第 3 章

类、对象与接口

本章内容

本章内容主要包括面向对象程序设计理念及类、对象、继承、接口、多态、内部类和 Lambda 表达式等内容。这些概念和内容会渗透到 Java 编程的各个角落，它们是面向对象程序设计的基础。

学习目标

- 了解类的组成，熟练掌握类成员的访问控制权限设定，方法重载的概念和技术要点，掌握静态变量、静态方法的设定方法和含义。
- 了解继承的概念，理解子类引用、父类引用与子类对象之间的关系。熟练掌握方法重写的概念和技术要点，理解引用类型变量自动转型与强制转型的前提条件。
- 了解接口与抽象类的区别，掌握抽象类与接口的定义，熟练掌握接口的实现。
- 理解多态的概念，掌握引用类型变量转型的本质。
- 了解内部类的定义和简单使用方法，熟悉匿名类的用法，深刻理解它与 Lambda 表达式之间的关系，可以熟练完成 Lambda 表达式的书写。

3.1 面向对象的程序设计概述

程序是世界的抽象，人们对世界的认识很大程度上影响到程序设计的思想。现实世界就是由各种对象组成的，如人、动物、植物等。对象都有各自的属性和与属性相关联的行为，如人类拥有知识、教师可以传播知识、学生可以学习知识等。通常人们会将属性及行为相同或相似的对象归为一类。类可以看成是对象的抽象，代表了此类对象所具有的共有属性和行为。程序代码中的类不仅包括数据（属性），还包括方法（行为）。类在实例化后就形成了程序中的对象，程序运行过程中的对象之间利用自身拥有的方法进行交互。这就是面向对象程序编写和运行的整体情况。

面向对象的程序设计思想是一种被广泛认可的、能让设计人员在处理复杂程序过程中受益的思想。在软件工程中，对复杂程序一般采用两种方法进行模块分解：功能分解和数据分解。功能分解是面向过程编程的基础，基于函数（方法）概念，以过程为中心来建立功能模块；数据分解则是面向对象编程的基础，依赖于类的概念，以数据为中心来建立数据模块，将数据与方法进行整合封装。面向过程的设计方法中，在数据与方法分离的情况下，用户不仅需要规划程序实现过程，还需要关注数据的稳定性、可靠性问题，而且需要随时屏蔽数据被其他方法错误修改的可能。虽然这种方式在软件当中设计难度较大，但因为代码量小、复用效率高、可充分利用系统性能等特点，在一些小型环境下或特定领域中仍然被普遍使用。

但是在大型软件设计过程中，人们更加注重分解模块的可靠性、可维护性，对代码简洁性和系统运行的极致性能做了一定的妥协。人们开始从面向数据的角度分解整个设计工作，将数据与方法进行共同封装这种方式下，整个系统中只需要关注数据流即可。这种封装既可以让方法在一个相对稳定的环境运行，避免数据串扰的发生，又丰富了数据对外的服务接口。这就是面向对象的编程思想产生的原因之一。

面向对象程序的基本组成单位是类。程序在运行时会按要求找到类，由类生成对象，对象之间通过交互通信、互相协作完成相应的功能。类是面向对象程序设计的核心，而对象则是面向对象程序运行的核心。

面向对象程序设计方法秉持的原则如下。

（1）从数据出发对系统中的类进行划分，将数据相关的方法和数据封装在一起，用类来管理方法和程序模型，使用对象与对象交互的方式完成程序运行流程。

（2）开闭原则（Open-Closed Principle，OCP）。封装的模块对于功能扩展保持开放，对模块内部的数据和方法运行保持封闭。其核心思想是这个模块可以在不被修改的前提下更容易被扩展，同时将数据和功能实现局限化到模块内部，降低模块之间的耦合性。

面向对象程序设计的特点如下。

1. 抽象

抽象就是忽略问题中与当前目标无关的某些方面，将与当前设计目标有关的方面使用计算机的语言表述出来的一种设计思路。目标对象的属性可以抽象为成员变量，使用 Java 中四类八种基础数据类型或引用变量类型的值来表示，如学生的年龄、姓名。目标对象的行为和功能使用方法的形式表示，如学生的学习知识的行为、相互交流的行为。

当然，对象的抽象要与系统设计目标相匹配。例如，描述一个人的年龄，一般来说，抽象为一个以年为单位的 int 类型值表示即可，在特殊情况下（记录新生儿信息时）也需要选择更高精度的以天单位的数值。因此，抽象就是找出真实对象的部分与系统相关特性，并在编程语言描述事物的能力限定范围内进行编码，这个抽象过程是一个平衡需求与实现的过程。对于面向对象的程序而言，抽象过程是以对象数据为核心，将数据和相关功能进行协同抽象（需求与实际代码功能的妥协）、共同封装的过程。

2. 封装

面向对象的封装特性与其抽象特性密切相关。封装是一种信息隐藏技术，就是利用“类”这个结构将数据和基于数据的操作封装在一起。对象的界面信息对用户开放，对象的内部细节对用户隐藏。封装的目的在于将对象的使用和设计实现分离，使用者不必知道实现的细节，只需使用设计者提供的接口访问对象即可，此种设计有助于减少代码的耦合度，也简化了其他用户使用该对象的学习过程。

3. 继承

继承体现了类与类之间一般与特殊的关系。继承可以让子类获得父类的属性和行为，为类成员的重用提供了方便。使用继承不仅可以使程序结构清晰，降低了编码和维护的工作量，而且可以使子类和父类拥有相同的对外服务接口，为面向对象程序的多态机制奠定了基础。

继承有单继承和多继承之分。单继承是指任何一个子类都只有一个直接父类；而多继承是指一个类可以有一个以上的直接父类。采用单继承的类层次结构为树状结构，采用多

继承的类层次结构为网状结构，设计及实现都比较复杂。Java 语言中类与类之间的关系为单继承。

4. 多态

多态是指一个程序中同一方法名的不同方法体共存的情况。由于多态机制的出现，我们在设计时可以更容易关注程序的整体逻辑，而不必过分纠结于程序具体实现细节。在面向对象程序设计中，多态机制可以用来提高程序的抽象度和简洁性。多态机制的使用可以在单一界面的基础上，实现行为结果的多样化。多态服务是 OCP 原则在面向对象程序设计中的主要体现，也是面向对象程序设计模式的核心优势之一。

3.2　类 与 对 象

Java 程序中“万物皆类”，所有的代码都封装在类体中。类不仅是生成对象的模板，也是引用类型变量引用对象时的准则，还是引用变量调用对象成员的权利列表。

3.2.1　建立 Java 中的类

类封装了数据及其与数据相关联的方法。如汽车类，封装了轮子、发动机、方向盘、油门、刹车等属性，及前进后退、左转右转、停止等方法。当你拥有一个汽车对象时，不光拥有它的属性，还可以调用它的方法。

建立类的 Java 语法为：

```
[访问控制符] class 类名{
    [访问控制符] 变量类型 成员变量 1[=初始值];
    [访问控制符] 变量类型 成员变量 2[=初始值];

    [访问控制符] 返回值类型 成员方法 1(参数列表) {
        //方法体
    }
    [访问控制符] 返回值类型 成员方法 2(参数列表) {
        //方法体
    }
}
```

提示

这里的“访问控制符”相关知识见 3.2.3 节，为保持知识的连贯性，在 3.2.3 节之前的类、成员变量、成员方法的访问控制符都暂时使用默认的，即使用空白、不写任何控制符的方式定义类、成员变量和成员方法。

在类中定义的成员变量和成员方法是类的一部分，实例化为对象后它是对象的一部分，在声明引用变量的时候，它也是引用变量持有对象的检验标准，也是引用变量调用对象成员的权利列表。

1. 类体的声明

类声明时使用关键字 class，其后紧跟的标识符为类名，之后的大括号内的代码块则是类体。类名满足标识符的相关规定，Java 官方版本中类名首字母皆为大写，我们也推荐大

家沿用此规则。另外需要注意一点，在设计类名时，我们也建议避免与 JDK 中官方类名重名，减少因重名而引起的代码误用的可能性。

代码 3-1 里定义三个类，分别示范了最简单的类 C1、仅包含成员变量的类 C2、既包含成员变量又包含成员方法的类 C3。

```
//代码 3-1  定义三个类

class C1{}              //最简单的类
class C2{               //有成员变量的类
    int i;
    String name;
}
class C3{               //既有成员变量也有成员方法的类
    int i;
    void f(){
        i++;
    }
}
```

补充知识：类、class 文件与 Java 源文件的关系

（1）Java 源文件是 Java 的源代码文件。一个 Java 源文件里面可以写很多个类、接口或枚举，但最多只能有一个类、接口或枚举可以声明为 public 的，且这个 public 的类、接口或枚举的标识符需要与 Java 源文件的文件名相同。

（2）class 文件是在 JVM 中运行的 Java 程序。它是 Java 源文件编译后的产物。Java 源文件中每一个类（包括普通类、抽象类、接口及内部类），编译后都会生成对应的 class 文件。

（3）一个 Java 源文件编译后可能产生多个 class 文件，建议在一个 Java 源文件中只写一个类、接口或枚举并将其声明为 public 的，以方便快速找到 class 文件所对应的类所在源文件位置。

2. 成员变量的声明

在类体中，类中的成员变量类型可以是四类八种基本类型的变量，也可以是引用类型的变量。在类被实例化而产生的对象当中，若对成员变量赋有初始值，则对象中该成员变量的值为初始值。若没有对成员变量赋初始值，则该成员变量值为默认值。数字类型的默认值都是等于 0 的值（0/0L/0.0f/0.0），char 类型的默认值是‘\0’，boolean 类型的默认值为 false，引用数据类型的默认值为 null（空指针），与数组中元素的默认值规则一致。

3. 成员方法的声明

类体中定义的方法称为成员方法，成员方法的定义方式与第 2 章中方法的定义方式是一样的。成员方法的定义中依然包括返回值类型、方法名、参数列表。成员方法可以自由访问本类中的成员变量及其他的成员方法。

注意

成员方法和成员变量不能使用 static 修饰，成员方法也无法直接访问本类中 static 修饰的变量和方法。成员变量需要在对象出现后方能出现，而 static 修饰的变量和方法在类被加载到内存中时就已经完成了初始化和加载。一般成员变量和 static 修饰的成员从生命周期上来看是不一致的，因此，不能相互调用也是必然的。

```
//代码 3-2　测试成员方法对成员变量的可访问性
class TestClass{
    int shareVar;
    int i;                                    //默认值为 0
    void f(){
        System.out.println(shareVar);         //可访问成员变量,默认为 0
    }
    void g(){
        shareVar++;                          //可修改成员变量,能影响 f()中的输出结果
        int i =5;                             //局部变量具有更高的访问优先级
        System.out.println(i+"|"+this.i);     //i 是局部变量,this.i 是成员变量
        f();
    }
}
```

代码 3-2 展示了如下内容。

(1) 类中成员变量为成员方法所共有,一处修改会影响类内所有成员方法的运行结果,因此用户需要规划好成员变量的修改时机。

(2) 若方法体内的局部变量与类中成员变量同名,则程序会优先访问局部变量,如需访问成员变量需要使用“this.成员变量”的方式显式指明。

(3) 成员方法之间可以相互调用,请注意避免死锁情况的发生。

4. 特殊成员方法——构造方法的声明

构造方法是一种特殊的成员方法。

(1) 构造方法是与类名相同的,但不声明返回值的特殊成员方法。虽然没有明确地写出返回值,但是构造方法实际返回值是本类类型的引用类型变量,这个引用变量持有类实例化后的对象,通过引用变量间的赋值将这个对象移交给用户定义的引用变量。

(2) 构造方法仅在类实例化过程中(对象生成时)被调用一次,其他任何情况都禁止调用,因此构造方法不在引用变量的权利列表当中。

(3) 构造方法与一般成员方法一样可以自由地访问成员变量,调用成员方法。为此,构造方法经常被用来进行对象成员变量初始化工作。

(4) 构造方法一般会被声明为 public 的。以方便在其他类中调用构造方法实例化对象。

(5) Java 要求每一个类都要拥有至少一个构造方法。如果没有,如代码 3-1 中的三个类,系统会为该类生成一个无参的、默认空实现的构造方法。一旦用户自定义声明构造方法,默认构造方法失效。

代码 3-3 演示了几种不同模式的构造方法。

```
//代码 3-3　几种不同模式的构造方法
class DefineConstructor1 {
    public DefineConstructor1() {
        System.out.println("Hello! DefineConstructor 1!");
    }
```

```
    }

    class DefineConstructor2 {
        public DefineConstructor2(int k) {
            System.out.println("Hello! DefineConstructor 2!" +k);
        }
    }

    class DefineConstructor3 {
        int age;
        String name;

        DefineConstructor3(int a, String n) {
            age =a;
            name =n;
            printName();
            System.out.println("Hello! DefineConstructor 3!"+age +"|" +name);
        }

        void printName() {
            System.out.println("My name is " +name);
        }
    }
```

在代码 3-3 中可以看到：

(1) 在类当中，用户可以自由定义构造方法。如代码中出现了有参数构造方法，也出现了无参数构造方法。

(2) 构造方法可以像一般成员方法一样，访问类体中成员变量和其他成员方法，如 DefineConstructor3()就调用了本类成员 printName()和 age。

(3) 构造方法一般用于成员变量的初始化，在对象实例化时使用指定值替代原有成员变量的默认值。

3.2.2 对象的引用和对象的生成

类是现实世界中对象在 Java 语言中的抽象表示。在 Java 中，一个完整的类的作用有两个：①以类为模板生成(实例化)对象；②以类为模板声明引用类型变量。

为了叙述方便，我们定义了类 Stu，其中包括两个成员变量 name 和 age，一个成员方法 getName()和一个双参数的构造方法 Stu()，如代码 3-4 所示。

```
//代码 3-4 Stu 类
public class Stu{
    String name;
    int age;
    public Stu(String n, int a){
        name =n;
        age =a;
```

```
    }
    String getName(){
        return name;
    }
}
```

1. 声明引用类型变量

声明以类为类型的引用变量的 Java 语法为：

```
类名 引用变量名
```

对于以类为模板的引用变量可以认为是其持有对象应该遵守的规则和标准。它规定了持有的对象应该有什么样的成员变量和成员方法，如同为引用变量设定访问对象的“权利列表”。如果现有声明了一个 Stu 类型的引用变量 st，它持有一个对象 A：

```
Stu st =对象 A
```

那么对象 A 必须满足 Stu 类的要求：有两个成员变量 name 和 age，有一个成员方法 name()。当使用 st 变量时，可以通过 st 找到对象 A，按照 Stu 类提供的“权利列表”访问对象 A 中(可访问的)成员变量和成员方法。如代码 3-5 所示。

注意

构造方法仅在对象实例化时使用，因此不在引用变量的“权利列表”当中。

当引用变量 st 的生命周期结束后，这个对象 A 没有一个来自于对象外部的引用，那么 JVM 就将对象 A 标定为“系统垃圾”，在适当的时候由 Java 的垃圾回收机制清空该块内存。

2. 实例化对象

对于一个完整的类(所有成员变量都有值，所有成员方法都有方法体)而言，JVM 会根据类的定义在内存中构造一个对象。这个过程就像使用图纸生产一辆汽车一样。每调用一次构造方法(图纸)，就生成一个对象(生产一辆汽车)。类的实例化的 Java 代码格式是：

```
new 构造方法名(参数列表);
```

注意：调用构造方法需要匹配的构造方法的声明。如一个类没有定义构造方法，只使用默认无参数空实现的构造方法，则实例化代码为“new 类名()”；实例化代码 3-4 中 Stu 类对象时，因为用户自定义构造方法，默认无参空实现的构造方法被停用，需要按 Stu 类中显式定义的构造方法的要求提供两个参数：

```
new Stu("李雷",19)
```

若此时使用“new Stu()”这个构造方法，则会提示未发现匹配的构造方法，编译错误。

3. 引用变量与对象的匹配

使用类声明引用变量和使用类实例化对象的代码是可以分开进行的。这就会出现引用变量由充当变量类型的“类 1”来声明，对象由“类 2”为实例化生成的情况。如果对象满足引用变量的“要求”，那么这个对象就可以由该引用变量来引用，如图 3-1 所示。

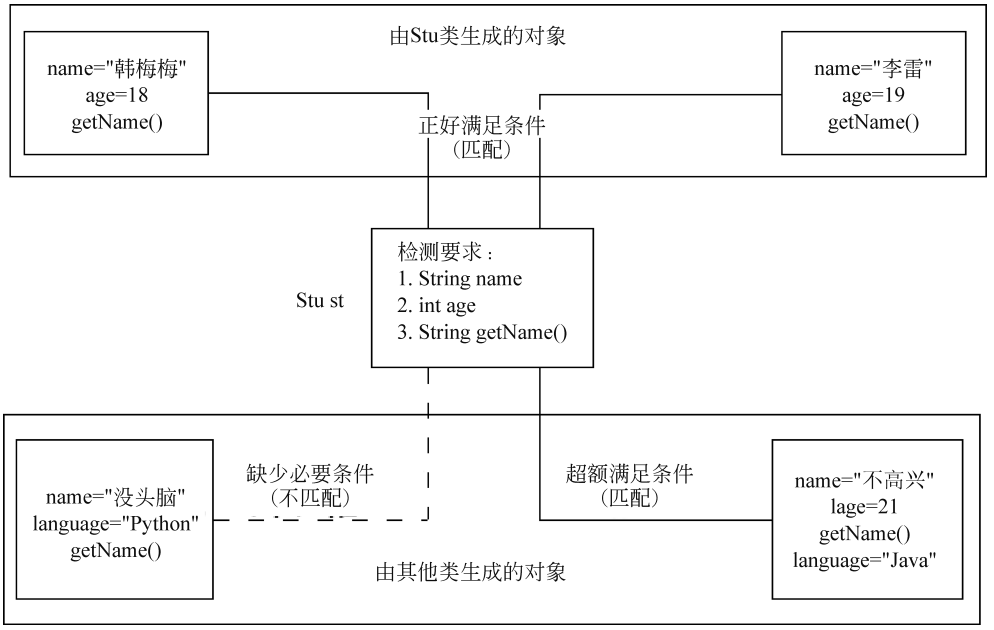

图 3-1　引用变量和对象的匹配

如图 3-1 所示，引用变量 st 按照 Stu 类模板对其可能引用对象提出了三个要求：两个属性、一个成员方法。

（1）使用 Stu 类生成的“李雷”与“韩梅梅”自然满足这些要求。

（2）“不高兴”这个对象，里面包含的成员比 Stu 类的要求更多，因此匹配 st 变量要求，可以被 st 引用。实际上，“不高兴”是由 Stu 类的子类生成的对象，其拥有成员自然不少于 Stu 类的要求。

（3）“没头脑”这个对象，因为不满足 st 变量的要求，不能被 st 引用。

代码 3-5 继续使用了代码 3-4 中定义的 Stu 类，演示了引用变量的两个作用。

（1）持有一个满足 Stu 类要求的对象“Stu st = new Stu("李雷",19)”。

（2）按照类 Stu 给出的权利列表（两个属性，一个方法），向其持有对象发出访问请求。对象因为满足这样的规则（已经被引用变量引用），可以正确响应引用变量的访问请求，访问对象成员 age、name 和 getName()。

```
//代码 3-5  TestStu 类
public class TestStu {
    public static void main(String[] args) {
        Stu st =new Stu("李雷",19);          //引用变量与对象匹配;调用 Stu 构造方法
        int age =st.age;                     //按权利列表调用属性,对象响应正确
        String name1 =st.name;               //按权利列表调用属性,对象响应正确
        String name2 =st.getName();          //按权利列表调用方法,对象响应正确
        System.out.println(age +"||" +name1 +"||" +name2);
    }
}
```

此外，从 Java 语法来看，引用变量也是一个变量，满足变量的三个要素：变量类型、变量

名、变量的值。这里的变量的值可以理解为对象。引用变量的值可以由两种方式得到。

(1) 由类的构造方法赋值。使用此方法,引用变量持有由构造方法返回的对象,如“Stu st = new Stu("李雷",19)”。

> **提示**
>
> 引用变量实际上是指向对象的指针,但是Java语言在建立之初就消灭了指针这个概念,引用变量除了赋值以外,不参与任何运算。引用变量指向对象一切工作都是JVM自动完成,在Java中,可以简单认为引用变量的值,不是地址,而是对象。

(2) 由其他引用类型变量赋值。使用此方法,引用变量持有使用其他引用变量传递过来的对象,Java语法为:

```
类A引用变量名 =其他引用变量;
Stu nfs =st;
```

3.2.3　访问控制

信息隐藏是面向对象程序设计最重要的功能之一,它通过为类中成员,包括static成员加载访问修饰符来完成。通过访问修饰符的加入,可以对类、类中成员(包含成员变量、成员方法、内部类等)、类中包含的static成员等内容进行可访问权限控制,达到屏蔽无权限访问的目的,为类内数据存储和方法运行打造了一个可控的运行环境。

Java将访问请求按请求发起者与本类之间关系分为以下四个层级。

(1) 当前类。来自于同一个类内部的访问请求。

(2) 当前包。来自于同一个包(package)其他类中的访问请求。

(3) 子类。来自于子类中的访问请求。这里子类可能与父类可以不属于同一个包。

(4) 无限制。来自任意类中的访问请求。

> **补充知识:包**
>
> Java使用包(package)来管理类。Java中的同一个包的判断标准有两个,源文件中package语句声明相同且编译后class文件在同一个目录,见4.1节。

1. 类的访问控制符

对于类、枚举、接口而言,修饰符可选public,若不声明,空缺就表示该修饰符是默认的(friendly)。public对应权限为“无限制”,即表示这个类在任何地方都是可见的,在任何地点发出的访问请求(如实例化、声明引用类型)都会被响应。friendly对应权限为当前包,表示只有来自同一个包中的访问请求才会被响应,而来自其他包的访问请求会被拒绝。类、枚举、接口的访问控制如表3-1所示。

表3-1　不同地方发出访问请求后,类、枚举、接口的访问控制

访问控制符	当前包	无限制
空缺(默认,friendly)	√	×
public	√	√

2. 成员的访问控制符

对于类内成员（成员变量，成员方法，内部类）而言，其访问控制（修饰）符有四种：public、protected、friendly（默认）、private，若不声明，空缺就表示该修饰符是默认的（friendly）。它们对于不同地方发起访问的响应方式如表 3-2 所示。

表 3-2　不同地方发出访问类内成员的请求后许可情况

访问控制符	当前类	当前包	子　类	无限制
private	√	×	×	×
空缺（默认，friendly）	√	√	×	×
protected	√	√	√	×
public	√	√	√	√

表 3-2 中，√表示该成员会响应访问请求，×表示该成员会拒绝该访问请求。

在实际使用中，public 和 private 这两种权限的使用频率是最高的。将成员变量 private 化，可以有效地管理外界对于封装数据的访问，一般来说，用户会使用 getter()方法和 setter()方法来分别控制成员变量是否可读写，并在这两个方法当中加入相应代码进行一些高级操作，对返回值进行加工，并对注入值进行过滤清洗。代码 3-6 将展示 private、public 权限的使用。

```
//代码 3-6　有访问控制符修饰的类成员
public class Girl {
    private double age =18.5;
    public int getAge() {
        innerLife();
        return (int)(18+(age-18) * 0.05);
    }
    public void setAge(double outB){
        age =18 +0.05 * (outB-18);
    }
    private void innerLife() {
        System.out.println("我不会告诉你我真实的年龄");
    }
}
```

从代码 3-6 可以看到：

（1）在类内部，成员方法可以自由访问成员变量和其他成员方法，如 getAge()可以自由访问本类中的成员 age 和 innerLife()；setAge()可以自由访问 age。

（2）public 的 getter()方法，命令方式：get＋变量名，可返回加工后的变量。保证了 Girl 类对象中成员变量的私密性。

（3）public 的 setter()方法，处理外部信息后，再存入成员变量。保证了 age 这个变量的安全性，可以按照设计要求进行清洗再加工。

3. 访问对象（非 static）成员的方法

在类中，定义了成员变量与成员方法。当类实例化为一个对象后，这个对象也包含相应

的成员变量和成员方法。访问该对象的成员需要持有对象的引用变量的权利列表中有该成员(见 3.3.5 节引用类型变量的转型)。

在 Java 中,引用类型变量通过运算符“.”访问对象内的成员变量或调用成员方法,注意这种访问需要确定本处代码与目标类之间的访问关系(当前类、当前包、子类或无限制)是否满足成员变量与成员方法的访问控制条件(public、friendly、protected 或 private)。

```
//代码 3-6(续)　访问控制符约束下,外界对类成员的访问
public class TestGirl {
    public static void main(String [] args){
        Girl girl =new Girl();
        //int girlAge =girl.age ;          //private 属性不可访问
        //girl.innerLife();                //private 方法不可访问

        int girlAge =girl.getAge();        //public 方法
        System.out.println(girlAge);

        girl.setAge(25);                   //public 方法
        girlAge =girl.getAge();            //public 方法
        System.out.println(girlAge);
    }
}
```

运行结果:

```
我不会告诉你我真实的年龄
18
我不会告诉你我真实的年龄
18
```

在代码 3-6(续)中,main()方法中声明了 Girl 类的一个对象 girl 来测试其成员变量的可访问性。相对于 Girl 类对象 girl 而言,来自 TestGirl 类中 main()方法的访问都属于“与对象毫无关系的对象”(无限制区域)发出的访问请求,只能访问 Girl 类对象中 public 权限的。如 public 修饰的 getter()/setter()方法、getAge()和 setAge()是可以访问的,而 private 的成员变量 age 和成员方法 innerLife()存在于对象 girl 当中,可以运行,但拒绝来自类外部(TestGirl 类中)的访问。

3.2.4　类定义中的多态——重载

在 Java 的语法中规定了“在类内,若成员方法的方法名相同,但参数列表不同,则构成方法的重载。对于参数列表,只要参数类型、参数个数或参数顺序三者其一不同即可认定两个参数列表不同”。方法重载是多态机制的一个重要表现形式,需要注意的是,重载只在类所定义的范围内有效。

```
//代码 3-7　成员方法的重载和调用
public class TestOverload {
    public static void main(String[] args) {
        Overload or =new Overload ();
```

```
        or.f();                    //调用无参方法()
        or.f(1);                   //调用单参方法(int)
        or.f(0.1);                 //调用单参方法(double)
        or.f(1, 0.1);              //调用双参方法(int,double)
    }
}

public class Overload {            //独立的 Java 源文件: Overload.java
    public void f() {
        System.out.println("f without any parameter!");
    }

    public void f(int i) {
        System.out.println("f with an integer parameter!");
    }

    public void f(double d) {
        System.out.println("f with a double parameter!");
    }

    public int f(int i, double j) {
        System.out.println("f with two integer parameters!");
    return 1;
    }
}
```

运行结果：

```
f without any parameter!
f with an integer parameter!
f with a double parameter!
f with two integer parameters!
```

在写重载方法时需要注意：

(1) 两个成员方法是否构成重载的两个条件：方法名相同且参数列表不同，与它们的返回值类型无关。如代码 3-7 中：

```
void f();
void f(int i);          //参数列表不同,构成重载,合法
//int f();              //参数列表相同,不构成重载,非法
```

(2) 使用 JVM 调用重载方法时，如无完全符合要求的方法，则 JVM 会尝试对参数自动转型，就近匹配。这时 JVM 选用重载方法按“就近原则”进行匹配。如代码 3-7 中，TestOverload 中 or.f(1)中参数为 int 类型变量，按照自动转型的原则(见 2.5.1 节)，f(int i)与 f(double d)都可以响应。按“就近原则”，JVM 使用了 f(int i)。假如 Overload 类中没有 f(int i)，只有两种方法 void f(float k)与 void f(double d)，按就近原则，JVM 会选择 void f(float k)。

(3) 构造方法允许重载。显式写构造方法时，都会写一个无参数的构造方法，这个做法

有助于简化子类的构造方法的代码(详见 3.3.4 节)。

3.2.5　类定义中的其他问题

1. this 指针

this 指针是类体内指向自身的一个引用,它是一个内部引用,无法在类体外使用。在程序中,一般使用“this.”来显式调用类内成员方法与成员变量,可以从形式上区别方法体内同名的局部变量。用户可以使用“this(参数列表)”来调用类体内其他重载的构造方法。注意:调用其他重载构造方法的语句“this()”必须写在构造方法的第 1 行。

```
//代码 3-8  this 指针与 this()方法的使用示例

public class TestThis {
    public static void main(String[] args) {
        CThis ct =new CThis();
    }
}

public class CThis {
    public int age;
    public CThis(int age) {
        this.age =age;              //不写 this,JVM 会优先选择局部变量
        System.out.println("CT with a parameter");
    }
    public CThis() {
        this(3);                    //调用本类其他的构造方法
        System.out.println("CT without any parameter");
    }
}
```

运行结果:

```
CT with a parameter
CT without any parameter
```

从代码 3-8 来看,使用 this 指针可以显式地调用类成员变量,this.age 就明确表示这里使用的是 CThis 类的成员变量,而不是 CThis 构造方法中的局部变量。而 this(3)则表示调用本类其他构造方法,被写在构造方法体的第一行。

2. static 关键字

static 用来声明类内静态成员变量和静态成员方法(2.7 节中定义的方法和 main()方法都是静态方法),也可以声明静态代码块。

```
public class 类名{
    [访问控制符] static 变量类型 静态变量名;
    [访问控制符] static 变量类型 静态方法名(参数列表){方法体语句块}
    static { 静态代码块 }
}
```

静态变量和静态方法可以直接通过类名来访问,Java 语法为:

```
类名.静态方法名(参数列表…)
类名.静态变量名
```

提示

（1）非静态的一般成员（成员变量和成员方法）属于对象，静态成员属于类。一般成员可以被子类继承，但静态成员无法被继承。

（2）用户只有在生成对象后方可访问非静态对象，而静态成员在类代码（第一次被代码引用时）加载后即可以访问。

（3）静态成员可通过类对象的引用变量访问，但不建议这样使用，因为这样做容易引起不必要的误解。

静态代码块是在类中独立于类成员的 static 语句块，可以有多个，且位置不受限制。多个静态代码块，JVM 加载类时会按代码顺序，从上到下依次执行这些静态的代码块。如代码 3-9 所示。

```
//代码 3-9  静态代码块,静态成员的定义
public class StaticCode {
    private static int staticA;          //static 变量
    public int normB;                    //成员变量,不能被 static 直接调用

    static {                             //第一顺序代码段
        StaticCode.staticA =3;           //访问 static 变量
        System.out.println("First: " +staticA);
        TestStaticCode.staticG();        //调用 static 方法
    }

    static {                             //第二顺序代码段
        StaticCode.staticA =5;
        System.out.println("Third: " +staticA);
    }

    public static void staticG() {       //static 方法
        StaticCode.staticA =4;
        System.out.println("Second: " +staticA);
    }

    public void f() {                    //成员方法,不能被 static 直接调用
        System.out.println("Hello World!");
    }

    public static void main(String[] args) {
        //未实例化对象,只是将 StaticCode 类加载到内存当中
        System.out.println("Main!");
    }
}
```

输出结果为：

```
First: 3
Second: 4
Third: 5
Main!          //先加载类,后执行 main 方法
```

如代码 3-9 展示的结果,在 JVM 调用 main()方法时,需要加载 StaticCode 类,在此过程中触发 static 代码块自动执行。而代码 3-10 的运行结果表明,在没有代码涉及 StaticCode 类(代码 3-9 中定义)之前,static 代码块是不会自动执行的。

```
//代码 3-10  类内静态元素使用注意事项
public class TestStaticCode {
    public static void main(String[] args) {
        System.out.println("内存为空,未涉及 StaticCode 类");

        StaticCode.staticG();
        System.out.println("第一次,涉及 StaticCode 类,先执行 static 代码块,
                           后执行 staticG()");
        System.out.println("共享静态变量: "+StaticCode.staticA);

        StaticCode t1 =new StaticCode();      //实例化一个对象,正常访问成员变量
        t1.f();
        t1.normB =1000;
        System.out.println(t1.normB);
        System.out.println("t1 静态变量: "+t1.staticA);

        t1.staticA =50;                       //通过对象,修改 static 变量

         //内存中共用的 staticA 被修改,所有对象都受到影响!!
        System.out.println("类名静态变量: "+StaticCode.staticA);
        //类名访问。受影响!

        StaticCode t2 =new StaticCode();      //第二次加载,静态代码不再执行
        System.out.println("t2 静态变量: "+t2.staticA);
        //新对象后访问,同样受影响!

        System.out.println("t2 成员变量: " +t2.normB);
        //一成员变量是独立的
    }
}
```

程序输出结果如下。

```
内存为空,未涉及 StaticCode 类
First: 3
Second: 4
Third: 5
Second: 4
第一次,涉及 StaticCode 类,先执行 static 代码块,后执行 staticG()
共享静态变量: 4
```

```
Hello World!
t1 成员变量: 1000
t1 静态变量: 4
类名静态变量: 50
t2 静态变量: 50
t2 成员变量: 0
```

代码 3-10 验证了如下结论：

(1) 静态变量及静态方法独立于该类的任何对象。在类未实例化对象之前，就可以通过类名直接访问，如代码中的 StaticCode.staticG()和 StaticCode.staticA 等。

(2) 静态代码在类第一次出现在代码中时，自动加载运行，且只运行一次。如在 TestStaticCode 的 main()中，开始时未直接加载 StaticCode 类，直到出现 StaticCode.staticG()语句，第一次加载 StaticCode 类，这时 StaticCode 中的两个静态代码块才依次运行；在第二次使用 StaticCode 类时“StaticCode t2 = new StaticCode()”，静态代码块将不再运行。

(3) 静态变量看起来也是类内成员，除了通过类名直接访问外，也可以通过引用变量访问，因此，也经常称静态变量为静态成员变量。如 staticA 变量可以通过 StaticCode.staticA 访问，也可以通过 t1.staticA 访问。然而，静态变量并不是真正的类内成员，如图 3-2 所示。

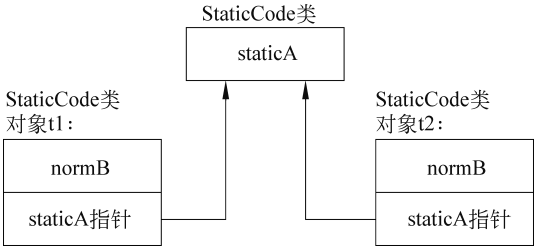

图 3-2 StaticCode 中静态成员与一般成员之间关系

类的成员变量和成员方法存在于对象当中，只有在对象实例化之后才能出现。静态成员独立于对象，在类代码加载后就会出现在内存当中，静态变量是类的所有实例所共有的，且是唯一的。一处修改，就会影响所有对象。成员变量存在于对象当中，彼此独立，如 t1 的 normB 和 t2 的 normB。

从生命周期来看，类的非静态成员的生命周期与静态成员也是不一样的。非静态成员诞生于对象实例化，而静态成员诞生于类代码加载；非静态成员消亡于对象被回收，而静态成员始终存在。因此，对于程序来说，只有一些必要的、常用的方法和变量才会被声明为静态，减少静态成员的使用有助于提高内存的利用率。

3. 关于类的一些约定俗成

(1) 将所有的成员变量声明为 private，利用 public 的 getter()、setter()方法分别去设定读写控制。getter()、setter()方法的命名规则为：

```
public 所读变量的类型 get+“要读取变量名”(){                    //… }
public void set+“设定值的变量名”(变量类型 变量名){             //… }
```

如对私有变量 int privateElement 的 getter()、setter()方法的典型写法如下。

```
public int getPrivateElement(){
    return privateElement;
}
public void setPrivateElement(int inputElement){
    this.inputElement =inputElement;
}
```

(2) 尽量避免将一个变量设定为 static 的,更不要使用它在不同对象之间传递值。因为这样会打破类的封装性,且容易出现不可预料的错误。一般声明公用变量,为避免被意外修改,会将其声明为常量——"static final"的。

(3) 纯粹服务性质与具体数据无关的方法,可将其声明为 public static 的。如 Java 中常用的工具类有 java.lang.Math 类、java.util.Arrays 类和 Collections 类等。

(4) 每个类单独放一个 Java 文件,并将其声明为 public 的。

3.3　继　　承

在面向对象程序设计中,继承所表达的就是类与类之间的包含关系。类 A 继承了类 B,A 是一种 B,属于类 A 的对象具有类 B 的全部性质(属性)和功能(方法)。

我们称被继承的类 B 为基类、父类或超类,而称继承类 A 为 B 的派生类或子类。从概念的角度来看,父类更通用,子类更具体。现实生活中,这样的例子很多,如摩托车和汽车都属于机动车,摩托车类和汽车类继承了机动车类。

Java 中,类 A 继承了类 B,使用关键字 extends 声明一个类是从另一个类继承而来,语法为:

```
[public] class 父类{ }
[public] class 子类 extends 父类{ }          //单继承,只有一个父类
```

Java 类与类之间只支持单继承,不支持多继承,即子类只允许拥有一个直接父类。

3.3.1　父类是共同代码的抽象

Java 继承是基于已存在的类建立新类的技术,子类只需要声明继承父类,即可复用父类中的代码。父类承载所有子类的通用设定和共用代码,子类专注于描述子类所特有的功能和特征。对于摩托车类和汽车类,设定如下。

```
摩托车类:      属性(牌照号),方法(启动,停止)
汽车类:        属性(牌照号),方法(启动,关窗)
```

代码如下。

```
public class Motor{
    private String license;
    public Motor(String license){
        this.license =license;
    }
    public void start(){
```

```
            System.out.pritnln(license +“starts”);
        }
        public void shutDown(){
            System.out.println(“The moter was parked at the roadside.”);
        }
    }
    public class Car {
        private String license;
        public Car (String license){
            this.license =license;
        }
        public void start(){
            System.out.pritnln(license +“starts”);
        }
        public void closeWindows(){
            System.out.println(“The car is closing the windows.”);
        }
    }
```

对于这两个类，可以发现其部分代码高度重复。对于方法，高度重复的代码可以提取出来成为一个新的方法，原先的方法调用新方法即可。对于类，同样可以提取相同代码形成一个新父类。这样子类就不需要重复性地写入代码，代码也更加简洁，也提高了代码的复用性。使用继承改造后的类如代码 3-11 所示。

```
//代码 3-11  继承中父类的构造
public class Vehicle{                    //父类
    private String license;
    public Vehicle (String license){
        this.license =license;
     }
    public void start(){
        System.out.println(license +"starts");
    }
}

public class Motor extends Vehicle {    //子类 1
    public Motor (String id){
        //license =id ;                  //不能直接访问父类 private 成员
        super(id);                       //调用父类 public 的构造方法
     }
    public void shutDown(){
        System.out.println("The moter was parked at the roadside.");
    }
}

public class Car extends Vehicle {      //子类 2
    public Car (String id){
```

```
        super(id);
    }
    public void closeWindows(){
        System.out.println("The car is closing the windows.");
    }
}
```

3.3.2　继承对于对象和引用变量的影响

子类继承父类，则子类对象既包含子类中定义的(非静态的)成员，也包含父类中的所有(非静态的)成员，就好像一个汤圆，声明汤圆这个子类对象的时候，其在核心内会包含黑芝麻芯这样的一个父类对象。

因为从引用的角度来看，子类对象既满足了子类类型引用的限制条件，也满足了父类类型引用的限制条件，因此，子类对象既可以由父类引用变量来持有，也可以由子类引用变量来持有。

```
//代码 3-12　父类变量持有子类对象
public class TestInherit {
    public static void main(String[] args) {
        Son s =new Son();
        System.out.println("father's varible: " +s.fatherInt);
        System.out.println("son's varible: " +s.sonInt);
        s.fatherFunction();
        s.sonFunction();

        Father f =s;                    //Son 对象,满足 Father 类引用类型要求
        System.out.println("father's varible: " +f.fatherInt);
        f.fatherFunction();

        //因为 Father 类没有 sonInt 和 sonFunction()成员,所以下面的语句非法
        //System.out.println("son's varible: " +f.sonInt);
        //f.sonFunction();
    }
}

public class Father {
    public int fatherInt =10;

    public void fatherFunction() {
        System.out.println("Father's function!");
    }
}

public class Son extends Father {       //Son 类继承自 Father 类,隐式包含
                                        //Father 类的成员变量及成员方法
    public int sonInt =11;
    public void sonFunction() {
```

```
            System.out.println("Sun's function! " +fatherInt);
        }
    }
```

运行结果：

```
father's varible: 10
son's varible: 11
Father's function!
Sun's function! 10
father's varible: 10
Father's function!
```

通过代码 3-12 可以看到：

(1) 通过子类引用变量 s，可以访问对象父类成员 fatherInt 与 fatherFunction()，证明子类对象中包含一个父类对象。

这种生成子类对象的同时，也生成父类对象的行为，方便了程序灵活配置子类对象行为，但也为程序运行速度带来麻烦。因此，有人嘲讽 Java 的这种行为是："You wanted a banana but you got a gorilla holding the banana"。①

(2) 子类对象匹配父类引用变量中对于成员的要求，可以被父类类型的引用变量所持有，如 Father f = s。

(3) 引用变量可调用对象成员的权利列表来源其所依赖的类，而不是其持有的对象。即使父类引用变量持有子类对象，但是由于父类的定义，该引用变量无法访问子类对象中特有的成员。如 f.sonFunction()和 f.sonInt 都是非法的。

3.3.3 重写与多态

方法重写(Override)的过程就是在子类中创建一个与父类成员方法声明一致(方法名相同、返回值类型相同、参数列表相同)但方法体不同的方法。方法重写有时也被称为方法覆盖、方法复写。

子类中方法的重写可以在子类中实现不同于父类的功能。之所以使用重写，而不是新增一个方法，其重要原因是为了在父类类型的引用变量权利列表不变的条件，通过加载不同子类对象，实现相同方法调用不同实现功能。这一机制就是面向对象程序中最重要的多态机制。多态实现的代码基础就是继承和子类对父类方法的重写。父类引用类型变量 a，根据父类定义获取一个可调用方法 functionA()的权利；子类对象被变量 a 所持有；当 a 准备使用方法 functionA()，JVM 从"汤圆"模型的子类对象的外层向内层查找方法 functionA()。因为重写的严格要求，JVM 会直接匹配子类中的 functionA()，执行子类的 functionA()而不是父类的 functionA()。多态的实现效果就是父类的引用变量通过加载不同的子类对象，调用父类中的方法名，但执行的却是子类中的方法体。

关于重写需要注意的是：

① 你想要一根"香蕉"，但你却得一只握着香蕉的大猩猩。一个包含了父类对象的子类对象。—编辑注