第3章 面向对象程序设计

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，并将操作指令和被操作的数据隔离开，分为代码段和数据段。随着程序复杂度越来越高，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地强化，将待操作的数据和相应的操作指令组织在一起，形成了著名的面向对象程序设计方法。在面向对象设计中，人们可以借鉴人类研究和认识世界的诸多理论和方法，将世界中的具体对象经过分类和抽象后变成一个个符号化的模板类，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。而在程序运行过程中，又将符号化模板类转换为的具体的时空化的实例对象，用一个设计好的类可以创建许多对象。用计算机术语讲，类就是对象的代码抽象，是创建对象的代码模板，对象是进程中的类的具体实例。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个可观测、具体的对象开始的，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独特的运动规律和内部结构状态的对象所组成的，对象和系统环境既保有联系又保持独立，即对象必须依赖系统才能存在，但同时它又是一个独立的、可观察的个体。正是不同对象之间的相互作用和通信构成了一个完整的系统。因此，程序员应当按照现实世界的本来面貌来理解程序系统，直接通过对象及其相互关系来反映进程系统和客观世界，这样建立起来的系统才能接近和满足客观世界的真实需求。

从方法论的角度来看，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


视频讲解

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以下两步。

(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,通过对象的使用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,而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,对象一般指的是一个独立的客观实体,它一般都有一定的内部结构,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,至少占有一定空间和时间。

在面向对象程序设计中,对象就是数据结构加上指令代码,或称数据与代码的组合,它是理解面向对象程序设计技术的关键。为了理解这一点,先来研究现实世界中的对象。我们周围的汽车、电视机、狗、猫、书桌、自行车等都是现实世界中的物理实体,也就是通常所说的客观对象。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、两个车轮和齿轮的数目等)和行为(刹车、加速等)。对事件的响应能力是对象通过行为或功能方法实现的,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。程序对象是现实世界的对象在计算机内部的模拟化产物,它们也有状态和行为,程序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(指令集合)来实现。

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,就是对象对内部数据和结构的一种隐藏和隔离。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,将数据和操作细节(方法)隐藏起来,只暴露必要的交互接口,这和客观世界中对象是类似的,例如小狗、小猫等。和对象的交互必须符合接口标准选择和限制,观察小狗、小猫和小鸟等动物对象就可以理解,这使得与对象的交互可按照统一的方式进行,这样就能比较容易地产生更为统一和健壮的系统程序。

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,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,但将其意识行为去掉,这样的人也只是“植物人”,不是一个真正的人。所以,物质结构决定了行为,行为又改变着物质结构,它们是不可分的一体两面。

在计算机程序设计中,只有设计出精巧的数据结构再配以合适的方法代码(即抽象和封装),加上继承和多态,才是真正的面向对象程序设计。

3.1.2 类

在物理或生物世界中,类代表一种抽象概念,例如猫作为一类动物的统称,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。而对象实际是程序运行时通过类创建的一个个实例，生成对象的过程叫作“实例化”。一个实例化的对象实际上是由若干个成员变量和成员方法组成的封装体。当创建一个对象时，系统将为该实例中的成员变量分配内存，然后利用成员方法去和系统或其他对象交互。

为了更好地理解类和对象的概念，这里用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，是一种代码模版，对应的就是此处讲的类代码模板。那么经过了胚胎发育，最后成长为一个生命体。例如，人、鸡、猫、狗、马、蛇等，就是 DNA 代码模板创建的一个个实例对象。生命对象具有生命周期，计算机程序中的程序对象也有构造、初始化、功能交互、死亡等周期。

3.1.3 类设计的 Java 语法

类是创建对象的代码模板，一般由两部分组成，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。在 Java 语言的语法中，类由两部分组成：类声明和类体。其基本格式如下：

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

其中，[修饰符]可以用 public 代表公开，private 代表私有，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，又包括以下两部分。

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，如自己定义的类，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，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)

方法是对对象功能行为的描述，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定义，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(简称为实参)。

```
[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(也称按值传递)，传递过程是按照顺序依次对应传递的。

(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，我们需要抽象一个复数类，读者在数学中应该学过，一个复数由两部分组成，即一个实部一个虚部，组成形如 $a+b\mathrm{i}$ 的形式，复数的各种运算读者也应该清楚，则一种可能的抽象如下例所示，图 3-1 对该类的结构进行了说明。

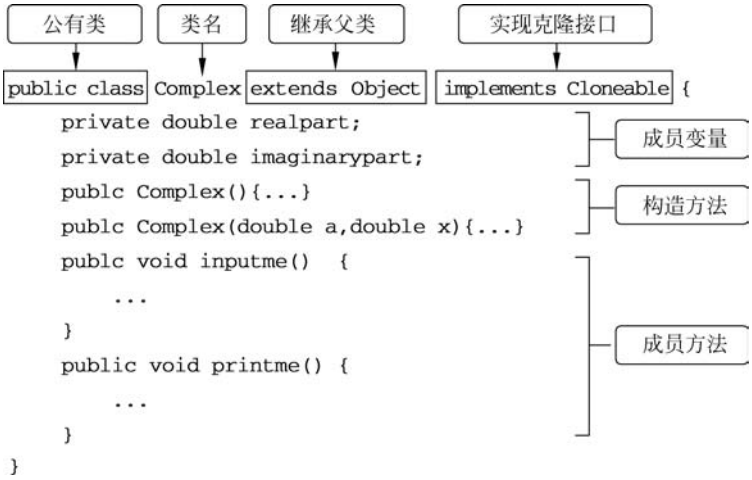

图 3-1 类设计示意图

【例 3-1】 复数类抽象。

```
import java.util.Scanner;
public class Complex extends Object implements Cloneable{
      private double realpart;
      private double imaginarypart;
      public Complex(){realpart = 0;imaginarypart = 0;}                    //默认构造方法
      public Complex(double s,double x){realpart = s;imaginarypart = x;} //构造方法
      public void inputme() {
        Scanner keyin = new Scanner(System.in);
        System.out.print("real:");
        realpart = keyin.nextDouble();
        System.out.print("imaginary:");
        imaginarypart = keyin.nextDouble();
      }
```

```
    public void printme() {
        String str = "" + realpart;
        if(imaginarypart < 0.0) str = str + imaginarypart + "i";
        else str = str + " + " + imaginarypart + "i";
        System.out.println(str);
    }
}
```

注意：和现实世界中对象一样，进程中的每个对象也需要一个构造过程（对应于现实世界中对象的生产过程），构造方法用来完成这一过程，对象一经创建，就可以和其他对象或系统进行交互，交互一般通过方法调用进行。在类中直接定义的变量并且没有 static 修饰符，被称为对象的成员变量，在创建对象后，每个对象都会有一份。

3.1.4 消息

由于不存在孤立系统，在进程中，一个孤独的对象是没有用的。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。在进程中，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，接收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。一个消息通常由以下 3 个部分组成。

（1）接收消息对象的名称。

（2）要执行方法的名称。

（3）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 点起床。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“起床”，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，可以理解为消息是方法调用的专署名词，适用于面向对象领域。

例如我们要测试前面定义的复数类 Complex，可用以下测试类来进行测试。

【例 3-2】 测试复数类 TestComplex。

```
public class TesstComplex {
public static void main(String[] args) {
      Complex m1 = new Complex(3.4,8.0);
      m1.inputme();            //调用 inputme()方法，即给 m1 对象发输入请求的消息
      m1.printme();            //调用 printme()方法，即给 m1 对象发打印输出的消息
    }
}
```

视频讲解

3.1.5 引用和引用变量

在第 2 章中已经介绍过引用，此处进一步阐述。引用就像现实世界中的空间地址，可以通过地址来找一个具体对象。Java 中的引用类似于 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 中的引用本质上也是内存地址,但是不能进行加减运算,用来说明此地址处有一个对象,理论上它也代表一个对象在内存中开始地址,其中 null(即 0)代表空引用,即此引用目前不指向任何有效对象。

如果用一个类定义一个变量,或通过数组形式定义的变量,或后面讲到的通过接口定义的变量,则该变量就是一个引用类型变量,可以存储一个内存地址了。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 2-9 和图 2-10。

在 Java 中通常会通过 new 来创建一个对象和引用进行关联,如例 3-2 中的:

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

这样不仅创建了一个对象和引用 m1 进行关联,同时也进行初始化。如果定义了一个引用,但没有指向任何对象,如用例 3-2 中的复数类定义一个变量 Complex 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异常发生,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,类似于户口本中的地址信息,派出所可以通过地址信息定位、找人,并可以修改户口或身份信息,例如从未婚修改为已婚等,但引用本身并不发生变化。

注意:如果用类来定义数组,则该数组变量也是一个引用变量,因为 Java 语言中数组为对象,同时每一个数组元素又都是存储对象的引用,用来指向该类的一个实例对象,所以 Java 中用类或接口定义的数组,类似于 C 语言中的指针数组概念。

3.1.6 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符号 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。

程序中一般在以下情况使用 this 关键字。

(1) 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。

(2) 在一个非静态成员方法中,如果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。

(3) 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。

【例 3-3】 测试 this 关键字。

```
public class TestThis {
  int x;                                    //对象成员变量
  TestThis(int x) {                         //形参变量,局部变量
    this.x = x;
  }
  public void passingValue(){
    System.out.println("x 等于 " + x);     //成员变量,即 this.x
  }
  public static void main(String args[]) {
    TestThis test = new TestThis(10);
```

```
        test.passingValue();
    }
}
```

对例 3-1 中的复数类进行改进，增加了加减乘除方法，在下例中综合演示了 this、引用变量、方法调用（即消息）等方面。

【例 3-4】 复数类的进化版及其测试。

```
//Complex1.java
import java.util.Scanner;
public class Complex1 extends Object implements Cloneable{
    private double realpart;
    private double imaginarypart;
    public Complex1() {
        realpart = 0;
        imaginarypart = 0;
    }                                    // 默认构造方法
    public Complex1(double s, double x) {
        realpart = s;
        imaginarypart = x;
    }                                    // 构造方法
    public void inputme() {
        Scanner keyin = new Scanner(System.in);
        System.out.print("real:");
        realpart = keyin.nextDouble();
        System.out.print("imaginary:");
        imaginarypart = keyin.nextDouble();
    }
    public String toString() {
        String str = "" + realpart;
        if (imaginarypart < 0.0)
            str = str + imaginarypart + "i";
        else
            str = str + "+" + imaginarypart + "i";
        return str;
    }
    public double getRealpart() {
        return realpart;
    }
    public void setRealpart(double realpart) {
        this.realpart = realpart;
    }
    public double getImaginarypart() {
        return imaginarypart;
    }
    public void setImaginarypart(double imaginarypart) {
        this.imaginarypart = imaginarypart;
    }

    public Complex1 add(Complex1 other) {        // 复数加法
        return new Complex1(this.realpart + other.realpart, this.imaginarypart + other.
```

```
imaginarypart);
    }
    public Complex1 sub(Complex1 other) {        // 复数减法
        return new Complex1(this.realpart - other.realpart, this.imaginarypart - other.
imaginarypart);
    }

    public Complex1 mut(Complex1 other) {        // 复数乘法
         double r = this.realpart * other.realpart - this.imaginarypart * other.
imaginarypart;
         double i = this.imaginarypart * other.realpart + this.realpart * other.
imaginarypart;
        return new Complex1(r, i);
    }

    public Complex1 div(Complex1 other) {        // 复数除法
        double denominator = other.realpart * other.realpart + other.imaginarypart *
other.imaginarypart;
         double r = (this.realpart * other.realpart + this.imaginarypart * other.
imaginarypart) / denominator;
         double i = (this.imaginarypart * other.realpart - this.realpart * other.
imaginarypart) / denominator;
        return new Complex1(r, i);
    }
}
```

3.1.7 匿名对象

所谓匿名对象,就是创建的对象没有特定引用指向它。如下例所示:

```
//TestAnonymous.java
  public class TestAnonymous {
    public static void main(String[] args) {
        new Complex1(3.0,5.0).printme();        //创建匿名对象并调用 printme 方法
    }
}
```

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。

视频讲解

3.1.8 方法重载

在同一个类中有多个同名的方法,但方法参数列表不同,执行代码也不同,称为方法重载。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 3-5】 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(Complex1 a, Complex1 b) {
        Complex1 f1 = new Complex1(a.getRealpart() + b.getRealpart(), a.getImaginarypart() +
b.getImaginarypart());
        System.out.println("两个复数相加得 " + f1);
    }
       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");
      Complex1 f1 = new Complex1(3.4,2.8);
      Complex1 f2 = new Complex1(1.6, - 7.8);
      f1.display();
      f2.display();;
      c.add(f1,f2);
   }
}
```

3.1.9 构造方法设计和对象的创建

前面已讲过，对象是类实例化后的产物，所谓实例化是按照类的设计创造对象的过程，就是给此对象分配内存空间并初始化，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，类似于现实世界中动物对象的分娩或孵化过程。

注意：① 构造方法是类中一个特殊的方法，特殊之处在于此方法要与类名同名，并且不能有返回类型。

② 构造方法不能有返回类型并不能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。

③ 构造方法可以重载，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 Complex()是默认构造方法，而 Complex(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，建议读者最好提供一个默认的构造方法。例如抽象一个学

生类和班级类:

【例 3-6】 学生类 Student。

```
//Student.java
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。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:

```
//TestStudent.java
public class TestStudent {
    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关键字 this 调用另一个构造方法，则该调用语句必须出现在第一句。

3.1.10　getter 方法和 setter 方法设计

前一节已初步设计好 Student 类，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 类中的 main 方法中，如果想直接给 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 编译时就不允许通过。封装带来两个好处，一是被封装的数据对外是不可见的，二是通过提供一系列的 getter 方法和 setter 方法去读写这些数据，通过这些方法中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部分转化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。

getter 方法和 setter 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 setter 和 getter 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，应该至少提供两套 getter 方法和 setter 方法，如上例所示。

3.1.11 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,也可以说它是所有类的一个框架设计,在此类中有两个重要的方法。

(1) toString()方法:用来将一个对象转换成字符串描述形式。

(2) 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,没有实际用处,所以在类中,要给出更有意义的、更具体的实现代码,在面向对象程序设计理论中这叫作方法重写,在继承中还会讲到。

```
public boolean equals(Object obj) {
    student anotherstu = (student)obj;
    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 3-7】 复数类完善版,添加了 getter 方法、setter 方法、equals()和 toString()方法。

```
//Complex2.java
import java.util.Scanner;
public class Complex2 {
     private double realpart;
     private double imaginarypart;
     Complex2(){realpart = 0;imaginarypart = 0;}
     Complex2(double s,double x){realpart = s;imaginarypart = x;}
     public double getRealpart(){ return realpart;}
     public double getImaginarypart(){ return imaginarypart;}
     public void setRealpart(double r){ realpart = r;}
     public void setImaginarypart(double i){ imaginarypart = i; }
     public boolean equals(Complex2 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 inputme() {
        Scanner keyin = new Scanner(System.in);
        System.out.print("real:");
        realpart = keyin.nextDouble();
        System.out.print("imaginary:");
        imaginarypart = keyin.nextDouble();
      }
      public void printme() {
         System.out.println(toString());
      }
     public Complex2 add(Complex2 other){
             return new Complex2(this.realpart + other.realpart, this.imaginarypart + other.
imaginarypart);
      }
      public Complex2 sub(Complex2 other){
             return new Complex2(this.realpart - other.realpart, this.imaginarypart - other.
imaginarypart);
      }
      public Complex2 mut(Complex2 other){
             double r = this.realpart * other.realpart - this.imaginarypart * other.
imaginarypart;
             double i = this.imaginarypart * other.realpart + this.realpart * other.
imaginarypart;
             return new Complex2(r,i);
      }
      public Complex2 div(Complex2 other){
             double denominator = other.realpart * other.realpart + other.imaginarypart *
other.imaginarypart;
             double r = (this.realpart * other.realpart + this.imaginarypart * other.
imaginarypart)/denominator;
             double i = (this.imaginarypart * other.realpart - this.realpart * other.
imaginarypart)/denominator;
             return new Complex2(r,i);
      }
      public static void main(String[] args) {
         Complex2 m1 = new Complex2(3.4,8.0);
         Complex2 m2 = new Complex2(3.4,8.0);
         System.out.println("m1 == m2 = " + (m1 == m2));
         System.out.println("m1.equals(m2) = " + m1.equals(m2));
         Complex2 m3 = new Complex2(4.4, - 8.9);
         System.out.println("m1 = " + m1);
         System.out.println("m3 = " + m3);
         Complex2 m4 = m2.add(m3);
         m4.printme();
      }
}
```

3.1.12 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外,每一个类都应该有自己独特的功能方法,如前面的复数类 Complex1 的加减乘除等方法,又比如针对学生类 Student 可以添加求总分以及输入数据等方法,代码如下:

```
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();
       }
   }
```

至此,读者应该学到要设计一个类,需要抽象数据成员,需要构造方法、getter 方法、setter 方法、toString 方法、equal 方法以及一些其他和业务相关的功能方法设计,这样的类才算完善,才能用来构造真正的业务程序,参考源代码 Student 类、Myclass 类、TestStudent 类、TestMyclass 类、MyclassTest 类的设计。有了这些基础知识,下面再来细化理解面向对象程序设计的几个基本原理。

视频讲解

3.2 面向对象程序设计的基本原理

从前面的叙述来看,面向对象程序设计其实就是在程序设计的发展历史中逐渐形成的一套设计理论,并且还在继续完善中。目前来说,整个面向对象程序设计理论主要建立在以下几条原理之上。

(1) 抽象原理。

(2) 封装原理。

(3) 继承原理。

(4) 多态原理。

(5) 组合原理。

3.2.1 抽象原理

抽象就是从大量的、普遍的、具体的个体中抽象出共有的属性和行为,从而形成一般化概念或符号化的过程。例如,植物、动物、质量、能量等。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

作具有特定行为的对象。人们可以忽略发动机、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，从整体的视角来描述和表述汽车对象。而在面向对象程序设计中，抽象的作用是把事物、系统或概念用数据和指令的方式进行编码，以便在计算机的进程中能够模拟系统或解决问题。

如何进行抽象？每个人都有不同的理解，并且这需要大量的实践锻炼才行。抽象能力似乎是人类具备的一种特殊能力，人类正是通过抽象和分类才发展出当前的文明水平。总而言之，在一个系统设计中，人们需要从特定视角和层次对系统中涉及的对象和概念进行分析和归类，然后抽取需要的数据描述和功能描述，再用程序编码的方式进行表述，这就是抽象原理在面向对象程序设计中的应用。

下面通过一个示例来展示抽象原理，假设在屏幕上用“*”打印矩形，可以把此矩形看成一个对象，用面向对象的思维来进行分析和抽象，所有的矩形都有宽(w)和高(h)，并且在屏幕上有一个位置，而位置是由形如(x,y)的坐标标识出来的，所以最简单的抽象就是通过(w、h、x、y)来定义一个矩形类(Rectangle)，然后提供一个 printme() 方法在屏幕上打印出这个矩形。如图 3-2 所示，参考 Java 代码示例如下：

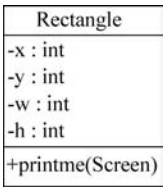

图 3-2　Rectangle 类

【例 3-8】 矩形类 Rectangle 抽象。

```
public class Rectangle {
    int x,y,w,h;
    Rectangle() {
       this(0,0,1,1);
    }
    public Rectangle(int x,int y,int w,int h) {
      this.x = x;
      this.y = y;
      this.w = w;
      this.h = h;
    }

    public void printme(Screen myscreen) {
        myscreen.setY(y);
        for(int i = 1;i <= h;i++)        {
          myscreen.setX(x);
          myscreen.repeat('*',w);
            myscreen.println();
        }
    }
}
```

从上面的分析中可以看到一个频繁出现的名词“屏幕”，并且在打印方法中要使用“屏幕”的接口方法来完成打印操作，所以“屏幕”应该也是一个对象，那么屏幕对象如何抽象呢？和矩形对象一样，经过分析会发现每一个屏幕需要一个宽度和高度，在屏幕上应该有一个表示当前输入/输出的位置信息，还要包含存储屏幕内容的存储空间(此处假设屏幕只处理文本字符)，所以也可以简单的抽象出 Screen 类，通过(w,h,x,y,data)来定义，并提供相应的

一些功能方法如定位、打印、显示等,如图 3-3 所示,参考 Java 代码如下:

【例 3-9】 屏幕类 Screen 抽象。

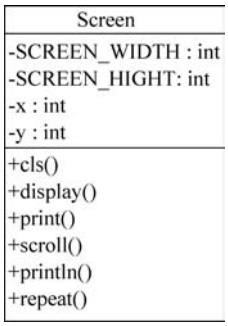

图 3-3 Screen 类

```
public class Screen {
    int SCREEN_WIDTH;
    int SCREEN_HEIGHT;
     int x,y;
     char[][] data;
    int getX() {
        return x;
    }
    public void setX(int x) {
         this.x = x;
    }
    public int getY() {
        return y;
    }
    public void setY(int y) {
        this.y = y;
    }
    public Screen() {
        SCREEN_HEIGHT = 50;
        SCREEN_WIDTH = 80;
             data = new char[SCREEN_HEIGHT][SCREEN_WIDTH];
    }
    public Screen(int r,int c) {
        SCREEN_HEIGHT = r;
        SCREEN_WIDTH = c;
        data = new char[SCREEN_HEIGHT][SCREEN_WIDTH];
    }
    public void cls() {
        for(int i = 0;i < SCREEN_HEIGHT;i++) {
            for(int j = 0;j < SCREEN_WIDTH;j++) {
                data[i][j] = ' ';
            }
        }
    }
    public void display() {
        for(int i = 0;i < SCREEN_HEIGHT;i++) {
            for(int j = 0;j < SCREEN_WIDTH;j++) {
                System.out.print(data[i][j]);
            }
            System.out.println();
        }
    }
    public void repeat(char ch,int m) {
        for(int i = 1;i <= m;i++) print(ch);
    }
    public void print(char ch) {
        if (y < SCREEN_HEIGHT && x < SCREEN_WIDTH) {
```

```
            data[y][x] = ch;
            x++;
            if (x == SCREEN_WIDTH) {
                y++;
                if (y == SCREEN_HEIGHT) {
                    scroll();                    //屏幕上滚一行
                       y = SCREEN_HEIGHT - 1;
                }
                x = 0;
            }
        }else {
            System.out.println("错误:超出屏幕了!");
        }
    }
    public void println() {
        if(++y == SCREEN_HEIGHT) {
        scroll();
        y = SCREEN_HEIGHT - 1;
     }
        x = 0;
    }
    public void scroll() {
        for(int i = 0;i < data.length - 1;i++) {
            data[i] = data[i + 1];
        }
        data[data.length - 1] = new char[SCREEN_WIDTH];
  }
}
```

示例测试,有了创建对象的矩形类和屏幕类代码模板就可以进行测试了,另外设计一个测试类,提供主方法和测试代码,运行效果如图 3-4 所示。

图 3-4　屏幕上打印矩形对象

【例 3-10】 测试类。

```
public class TestRectangle {
    public static void main(String[] args) {
        Screen myscreen = new Screen();
        Rectangle rc1 = new Rectangle(0,0,6,5);       //第 0 行 0 列的 5 行 6 列的矩形
        rc1.printme(myscreen);
        Rectangle rc2 = new Rectangle(32,4,5,7);      //第 4 行 32 列的 7 行 5 列矩形
        rc2.printme(myscreen);
        myscreen.display();
    }
}
```

视频讲解

3.2.2　封装原理

一般情况下,人们只能看到物质的“外壳”,而看不到其内部结构,对象的内部数据和结构对外是不可见的。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,也就是说人类都是内聚性很强的对象个体,但又留有眼、耳、鼻、口等接口,通过这些接口在这个世间生存和忙碌。仔细观察动物世界中的各种动物、人造物品(如手机、打印机、汽车)等都具有很好的封装性。

在程序设计的发展历史中,人们发现模块的内聚性越强,耦合性越弱,对程序的设计和协同开发越好,可以大大提高程序软件的可维护性和扩展性,所以在面向对象程序设计中,将对象的内部数据和结构对外做信息隐藏,让外部不可访问,提供一系列的公有接口用来进行信息和能量交换如图 3-5 所示,这就是封装原理在面向对象程序设计中应用。

对于提供了私有属性的对象,如何访问和修改其值呢? 一般情况下应该根据需要提供相应的 getter 方法(读取值)和 setter 方法(修改值),读值方法和赋值方法的基本理念就是对数据进行必要的隔离和过滤,使外界无法直接访问和修改这些数据。因为其他对象不应该直接操作另一个对象中数据,而该对象的读值方法和赋值方法可提供对内部数据的访问,访问时进行必要过滤和限制,甚至复杂的解码过程,类似于动物的消化系统接口。读值方法和赋值方法有时又被称为访问方法(即 getter 方法)和设值方法(即 setter 方法)。

在上一节中抽象了屏幕类(Screen)、矩形类(Rectangle),并且用一个测试程序完成了测试。但有个问题,如果在测试程序中,直接修改矩形对象的内部数据,就会造成数据混乱,这些矩形对象已经不是原来的矩形对象了。

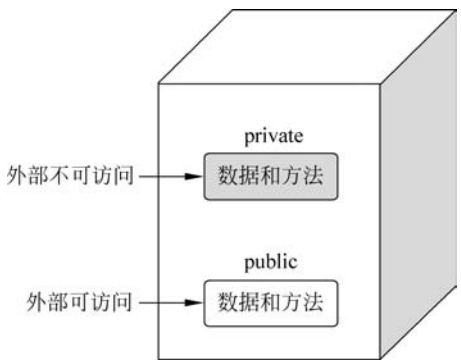

图 3-5　封装示意图

图 3-6　对象内容被破坏

【例 3-11】 没有封装的对象程序演示。

```
public class TestNoEnCapsulation {
    public static void main(String[ ] args) {
        Screen myscreen = new Screen();
        myscreen.init();
        Rectangle rc1 = new Rectangle(0,0,6,5);
        rc1.h = 3;                                    //数据被任意修改,对象被破坏
        rc1.x = 10;
        rc1.printme(myscreen);
        Rectangle rc2 = new Rectangle(32,4,5,7);
        rc2.w = 10;
        rc2.printme(myscreen);
```

```
        myscreen.data[5][33] = '中';                    //数据被非法修改,对象内容被破坏
        myscreen.display();
    }
}
```

如果修改屏幕对象数据是非法或出错的(如将前例中屏幕对象的宽带修改为－3,则程序就会出错),那么矩形对象就无法显示了,如图 3-7 所示。

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 80
        at myjava2019.chap3.test0.Screen.init(Screen.java:39)
        at myjava2019.chap3.test0.Test.main(Test.java:7)
```

图 3-7　没有封装数据被其他类修改后出错

【例 3-12】 无封装对象数据修改后出错。

```
public class TestNoEnCapsulation1 {
    public static void main(String[] args) {
        Screen myscreen = new Screen();
        myscreen.init();
        myscreen.SCREEN_WIDTH = 100;                    //相当于创建好的屏幕对象拉宽
        Rectangle rc1 = new Rectangle(0,0,6,5);
        rc1.printme(myscreen);
        Rectangle rc2 = new Rectangle(32,4,5,7);
        rc2.printme(myscreen);
        myscreen.display();
    }
}
```

想要防止数据被非法修改,就需要使用封装技术。在 Java 语言中,实现封装的关键字是 private,提供公有接口的关键字是 public。实现封装需要两步:第一步,将对象内部的属性数据用 private 修饰,这样其他对象就无法直接访问和修改了,并且有些属性在对象创建后再不允许修改,则此类属性应该定义为常量;第二步,对于需要访问的属性提供读值方法 getter,并需要特定代码对数据进行处理,根据安全需要可隐藏某些数据,对于需要修改的属性提供写值方法 setter,并且在方法中提供约束和过滤代码,保证合法数据进入,阻挡非法数据进入。实现封装后屏幕类的代码如下。

【例 3-13】 屏幕类的封装版。

```
public class Screen {
    private final int SCREEN_WIDTH;
    private final int SCREEN_HEIGHT;
    private int x;
    private int y;
    private char[][] data;
    public int getX() {
        return x;
    }
    public void setX(int x) {
        if (x < SCREEN_WIDTH)     this.x = x;
```

```
    }
    public int getY() {
        return y;
    }
    public void setY(int y) {
        if (y < SCREEN_HEIGHT)     this.y = y;
    }
    public Screen() {
        SCREEN_HEIGHT = 50;
        SCREEN_WIDTH = 80;
        data = new char[SCREEN_HEIGHT][SCREEN_WIDTH];
    }
    public Screen(int r, int c) {                    //通过判断对输入的数据进行过滤
        if (r >= 1 && r <= 1000)
            SCREEN_HEIGHT = r;
        else
            SCREEN_HEIGHT = 50;
        if (c >= 1 && c <= 1000)
            SCREEN_WIDTH = c;
        else
            SCREEN_WIDTH = 80;
        data = new char[SCREEN_HEIGHT][SCREEN_WIDTH];
    }

    public void init() {
        for (int i = 0; i < SCREEN_HEIGHT; i++) {
            for (int j = 0; j < SCREEN_WIDTH; j++) {
                data[i][j] = ' ';
            }
        }
    }
    public void display() {
        for (int i = 0; i < SCREEN_HEIGHT; i++) {
            for (int j = 0; j < SCREEN_WIDTH; j++) {
                System.out.print(data[i][j]);
            }
            System.out.println();
        }
    }

    public void repeat(char ch, int m) {
        for (int i = 1; i <= m; i++)
            print(ch);
    }
    public void print(char ch) {
        if (y < SCREEN_HEIGHT && x < SCREEN_WIDTH) {
            data[y][x] = ch;
            x++;
```

```
            if (x == SCREEN_WIDTH) {
                y++;
                if (y == SCREEN_HEIGHT) {
                    scroll();
                    y = SCREEN_HEIGHT - 1;
                }
                x = 0;
            }
        } else {
            System.out.println("错误:超出屏幕了!");
        }
    }
    public void println() {
        y++;
        if (y == SCREEN_HEIGHT) {
            scroll();
            y = SCREEN_HEIGHT - 1;
        }
        x = 0;
    }
    public void scroll() {
        for (int i = 0; i < data.length - 1; i++) {
            data[i] = data[i + 1];
        }
        data[data.length - 1] = new char[SCREEN_WIDTH];
    }
}
```

修改后,外部就无法修改屏幕(Screen)类对象的内部数据了,并且相应方法的代码也做了过滤处理,使非法数据无法进入。

3.2.3 继承原理


视频讲解

继承也是存储的另一种形式,是“数据和指令代码”的动态存储方式,人类的学习、工作都依赖于此。继承不是简单的复制,其基本内涵中有发展和改变的含义,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,从已存在的类产生新类的机制也被定义为继承。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的流程,提高工作效率。

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。通过对父类的设计作一些局部的修改,可以使得子类对象具有更好的适应能力和强大的生存能力。

如果从一个抽象模型中剔除足够多的细节,则它将变得更通用,能适应多种情况或场合,这样的抽象常常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。例如,前面的示例,如果想在屏幕中不但可以打印矩形,还可以打印菱形、直角

三角形、圆形等一系列形状,则应该分层抽象类如图 3-8 所示。

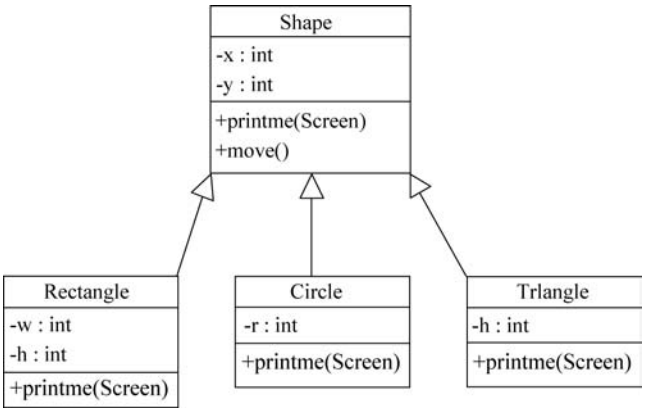

图 3-8 继承示意图

面向对象程序设计的最强大功能之一就是代码重用。面向过程的结构化设计提供的代码重用非常有限,基本上限定在编写一个功能模块,然后在进程中多次调用它,或者是对特定的编码复制粘贴再修改。但是在面向对象的设计中代码重用已经很完善了,通过定义类之间的关系,通过组织和识别不同类之间的共性,不仅可以实现代码重用,还可以指导人们对复杂问题分层抽象、分层处理,继承就是实现该功能的主要原理。

在面向对象程序设计中,如何实现继承,不同语言有不同的实现机制,在 Java 语言中,通过关键字 extends 来指明一个子类从一个父类扩展而来。举例说明,以图 3-8 来演示。

【例 3-14】 继承原理演示。

```
//Shape.java
public class Shape {
    protected int x;
    protected int y;
    public int getX() {
        return x;
    }
    public void setX(int x) {
        if(x>=0&&x<1000) this.x = x;
        else this.x=0;
    }
    public int getY() {
        return y;
    }
    public void setY(int y) {
        if(y>=0&&y<1000) this.y = y;
        else this.y=0;
    }
    public Shape() {}
    public Shape(int x,int y) {
        if(x>=0&&x<1000) this.x = x;
        else this.x=0;
        if(y>=0&&y<1000) this.y = y;
```

```
        else this.y = 0;
    }
     public void printme(Screen sc) {
          sc.setY(y);
        sc.setX(x);
     System.out.println();
    }
    public void move(int x,int y) {
        if(x >= 0&&x < 1000) this.x = x;
        else this.x = 0;
        if(y >= 0&&y < 1000) this.y = y;
        else this.y = 0;
    }
}
//Lingxing.java
public class Lingxing extends Shape {
    private int h;
    public Lingxing() {
            this(0,0,7);
        }
    public Lingxing(int x,int y,int h) {
            super(x,y);
            this.h = h;
        }
    public void printme(Screen myscreen) { // 覆盖父类中 printme()方法
        myscreen.setY(y);
        for (int i = 1; i <= (h + 1) / 2; i++) {
            myscreen.setX(x);
            myscreen.repeat(' ', h / 2 + 1 - i);
            myscreen.repeat('*', 2 * i - 1);
            myscreen.println();
        }
        for (int i = h / 2; i >= 1; i-- ) {
            myscreen.setX(x);
            myscreen.repeat(' ', h / 2 + 1 - i);
            myscreen.repeat('*', 2 * i - 1);
            myscreen.println();
        }
    }
}
//Circle.java
public class Circle extends Shape{
    private int r;
    public Circle(int x,int y,int r) {
        super(x,y);
        this.r = r;
    }
    public void printme(Screen sc) {                    //覆盖父类中的 printme()方法
        // x * x + y * y = r * r
        sc.setY(y);
        for(int y = 0;y <= 2 * r;y += 2) {
```

```
                int lx = (int)Math.round(r - Math.sqrt(2 * r * y - y * y));
                int len = 2 * (r - lx);
                sc.setX(this.x + lx);
                sc.print('*');
                for(int j = 0;j <= len;j++) {
                    sc.print('*');
                }
                sc.print('*');
                sc.println();
            }
        }
}
//Triangle.java
public class Triangle extends Shape{
    private int h;
    public Triangle() {
        this(0,0,7);
    }
    public Triangle(int x,int y,int h) {
        super(x,y);
        this.h = h;
    }
    public void printme(Screen myscreen) {                    //覆盖父类中 printme()方法
        myscreen.setY(y);
        for(int i = 1;i <= h;i++)
        {
            myscreen.setX(x + h - i);
            myscreen.repeat('*',2 * i - 1);
            myscreen.println();
        }
    }
}
//TestInherit.java 测试类
public class TestInherit {
    public static void main(String[] args) {
        Screen myscreen = new Screen(25,80);
        myscreen.cls();
        Lingxing mylx = new Lingxing(0,0,9);
        mylx.printme(myscreen);
        Lingxing mylx2 = new Lingxing(20,1,12);
        mylx2.printme(myscreen);
        Rectangle rc = new Rectangle(14,1,5,7);
        rc.printme(myscreen);
        Triangle tr = new Triangle(56,2,7);
        tr.printme(myscreen);
        Circle c = new Circle(34,0,10);
        c.printme(myscreen);
        myscreen.display();
```

```
    }
}
```

注意：Java 中实现继承的关键字是 extends。

测试类运行结果如图 3-9 所示。

图 3-9　测试类运行截图

从该示例可以看出，Shape 类可以看成各种图形的抽象父类，从而可以派生出各种具体的图形类，如 Rectangle、Triangle 等，子类自动拥有父类中的成员变量 x、y，同时继承了父类中的各种公有成员方法，各子类有根据自己形状的特点，给出了 printme()方法的覆盖实现，从而为实现多态打好了基础。

继承提供的是 is-a 关系，即父类相对于子类更为抽象，子类更为具体，子类对象同样隶属父类型。生物类是动物类的父类、动物类是人类的父类，张三是一个人类对象，同样的，张三也是一个动物类对象或者生物类对象。

3.2.4　多态原理

视频讲解

多态原理是生物多样性在面向对象程序设计中的应用，指的是在一个系统中同一消息可能会引发多种反应。例如，多个动物面对同样的刺激、消息等，不同的动物的反应是不一样的。在面向对象程序设计中，如果有许多不同的对象，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，但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原理，多态原理如图 3-10 所示，给不同的打印机发送相同的打印消息，不同的打印机有不同的打印实现方式。

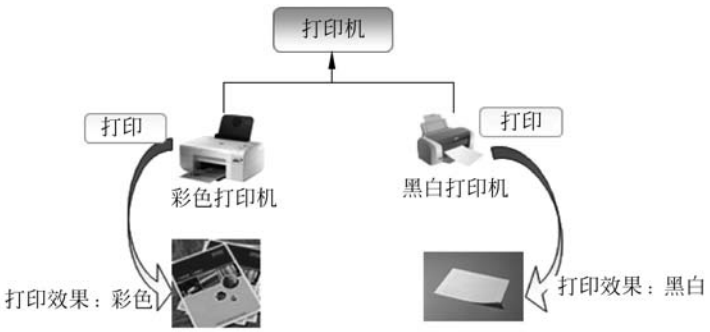

图 3-10　多态示意图

在具体实现上是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译时并不确定,只有在程序运行期间才能确定,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代码实现上,从而导致该引用调用的具体方法代码随之改变,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,让程序可以选择不同的代码来运行,这就是多态性实现技术,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重要。在类的层次结构中,每个子类都是它的父类的特殊化(specialization)或具体化。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,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。继续用前面的示例程序,重新写一个测试类,采用多态原理,测试类的代码如下:

【例 3-15】 多态原理演示。

```
//TestPolymorphism.java
public class TestPolymorphism {
    public static void main(String[] args) {
        Screen myscreen = new Screen(25,80);
        myscreen.cls();
        Shape shapes[] = new Shape[5];                  //通过父类定义了有 5 个引用变量的数组
        shapes[0] = new Lingxing(0,0,9);                //指向一个菱形对象
        shapes[1] = new Lingxing(20,1,12);
        shapes[2] = new Rectangle(14,1,5,7);            //指向一个矩形对象
        shapes[3] = new Triangle(56,2,7);               //指向一个三角形对象
        shapes[4] = new Circle(34,0,10);                //指向一个圆形对象
        for(int i = 0;i < shapes.length;i++) {
            shapes[i].printme(myscreen);                //方法调用相同,但因对象不同执行代码
                                                        //也不同,这就是多态原理
        }
        myscreen.display();
    }}
```

程序运行结果和图 3-9 一样。

3.2.5 组合原理


视频讲解

在现实世界中，一个复杂对象总是由许多子对象构造而成。例如，汽车对象包含了发动机、轮胎、方向盘等子对象，一个宠物狗对象也会包含心、肝、脾、肺等子对象。在面向对象程序设计中，常用组合来完成从简单对象到复杂对象的构造过程，一个复杂对象常常是由多个简单的成员对象组合而成的。相对于继承的 is-a 关系，组合是 has-a 关系，即整体和部分的关系。

使用组合的原因是通过组合可以降低构建系统的复杂性，这也是人们解决复杂问题的通用方式。研究表明在人的短期记忆中一次性最多只能记住 7 组数据，所以人们更喜欢使用抽象概念，每个抽象概念下又由诸多具体数据组合而成。例如，在日常生活中，人们不会说拥有了一个很大的物件，它包括一个方向盘、四个车轮、一个引擎等，人们会说有了一辆车。车就是一个更为抽象化的概念，这有助于交流和保持清醒的头脑。

在现实世界中，人们会为生产的产品组件制定标准，这样一件产品的某个组件出问题了就可以用另一个组件实现同样标准的替换。在面向对象程序设计中，可以使用组合来完成类似的功能，这意味着实现了组件的标准化和重用。例如，抽象一个汽车类，汽车对象中有一个方向盘，只要方向盘的接口都是一样的，那么就无须考虑为具体的汽车对象安装特定的方向盘，只要找一个一样接口的方向盘装上即可。

使用组合的另一个优势是可以分别构建系统和子系统，而且更重要的是这些系统可以被独立测试和维护。毫无疑问，大型软件系统是相当复杂的，为了构建高质量的软件，必须遵循一种规则来取得成功，这个规则就是尽可能地保持简单。和大型工程项目一样，为了能够简化管理和提高效率，人们常常使用分解和分层的技术将复杂的项目分解成一个个容易实现的小项目。类似的，为了让大型的软件系统能够正常工作并且易于维护，必须将其分割为更小且更容易管理的单位。1962 年发表的标题为“架构的复杂性”一文中，作者 Herbert simon 作为诺贝尔奖的获得者总结了以下对稳定的系统的思考。

(1) 稳定的复杂系统通常具有一定的层次结构，每个系统由更简单的子系统构建而成，这些子系统又由更简单的子系统构建而成。这种方式是软件开发过程中基本的解耦方式，所以读者要尽可能学习和熟悉它。在面向对象的设计中，组合适用于这条准则，即通过简单的对象来构筑复杂的对象。

(2) 稳定的复杂系统是可分解的。这意味着可以识别组成系统的各个部分，以及这些部分之间的交互关系。在稳定的系统中，组成部分之间的交互要少于组成部分内部的交互。例如立体音响系统由更简单的器件组成，即话筒、按钮和扩音器。这种方式比集成系统更稳定，因为集成系统不容易解耦。

(3) 稳定的复杂系统往往由不同类型的子系统以不同的方式组合而成。而这些子系统又由更小的部分组合而成。

(4) 可工作的复杂系统往往是从可工作的简单系统演化而来。人们往往不会从头建立新系统(即重新发明轮子)，而是基于已经经过验证的系统来构建新系统。

在 Java 语言和.NET 框架中，组合概念更加重要。因为对象可以被动态加载，所以解耦设计很重要。例如，如果发布了一个应用程序，后来由于修复缺陷或者维护的目的，需要重新创建其中一个类文件，那么只需用重新发布这个特定的类文件即可。如果所有的代码

都在单个文件中,则需要重新发布整个应用程序。

组合通常有两种方式,即联合和聚合。这些方式代表了对象之间不同的协作关系。任何组合类型都是 has-a 关系。然而,联合和聚合的微小区别在于部分如何构成整体。在聚合中,通常只看到整体,如手机或电视机,而在联合中,通常看到的是组成整体的部分,如计算机、打印机、鼠标、键盘构成的办公系统,音响、功放、麦克风、DVD 播放机、电视机等构成的家庭音响和影院系统等。

1. 聚合及其实现技术

组合最直观的方式就是聚合,聚合意味着一个复杂的对象由许多小对象构成。例如,智能手机,外表看就是一个整体对象,但实际上它是由许多小部件组合而成,再比如电视机是进行娱乐活动的统一平台,当看电视时,人们只看到了一个电视机,但电视机内部由很多子系统如音频子系统、显像子系统、信号接收子系统和解码子系统等组合而成。简单地讲,聚合往往是一个整体的封装对象,对象内部又可以分解为许多的标准小部件,每个小对象又都是具有特定功能和标准接口的封装体。下面用面向对象式的思维来设计一个简易的、具有简单计算功能的 Computer 类,该简易计算机拥有中央处理器 CPU、存储器 Memory、显示屏和一个小键盘,它们组装在一起形成了一个整体设备如图 3-11 所示,为用户提供计算服务。

图 3-11　简易计算机

【例 3-16】 聚合演示。

```
//CPU.java
public class CPU {                                    //抽象的计算控制模块
    private double ax, bx;
    private String instruct;
    private Memory memo;
    public CPU(Memory memo) {
        ax = bx = 0;
        instruct = " + ";
        this.memo = memo;
    }
    public String getInstruct() {
        return instruct;
    }
    public void setInstruct(String instruct) {
        this.instruct = instruct;
    }
    public void calculate() {
        ax = memo.getFirstnum();
        bx = memo.getSecondnum();
        switch (instruct) {
        case " + ":
            ax = ax + bx;
            break;
        case " - ":
```

```
            ax = ax - bx;
            break;
        case "*":
            ax = ax * bx;
            break;
        case "/":
            ax = ax / bx;
            break;
        default:
        }
        memo.setResult(ax);
    }
}
//Memory.java
public class Memory {                                    //抽象的存储子模块
    private double firstnum;
    private double secondnum;
    private double result;
    @Override
    public double getFirstnum() {
        return firstnum;
    }
    public void setFirstnum(double firstnum) {
        this.firstnum = firstnum;
    }
    public double getSecondnum() {
        return secondnum;
    }
    public void setSecondnum(double secondnum) {
        this.secondnum = secondnum;
    }
    public double getResult() {
        return result;
    }
    public void setResult(double result) {
        this.result = result;
    }
}
//Keyboard.java
import java.util.Scanner;
public class Keyboard {                                  //抽象输入键盘模块
    private Scanner keyin = new Scanner(System.in);
    public double inputDouble() {
        return keyin.nextDouble();
    }
    public String inputString() {
        return keyin.next();
    }
}
//Screen.java
import java.io.PrintStream;
```

```
public class Screen {                                    //抽象屏幕显示模块
    private PrintStream out;
    public Screen() {
        this.out = System.out;
    }
    public Screen(PrintStream out) {
        this.out = out;
    }
    public void print(String str) {
        out.print(str);
    }
    public void println(String str) {
        out.println(str);
    }
    public void println() {
        out.println();
    }
    public PrintStream getOut() {
        return out;
    }
    public void setOut(PrintStream out) {
        this.out = out;
    }
}
//Computer.java
public class Computer {                                  //抽象的聚合后的简易计算机
    private CPU cpu;                                     //组合子对象 cpu 处理计算
    private Memory memory;                               //组合子对象 memory 处理存储
    private Keyboard keyboard;                           //组合子对象 keyboard 处理输入
    private Screen screen;                               //组合子对象 screen 处理输出
    public Computer() {
        memory = new Memory();
        cpu = new CPU(memory);
        keyboard = new Keyboard();
        screen = new Screen();
    }
    public Computer(CPU cpu, Memory memory, Keyboard keyboard, Screen screen) {
        super();
        this.cpu = cpu;
        this.memory = memory;
        this.keyboard = keyboard;
        this.screen = screen;
    }
    public void doWork() {                               //模拟计算机开机工作方法
        screen.print("第一个操作数:");
        memory.setFirstnum(keyboard.inputDouble());
        screen.print("运算符:");
        cpu.setInstruct(keyboard.inputString());
        screen.print("第二个操作数:");
        memory.setSecondnum(keyboard.inputDouble());
        cpu.calculate();
```

```
        screen.println("计算结果:" + memory.getResult());
    }
}
//TestComputer.java
public class TestComputer {
   public static void main(String[] args) {
      Computer mycomputer = new Computer();          //创建一个 Computer 对象
      mycomputer.doWork();                           //执行计算任务
   }
}
```

从该例演示来看,一个 Computer 对象聚合了 4 个子对象,但通过封装后看到的只是一个整体的 Computer 对象,看不到内部的子对象和组成结构,4 个内部子对象协同工作完成整个 Computer 对象工作。

2. 联合及其实现技术

联合代表若干独立的对象可以连接成一个更复杂、功能更强大的对象。例如,家庭影院系统中,电视机、音响、DVD 播放机等各种各样的组件都是独立的,都可以提供特定的功能,但通过一些插接线连接后构成了一个功能更强大系统。同样的,计算机、打印机、麦克风、音响、摄像头等也都是独立存在的小对象,将它们连接在一起就会组成功能更丰富、效率更高的复杂对象。简单地说,联合是将若干独立的子对象连接起来形成具有复杂功能的大对象。例 3-17 抽象了打印机、摄像头、音箱、麦克风等小对象,通过联合可完成更复杂的功能。

【例 3-17】 联合演示。

```
//Printer.java
public class Printer {
    private String brand;
    public Printer(String brand) {
        this.brand = brand;
    }
    public void print(String msg) {
        // TODO Auto-generated method stub
        System.out.println("在" + brand + "打印机上打印:" + msg);
    }
}
//Camera.java
public class Camera {
    private String brand;
    public Camera(String brand) {
        this.brand = brand;
    }
    public byte[] getData() {
        String data = "从" + brand + "摄像头上获取的视频字节流";
        return data.getBytes();
    }
}
//SoundBox.java
public class SoundBox {
    private String brand;
```

```
    public SoundBox(String bd) {
        brand = bd;
    }
    public void play() {
        System.out.println("在" + brand + "上播放歌曲让世界充满爱.....");
    }
}
//Microphone.java
public class Microphone {
    private String brand;
    public Microphone(String brand) {
        this.brand = brand;
    }
    public byte[] getData() {
        String msg = "在" + brand + "麦克风上获取的音频数据流";
        return msg.getBytes();
    }
}
//Computer1.java
public class Computer1 {
    private String brand;
    private Disk mydisk = new Disk("西部数据");        //磁盘为聚合对象
    public Computer1(String brand) {
        this.brand = brand;
    }
    public void playMusic(SoundBox sb) {
        sb.play();
    }
    public byte[] inputVideo(Camera cm) {
        return cm.getData();
    }
    public void print(Printer out,String msg) {
        out.print(msg);
    }
    public byte[] inputAudio(Microphone mh) {
        return mh.getData();
    }
    public void saveData(byte[] data) {
        mydisk.saveData(data);
    }
}
//TestUnion.java
public class TestUnion {
    public static void main(String[] args) {
        Computer1 mycomputer = new Computer1("联系昭阳 450 电脑");
        Printer myprinter = new Printer("Brother DCP - 7057 打印机");
        Camera mycamera = new Camera("奥尼剑影摄像头");
        SoundBox mysound = new SoundBox("好牧人 V8 音箱");
        Microphone mymc = new Microphone("飞利浦麦克风");

        mycomputer.playMusic(mysound);                  // 通过音箱播放音乐
```

```
        mycomputer.saveData(mycamera.getData());
        mycomputer.print(myprinter, "摄像头输入的数据被保存到磁盘上了!");
        mycomputer.saveData(mymc.getData());
        mycomputer.print(myprinter, "麦克风输入的数据也被保存到磁盘上了!");
    }
}
```

从上面的演示可以看出,在联合中每个对象都是独立的,摄像头、打印机、音箱等都不是计算机的组成部分,但通过引用把它们连接起来,就可以在计算机对象中调用打印机的方法或从摄像头获取数据流。

总而言之,聚合是指一个复杂对象由其他子对象组合而成,内聚强、耦合强。而当一个对象需要其他独立对象的服务时,则建议使用联合,联合内聚弱(或者没有),耦合弱。


视频讲解

3.3 Java 语言中的访问权限修饰符

初步学习了面向对象的基本原理后,再来系统学习 Java 语言中的访问修饰符。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 中的访问控制分为以下 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,对子类及同一个包中的类公开。

(3) 默认级别:没有访问控制修饰符,对同一个包的类和对象公开。

(4) 私有级别:用 private 修饰,只有本类对象可以访问,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 来修饰。

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	×	×	×	√

在 Java 中,封装是通过 private 修饰符实现,公开接口是通过 public 修饰符实现的,protected 修饰符一般用在继承体系中,用来给子类公开方法或数据成员。

3.4 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,内存必须由人负责管理,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。在 Java 语言中,JVM 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 JVM 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。程序设计人员花费大量的精力来防止程序中经常出现的失误问题,例如经常忘记释放资源,或者错误地释放正在使用的资源。因此,Java 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,但也有可能使对象复活,不再回收该对象的内存。

3.5 程序建模示例

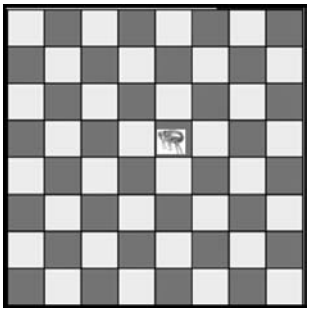

图 3-12 跳蚤实验

【程序建模示例 3-1】 一个房间内铺有 m 行 n 列瓷砖,如图 3-12 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。

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,用二维数组每一个元素的值纪录跳蚤经过此瓷砖的次数。再抽象一个跳蚤类,保存有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());
      }
}
```

程序的一次执行结果如图 3-13 所示。

【程序建模示例 3-2】 采用面向对象的方式抽象一个能够处理多项式的类 Polynomial,该类对象可以表示有限次幂多项式如图 3-14 所示,系数为实数、指数为正整数。请编写程序实现多项式的表示、加法、减法和乘法等操作,可以比较两个多项式是否相等,可以将其转换为字符串,可以修改多项式的某一项。建议另外抽象一个类用来表示单个幂项,例如抽象一个 Item 类,Item(10,5)就表示 $5x^{10}$,Item(20,6)表示 $6x^{20}$ 等,给 Polynomial 类提供一个 add(Item item)方法用来修改多项式的某一项。

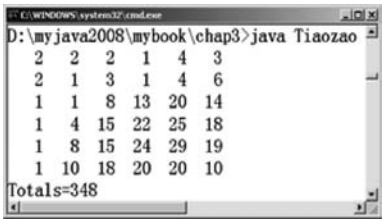

图 3-13 跳蚤程序执行结果

$$p(x) = 5x^{10} + 9x^7 - x - 10$$

图 3-14 多项式抽象

```
//Polynomail.java
import java.util.Arrays;
public class Polynomial {
    private int maxindex;
    private double[ ] coefficient;
    public Polynomial() {
        maxindex = 0;
        coefficient = new double[1];
        coefficient[0] = 0.0;
    }
    public Polynomial(int m, double... c) {
        maxindex = m;
        coefficient = new double[maxindex + 1];
        int j = 0;
        for (int i = maxindex; i >= 0 && j < c.length; i--) {
            coefficient[i] = c[j++];
        }
    }

    public Polynomial(int... arg) {
        if (arg.length > 0) {
            maxindex = arg[0];
            coefficient = new double[maxindex + 1];
            int j = 1;
            for (int i = maxindex; i >= 0 && j < arg.length; i--) {
                coefficient[i] = arg[j++];
            }
        }
    }

    public String toString() {
        StringBuilder str = new StringBuilder();
        if(maxindex == 1) {
            str.append(coefficient[maxindex] + "*x" );
        }else if(maxindex == 0){
            str.append(coefficient[maxindex]);
        }else {
            str.append(coefficient[maxindex] + "*x^" + maxindex);
        }

        for (int i = maxindex - 1; i >= 0; i--) {
            if (coefficient[i] > 0) {
                str.append("+" + coefficient[i]);
                if (i > 0) {
                    if (i == 1) {
                        str.append("*x");
                    } else {
                        str.append("*x^" + i);
                    }
                }
```

```
            } else if (coefficient[i] < 0) {
                str.append(coefficient[i]);
                if (i > 0) {
                    if (i == 1) {
                        str.append("*x");
                    } else {
                        str.append("*x^" + i);
                    }
                }
            }
        }
        return str.toString();
    }

    public void add(Item item) {
        if (item.getIndex() > maxindex) {
            int newmaxindex = item.getIndex();
            double[] newcoefficient = new double[newmaxindex + 1];
            for (int i = 0; i <= maxindex; i++) {
                newcoefficient[i] = coefficient[i];
            }
            for (int i = maxindex + 1; i < newmaxindex; i++) {
                newcoefficient[i] = 0;
            }
            newcoefficient[newmaxindex] = item.getCoefficient();
            maxindex = newmaxindex;
            coefficient = newcoefficient;
        } else {
            coefficient[item.getIndex()] += item.getCoefficient();
        }
        normalize();                                    // 最高系数为 0,规范化处理
    }

    public void normalize() {
        if (maxindex == 0)
            return;
        int i = maxindex;
        while (i > 0&&Math.abs(coefficient[i])< 1e - 5) {
            i--;
        }
        maxindex = i;
    }

    public Polynomial add(Polynomial another) {
        int newmaxindex = Math.max(maxindex, another.maxindex);
        double[] newcoefficient = new double[newmaxindex + 1];
        int i = newmaxindex, j = 0;
        while (i > another.maxindex) {
            newcoefficient[j++] = coefficient[i--];
        }
```

```
        while (i > maxindex) {
            newcoefficient[j++] = another.coefficient[i--];
        }
        while (i >= 0) {
            newcoefficient[j++] = coefficient[i] + another.coefficient[i];
            i--;
        }

        Polynomial tmp = new Polynomial(newmaxindex, newcoefficient);
        tmp.normalize();
        return tmp;
    }

    public Polynomial sub(Polynomial another) {
        int m = another.maxindex;
        for (int i = m; i >= 0; i--) {
            another.coefficient[i] = -another.coefficient[i];
        }
        return add(another);
    }

    public Polynomial mut(Polynomial another) {
        int m = maxindex;
        int n = another.maxindex;
        int newmaxindex = m + n;
        double[] newcoefficient = new double[newmaxindex + 1];
        Polynomial p = new Polynomial(newmaxindex, newcoefficient);
        for (int i = m; i >= 0; i--) {
            double thisco = coefficient[i];          // 系数
            for (int j = n; j >= 0; j--) {
                double anotherco = another.coefficient[j];
                double newco = thisco * anotherco;
                int newindex = i + j;
                Item item = new Item(newindex, newco);
                p.add(item);
            }
        }
        return p;
    }
    // 用递归方法解决除法
    // 如果最高幂次小于 another 的最高次幂,则结束递归,返回商和余数
    // 否则,求出该对象最高次幂和 another 对象的最高次幂的差值,构造一个 Polynomial 对象 obj,只包含一项 item,将其添加到商 results[0]中,然后调用 sub(another.mut(obj)),消除最高次幂项,得余式 results[1]
    // 继续递归调用 results[1].div(results,another);
    private Polynomial[] div(Polynomial[] results,Polynomial another) {
        if(this.maxindex < another.maxindex) {
            return results;
        }else{
            int newindex = this.maxindex - another.maxindex;
            double newcoefficient = this.coefficient[this.maxindex]/another.coefficient
```

```
[another.maxindex];
            Item item = new Item(newindex,newcoefficient);
            Polynomial obj = new Polynomial();
            obj.add(item);
            results[0].add(item);
            results[1] = this.sub(another.mut(obj));
            results[1].div(results,another);        //余项继续递归除法
        }
        return results;
    }
    public Polynomial div(Polynomial another) {
        Polynomial[] results = new Polynomial[2];
        results[0] = results[1] = new Polynomial();
        results = div(results,another);
//      System.out.println("商:" + results[0]);
//      System.out.println("余式:" + results[1]);
        return results[0];
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + Arrays.hashCode(coefficient);
        result = prime * result + maxindex;
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Polynomial other = (Polynomial) obj;
        if (!Arrays.equals(coefficient, other.coefficient))
            return false;
        if (maxindex != other.maxindex)
            return false;
        return true;
    }
}
//TestPolynomail.java
public class TestPolynomial {
    public static void main(String[] args) {
        Polynomial p = new Polynomial(10, 5, 0, 0, 9, 0, 0, 0, 0, 0, -1, -10);
        Polynomial p1 = new Polynomial(10, 6, 0, 0, 0, 15, 0, 0, 3, 0, -1, 20);
        System.out.println("多项式 p:" + p);
        Item item = new Item(9, 11);
```

```
        p.add(item);
        System.out.println("添加 11x^9 后的多项式 p:" + p);
        item = new Item(11, 20);
        p.add(item);
        System.out.println("添加 20x^11 后的多项式 p:" + p);
        System.out.println("多项式 p1:" + p1);
        Polynomial p2 = p.add(p1);
        System.out.println("p2 = " + p2);
        Polynomial p3 = p.sub(p1);
        System.out.println("p3 = " + p3);
        Polynomial p4 = p.mut(p1);
        System.out.println("p4 = " + p4);
        /**********************************************/
        Polynomial dividend = new Polynomial(3,1,1,7,9);      //x^3 + x^2 + 7x + 9
        System.out.println("dividend = " + dividend);
        Polynomial divisor = new Polynomial(2,1,5,6);         //x^2 + 5x + 6
        System.out.println("divisor = " + divisor);           //x - 4
        Polynomial results = dividend.div(divisor);           //21x + 33
        System.out.println("商:" + results);
    }
}
```

3.6 本章小结

本章详细介绍了面向对象程序设计的基本概念、基本原理及其实现技术，包括抽象原理、封装原理、继承原理、多态原理和组合原理。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如何进行抽象、类的封装技巧、方法重载、getter 方法、setter 方法以及其他功能方法的设计技术。

如果一个对象公共接口的所有特性都与这个对象表示的概念有关，则称这个类的设计是内聚的。如果一个类可以通过具体的设计变得更具体，则应该采用继承思想，通过该类派生出一个子类，在子类中可以添加新属性或新方法，也可以在子类中重写从父类中继承下来的方法。如果一个对象的方法以某种方式使用了另一个对象，则说明两个类之间有依赖关系，需要采用联合模式。如果一个对象由许多小对象组成，则该类的设计中需要采用组合模式。

最后介绍了内存管理中的一种策略，即垃圾回收策略。Java 语言通过 JVM 实现了用垃圾回收策略代替 C 语言中的程序员需要自己管理内存的方案。

第 3 章 习 题

一、单选题

1. 下列不属于面向对象原理的是(　　)。

A. 封装　　B. 代理　　C. 多态　　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 包中的 Object 类的(　　)方法将比较两个对象的内容是否相等,如果相等则返回 true,不相等则返回 false。

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

5. 当编译并运行下列程序段时,将会发生什么情况?(　　)

```
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

6. 对下列定义的类,如何修改 salary 属性使得它既能被封装,又能被访问和修改?(　　)

```
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

7. 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

8. 关于构造方法,下列说法错误的是(　　)。

A. 构造方法可以重载

B. 构造方法用来初始化该类的一个新的对象

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

9. 在 Java 中,为了使一个名为 Example 的类成功地编译和运行,必须满足以下哪个条件?(　　)

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

10. 给出以下代码,该程序的输出结果是什么?(　　)

```
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

11. 编译并执行下列程序段,将会输出什么结果?(　　)

```
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

12. 下列哪一个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、编程题

1. 设计一个 Timer 类,属性包括时、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,输入等差数列的开始数、个数 n、等差 d,输出该等差数列的和。

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

5. 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
    ……
```

6. 设计一个名为 MyInteger 的类。该类包含:

(1) 一个名为 value 的 int 数据字段,存储由该对象表示的 int 值。

(2) 为指定的整型值创建 MyInteger 对象的构造函数。

(3) 返回整型值的 getter 方法。

(4) isEven()、isOdd()和 isPrime()方法如果该对象中的值是偶数、奇数或素数,则分别返回 true。

(5) 方法 equals(int)和 equals(MyInteger),如果该对象中的值等于指定的值,则返回 true。

(6) toString()方法,返回该整型值的字符串形式。

设计一个测试类,对以上属性或方法进行测试。

三、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. 请说明当比较两个对象时,使用“==”比较两个引用变量和使用 equals()方法比较两个引用变量时的区别。

5. 类的实例变量在什么时候会被分配内存空间?

6. 什么是封装? Java 中如何实现封装?

7. 什么是继承和多态?

8. 什么是组合？有哪些实现方式？

9. 什么是匿名对象？

10. 解释 Java 的内存垃圾回收机制。

11. 为什么下列代码会导致 NullPointerException？

```
public class Test {
    private String text;
    public Test(String s) {
        String text = s;
    }
    public static void main(String[] args) {
        Test test = new Test("ABC");
  System.out.println(test.text.toLowerCase());
    }
}
```