第3章 面向对象程序设计原理和Java语言实现

正如第1章所提到的，最初的计算机程序设计语言主要集中在指令和语义的抽象上，并将操作和被操作的数据隔开，分为代码段和数据段。随着程序的复杂度越来越高，这种方法的局限性越来越大，后来计算机科学家将程序设计中模块化的思想进一步强化，将待操作的数据和相应的操作行为组织在一起，形成了著名的面向对象程序设计方法。在面向对象设计中，用户可以从大量的实际对象当中抽象出一个个类，类中有描述对象内部结构状态的数据部分，还有描述此对象具有的功能和行为的指令部分。在计算机进程中对象又是类的具体化实例，用一个设计好的类可以创建许多对象。简单地说，类就是对象的代码抽象，是创建对象的代码模板，对象是类的具体化的实例。对象比结构化程序设计中讲到的函数、过程更具有实际意义，人类认识世界的基础就是从一个一个基本的实际对象开始的，所以，面向对象程序设计具有现实的认知基础。

从世界观的角度来看，面向对象的基本哲学认为世界是由各种各样具有自己的运动规律和内部状态的对象所组成的，不同对象之间的相互作用和通信构成了完整的现实世界。因此，程序员应当按照现实世界的本来面目来理解世界，直接通过对象及其相互关系来反映世界，这样建立起来的系统才能符合现实世界的本来面目。

从方法学的角度来看，面向对象的方法是面向对象的世界观在程序开发方法中的直接运用。它强调系统的结构应该直接与现实世界的结构相对应，应该围绕现实世界中的对象来构造系统，而不是围绕功能来构造系统。

从程序设计的角度考虑，程序对象应该是将组成对象的数据代码和对象具备的功能代码封装成一个整体，符合强内聚和弱耦合的原则。当然，面向对象的程序设计语言必须有描述对象内部结构及其相互之间关系的语法和规则。

3.1 面向对象程序设计的基本概念

对象和类是面向对象程序设计的核心概念，程序员使用对象和类进行程序设计。类可以分成两种，一种是程序员可以直接使用的类，是由JDK提供的或其他人写好的；另一种需要程序员自行设计。设计一个类大致可分为下面两步：

(1) 对现实世界的实体进行抽象，抽取其合适的状态和行为，形成思维中的类；

(2) 用Java语言来描述思维中的类，使思维中的类变成一个Java语言类。

对象是类的实例，同一个类可以建立多个对象实例，通过使用对象可以达到程序设计的

目的。对象和类既有区别又有联系,类是创建实例对象的代码模板,对象则是按照类创建出来的一个个实例,有点像汽车的设计图纸和汽车的关系。采用面向对象程序设计技术的原因主要有两个,一是我们认识、研究世界乃至于改造世界都是以“对象”为基本单位进行的,将这一人类活动衍生到计算机编程中顺理成章;二是为了提高程序设计的效率,尤其是在越来越复杂的问题环境中解决模块的颗粒度问题,即内聚性和耦合性的分界线问题。

3.1.1 对象

在现实世界中,对象一般指一个独立于我们的客观实体,它一般有一定的内部结构,对外表现出一定的属性和行为,并且和周围的世界有一定的交互性,至少占有一定的空间和时间。

在面向对象程序设计中,对象就是数据空间结构加上指令代码,或称数据与代码的组合。换句话说,面向对象编程中的最基本的概念就是“对象”(Object),它是理解面向对象程序设计技术的关键。为了理解这一点,我们首先来环顾现实世界中的对象。我们周围的汽车、电视机、狗、书桌、自行车等都是现实世界中的物理实体,也就是我们所说的对象。现实世界中的对象具有 3 个特征,即状态、行为和事件响应能力,例如,自行车有状态(传动装置、步度、两个车和齿轮的数目等)和行为(刹车、加速等)。对事件的响应能力是通过行为方法实现的,它代表了一个对象和周围世界或其他对象的一种交互能力。其次,我们再来看看软件对象,软件对象是现实世界中的对象在计算机内部的模拟化产物,它们也有状态和行为。软件对象把状态用数据来表示并存放在变量中,而行为则用方法(程序代码)来实现。

软件对象用于模拟现实世界中的实体,其对象模型是根据现实世界中实体的状态和动作来确定的,程序设计中的对象用特定的数据结构表示其内部组织结构,用操作数据的方法描述其可产生的行为。在程序设计中,不仅现实世界中的物体可以表示为一个对象,一个抽象的概念也可以是一个对象。例如在图形用户界面的交互式窗口系统中,“事件”就是一个普通的对象,用来表达用户对鼠标或键盘的操作。

把一个对象的数据加以包装并置于其方法的保护之下称为封装。所谓封装,就是对内部数据的隐藏。封装实现了把数据和操作这些数据的代码包装成一个对象,而将数据和操作细节(方法)隐藏起来。如果增加某些限制,使得对数据的访问可按照统一的方式进行,这样就能比较容易地产生更为健壮的程序代码。

面向对象程序设计还体现了另外一个哲学思想,即意识和物质的不可分性,行为和肉体的不可分性。以人的大脑为例,如果没有了大脑的神经元细胞物质组成部分,也就无所谓意识,大家都会明白,这样的人是不会存在的,因为意识没有载体。同样,如果一个人的大脑的物质组成正常,将其意识行为去掉,大家明白这样的人只是“植物人”,不是一个真正的人。所以,结构决定了行为,行为又改变着结构,它们是不可分的一体两面。

我们只有设计出精巧的数据结构再配以合适的方法代码加上继承和多态,才是真正的面向对象程序设计。

3.1.2 类

在我们的物理或生物世界中,类代表一种抽象概念,例如猫代表一类动物的统称,它们都具有一些基本的、相同的属性和行为。在科学研究中我们使用类属概念将世界分门别类,

再进行归纳、演绎和研究。

在计算机程序设计中，类是一个蓝图或模板，定义了某种类型的所有对象具有的数据特征和行为特征。在 Java 语言中，程序设计的基本单位就是类，也就是说，一个 Java 程序是由许多设计好的类组成的。对象实际上是程序运行时通过类创建的一个实例，生成实例的过程称为“把一个对象实例化”。一个实例化的对象实际上是由若干个实例变量和实例方法组成的。当创建出一个类的实例时，系统将为实例变量指定内存，然后即可利用实例方法去做某些工作。

为了更好地理解类和对象的概念，我们拿生命现象中的相关概念进行类比。如果将胚胎细胞中的 DNA 分子链看成是由 4 种基本的核糖核酸 ATGC 组成的编码序列，是一种代码模板，对应的就是我们此处讲的类。那么经过胚胎发育，最后成长为一个个生物个体，例如人、猫、狗、马等，就是由此代码模板创建的一个个实例对象，实例对象具有生命周期，对应于内存中对象的初始化、消息交互、死亡等。

3.1.3　消息

跟我们所处的世界一样，不存在孤立系统，所以在程序中，一个孤立的对象是没有用的。对象往往是作为一个组员出现在包含有许多其他对象的大程序或应用软件之中，通过这些对象的相互作用可以实现高层次的操作和更复杂的功能。某对象通过向其他对象发送消息与其他对象进行交互和通信。例如，当对象 A 要执行对象 B 中的方法时，对象 A 便发送一个消息给 B。有时，接受消息的对象需要更多的信息以便能精确地知道做什么。消息以参数的形式传递给某个方法，一个消息通常由下面 3 个部分组成：

- 接受消息对象的名称；
- 要执行方法的名称；
- 方法需要的参数。

举例说明，胡亥给李斯打电话，叫他准备明天早上 10 点上朝；在这里，胡亥是消息的发送者，李斯是消息的接收者，将要执行的方法是“上朝”，参数是第二天早上 10 点。

消息的优点在于提供了对象交互的统一手段，不同进程中或不同计算机上的对象也可以通过消息相互作用。在计算机程序中，消息实际上就是一个方法调用过程，是一个类或对象调用其他对象或类方法的过程。

3.2　面向对象程序设计的基本原则

面向对象程序设计原则也就是我们在设计程序时应该遵循的基本设计思路，这些原则是若干计算机程序员积累下来的对我们设计程序很有用的技巧和指导原则。在第 1 章中已经简单地对这些原则进行了介绍，本章进一步强化和细化这些概念。

3.2.1　抽象原则

抽象就是从大量的、普遍的个体中抽象出共有的属性和行为，从而形成一般化概念的过程。在现实世界中，人们正是通过抽象来理解复杂的事物。例如，人们并没有把汽车当作成

百上千的零件组成来认识,而是把它当作具有自己特定行为的对象。我们可以忽略发动机、液压传输、刹车系统等如何工作的细节,而习惯于把汽车当作一个整体来认识。

如果从一个抽象模型中去除足够多的细节,则它将变得足够通用,足以适用于多种情况或场合,这样的抽象在程序设计中非常有用。经过对大量事物的抽象和归类,可以形成相应的类属层次,图 3-1 所示为自然界中各事物的一个分类抽象。

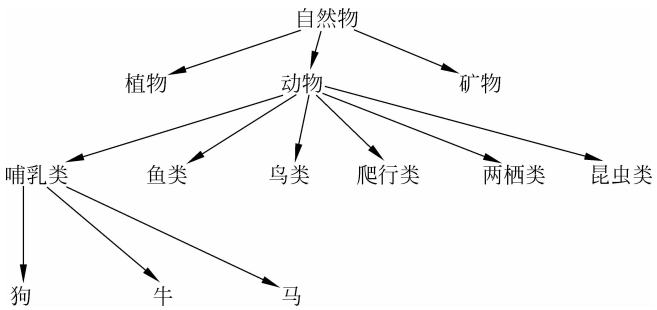

图 3-1 抽象示意图

3.2.2 封装原则

封装原则是一个自然法则,正如我们看到的、学习到的,以及我们周围的一切实体,包括动物、植物、各种人造物品都是封装的。一般情况下,我们只能看到这些物体的“外壳”,而看不到其内部结构。

这种将内部结构和功能对外隐藏,只留下必要的接口和外界进行能量或信息交流的机制就是封装。例如人类,内脏、血管、神经都被封装在皮肤里面,对外表现出来的仅仅是皮肤和五官接口,也就是说,我们都是内聚性很强的对象个体,但我们又留有眼、耳、鼻、口等接口,我们通过这些接口在世间生存和忙碌。如果仔细观察,动物世界中的大多数动物都具有很好的封装性。

在面向对象程序设计中,我们应该遵循同样的原则,将对象的内部结构对外做信息隐藏,让外部不可访问,但提供一系列的公有接口,用来进行信息和能量交换。在 Java 语言中,实现封装的关键字是 private,提供接口的关键字是 public,如图 3-2 所示。

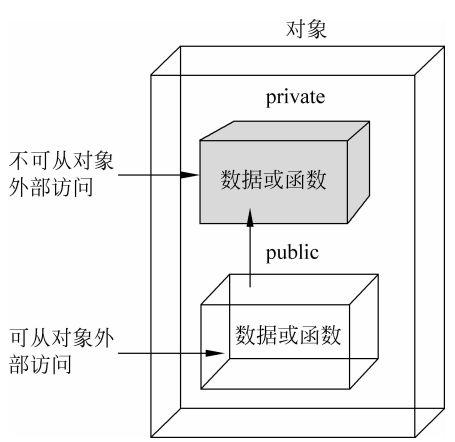

图 3-2 封装示意图

3.2.3　继承原则

继承原则也是一个自然法则，正如第 1 章提到的一样，如果没有继承，我们的生物世界就会永远处在生物链的底端，就不会出现如此丰富多样的生命世界。继承是发展的一部分，只有不断地继承旧的、成熟的东西，才能发展出更新的、更先进的东西，否则，我们就会在原地踏步，永远重复。继承也是存储的另一种形式，是“代码”的动态存储方式。我们的学习、工作都依赖于此，如果没有继承，则我们早晨起床后就会是一个“新”的我，跟昨天没有关系。我们的生长和记忆都是在继承昨天的基础上开始的。继承不是简单的复制，其基本内涵就有改变的含义，所以有继承才有进化，这也是“生命是程序”的另一个证据。

在面向对象程序设计中，我们将从已经存在的类产生新类的机制称为继承。原来存在的类叫父类(或叫基类)，新类叫子类(或叫派生类)，子类中会自动拥有父类中的设计代码，还可以改写原来的代码或添加新的代码。继承带来的好处有两个方面，一方面可减少程序设计的错误，另一方面做到了代码复用，可简化和加快程序设计，提高了工作效率。

继承不仅仅是简单的拥有父类的设计代码，继承机制本身就具有进化的能力，跟生物世界一样，子代总是比父代更能适应环境。我们通过对父类的设计做一些局部的修改，使得子类对象具有更好的适应能力和强大的生存能力。图 3-3 所示为一个继承示意图。

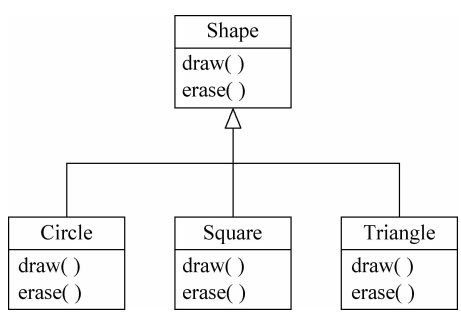

图 3-3　继承示意图

3.2.4　多态原则

多态原则是生物多样性在面向对象程序设计中的应用，正如第 1 章中说过的一样，面对同样的刺激、消息等，不同动物的反应是不一样的。在面向对象程序设计中，如果我们有许多不同的对象，每个对象都具有相应的行为模式(即执行代码)，对每个对象发送同样的消息，每个对象的执行代码是不一样的，这就是面向对象程序设计中的多态。

在具体实现上指程序中定义的引用变量所指向的具体对象和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定，即一个引用变量到底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在程序运行期间才能决定。因为在程序运行时才确定具体的对象，这样不用修改源程序代码就可以让引用变量绑定到各种不同的类实现上，从而导致该引用调用的具体方法随之改变，即不修改程序代码就可以改变程序运行时所绑定的具体代码，让程序可以选择多个运行状态，这就是多态性，多态性增强了软件的灵活性和扩展性。

多态性(polymorphism)是面向对象编程的基础属性,它允许多个方法使用同一个接口,从而导致在不同的上下文中对象的执行代码可以不一样。Java 从多个方面支持多态性,其中两个方面最为突出,一是每个方法都可以被子类重写;二是设立 interface 关键字。另外,Java 还支持通过方法重载实现的静态多态方式,即通过在编译时根据方法的参数不同选择编译不同的实现代码来实现多态,在运行时不再根据上下文改变。

由于超类(父类)中的方法可以在派生类(子类)中重写,因此,创建类的层次结构非常简单。在类的层次结构中,每个子类都是它的父类的特化(specialization)。从类属关系上来讲,属于底层类的对象肯定属于高层类。例如,小学生类是学生类的子类,学生类是人类的子类等,如果张三是一个小学生,则张三一定是一个学生,并且张三一定是一个人。在 Java 中,父类的引用可以指向子孙类对象,从而可以通过父类引用来调用子类对象的方法。在 Java 中,多态是通过动态绑定实现的,在通过父类的引用调用某子类对象的一个方法时会自动执行由该子类重写后的版本。因此,可以用父类来定义对象的形式并提供对象的默认实现,而子类根据这种默认实现进行修改,以更好地适应具体情况的要求。总之,在父类中定义的一个接口可以作为多个不同实现的基础。多态性原理如图 3-4 所示。

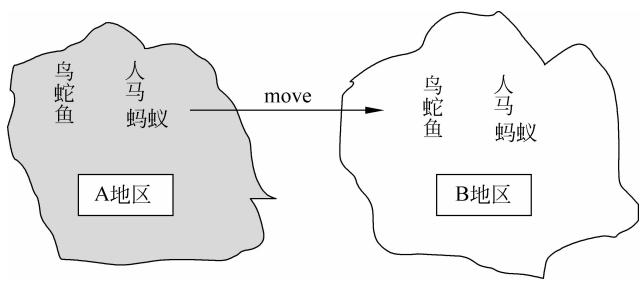

图 3-4 多态示意图

3.3 面向对象程序设计思想综述

面向对象最初是一种编程方式,现在成为了一种思想,一种在具体编程、软件设计、企业架构设计等领域广为应用的核心思想。在维基百科中给出了面向对象程序设计的基本概念,即使用“对象”这一概念将数据和操作数据的方法统一起来封装成一个整体,每个对象都有表示自己内部结构的数据,我们称之为属性,同时还有可以操作这些数据以改变对象的内部结构的函数[①](在 Java 中称为方法),以便对外界表现出某种行为。对象之间通过“消息”来交互,而消息本质上就是一个对象或系统对另一个对象发起的一次函数调用。一般来讲,一个完整的应用程序的功能由多个对象的协同工作来完成。在 Sun 公司的网站中较为通俗地说明了“对象”就是将相关的状态和行为捆绑在一起的基本软件单位,常用来模拟我们客观世界中和日常生活中的实体或头脑中的概念。

在面向对象程序设计中,程序设计的目标逐渐发生了改变,设计的重点不再是流程设计,而是抽象出一套代码模板(类似于细胞中的染色体上的基因码段),这套代码模板称为

① 计算机程序中的函数指的就是经过抽象得到的固定、有序的指令集,可以完成一个独立的计算功能。

类。在程序运行时，用这些类创建出需要的对象，可以是一个也可以是多个，每一个对象都有自己的状态（数据）和自己的行为（操作）。这些对象互相联系、协同工作，完成整个程序的功能。正如在生命体中，决定每一种细胞的基因是确定的，由这部分基因表达后形成特定种类的细胞，在这里细胞可以看成是对象，基因可以认为是类（代码模板），而染色体就是类代码的串行存储结构，类似于前后相互连接的火车车厢，一种基因组的抽象如图 3-5 所示。

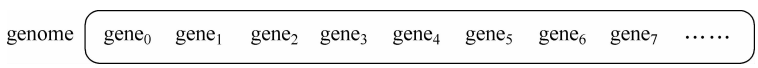

图 3-5　染色体 DNA 分子链中的基因组织图

3.3.1　类设计的一般规则

类是创建对象的代码模板，一般由两个部分组成，即描述对象状态和结构的成员变量和描述对象行为的成员方法。类是面向对象程序设计的基本要素，通过设计类来实现前面我们讲过的抽象、封装、继承以及多态，对象是由类创建的，当使用一个类创建一个对象时，我们也常说这个类实现了一个实例。

在语法上，类由两个部分组成，即类声明和类体。其基本格式如下：

```
class 类名[extends 父类名] [implements 接口名] {类体的内容 }
```

其中，class 是定义类的关键字，extends 是继承关键字，implements 是实现接口的关键字，类体部分代表此类的主体部分，又包括两个部分。

1. 成员变量（用来描述对象的属性）

```
[修饰符] 类型 变量名[ = 初值] [,变量名[ = 初值] … ];
```

说明：

（1）类型：可以是 Java 的基本类型，例如 int、float 等，也可以是复杂类型，例如我们自己定义的类，或者数组、接口等。

（2）变量名：必须是合法的 Java 标识符。

（3）修饰符：说明变量的访问权限和某些使用规则，可以是 public、private、protected、static、final 等，在后面会一一讲到。

（4）当成员变量含有自己的初始化表达式时，可以对变量初始化，即赋初值。

2. 成员方法（用来描述对象的行为）

方法是对实体行为的描述，对象通过执行它的方法对传来的消息做出响应。方法的定义只能在类中进行，它是完成某种功能的程序块，一个类或对象可以有多个方法。

方法的定义指描述方法的处理过程及其所需的参数，并用一个方法名来标识这个处理过程。方法定义中的形式参数并没有实际值，仅仅是为了描述处理过程而引入的占位符。

方法的使用就是通过向实例对象发送消息执行方法所定义的处理功能。在使用方法时给出参数的实际值，这些实际值称为实际参数（简称实参）。

```
[修饰符] 返回类型 方法名([形式参数列表])[throws 异常列表]
{  方法体  }
```

说明：

(1) 返回类型：说明此方法执行完后会返回一个值，这里指的是返回值的数据类型，可以是基本类型，也可以是复杂类型。如果返回类型为 void，表示返回值为 null，即不返回任何值。

(2) 方法名：方法的名称，必须是合法的 Java 标识符。

(3) 形式参数列表：说明使用此方法所需要的参数列表，可以有 0 个或多个，多个参数之间用逗号“,”隔开。在方法执行时，调用者会将调用时的实际参数值复制(传递)一份到形参变量中，传递过程是按照顺序依次对应传递。

(4) 修饰符：说明此方法的访问权限和某些使用规则，可以是 public、private、protected、static、abstract 和 final 等。

(5) 方法体：用一对花括号“{}”括起来，包含局部变量定义和相应的执行语句。

(6) 异常列表：说明本方法有可能产生的异常，需要调用者处理，在后面会详细讲解。

举例说明，我们需要抽象复数类，大家在数学中应该学过，一个复数由两个部分组成，即一个实部、一个虚部，组成形如 $a+bi$ 的形式，复数的各种运算大家也应该清楚，则一种可能的抽象如下例所示。

【例 3-1】 复数类抽象。

```
import java.util.Scanner;
public class fushu {
      private double realpart;
      private double imaginarypart;
      fushu(){realpart = 0;imaginarypart = 0;}                    //默认构造方法
      fushu(double s,double x){realpart = s;imaginarypart = x;}   //构造方法
      public void input() {
        Scanner keyin = new Scanner(System.in);
        System.out.print("real:");
        realpart = keyin.nextDouble();
        System.out.print("imaginary:");
        imaginarypart = keyin.nextDouble();
      }
      public void display() {
            String str = "" + realpart;
            if(imaginarypart < 0.0) str = str + imaginarypart + "i";
            else str = str + " + " + imaginarypart + "i";
            System.out.println(str);
      }
      public static void main(String[] args) {
         fushu m1 = new fushu(3.4,8.0);
         m1.input();
         m1.display();
      }
}
```

3.3.2　引用和引用变量

在第 2 章中已经介绍过引用了，此处进一步阐述。引用其实类似于 C 语言中的指针概念，但在 C 语言中指针是一个内存地址，用一个大于 0 的正整数来表示，可以进行加减运算。Java 中的引用不能进行加减运算，含有指向一个对象的意思，理论上它也代表一个对象的内存开始地址，其中，null(即 0)代表空引用，即不指向任何对象。

如果用一个类定义一个变量或数组变量，或后面讲到的接口变量，则该变量就是一个引用类型变量，可以存储引用(即一个对象的地址信息)了。基本类型变量和引用类型变量的存储结构示意图参考第 2 章的图 2-6 和图 2-7。

在 Java 中通常通过 new 创建一个对象和引用进行关联，例如：

```
fushu m1 = new fushu(3.4,8.0);
```

这样不仅创建了一个对象和引用 m1 进行关联，同时也进行了初始化。如果我们定义了一个引用，但没有指向任何对象，例如用上例中的复数类定义一个变量，即“fushu myvar;”，此时 myvar 的值为 null，即没有指向任何实际对象。如果调用它的成员方法或访问成员变量就会导致发生异常，因为对象还不存在，所以对象的属性和方法都不存在。

引用可以作为方法的参数，通过引用来传递对象，从而可以改变对象的内部状态，类似于你给派出所自己的身份证号码，派出所可以通过身份证号码定位你的信息，并可以修改你的信息，例如将未婚修改为已婚等，但引用本身并不发生变化。

注意：如果用类来定义数组，则该数组变量就是一个引用变量，因为 Java 语言中的数组为对象，同时每一个数组元素也是引用变量，用来指向该类的一个实例对象，这类似于 C 语言中的指针数组概念。

3.3.3　this 关键字

Java 用 this 引用指向对象自己，也就是说，当一个对象创建好后，Java 虚拟机就会给它分配一个引用自身的指针 this。在使用对象的成员变量和成员方法时，如果没有指定相应的对象约束，则默认使用的就是 this 引用。

在程序中，我们一般在以下情况使用 this 关键字：

- 在类的构造方法中，通过 this 语句调用这个类的另一个构造方法。
- 在一个非静态成员方法中，局部变量或形参变量与非静态成员变量同名，成员变量被屏蔽，要使用 this. varname 这种形式来指代成员变量。
- 在一个方法调用中，可以使用 this 将当前实例的引用作为参数进行传递。

3.3.4　匿名对象

所谓匿名对象，就是创建的对象没有特定的引用指向它，如下例所示：

```
public class StudentTest {
   public static void main(String[ ] args) {
     new Student("张三",'M',23). introduceMe();                    //匿名对象
```

```
    }
}
```

匿名对象在使用完后即变成垃圾对象,等待垃圾回收器回收。

3.3.5 方法重载

在同一个类中有多个同名的方法,但方法的参数列表不同,执行代码也不同,我们称之为方法重载。Java 中的方法重载也是实现多态性的方法之一,但方法重载是静态绑定的,即在编译时已确定好要执行的方法代码。方法重载主要通过实参列表和形参列表的配对来确定要使用哪个方法,例如下例。

【例 3-2】 方法重载演示。

```
class Calculation {
  public void add( int a, int b) {
    int c = a + b;
    System.out.println("两个整数相加得 " + c);
  }
  public void add( float a, float b) {
    float c = a + b;
    System.out.println("两个浮点数相加得" + c);
  }
  public void add( String a, String b) {
    String c = a + b;
    System.out.println("两个字符串相加得 " + c);
  }
  public void add(fushu a,fushu b) {
       fushu f = new fushu(a.shibu + b.shibu,a.xubu + b.xubu);
       System.out.println("两个复数相加得 " + f);
  }
}
class CalculationDemo {
  public static void main(String args[]) {
    Calculation c = new Calculation();
    c.add(10,20);
    c.add(40.0F, 35.65F);
      c.add("早上", "好");
      fushu f1 = new fushu(3.4,2.8);
      fushu f2 = new fushu(1.6, -7.8);
      f1.display();
      f2.display();;
      c.add(f1,f2);
  }
}
```

3.3.6 构造方法设计和对象的创建

前面已经讲过,对象是类实例化后的产物,所谓实例化就是按照类的设计创造对象,就是给此对象分配内存并初始化,即要进行一系列的构造工作,使其变成一个合适的、可用的

对象，这就是构造方法所完成的工作。构造方法是类中的一个特殊方法，特殊之处在于此方法要与类名同名，并且不能有返回类型。

注意：构造方法不能有返回类型并不代表它不能有返回值，实际上它要返回对象在内存中的开始地址。

构造方法可以重载，并且我们把没有任何参数的构造方法称为默认构造方法。如例 3-1 中的 fushu()是默认构造方法，而 fushu(double s,double x)则是带有两个形参的构造方法。在 Java 中，如果程序员没有提供构造方法，则 Java 编译器会自动提供一个默认的构造方法，如果程序员提供了构造方法，则 Java 编译器不再提供任何构造方法。

除了特殊的设计模式以外，建议大家最好提供一个默认的构造方法，例如我们抽象一个学生类。

【例 3-3】 学生类 Student。

```
import java.util.Scanner;
public class Student {
    private String name;
    private char sex;
    private int  age;
    private String[] coursenames;
    private double[] coursescores;
    public Student(){                                          //默认构造方法
        name = "unknown name!";
        sex = 'M';
        age = 0;
        coursenames = new String[3];
        coursescores = new double[3];
        coursenames[0] = new String("语文");
        coursenames[1] = new String("数学");
        coursenames[2] = new String("英语");
        coursescores[0] = coursescores[1] = coursescores[2] = 0.0;
    }
    public Student(String n,char s,int a){                     //带参数的构造方法
        name = n;
        sex = (s == 'F')?s:'M';                                //过滤数据
        if(a >= 0&&a <= 40) age = a;                           //过滤数据
        else age = 18;
        coursenames = new String[3];
        coursescores = new double[3];
        coursenames[0] = new String("语文");
        coursenames[1] = new String("数学");
        coursenames[2] = new String("英语");
        coursescores[0] = coursescores[1] = coursescores[2] = 0.0;
    }
    public void introduceMe() {
        System.out.println("我的名字是:" + name);
        System.out.println("我的性别和年龄分别是:" + sex + " 和 " + age);
        System.out.println("我的成绩还没有输入!");
    }
}
```

在定义完类后,就可以用类来创建对象并使用对象了。关键字 new 通常称为创建运算符,用于分配对象内存,并将该内存初始化为默认值,然后调用构造方法来执行对象具体初始化。例如下例:

```
public class StudentTest {
   public static void main(String[ ] args) {
     Student stu1 = new Student();                          //调用默认构造方法
     Student stu2 = new Student("张三",'M',23);             //调用带参数的构造方法
     stu1.introduceMe();
     stu2.introduceMe();
   }
}
```

在 new 分配内存后,各种类型变量的默认初始值如表 3-1 所示,也就是调用构造方法之前的值。

表 3-1 成员变量的默认值

类 型	默认值	类 型	默认值
byte	(byte)0	char	'\u0000 '
short	(short)0	float	0.0F
int	0	double	0.0D
long	0L	对象引用	null
boolean	false		

注意:*如果一个构造方法通过 this 调用另一个构造方法,则该调用语句必须出现在第一句。*

3.3.7 get 方法和 set 方法设计

前一节已初步设计好 Student 类,我们对它内部的数据做了封装,使得类外不能直接访问,例如在 StudentTest 类的 main 方法中,我们想直接给 stu1 对象的年龄 age 赋值,即"stu1.age=200;",这是错误的,这种破坏封装的语句在 Java 编译时不允许通过。封装给我们带来了两个好处,一是被封装的数据对外是不可见的,二是我们通过提供一系列的 get 方法和 set 方法去读写这些数据。在这些方法中我们可以过滤传进来的数据,就像人的消化系统一样,所有的食物经过消化系统后变成了对人有用的营养,而非法数据则被过滤,这就是对象对外提供的交换接口。

get 方法和 set 方法的编写也很简单,一般以 get 和 set 开头,后面单词的第一个字母一般大写。例如给 Student 类加上合适的 set 和 get 方法:

```
public String getName(){return name;}
public void setName(String n){name = n;}
public char getSex(){return sex;}
public void setSex(char s){sex = (s == 'F')?s:'M';}
public int getAge(){return age;}
public void setAge(int a){age = (a >= 0&&a <= 40)?a:18;}
public String[ ] getCoursenames(){return coursenames;}
```

```
public String getCoursename(int i){return coursenames[i];}
public void setCoursenames(String[] cn){coursenames = cn;}
public void setCoursename(String cn,int i) {
    coursenames[i] = cn; }
public double[] getCoursescores(){return coursescores;}
public double getCoursescore(int i){return coursescores[i];}
public void setCoursescores(double[] cs){coursescores = cs;}
public void setCoursescore(double cs,int i){coursescores[i] = cs;}
```

注意：针对具有多值的成员变量，一般是数组或集合，我们应该至少提供两套 get()方法和 set()方法，如上例所示。

3.3.8　toString 方法和 equals 方法设计

在上一章介绍了 Object 类，说 Object 类是 Java 中所有其他类的根，也可以说，它是我们所有类的一个框架设计，在此类中有两个重要的方法。

- toString()方法：用来将一个对象转换成字符串描述形式。
- equals()方法：用来比较两个对象的内容是否一样。

它们的原始实现非常简单，我们需要在类中给出更有意义的、更具体的执行代码，实际上这叫方法重写，在下一章中会讲到。

```
public boolean equals(Student anotherstu) {
  boolean flag = true;
  if(name.equals(anotherstu.getName()) flag = false;
  else if(age == anotherstu.getAge()) flag = false;
  else if(sex == anotherstu.getSex()) flag = false;
  return flag;
}
public String toString() {
  String myinfo = "   name:" + name + "\tsex:" + sex + "\tage:" + age;
  myinfo = myinfo + "\n=========================================\n";
  for(int i = 0;i < coursenames.length;i++) {
    myinfo = myinfo + "    " + coursenames[i] + "   \t";
  }
  myinfo = myinfo + "\n";
  for(int i = 0;i < coursescores.length;i++)  {
    myinfo = myinfo + "    " + coursescores[i] + "   \t";
  }
  myinfo = myinfo + "\n=========================================";
  return myinfo;
}
```

【例 3-4】 复数类新版，添加了 equals()和 toString()方法。

```
import java.util.Scanner;
public class fushu {
     private double realpart;
     private double imaginarypart;
     fushu(){realpart = 0;imaginarypart = 0;}
```

```
    fushu(double s,double x){realpart = s;imaginarypart = x;}
    public boolean equals(fushu another) {
       return (this.realpart == another.realpart
          && this.imaginarypart == another.imaginarypart);
    }
    public String toString() {
          String str = "" + realpart;
          if(imaginarypart < 0.0) str = str + imaginarypart + "i";
          else str = str + " + " + imaginarypart + "i";
          return str;
    }
    public void input() {
      Scanner keyin = new Scanner(System.in);
      System.out.print("real:");
      realpart = keyin.nextDouble();
      System.out.print("imaginary:");
      imaginarypart = keyin.nextDouble();
    }
    public void display() {
       System.out.println(toString());
    }
    public static void main(String[] args) {
       fushu m1 = new fushu(3.4,8.0);
       fushu m2 = new fushu(3.4,8.0);
       System.out.println("m1 == m2 = " + (m1 == m2));
       System.out.println("m1.equals(m2) = " + m1.equals(m2));
       fushu m3 = new fushu(4.4, - 8.9);
       System.out.println("m1 = " + m1);
       System.out.println("m3 = " + m3);
       m2.input();
       m2.display();
    }
}
```

3.3.9 其他功能方法设计

在类的设计中,除了针对封装属性提供的接口和重写从父类中继承的方法以外,每一个类都应该有自己独特的功能方法,例如针对学生类求总分以及输入数据等。

```
public double total() {
    double sum = 0.0;
    for(int i = 0;i < coursescores.length;i++)  {
       sum += coursescores[i];
    }
    return sum;
  }
  public void inputData() {
     Scanner in = new Scanner(System.in);
     System.out.println("请输入" + name + "的成绩: ");
     for(int i = 0;i < coursescores.length;i++) {
```

```
            System.out.print(coursenames[i] + ":");
            coursescores[i] = in.nextDouble();
        }
    }
```

到此为止，我们的学生类才初具模型，在后面的章节中会有更完整的代码。

3.4 方法递归

所谓递归(recursion)，在数学上就是利用自身结构来描述自己。例如：

- 自然数：1是一个自然数，一个自然数的后继者是一个自然数。
- 阶乘运算：0!＝1，如果 $n>0$，那么 $n!=n*(n-1)!$。
- 斐波纳契数列：f(0)＝1，f(1)＝1，f(n)＝f(n－1)＋f(n－2)。

在Java语言程序设计中，在一个方法内部直接或间接地调用自身，我们称之为递归方法。编写递归方法必须满足以下3个条件：

(1) 知道递归公式的描述。

(2) 每次递归调用必须使得其进程逐步接近递归的终止条件。

(3) 必须要有递归终止条件。

【例 3-5】 递归演示。

```
public class RecursionDemo {
    public long fac(int n) {
        if(n == 0)                                          //终止条件
          return 1;
        else
          return (n * fac(n - 1));                          //递归公式
    }
    public long fbnc(int n) {
        if(n == 0||n == 1)                                  //终止条件
          return 1;
        else
          return(fbnc(n - 1) + fbnc(n - 2));                //递归公式
    }
    public static void main(String[ ] args) {
        RecursionDemo rcs = new RecursionDemo();            //调用默认构造方法创建对象
        System.out.println("5!= " + rcs.fac(5));            //调用对象的 fac()方法
        System.out.println("f(20) = " + rcs.fbnc(20));      //调用对象的 fbnc()方法
    }
}
```

3.5 Java语言中的访问权限

Java语言采用访问控制修饰符来控制类及成员方法和成员变量的访问权限，Java中的访问控制分为下面4个级别。

(1) 公开级别：用 public 修饰，对外完全公开。

(2) 受保护级别：用 protected 修饰，对子类和同一个包中的类公开。

(3) 默认级别：没有访问控制修饰符，对同一个包中的类公开。

(4) 私有级别：用 private 修饰，只有类本身可以访问，不对外公开。

注意：访问级别仅仅适用于类及类的成员，而不适用于局部变量。局部变量只能在方法内部被访问，不能用 public、protected 或 private 修饰。

成员变量、成员方法和构造方法可以处于 4 个级别中的一个，而类又分为顶层类和内部类，顶层类只可以处于公开或默认级别，因此不能用 private 和 protected 类修饰，内部类可以有各种访问权限。表 3-2 列出了 Java 语言中的访问权限。

表 3-2　4 种访问级别的访问范围

访问控制	private 成员	默认的成员	protected 成员	public 成员
同一类中的成员	√	√	√	√
同一包中的其他类	×	√	√	√
不同包中的子类	×	×	√	√
不同包中的非子类	×	×	×	√

3.6 内部类和匿名类

3.6.1 内部类

内部类是指在一个外部类的内部再定义一个类，内部类作为外部类的一个成员，是依附于外部类存在的。内部类可以是静态的，可以用 protected 和 private 修饰(外部类只能使用 public 和默认的包访问权限)。内部类主要有成员内部类、局部内部类、静态内部类、匿名内部类。内部类允许把一些逻辑相关的类组织在一起，并且能控制内部类代码的可视性，将包含此内部类的类称为外部类或顶层类。

【例 3-6】 内部类演示 1。

```
class A {
    int a;
    public A(){a = 29;}
    public void print() {
       System.out.println("a = " + a);
       B myb = new B();                          //使用内部类创建对象
       myb.display();
    }
    class B {                                    //内部类,默认访问级别
       int b;
       B(){b = 78;}
       public void display() {
         System.out.println("a = " + a);         //可以直接访问外部类的成员变量
         System.out.println("b = " + b);
       }
```

```
    }
}
class innertest {
    public static void main(String[] args) {
        A mya = new A();
        mya.print();
        A.B innerobj = new A().new B();                    //在其他类中创建内部类对象
        innerobj.display();
    }
```

【例 3-7】 内部类演示 2。

```
class innerouter {
  private class inner {                                    //内部类,私有访问级别
    private String name;
    private int age;
    public int step;
    inner(String s,int a) {
      name = s;
      age = a;
      step = 0;
    }
    public void run() {
        step++;
    }
  }
  public static void main (String args[]) {
    innerouter a = new innerouter();                       //创建外部类对象
    innerouter.inner d = a.new inner("Tom",3);             //创建内部类对象
    d.step = 25;                                           //访问内部类的属性
    d.run();                                               //调用内部类的方法
    System.out.println(d.step);
    }
}
```

注意：如果一个内部类由 static 修饰，则此类会自动变成和顶层类同级，即不能再直接访问外部类非静态成员变量。

3.6.2　匿名类

匿名内部类(简称匿名类)就是没有名字的内部类。匿名类是一种特殊的内部类，这种类没有名字，通过 new 关键字直接创建某一个类的匿名子类的对象来使用。那么，在什么情况下需要使用匿名内部类呢？如果满足下面的一些条件，使用匿名内部类是比较合适的：

(1) 只用到类的一个实例。

(2) 类在定义后马上用到。

(3) 类非常小(Sun 公司推荐是在 4 行代码以下)。

(4) 给类命名并不会导致代码更容易被理解。

在使用匿名内部类时，用户要记住下面几个原则：

(1) 匿名内部类不能有构造方法。

(2) 匿名内部类不能定义任何静态成员、方法和类。

(3) 匿名内部类不能是 public、protected、private、static。

(4) 只能创建匿名内部类的一个实例。

(5) 一个匿名内部类一定是在 new 的后面,用其隐含实现一个接口或实现一个类。

(6) 因匿名内部类为局部内部类,所以局部内部类的所有限制都对其生效。

下例用于演示匿名类的使用技巧。

【例 3-8】 匿名类演示。

```
class Anonymousedemo {
    Anonymousedemo(){
       System.out.println("默认构造方法!");
    }
    Anonymousedemo(int x){
       System.out.println("带一个参数的构造方法!");
    }
    void method(){
      System.out.println("一成员方法");
    }
    public static void main(String[] args)
    {
      new Anonymousedemo().method();                    //创建匿名对象并调用成员方法
      Anonymousedemo a = new Anonymousedemo(){          //匿名类,实际上是子类
         void method(){
           System.out.println("匿名类中的成员方法");
         }
      };
      a.method();
  }
}
```

3.7 Java 的垃圾回收机制

垃圾回收作为一种内存管理技术已经存在了很长时间,但是 Java 使它焕发出崭新的活力。在 C++等语言中,内存必须人工管理,程序员必须显式地释放不再使用的对象。这是问题产生的根源,因为忘记释放不再使用的资源,或者释放了正在使用的资源都是很常见的事情。Java 代替程序员完成了这些工作,从而防止了此类问题的发生。在 Java 中,所有的对象都是通过引用访问的,这样,当垃圾回收器发现一个没有引用的对象时,就知道此对象已经不被使用,并且可以回收了。如果 Java 允许对象的直接访问(与简单数据类型的访问方式类似),那么这种有效的垃圾回收方法将无法实现。

Java 的垃圾回收策略在普遍意义上反映了 Java 的理念,那就是简化 Java 程序员的工作复杂度,提高程序员的编程效率。Java 设计人员花费大量的精力来防止其他编程语言经常出现的典型问题,例如程序员经常忘记释放资源,或者错误地释放正在使用的资源。因此,使用垃圾回收策略有效地避免了此类问题的发生。

Java 的垃圾回收具有以下特点：

(1) 只有当对象不再被程序中的任何引用变量引用时，它的内存才可能被回收。

(2) 程序无法迫使垃圾回收器立即执行垃圾回收操作。

(3) 当垃圾回收器将要回收无用对象的内存时，先调用该对象的 finalize()方法，该方法释放对象所占的相关资源，但也有可能是对象复活，不再回收该对象的内存。

3.8　程序建模示例

【程序建模示例 1】 Hanoi 塔问题的递归解法。

在印度有这么一个古老的传说：在世界中心贝拿勒斯(在印度北部)的圣庙里，一块黄铜板上插着 3 根宝石针。印度教的主神梵天在创造世界的时候，在其中一根针上从下到上穿好了由大到小的 64 片金片，这就是所谓的汉诺塔。不论白天黑夜，总有一个僧侣在按照下面的法则移动这些金片：一次只移动一片，不管在哪根针上，小片必须在大片的上面，如图 3-6 所示。僧侣们预言，当所有的金片都从梵天穿好的那根针上移到另外一根针上时，世界将在一声霹雳中消灭，而梵塔、庙宇和众生也将同归于尽。

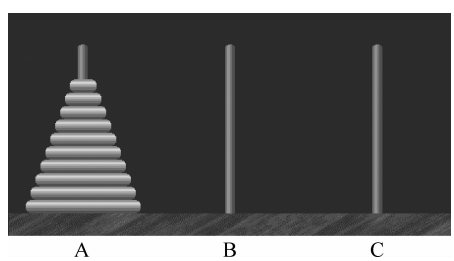

图 3-6　汉诺塔演示

分析：不管这个传说的可信度有多大，如果考虑把 64 片金片由一根针移到另一根针上，并且始终保持上小下大的顺序，需要移动多少次呢？这里需要用到递归方法，假设有 n 片，移动次数是 $f(n)$，则移动次数如表 3-3 所示。

表 3-3　所需移动次数

金片数	所需移动次数 $f(n)$	金片数	所需移动次数 $f(n)$
1	1	…	…
2	3	n	2^n-1
3	7		

如表 3-3 所示，当 $n=64$ 时，$f(64)=2^{64}-1=18\ 446\ 744\ 073\ 709\ 551\ 615$。假如每秒钟一次，共需多长时间呢？一个平年 365 天，大约有 31 536 000s，闰年 366 天，有 31 622 400s，平均每年 31 556 952s，计算一下，18 446 744 073 709 551 615/31 556 952=584 554 049 253.855 年。

这表明移完这些金片需要 5845 亿年以上，而地球存在至今不过 45 亿年，太阳系的预期寿命据说也就数百亿年。真的过了 5845 亿年，不说太阳系和银河系，至少地球上的一切生命，连同梵塔、庙宇等，都早已灰飞烟灭。

下例使用递归方法简单地演示了 Hanoi 塔移法：

```
//Hanoi.java
import java.util.Scanner;
public class Hanoi{
  void moves(char a,char c){
     System.out.println("From " + a + " to " + c);
  }
  void hanoi(int n,char a,char b,char c){
     if(n == 1) moves(a,c);
     else {
        hanoi(n - 1,a,c,b);
        moves(a,c);
        hanoi(n - 1,b,a,c);
     }
  }
  public static void main(String[ ] args) {
     int n;
     Scanner keyin = new Scanner(System.in);
     Hanoi test = new Hanoi();
     System.out.print("Enter a number:");
     n = keyin.nextInt();
     test.hanoi(n,'A','B','C');
  }
}
```

大家还可以上网看到很多基于图形界面的 Hanoi 塔演示，我们的源代码包中包含了 Bob Kirkland 的基于图形界面和多线程的一个小应用程序的演示。

【程序建模示例 2】 一个房间内铺有 m 行 n 列瓷砖，如图 3-7 所示。一个跳蚤随机从一个瓷砖开始，每次随机选择一个方向，前进一个瓷砖，当碰到墙时，代表此方向不能前进，试编程模拟此过程，当跳蚤遍历所有瓷砖时，输出每块瓷砖被经历的次数和跳蚤跳跃的总次数。

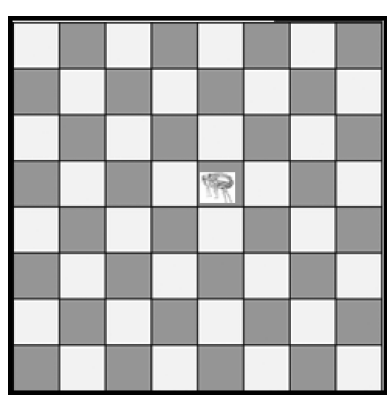

图 3-7 跳蚤实验

分析：我们抽象一个房子类，保存有 m 和 n 的值，以及一个模拟地板瓷砖的二维数组，用二维数组中每一个元素的值记录跳蚤经过此瓷砖的次数。再抽象一个跳蚤类，保存跳蚤

所在的瓷砖位置，然后随机产生一个方向进行跳跃。一种可能的程序模拟如下：

```
//Tiaozao.java
class House{
  private int m;
  private int n;
  private int[][] a;
  public House(){
    m = 10;n = 10;
    a = new int[m][n];
    for(int i = 0;i < m;i++)
      for(int j = 0;j < n;j++) a[i][j] = 0;
  }
  public House(int m,int n){
    this.m = m;this.n = n;
    a = new int[m][n];
    for(int i = 0;i < m;i++)
      for(int j = 0;j < n;j++) a[i][j] = 0;
  }
  public int getM(){return m;}
  public int getN(){return n;}
  public int[][] getA(){return a;}
  public int getElement(int i,int j){return a[i][j];}
  public void setElement(int i,int j,int v){ a[i][j] = v; }
  public boolean checkZero(){
    for(int i = 0;i < m;i++)
      for(int j = 0;j < n;j++) {
        if(a[i][j] == 0) return true;
      }
    return false;
  }
  public void display() {
    for(int i = 0;i < m;i++){
       for(int j = 0;j < n;j++) {
          System.out.print("" + a[i][j] + " ");
      }
      System.out.println();
    }
  }
}
public class Tiaozao{
   private static final int UP = 0;
   private static final int DOWN = 1;
   private static final int RIGHT = 2;
   private static final int LEFT = 3;
   private int x,y;
   private int totals;
   private House ahouse;
   public Tiaozao(House h){
      ahouse = h;
      totals = 0;
```

```
        x = (int)(Math.random() * ahouse.getM());
        y = (int)(Math.random() * ahouse.getN());
    }
    public int getTotals(){return totals;}
    public boolean walk(int direction) {
        System.out.println("x = " + x + ",y = " + y + ",direction = " + direction);
        switch(direction)     {
          case UP: if(y == 0) return false;
                   else {
                     ahouse.setElement(x,y,ahouse.getElement(x,y) + 1);
                     y = y - 1;
                   }
                   return true;
          case DOWN: if(y == ahouse.getN() - 1) return false;
                   else {
                     ahouse.setElement(x,y,ahouse.getElement(x,y) + 1);
                     y = y + 1;
                   }
                   return true;
          case LEFT: if(x == 0) return false;
                   else {
                     ahouse.setElement(x,y,ahouse.getElement(x,y) + 1);
                     x = x - 1;
                   }
                   return true;
          case RIGHT: if(x == ahouse.getM() - 1) return false;
                   else {
                     ahouse.setElement(x,y,ahouse.getElement(x,y) + 1);
                     x = x + 1;
                   }
                   return true;
          default: System.out.println("非法移动!");return false;
        }
    }
    public void move() {
         int nextdirection;
         boolean success;
         do{
           nextdirection = (int)(Math.random() * 4);
           success = walk(nextdirection);
           if(success) totals++;
         }while(ahouse.checkZero());
    }
    public static void main(String[] args) {
        House ahouse = new House(4,4);
        Tiaozao atiaozao = new Tiaozao(ahouse);
        atiaozao.move();
        ahouse.display();
        System.out.println("Totals = " + atiaozao.getTotals());
    }
}
```

程序的执行结果如图 3-8 所示。

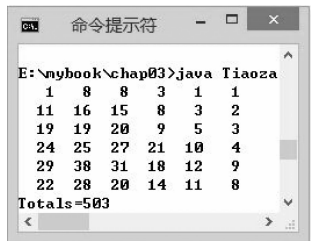

图 3-8　跳蚤程序的执行结果

3.9　本章小结

本章详细介绍了面向对象程序设计的基本概念和基本的设计原则，包括抽象原则、封装原则、继承原则、多态原则，还介绍了对象和类的基本概念，类和对象的关系以及类设计的一般规则。类是 Java 语言程序设计的基本元素，它定义了一个对象的结构和功能，类中主要包含属性和方法。本章通过程序实例演示了数据的封装技巧、方法重载、get 方法、set 方法以及功能方法的设计技术。在本章中还介绍了构造方法、方法递归等基本概念，辨析了引用和引用变量，简述了内部类和匿名类的概念，最后通过程序建模演示了面向对象程序设计中的抽象、封装等设计技巧。

习题

一、选择题

1. 下列不属于面向对象编程的 3 个特征的是(　　)。

A. 封装　　B. 指针操作　　C. 多态性　　D. 继承

2. (　　)是一组有相同属性、共同行为和共同关系的对象的抽象。

A. 类　　B. 方法　　C. 属性　　D. 以上都不对

3. (　　)是指在调用一个方法时，每个实际参数“值”的副本都将被传递给此方法形参。

A. 按引用传递　　B. 按值传递　　C. 按对象传递　　D. 按形参传递

4. java.lang 包的 Object 的(　　)方法将比较两个对象是否相等，如果相等则返回 true。

A. toString()　　B. compare()　　C. equals()　　D. none of above

5. (　　)是指子类中的一个方法与父类中的方法有相同的方法名，并具有相同参数和类型的参数列表。

A. 重载方法　　B. 覆盖方法　　C. 强制类型转换　　D. none of above

6. 当编译并运行下列程序段时，将会(　　)。

```
class VarField {
  int i = 99;
  void amethod(){
    int i;
    System.out.println(i);
  }
}
public class VarInit{
  public static void main(String args[])
  {
      VarField m = new VarField();
      m.amethod();
  }
}
```

A. 输出 99　　B. 输出 0　　C. 编译时出错　　D. 执行时出错

7. 对于下列定义的类,通过(　　)可以使它既符合类的封装性,又能操作类中的属性。

```
class Staff{
    int salary;
}
```

A. 将属性 salary 定义为 private

B. 将属性 salary 定义为 public

C. 将属性 salary 定义为 private,并且定义 public 的 get 和 set 方法访问属性 salary

D. 将属性 salary 定义为 public,并且定义 public 的 get 和 set 方法访问属性 salary

8. 关于对象的删除,下列说法正确的是(　　)。

A. 必须由程序员完成对象的清除

B. Java 把没有引用的对象作为垃圾收集起来并释放

C. 只有当程序中调用 System.gc()方法时才能进行垃圾收集

D. Java 中的对象都很小,一般不进行删除

9. 关于构造方法,下列说法错误的是(　　)。

A. 构造方法不可以进行方法重写

B. 构造方法用来初始化该类的一个新对象

C. 构造方法具有和类名相同的名称

D. 构造方法不返回任何数据类型

10. 在 Java 中,为了使一个名为 Example 的类成功地编译和运行,必须满足(　　)。

A. Example 类必须定义在 Example.java 文件中

B. Example 类必须声明为 public 类

C. Example 类必须定义一个正确的 main()方法

D. Example 类必须导入 java.lang 包

11. 给出以下代码,该程序的输出结果是(　　)。

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

```
    Float f1 = new Float("10.4F");
    Float f2 = new Float("10.4f");
    System.out.print(f1 =  = f2);
    System.out.print("\t" + f1.equals(f2));
  }
}
```

A. true false　　B. true true　　C. false true　　D. false false

12. 编译并执行下列程序段，将会输出(　　)。

```
class Test1{
    private int i = 100;
    public Test1(){}
    public void putI(int n){i = n;}
    public int getI(){return i;}
}
class Test2{
    public void method1(){
        int i = 200;
        Test1 obj1 = new Test1();
        obj1.putI(20);
        method2(obj1,i);
        System.out.print(obj1.getI());
    }
    public void method2(Test1 v,int i){
        i = 0;
        v.putI(30);
        Test1 obj2 = new Test1();
        v = obj2;
        System.out.print(v.getI() + "," + i + ",");
    }
}
public class Main{
    public static void main(String[] args){
        Test2 obj = new Test2();
        obj.method1();
    }
}
```

A. 20,200,0　　B. 100,30,0　　C. 100,0,30

D. 200,30,0　　E. 200,20,30

13. 下列(　　)选项能较好地体现面向对象的封装性。

A. 类中的方法全部是私有的，以避免意外地修改成员变量的值

B. 类中的属性都是公有的，以便其他对象方便地访问

C. 类中的所有属性都是私有的，以防止意外地被修改

D. 一般情况下，类的属性是私有的，方法是公有的，通过公有方法来访问或修改私有属性

二、简答题

1. 什么是对象？什么是类？类和对象有什么关系？

2. 如何定义一个类？类中包含哪几个部分？

3. 如何创建对象？如何对对象进行初始化？

4. 请说明比较两个对象时，使用“==”比较两个引用变量和使用 equals()方法比较两个引用变量的区别。

5. 类的实例变量在什么时候会被分配内存空间？

6. 什么是匿名对象？什么是内部类？

三、编程题

1. 设计一个 Timer 类，属性包括小时、分、秒，然后编写含有 main 方法的类创建一个 Timer 对象进行测试。

2. 对学生成绩管理系统进行完善，补充修改、删除等功能。

3. 编写一个类，用该类创建的对象可以计算等差数列的和。

4. 试对平面直角坐标上的点、直线、三角形等简单图形编写一个程序，并在程序中求出对应图形的面积和周长。

5. 修改课本中的复数类，使其能进行复数的四则混合运算。

6. 编程打印出杨辉三角形(要求打印出 10 行，如下图所示)。

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5   10  10  5   1
    ……
```