的示例中，在同一个类中定义两个获得分数的方法，它们方法名一致，但是参数类型不同，因此它们是两个不同的方法。

当一个程序在调用 TestOverload 的 getScore()方法时，编译器会通过传入的参数决定执行哪一个方法。例如：

```
TestOverload tt = new TestOverload();
tt.getScore("34243199808062077");          //通过身份证号查询考试成绩
tt.getScore(12345678);                     //通过学号查询考试成绩
```

方法重载跟返回类型和修饰符没有关系，一个类中定义多个相同方法名的方法，参数列表不同，这就是方法重载，即使返回类型不同，修饰符不同，它们也是方法重载。例如，下面演示的代码也是方法重载：

```
class TestOverload{
    public static void test(){
        System.out.println("方法 1");
    }
    public int test(int x){
        return x;
    }
}
```

小脑袋大问号

学弟：学姐，方法重载是在一个类中定义多个相似的方法，方法名相同，参数列表不同，对不对？

学姐：是的，这就是方法重载。

学弟：那么为什么要定义那么多相似的方法呢？方法重载有什么作用呢？

学姐：例如，上面的示例提供两个查询分数的方法，一个是用学号查询，另一个是通过身份证号查询，使用者在查询分数时就可以通过两种途径进行查询，这样就能同时满足用户的不同需求。再如，打印语句 println()，Sun 公司在封装此方法时也写了很多种重载的样式，有 println(int)、println(double)、println(String)等，这样做的目的是满足更多的需求，使用者无论想要打印什么类型的数据都可以匹配上合适的方法。

总体上说，多个同名方法与不同的参数列表形成方法重载，以适应不同的使用场景和需求。在实际开发中，使用方法重载可以写出更简洁、清晰、易于维护和扩展的程序。

3. 方法的覆盖

Java 中的方法覆盖(Override)指的是子类在继承得到父类某个方法对父类的实现不满意，于是在子类重新实现一个具有相同方法签名的过程。

当子类重写了父类中定义的方法后，继承自父类中的该方法将被子类中的方法所替代，当通过子类的引用调用该方法时，实际执行的是子类中的方法。但是如何通过父类的引用调用该方法，还是会执行父类的方法。这样，可以在子类中实现一些不同于父类的功能，并且不会破坏父类的原有实现。

方法覆盖的时候需要注意以下几点。

（1）子类要覆盖的方法的访问权限修饰符不能小于父类的访问权限修饰符。例如，父类的方法修饰符是 public，子类在覆盖的时候只能是 public，不能写成 private。

（2）方法覆盖时子类的方法签名需要和父类一模一样。

（3）子类要覆盖的方法返回类型可以和父类方法返回类型一模一样，也可以变成父类方法返回类型的子类类型，这也是常说的协变返回类型。

（4）子类方法不能抛出比父类方法声明更宽泛的异常，但可以不抛出异常或者抛出更严格的异常。

小脑袋大问号

学弟：学姐，这里的访问权限修饰符，除了 public 和 private 之外还有其他访问权限修饰符吗？

学姐：有，后面的章节我们会逐一介绍，现在只需要了解针对方法权限范围来看 public > protected > default > private。

学弟：好的，还有一个问题，“方法返回类型可以变成父类方法返回类型的子类类型，也叫协变返回类型”这句话我不是很懂，学姐能详细介绍一下吗？

学姐：方法覆盖时返回类型可以和父类保持一致，例如下面演示的代码。

```
class Animal{
    public Animal produce(){          //生育的方法
        …
    }
}
class Dog extends Animal{
    @Override
    public Animal produce(){          //子类覆盖的方法返回类型和父类保持一致
        …
    }
}
```

子类 Dog 在覆盖 produce()方法时，返回类型和父类方法保持一致是可以的，但是子类方法返回类型选择父类(Animal)返回类型(Animal)的子类类型(Dog)更合适，也就是可以改成以下版本。

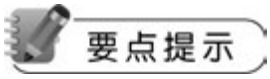
```
class Animal{
    public Animal produce(){          //生育的方法
        …
    }
}
class Dog extends Animal{
    @Override
    public Dog produce(){             //这里的返回类型变成父类返回类型的子类类型
        …
    }
}
```

如果 Animal 有两个子类——Dog 和 Cat，那么在任何一个子类中覆盖 produce()方法时，返回类型写成 Animal、Cat、Dog 中的任意一个代码都可以通过。不过还是选择一个合适的数据类型比较好理解。

还有一个值得一提的内容是，从 JDK 5.0 开始方法覆盖时可以在子类要覆盖的方法上面加上@Override 注解，用于标记某个方法是对父类方法的覆盖(重写)，编译器在编译时会检查该方法是否正确地覆盖了一个父类方法。当使用@Override 注解时，如果没有发生覆盖，则会发生编译器错误，避免由于方法签名不完全匹配而产生的不易被发现的错误。

思考：重载与覆盖各自的作用和使用场景是什么？

▶ 3.7.4　abstract 关键字



通过 abstract 可以实现多态。抽象类和抽象方法之间的关系为有抽象方法的类，一定是抽象类；抽象类不一定有抽象方法。当子类继承抽象类时，必须要将抽象类中的抽象方法全部实现(或者称为重写)。

在 Java 中，abstract 是一个修饰符，可以应用于修饰类、方法。它用来定义抽象类和抽象方法。抽象类和抽象方法是 Java 中的一种特殊类型，它们不需要提供实现，而是需要被子类或实现类实现或继承。

1. 抽象类

抽象类使用 abstract 修饰的类，不能被实例化。抽象类通常用来定义一些基本的方法和属性，这些方法和属性可以被子类继承和实现，以完成具体的功能。抽象类中可以包含抽象方法，也可以包含非抽象方法。

下面演示一段抽象类的定义。

```
abstract class Shape{
    int length;
    int width;
    public Shape(int length,int width){
        this.length = length;
        this.width = width;
    }
    public abstract void showArea();
    public abstract void showLength();
}
```

在这个示例中，Shape 类是一个抽象类。它有两个属性 length 和 width，还有一个公共的构造函数，同时，它还声明了两个抽象方法 showArea()和 showLength()，这两个方法没有方法体，子类需要实现它。

下面创建 Shape 类的子类并且实现它的两个方法：

```
class MyRectangle extends Shape{
    public MyRectangle(int length,int width){
        super(length,width);
    }
```

```
    public void showArea(){
        System.out.println(length * width);
    }
    public void showLength(){
        System.out.println((length + width) * 2);
    }
}
```

MyRectangle 是 Shape 的子类，提供一个构造方法给两个属性赋值，并且覆盖了父类的两个抽象方法。

小脑袋大问号

学弟：学姐，你说抽象类是类吗？

学姐：当然啦。

学弟：那抽象类有构造方法吗？

学姐：Java 中所有的类都有构造方法，抽象类也是类，所以一定有构造方法。

学弟：抽象类中有构造方法但是不能创建对象，那么抽象类中的构造方法有什么作用呢？

学姐：抽象类一般会伴随着子类存在，所以抽象类中的构造方法在子类构造方法中必须通过 super()显式调用，且必须放在子类构造方法的第一行。

2. 抽象方法

抽象方法是一种没有实现的方法，它没有方法体，只有方法的声明。抽象方法使用关键字 abstract 进行定义。抽象方法表示当前类必须包含该方法但是具体的实现细节待留给子类去实现。抽象方法不能用 final、static 和 private 修饰，因为这些修饰符与抽象方法的本质不符。

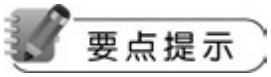
下面演示一段抽象方法的定义：

```
abstract class Animal{
    public abstract void eat();
    public void sleep(){
        System.out.println("动物需要休息");
    }
}
class Cat extends Animal{
    @Override
    public void eat(){
        System.out.println("猫吃鱼");
    }
}
class Dog extends Animal{
    @Override
    public void eat(){
        System.out.println("狗吃肉");
    }
}
```

上述代码父类 Animal 定义一个抽象方法和一个普通方法，由于类中存在抽象方法，那么所属的类一定要变成抽象类，抽象类中也可以定义普通方法。子类通过继承 Animal 类覆盖父类的抽象方法。

注意：抽象类不能使用 new 创建对象，抽象方法无函数体。建议在方法实现不确定时采用 abstract。

3.8　单例模式



Windows 任务管理器在正常情况下只能打开唯一一个任务管理器进程，即只有唯一的实例对象。Java 通过单例模式可以实现类的唯一实例。

在 Java 中，单例模式是一种常用的设计模式，它保证一个类有且只有一个实例，并提供了全局访问方式。通过单例模式可以避免创建多个相同的对象，避免资源的浪费，提高系统性能。

单例模式有多种实现方式，下面介绍其中比较常用的两种实现方式。

1. 饿汉式单例模式

在饿汉式单例模式中，类在被加载时就创建了一个实例，因此线程安全性比较好，但会增加内存消耗。示例代码如下：

```
public class Singleton {
    //私有化构造方法,防止外界随意创建 Singleton 对象
    private Singleton() {}
    //创建唯一实例,使用 private、static 修饰
    //使用 static 修饰的目的是防止内存中创建多个 Singleton 对象
    //使用 private 修饰的目的是防止外界随意地修改 instance 对象的值
    //例如:Singleton.instance = null;
    private static Singleton instance = new Singleton();
    //提供静态方法返回唯一实例
    //使用 static 修饰的目的是让 getInstance()更加方便地调用
    public static Singleton getInstance() {
        return instance;
    }
}
```

在使用时，通过类名.getInstance()方法获得 Singleton 对象。例如：

```
public class Test{
    public static void main(String[] args){
        Singleton s1 = Singleton.getInstance();
        Singleton s2 = Singleton.getInstance();
        System.out.println(s1 == s2);                    //true
    }
}
```

上述代码中 s1 和 s2 使用连等号比较实质是比较引用对象的地址，最终显示 true，表示它们引用对象的地址相等，代表 s1 和 s2 是内存中的同一个对象。

2. 懒汉式单例模式

懒汉式是最常见的单例模式之一，这种实现方式的主要特点是在需要时才去创建实例对象。具体实现方式是：在定义单例对象时，先不初始化，而是等到第一次使用时才去创建并返回其实例。

懒汉式单例模式的优点在于它可以延迟实例的创建，从而节省了系统资源。但是也有一些缺点，它的线程安全性存在隐患，在多线程的情况下，由于不同线程可能同时进入单例对象的创建方法，导致多次创建对象从而违反了单例模式的原则。

示例代码如下：

```
public class Singleton {
    //私有化构造方法,防止外界随意地创建 Singleton 对象
    private Singleton() {}
    //创建唯一实例,使用 private、static 修饰
    private static Singleton instance;
    //提供静态方法返回唯一实例
    public static synchronized Singleton getInstance() {
        if(instance == null){              //判断对象是否为空
            instance = new Singleton();    //如果对象为 null,那么创建对象
            return instance;               //返回创建的对象
        }else{
            return instance;               //如果对象不为空,那么直接返回对象
        }
    }
}
```

在使用时，同样通过“类名.getInstance()”方法获得 Singleton 对象，例如：

```
public class Test{
    public static void main(String[] args){
        Singleton s1 = Singleton.getInstance();
        Singleton s2 = Singleton.getInstance();
        System.out.println(s1 == s2);      //true
    }
}
```

在懒汉式单例模式中，还有一种使用双重校验锁的方式用来保证线程安全，后面介绍线程安全会再次介绍。另外，在使用单例模式时，需要注意：

(1) 构造方法必须是私有的，防止外部随意创建实例。

(2) 声明实例变量必须使用 private、static 修饰，保证只有一个实例，并且防止外界随意地修改值。

(3) 提供一个静态方法返回唯一实例，这个方法通常命名为 getInstance()，可以通过类名直接调用。

(4) 在多线程环境下，需要考虑线程安全性问题，例如使用双重校验锁、静态内部类等方式保证线程安全。后面章节会进行介绍。

注意：getInstance()方法中需要使用同步锁 synchronized 防止多线程同时进入造成类被多次实例化。

3.9 内部类

内部类(Inner Class)是 Java 中的一个非常重要的概念，它是一个定义在另一个类或者代码块中的类。内部类与外部类关系密切，可以直接访问外部类的某些属性和方法。根据定义的位置、修饰符和定义的方式，内部类分为成员内部类、静态内部类、局部内部类和匿名内部类。内部类的优点如下。

(1) 内部类与外部类可以方便地访问彼此的私有域(例如私有方法、私有属性)。

(2) 内部类是另外一种封装，对外部的其他类隐藏。

(3) 内部类可以实现 Java 的单继承局限。

▶ 3.9.1　成员内部类

在一个类中除了可以定义成员变量、成员方法，还可以定义类。定义在另一个类中的类叫作成员内部类，它是一种嵌套类。在 Java 中，成员内部类通常是非静态的。成员内部类可以访问外部类的所有成员，包括私有成员，并且可以保留对其外部类实例的引用。

成员内部类的语法格式如下：

```
class OuterClass{
    //...
    class InnerClass{
        ...
    }
}
```

在这个例子中，InnerClass 就是一个成员内部类，它被定义在 OuterClass 中。在内部类 InnerClass 中可以直接访问 OuterClass 中的成员变量和方法，例如：

```
class OuterClass {
        private int i = 10;
        public String str = "etoak";
        private void print() {
            System.out.println("OuterClass");
        }
        class InnerClass {
            public void print() {
                System.out.println("InnerClass");
                System.out.println(i);
                System.out.println(str);
                OuterClass.this.print();
            }
        }
}
```

在这个例子中，InnerClass 中的 print()方法可以直接访问 OuterClass 中的私有成员变量 i、公共成员变量 str 和私有方法 print()。同时，使用 OuterClass.this 来引用 OuterClass 的实例，这可以保留对外部类对象的引用。

需要注意的是，成员内部类的实例不能脱离外部类的实例而单独存在。每个内部类实例都隐式地保留了对其外部类实例的引用。因此，如果要创建内部类的实例，必须先创建外部类的实例，例如：

```
OuterClass ou = new OuterClass();
OuterClass.InnerClass in = ou.new InnerClass();
```

在这个例子中，先创建了 OuterClass 的实例 ou，然后通过 ou 来创建 InnerClass 的实例 in。除此之外，还可以使用 OuterClass.InnerClass 语法来直接创建 InnerClass 的实例，例如：

```
OuterClass.InnerClass inner = new OuterClass().new InnerClass();
```

这个语法可以一次性创建 OuterClass 和 InnerClass 的实例，但是需要注意的是，这种方式并不会保留对外部类实例的引用。

在成员内部类中不能出现任何静态的成员，因为这违背成员变量可以访问外部的所有成员的原则，而且成员内部类依赖于外部类存在，也就是说每个成员内部类的对象都隐式地关联着一个外部类对象。因此，成员内部类不能拥有静态成员，因为静态成员是不依赖于对象而存在的，而成员内部类的对象必须依赖于外部类对象。

小脑袋大问号

学弟：上面演示的外部类和内部类的代码，是不是可以将成员内部类看成成员属性，都是依赖于外部类对象而存在？

学姐：可以这么理解，成员变量、成员方法、成员内部类基本相同，可以按照相同的调用规则去理解。

学弟：OutClass.this 表示什么呢？

学姐：要想在内部类生成外部类对象的引用，可以使用外部类的名字.this，这样生成的引用会自动具有正确的类型，而且是可以在编译时确定并检查的，所以没有任何运行时开销。

学弟：如果在这一行直接写 this 的话会被认为是外部类对象还是内部类对象？

学姐：如果在内部类直接使用 this 则会被理解成内部类对象。

▶ 3.9.2 静态内部类

静态内部类是一种定义在另一个类中并被 static 关键字修饰的内部类。与成员内部类不同，静态内部类不依赖于外部类的实例，因此，可以拥有静态成员和方法，而成员内部类不可以。

静态内部类的定义方式如下：

```
public class OuterClass {
    …
    public static class InnerClass{
        …
    }
    …
}
```

在上述代码中，InnerClass 就是一个静态内部类，可以看到它的定义方式与外部类的定义方式类似，只是在 class 关键字前加上了 static 修饰符。在 InnerClass 的内部可以访问外部类的所有静态成员，但是不能访问外部类的实例属性和实例方法。例如：

```
class OuterClass {
    private int i = 10;
    public static String str = "etoak";
    private static void print() {
        System.out.println("OuterClass 静态方法");
    }
    public void test(){
        System.out.println("OuterClass 实例方法");
    }
    static class InnerClass {
        static int x = 45;
        int y = 50;
        public void innerprint() {
            System.out.println(str);              //访问外部类静态属性
            print();                              //访问外部类静态方法
            System.out.println(x);                //访问本类静态属性
            System.out.println(y);                //访问本类实例属性
            //静态方法中不能直接访问外界类的实例属性和实例方法
```

```
            //System.out.println(i);
            //test();
        }
        public static void staticInnerPrint(){
            System.out.println(str);          //访问外部类静态属性
            print();                          //访问外部类静态方法
            System.out.println(x);            //访问本类静态属性
            //静态方法中不能直接访问本类实例成员
            //System.out.println(y);
            //如果想要在静态方法中访问本类实例成员则可以通过以下方式
            System.out.println(new InnerClass().y);
            //静态方法中不能直接访问外界类的实例属性和实例方法
            //System.out.println(i);
            //test();
        }
    }
}
```

在这个例子中，InnerClass 中的 innerprint()方法可以直接访问 OuterClass 中的静态成员 str 和静态方法 print()，但是不能访问外部类的实例变量 i 和实例方法 test()。静态内部类可以访问本类的哪些成员取决于所在的方法。

在上述代码中，静态内部类 InnerClass 还包含一个静态方法 staticInnerPrint()。在这个方法中，它可以访问静态内部类的静态变量 x，以及通过实例化内部类来访问它的实例变量 y。

静态内部类的实例化不依赖于外部类的实例，可以直接通过类名进行实例化。下面演示如何创建静态内部类对象和调用静态内部类方法。

```
OuterClass.InnerClass in = new OuterClass.InnerClass();
in.innerprint();                              //调用静态内部类实例方法
OuterClass.InnerClass.staticInnerPrint();     //调用静态内部类静态方法
```

在上面的代码中，直接通过类名调用了 staticInnerPrint()方法，因为它是一个静态方法。

注意：静态内部类的静态方法中不能直接访问本类实例成员；静态内部类的静态方法中不能直接访问外界类的实例属性和实例方法。

▶ 3.9.3　局部内部类

在 Java 中，局部内部类是指定义在一个方法或者一个作用域内的内部类。与成员内部类和静态内部类不同，局部内部类只在定义它的方法或者作用域内有效，外部的代码无法访问它。

局部内部类的定义方式如下：

```
public class OuterClass {
    public void outerMethod() {
        class LocalInnerClass {
            …
        }
        …
    }
    …
}
```

在上述代码中，LocalInnerClass 就是一个局部内部类，它被定义在 outerMethod()方法中。对于局部内部类可以访问外部类的所有的成员，并且可以访问定义它的方法或者作用域

的 final 变量，但是局部内部类中不能定义静态的成员。例如：

```
class OuterClass {
    private int i = 10;
    public static String str = "etoak";

    public void outerMethod(){
        int x = 45;
        class LocalInnerClass{
            public void show(){
                System.out.println(i);          //10
                System.out.println(str);        //etoak
                System.out.println(x);          //45
            }
        }
        //局部内部类只在当前的方法体有效
        new LocalInnerClass().show();
    }
}
```

在上面的示例中，OuterClass 包含一个名为 outerMethod()的方法，该方法包含一个局部内部类 LocalInnerClass。局部内部类定义在 outerMethod()方法中，并且可以访问该方法的局部变量 x。

首先，在 outerMethod()方法中将局部内部类实例化为一个对象，然后调用它的方法 show()。show()在控制台上打印一些文本，其中包括外部变量的值。

需要注意的是，局部内部类只能访问所在方法或者作用域中的 final 修饰的局部变量。

小脑袋大问号

学弟：局部内部类可以访问外部类的所有成员和所在方法体里定义的局部变量。

学姐：所在方法体中的局部变量需要是最终变量，不然都会报错。

学弟：上面演示的代码局部变量 x 没有用 final 修饰啊。

学姐：在 Java 8.0 之前的版本中，局部内部类只能访问所在方法的 final 变量，需要明确加上 final 修饰符。在 Java 8.0 版本之后，如果变量值在初始化后没有更改，可以省略 final 关键字，但实际上该变量仍然是 final 的，也不应该再更改它。

学弟：为什么一定不能修改呢？

学姐：当局部内部类访问所在方法的局部变量时，Java 实际上创建了一个对该局部变量的副本。如果允许修改原始变量，则可能会导致多线程同步问题。如果局部内部类在一个线程中修改变量，而在另一个线程尝试读取该值，那么它可能会读到锁定之前或之后的变量值，这可能导致错误结果。使用 final 修饰符可以确保被引用的变量是不可变的。因此在局部内部类可以引用变量，但不能修改变量，从而消除了在多线程环境中出现同步问题的风险。

学弟：也就是 Java 8.0 之后，如果只是在初始化之后没有更改变量，则可以省略 final 关键字。不过变量仍然保持 final 特性，即不能在局部内部类中修改它们的值。

▶ 3.9.4 匿名内部类

匿名内部类是没有类名的局部内部类，通常用于实现接口或继承类。它被称为匿名内部

类，因为它没有单独的类名，并且它的定义和实例化通常出现在同一行代码中。

通常情况下，匿名内部类被用于创建一个实现某个接口或继承某个类的对象，并且只需要使用一次，或者在编写内部类类名并不重要的情况下。

下面是一个使用匿名内部类的简单示例，该部分内容可以学完第 5 章之后再回来学习会更好理解一点。

```
public class TestTreeSet{
    public static void main(String[] args){
        TreeSet<Integer> set = new TreeSet<>(new IntegerComparator());
        Collections.addAll(set,45,66,10,93,88,51);
        //实现对 TreeSet 集合中的元素进行降序排序
        System.out.println(set);
    }
}
class IntegerComparator implements Comparator<Integer>{
    @Override
    public int compare(Integer x,Integer y){
        return y.compareTo(x);
    }
}
```

使用匿名内部类的版本：

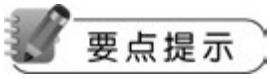
```
public class TestTreeSet{
    public static void main(String[] args){
        //创建一个实现 Comparator 接口的匿名内部类
        Comparator<Integer> cc = new Comparator<Integer>(){
            @Override
            public int compare(Integer x,Integer y){
                return y.compareTo(x);
            }
        };
        TreeSet<Integer> set = new TreeSet<>(cc);
        Collections.addAll(set,45,66,10,93,88,51);
        //实现对 TreeSet 集合中的元素进行降序排序
        System.out.println(set);
    }
}
```

在上述代码中，通过使用 new 关键字和 Comparator 接口的实现来创建一个匿名内部类，并重写 compare()方法。然后，将该匿名内部类实例化为一个 Comparator 对象，并将其用作 TreeSet 构造方法的参数进行定制排序。

通过上述代码也可以看出，匿名内部类比较适合使用一次的场景，如果代码中多次需要创建实现某个接口的对象，还是用常规的定义类的方式更合适。

思考：能否将 4 种类型的内部类的特点进行总结？

3.10　接口



接口用于描述类具有的功能清单，但并不给出具体实现，因此接口无法直接实例化，当某个类要使用接口时，再去实现接口中的这些方法。实现类需要遵从接口中描述的统一规则进行定义，因此，接口是对外提供的一组规则、标准。其另外一个重要特点是接口可以多继承。

在Java中，接口是一种特殊的类型，它可以定义一组方法的签名，但不提供任何实现，而是在实现接口的类中实现。接口是一种抽象类型，不能被实例化。

接口可以被类实现，实现类必须实现接口中定义的所有方法，否则就需要将实现类改成抽象的，实现类可以实现多个接口。

那么如何定义接口呢？接口中可以定义哪些内容？本小节将针对该部分内容进行讲解。接口中定义属性和普通类中定义属性相似，但是接口中的属性默认加上三个修饰符，分别是public、static、final。

接口中定义方法可以只声明方法签名，系统默认在方法前加上两个修饰符，分别是public、abstract。Java 8.0接口中引入了默认方法(加default)和静态方法(加static)，这使得接口中可以包含预定义的实现，而不仅仅是方法签名。在Java 9.0接口中允许出现私有方法。

接口的语法定义如下：

```
interface MyInterface {
    int x = 45;                    //接口中的属性前面默认加上 public、static、final
    void method1();                //接口中的方法前面默认加上 public、abstract
    String method2(String x);
    public static void method3(){ //Java 8.0 引入的静态方法
        //...具体实现
    }
}
```

实现接口的类的语法如下：

```
class ClassName implements MyInterface{
    @Override
    public void method1(){
        //...方法覆盖
    }
    @Override
    public String method2(String x){
        //...方法覆盖
    }
}
```

在Java中，接口是一种非常有用的特性，它允许开发人员定义一组方法签名，以便被实现类实现。接口是一种面向对象编程的重要概念，因此了解它们是非常重要的。

思考：在实现多态时，abstract与接口各有哪些优势？

小脑袋大问号

学弟：学姐，这里定义的接口为什么不能写成普通的父类呢？

学姐：可以呀，其实接口很像之前的抽象类，里面只定义方法签名，没有实现，留给子类去实现。

学弟：那抽象类和接口有什么区别呢？

学姐：咱们在这里之所以定义接口而没有定义抽象类是因为Java中的类只能单继承，即一个类只能继承一个类。我们想象一下，如果ClassName类已经继承了一个类，那么再继承MyInterface类就会失败。所以，类可以在不必改变自己的继承结构的情况下再实现接口，从而提供某些功能。

而且 Java 中的接口可以用来实现多态和解耦合，因为一个类可以实现多个接口，并且接口的实现可以在运行时动态地替换，还可以提供一种将类组织在一起的方式，从而创建更强大的抽象类型。

3.11　本章小结

本章详细介绍了面向对象的基础知识，首先介绍了类和对象的基础概念，以及什么是面向对象的思想，类的封装、继承和对象的创建使用；其次介绍了多态在项目开发中的应用；最后介绍了 Java 中常用的修饰符、接口的定义和使用。熟练掌握本章内容，能够更快速和高效地进行后续的学习。

3.12　习题

1. 下面的代码存在方法重载吗？如果存在，则_____和_____满足方法重载？

```
class A{
    public void test(){
        System.out.println("A 类的 test 方法");
    }
  }
class B extends A{
    public void test(int x){
        System.out.println("B 类的 test 方法");
    }
}
```

2. 如果希望在任何包下面的子类有权访问父类的方法，以下(　　)是较为严格的修饰符。

A：public　　B：private　　C：protected　　D：transient　　E：default

3. 请简述构造方法首行的 super()和 this()方法之间的区别。
4. 请简述抽象类和接口之间的区别。
5. 封装一个 Student 类型，具备的属性有 name、age、stuId，提供 eat()和 study()方法。
   封装一个 Teacher 类型，具备的属性有 name、age、salary，提供 eat()和 teach()方法。
   封装一个 Police 类型，具备的属性有 name、age、id，提供 eat()方法。
   封装一个 RestRoom 类型，提供 sleep()方法。要求学生、老师、医生都可以休息。
   使用封装、继承、多态实现。

习题答案