第 7 章

类和对象

（视频讲解：46 分钟）

在 Java 语言中经常被提到的两个词汇是类与对象，实质上可以将类看作是对象的载体，它定义了对象所具有的功能。学习 Java 语言必须要掌握类与对象，这样可以从深层次去理解 Java 这种面向对象语言的开发理念，使程序员更好、更快地掌握 Java 编程思想与编程方式，因此掌握类与对象是学习 Java 语言的基础。本章详细介绍了类的各种方法以及对象，讲解过程中为了使初学者更容易入门，笔者列举了大量实例。

通过阅读本章，您可以：

- 了解面向对象编程思想
- 掌握如何定义类
- 掌握类的成员变量、成员方法
- 掌握权限修饰符
- 掌握局部变量以及作用范围
- 掌握 this 关键字
- 掌握如何创建对象
- 掌握使用对象获取对象的属性和行为
- 掌握对象的创建、比较和销毁

7.1 面向对象概述

视频讲解：光盘\TM\lx\7\面向对象概述.exe

在程序开发初期人们使用结构化开发语言，但是随着时间的流逝，软件的规模越来越庞大，结构化语言的弊端也逐渐暴露出来，开发周期被无休止地拖延，产品的质量也不尽如人意，人们终于发现结构化语言已经不再适合当前的软件开发。这时人们开始将另一种开发思想引入程序中，即面向对象的开发思想。面向对象思想是人类最自然的一种思考方式，它将所有预处理的问题抽象为对象，同时了解这些对象具有哪些相应的属性以及展示这些对象的行为，以解决这些对象面临的一些实际问题，这样就在程序开发中引入了面向对象设计的概念，面向对象设计实质上就是对现实世界的对象进行建模操作。

7.1.1 什么是对象

现实世界中，随处可见的一种事物就是对象，对象是事物存在的实体，如人类、书桌、计算机、高楼大厦等。人类解决问题的方式总是将复杂的事物简单化，于是就会思考这些对象都是由何种部分组成的。通常都会将对象划分为两部分，即动态部分与静态部分。静态部分，顾名思义，就是不能动的部分，这部分被称为“属性”，任何对象都会具备其自身属性，例如一个人，它包括高矮、胖瘦、性别、年龄等属性。然而具有这些属性的人会执行哪些动作也是一个值得探讨的部分，这个人可以哭泣、微笑、说话、行走，这些是这个人具备的行为（动态部分），人类通过探讨对象的属性和观察对象的行为了解对象。

在计算机的世界中，面向对象程序设计的思想要以对象来思考问题，首先要将现实世界的实体抽象为对象，然后考虑这个对象具备的属性和行为。例如，现在面临一只大雁要从北方飞往南方这样一个实际问题，试着以面向对象的思想来解决这一实际问题，步骤如下：

（1）首先可以从这一问题中抽象出对象，这里抽象出的对象为大雁。

（2）识别这个对象的属性。对象具备的属性都是静态属性，例如大雁有一对翅膀、黑色的羽毛等。这些属性如图 7.1 所示。

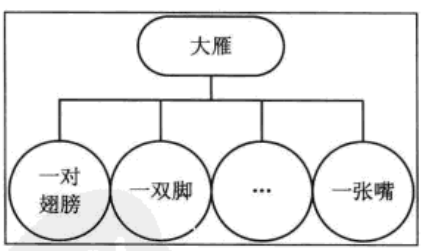

图 7.1 识别对象上的属性

（3）识别这个对象的动态行为，即这只大雁可以进行的动作，如飞行、觅食等，这些行为都是因为这个对象基于其属性而具有的动作。这些行为如图 7.2 所示。

（4）识别出这些对象的属性和行为后，这个对象就被定义完成，然后可以根据这只大雁具有的特性制定这只大雁要从北方飞向南方的具体方案以解决问题。实质上究其本质，所有的鸟类都具有以上的属性和行为，可以将这些属性和行为封装起来以描述鸟类。由此可见，类实质上就是封装对象属性和行为的载体，而对象则是类抽象出来的一个实例，两者之间的关系如图 7.3 所示。

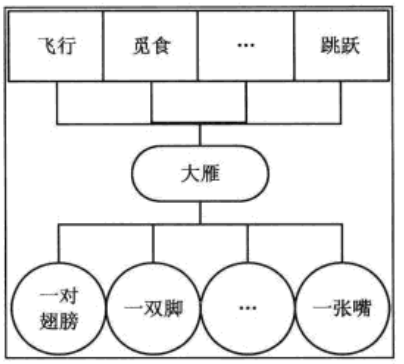

图 7.2 对象具有的属性和行为

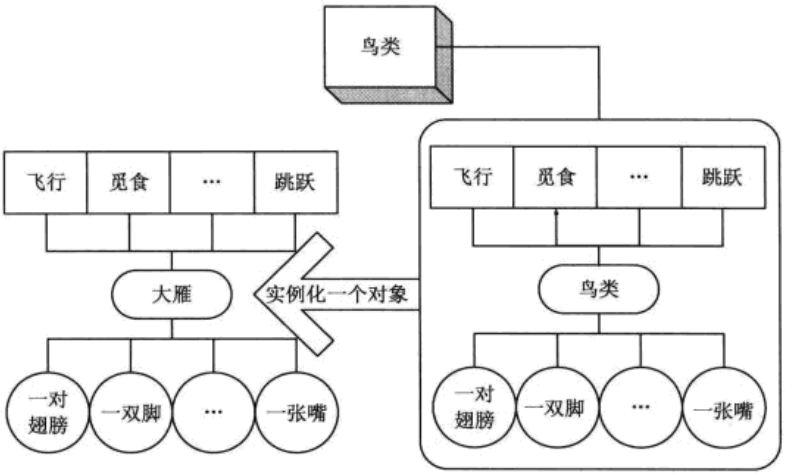

图 7.3 描述对象与类之间的关系

7.1.2 什么是类

不能将所谓的一个事物描述成一类事物，如一只鸟不能称为鸟类，如果需要对同一类事物统称，就不得不说明类这个概念。

类就是同一类事物的统称，如果将现实世界中的一个事物抽象成对象，类就是这类对象的统称，如鸟类、家禽类、人类等。类是构造对象时所依赖的规范，例如，一只鸟具有一对翅膀，而它可以通过这对翅膀飞行，而基本上所有的鸟都具有翅膀这个特性和飞行的技能，这样的具有相同特性和行为的一类事物就称为类，类的思想就是这样产生的。在图 7.3 中已经描述过类与对象之间的关系，对象就是符合某个类定义所产生出来的实例。更为恰当的描述是：类是世间事物的抽象称呼，而对象则是这个事物相对应的实体。如果面临实际问题，通常需要实例化类对象来解决。例如解决大雁南飞的问题，这里只能拿这只大雁来处理这个问题，不能拿大雁类或鸟类来解决。

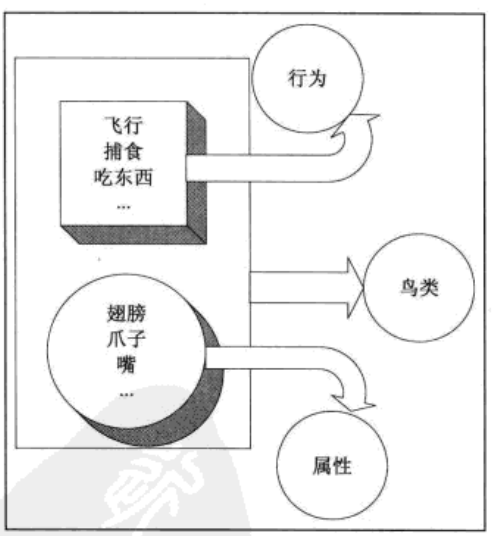

图 7.4 鸟类结构

类是封装对象的属性和行为的载体，反过来说具有相同属性和行为的一类实体被称为类。例如一个鸟类，鸟类封装了所有鸟的共同属性和应具有的行为，其结构如图 7.4 所示。

定义完成鸟类之后，可以根据这个类抽象出一个实体对象，最后通过实体对象来解决相关一些实际问题。

在 Java 语言中，类中对象的行为是以方法的形式定义的，对象的属性是以成员变量的形式定义的，而类包括对象的属性和方法，有关类的具体实现会在后续章节中进行介绍。

7.1.3 面向对象的特点

面向对象程序设计具有以下几个特点。

1. 封装

封装是面向对象编程的核心思想，将对象的属性和行为封装起来，而将对象的属性和行为封装起来的载体就是类，类通常对客户隐藏其实现细节，这就是封装的思想。例如，用户使用计算机，只需要使用手指敲击键盘就可以实现一些功能，用户无须知道计算机内部是如何工作的，即使用户可能碰巧知道计算机的工作原理，但在使用计算机时并不完全依赖于计算机工作原理这些细节。

采用封装的思想保证了类内部数据结构的完整性，应用该类的用户不能轻易直接操纵此数据结构，而只能执行类允许公开的数据。这样避免了外部对内部数据的影响，提高程序的可维护性。

使用类实现封装特性如图 7.5 所示。

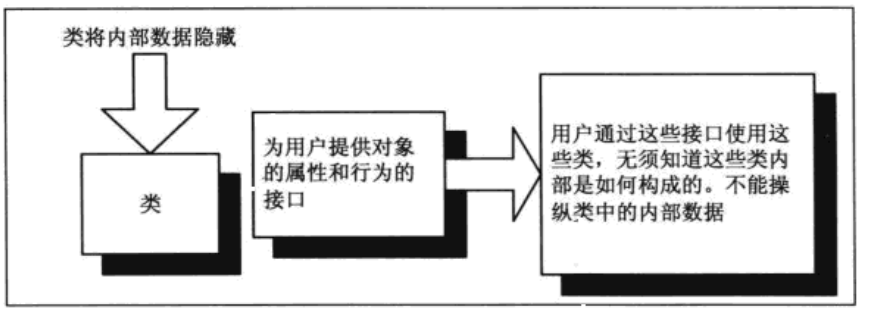

图 7.5 封装特性示意图

2. 继承

类与类之间同样具有关系，如一个百货公司类与销售员类相联系，类之间这种关系被称为关联。关联是描述两个类之间的一般二元关系，例如一个百货公司类与销售员类就是一个关联，再如学生类以及教师类也是一个关联。两个类之间的关系有很多种，继承是关联中的一种。

当处理一个问题时，可以将一些有用的类保留下来，当遇到同样问题时拿来复用。假如这时需要解决信鸽送信的问题，我们很自然就会想到图 7.4 所示的鸟类。由于鸽子属于鸟类，鸽子具有鸟类相同的属性和行为，便可以在创建信鸽类时将鸟类拿来复用，并且保留鸟类具有的属性和行为。不过，并不是所有的鸟都有送信的习惯，因此还需要再添加一些信鸽具有的独特属性以及行为。鸽子类保留了鸟类的属性和行为，这样就节省了定义鸟和鸽子共同具有的属性和行为的时间，这就是继承的基本思想。可见软件的代码使用继承思想可以缩短软件开发的时间，复用那些已经定义好的类可以提高系统性能，减少系统在使用过程中出现错误的几率。

继承性主要利用特定对象之间的共有属性。例如，平行四边形是四边形（正方形、矩形也都是四边形），平行四边形与四边形具有共同特性，就是拥有 4 个边，可以将平行四边形类看作四边形的延伸，平行四边形复用了四边形的属性和行为，同时添加了平行四边形独有的属性和行为，如平行四边形的

对边平行且相等。这里可以将平行四边形类看作是从四边形类中继承的。在Java语言中将类似于平行四边形的类称为子类，将类似于四边形的类称为父类或超类。值得注意的是，可以说平行四边形是特殊的四边形，但不能说四边形是平行四边形，也就是说子类的实例都是父类的实例，但不能说父类的实例是子类的实例，图形类之间的继承关系如图7.6所示。

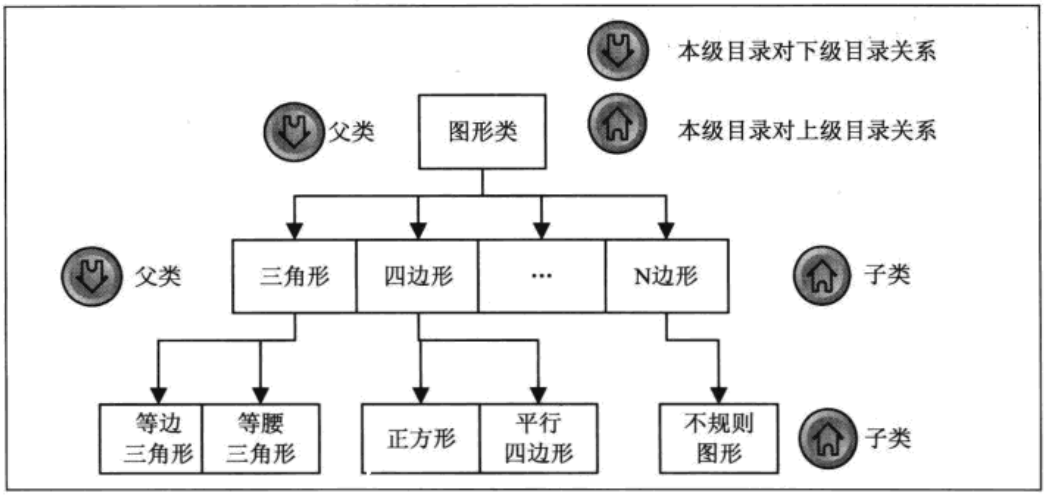

图7.6 图形类层次结构示意图

从图7.6中可以看出，继承关系可以使用树形关系来表示，父类与子类存在一种层次关系。一个类处于继承体系中，它既可以是其他类的父类，为其他类提供属性和行为，也可以是其他类的子类，继承父类的属性和方法，如三角形既是图形类的子类同时也是等边三角形的父类。

3. 多态

前面已介绍了继承，了解了父类和子类，其实将父类对象应用于子类的特征就是多态。依然以图形类来说明多态，每个图形都拥有绘制自己的能力，这个能力可以看作是该类具有的行为，如果将子类的对象统一看作是超类的实例对象，这样当绘制任何图形时，可以简单地调用父类也就是图形类绘制图形的方法即可绘制任何图形，这就是多态最基本的思想。

多态性允许以统一的风格编写程序，以处理种类繁多的已存在的类以及相关类。该统一风格可以由父类来实现，根据父类统一风格的处理，就可以实例化子类的对象。由于整个事件的处理都只依赖于父类的方法，所以日后只要维护和调整父类的方法即可，这样既降低了维护的难度，又节省了时间。

在提到多态的同时，不得不提到抽象类和接口，因为多态的实现并不依赖具体类，而是依赖于抽象类和接口。

再回到绘制图形的实例上来。作为所有图形的父类图形类，它具有绘制图形的能力，这个方法可以称为“绘制图形”，但如果要执行这个“绘制图形”的命令，没人知道应该画什么样的图形，并且如果要在图形类中抽象出一个图形对象，没有人能说清这个图形究竟是什么图形，所以使用“抽象”这个词汇来描述图形类比较恰当。在Java语言中称这样的类为抽象类，抽象类不能实例化对象。在多态的机制中，父类通常会被定义为抽象类，在抽象类中给出一个方法的标准，而不给出实现的具体流程。实质上这个方法也是抽象的，例如图形类中的“绘制图形”方法只提供一个可以绘制图形的标准，并

没有提供具体绘制图形的流程，因为没有人知道究竟需要绘制什么形状的图形。

在多态的机制中，比抽象类更为方便的方式是将抽象类定义为接口。由抽象方法组成的集合就是接口。接口的概念在现实中也极为常见，如从不同的五金商店买来螺丝和螺丝钉，螺丝很轻松地就可以拧在螺丝钉上，可能螺丝和螺丝钉的厂家不同，但这两个物品可以很轻易地组合在一起，这是因为生产螺丝和螺丝钉的厂家都遵循着一个标准，这个标准在 Java 语言中就是接口。依然拿“绘制图形”来说明，可以将“绘制图形”作为一个接口的抽象方法，然后使图形类实现这个接口，同时实现“绘制图形”这个抽象方法，当三角形类需要绘制时，就可以继承图形类，重写其中“绘制图形”方法，改写这个方法为“绘制三角形”，这样就可以通过这个标准绘制不同的图形。

7.2 类

视频讲解：光盘\TM\lx\7\类.exe

在 7.1.2 节中已经讲解过类是封装对象的属性和行为的载体，而在 Java 语言中对象的属性以成员变量的形式存在，而对象的方法以成员方法的形式存在。本节将介绍在 Java 语言中类是如何定义的。

7.2.1 类的构造方法

在类中除了成员方法之外，还存在一种特殊类型的方法，那就是构造方法。构造方法是一个与类同名的方法，对象的创建就是通过构造方法完成的，每当类实例化一个对象时，类都会自动调用构造方法。构造方法的特点如下：

☑ 构造方法没有返回值。

☑ 构造方法的名称要与本类的名称相同。

注意 在定义构造方法时，构造方法没有返回值，但这与普通没有返回值的方法不同，普通没有返回值的方法使用 public void methodEx()这种形式进行定义，但构造方法并不需要 void 关键字进行修饰。

构造方法的定义语法格式如下：

```
public book(){
    //...构造方法体
}
```

☑ public：构造方法修饰符。

☑ book：构造方法的名称。

在构造方法中可以为成员变量赋值，这样当实例化一个本类的对象时，相应的成员变量也将被初始化。

注意 如果在类中定义的构造方法都不是无参的构造方法，则编译器不会为类设置一个默认的无参构造方法，当试图调用无参构造方法实例化一个对象时，编译器会报错。所以只有在类中没有定义任何构造方法时，编译器才会在该类中自动创建一个不带参数的构造方法。

在上文中介绍过 this 关键字，了解了 this 可以调用类的成员变量和成员方法，事实上 this 还可以调用类中的构造方法。

【例 7.1】 在项目中创建 AnyThting 类，该类中使用 this 关键字调用构造方法。

```
public class AnyThting {
    public AnyThting(){                                    //定义无参构造方法
        this("this 调用有参构造方法");                      //使用 this 关键字调用有参构造方法
        System.out.println("无参构造方法");
    }
    public AnyThting(String name){                         //定义有参构造方法
        System.out.println("有参构造方法");
    }
}
```

在例 7.1 中可以看到定义了两个构造方法，在无参构造方法中可以使用 this 关键字调用有参的构造方法。但使用这种方式值得注意的是，只可以在无参构造方法中的第一句使用 this 关键字调用有参构造方法。

7.2.2 类的主方法

主方法是类的入口点，它定义了程序从何处开始；主方法提供对程序流向的控制，Java 编译器通过主方法来执行程序。主方法的语法格式如下：

```
public static void main(String[] args){
    //方法体
}
```

在主方法的定义中可以看到主方法具有以下特性：

☑ 主方法是静态的。要直接在主方法中调用其他方法，则该方法必须也是静态的。

☑ 主方法没有返回值。

☑ 主方法的形参为数组。

用 args[0]～args[n]分别代表程序的第一到第 n 个参数，可以使用 args.length 获取参数的个数。

【例 7.2】 在项目中创建 TestMain 类，在主方法中编写如下代码，并在 Eclipse 中设置程序参数。

```
public class TestMain {
    public static void main(String[] args) {              //定义主方法
        for(int i=0;i<args.length;i++){                    //根据参数个数做循环操作
```

```
            System.out.println(args[i]);                    //循环打印参数内容
        }
    }
}
```

运行结果如图 7.7 所示。

在 Eclipse 中设置程序参数的步骤如下：

（1）在 Eclipse 中选择菜单栏中的“运行”/“运行”命令，弹出“运行”对话框。

（2）选择“自变量”选项卡，在“程序自变量”文本框中输入相应的参数，每个参数间按 Enter 键隔开。具体设置如图 7.8 所示。

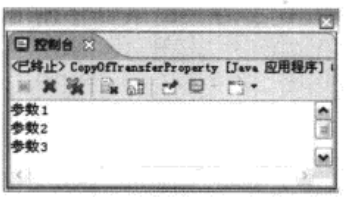

图 7.7　带参数的程序

图 7.8　Eclipse 中的“运行”对话框

7.2.3　成员变量

在 Java 语言中对象的属性称为成员变量，也可以称为属性。为了了解成员变量，首先定义一个图书类，成员变量对应于类对象的属性，在 Book 类中设置 id、name 和 category 3 个成员变量，分别对应于图书编号、图书名称和图书类别这 3 个图书属性。

【例 7.3】　在项目中创建 Book 类，并在该类中定义并使用成员变量。

```
public class Book{
    private int id;                                 //定义一个 int 型的成员变量
    private String name;                            //定义一个 String 型的成员变量
    private String category;                        //定义一个 String 型的成员变量
    public String getName(){                        //定义一个 getName()方法
        int id=0;                                   //局部变量
        setName("Java");                            //调用类中其他方法
        return id+this.name;                        //设置方法返回值
```

```
    }
    private void setName(String name){                    //定义一个 setName()方法
        this.name=name;                                   //将参数值赋予类中的成员变量
    }
    public Book getBook(){
        return this;                                      //返回 Book 类引用
    }
}
```

根据以上代码，读者可以看到在 Java 语言中使用 class 关键字来定义类，Book 是类的名称。同时在 Book 类中定义了 3 个成员变量，成员变量的类型可以设置为 Java 语言中合法的数据类型，其实成员变量就是普通的变量，可以为它设置初始值，也可以不为其设置，如果不设置初始值，则会有默认值。在 3 个成员变量前面读者应该注意到 private 关键字，它用来定义一个私有成员（关于权限修饰符的说明会在 7.2.8 节中进行介绍）。

7.2.4 成员方法

在 Java 语言中使用成员方法对应于类对象的行为。以 Book 类为例，它包含 getName()和 setName()两个方法，分别为获取图书名称和设置图书名称的方法。定义成员方法的语法格式如下：

```
权限修饰符 返回值类型 方法名(参数类型 参数名){
    ...//方法体
    return 返回值;
}
```

一个成员方法可以有参数，这个参数可以是对象也可以是基本数据类型的变量，同时成员方法有返回值和不返回任何值的选择，如果方法需要返回值可以在方法体中使用 return 关键字，使用这个关键字后，方法的执行将被终止。

注意

Java 语言中的成员方法无返回值可以使用 void 关键字表示。

成员方法的返回值可以是计算结果也可以是其他想要的数值、对象，返回值类型要与方法返回的值类型一致。

在成员方法中可以调用其他成员方法和类成员变量，例如在 getName()方法中就调用了 setName()方法将图书名称赋予一个值。同时在成员方法中可以定义一个变量，这个变量为局部变量（局部变量的内容会在 7.2.5 节中进行介绍）。

说明 如果一个方法中含有与成员变量同名的局部变量，则方法中对这个变量的访问以局部变量进行访问。例如变量 id，在 getName()方法中 id 的值为 0，而不是成员变量中 id 的值。

7.2.5 局部变量

在 7.2.4 节中已经讲述过成员方法，如果在成员方法内定义一个变量，那么这个变量被称为局部变量。

例如，在例 7.1 中定义的 Book 类中，getName()方法的 id 变量即为局部变量。实际上方法中的形参也可作为一个局部变量。例如在定义 setName(String name)方法时，String name 这个形参就被看作是局部变量。

局部变量是在方法被执行时创建，在方法执行结束时被销毁。局部变量在使用时必须进行赋值操作或被初始化，否则会出现编译错误。

【例 7.4】 在项目中创建一个类文件，在该类中定义 getName()方法并进行调用。

```
public String getName(){                    //定义一个 getName()方法
    int id=0;                               //局部变量
    setName("Java");                        //调用类中其他方法
    return id+this.name;                    //设置方法返回值
}
```

如果将 id 这个局部变量的初始值去掉，编译器将出现错误。

7.2.6 局部变量的有效范围

可以将局部变量的有效范围称为变量的作用域，局部变量的有效范围从该变量的声明开始到该变量的结束为止。局部变量的作用范围如图 7.9 所示。

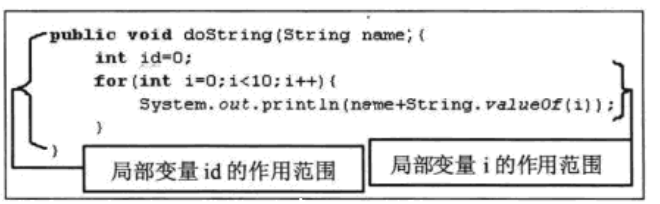

图 7.9 局部变量的作用范围

在相互不嵌套的作用域中可以同时声明两个名称、类型相同的局部变量，如图 7.10 所示。

图 7.10 在不同嵌套区域可以定义相同名称、类型的局部变量

但是在相互嵌套的区域中不可以这样声明，如果将局部变量 id 值在方法体的 for 循环中再次定义，

编译器将会报错，如图 7.11 所示。

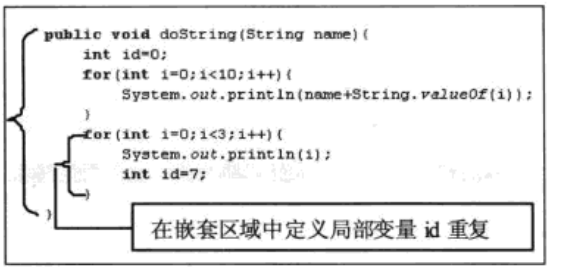

图 7.11 在方法体的嵌套区域中不可以定义相同名称和类型的局部变量

注意 在作用范围外使用局部变量是一个常见的错误，因为在作用范围外没有声明局部变量的代码。

7.2.7 静态变量、常量和方法

在介绍静态变量、常量和方法之前首先需要介绍 static 关键字，因为由 static 修饰的变量、常量和方法被称作静态变量、常量和方法。

有时在处理问题时，会需要两个类在同一个内存区域共享一个数据。例如，在球类中使用 PI 这个常量，可能除了本类需要这个常量之外，在另外一个圆类中也需要使用这个常量。这时没有必要在两个类中同时创建 PI 这个常量，因为这样系统会将这两个不在同一个类中定义的常量分配到不同的内存空间中。为了解决这个问题，可以将这个常量设置为静态的。PI 常量在内存中被共享的布局如图 7.12 所示。

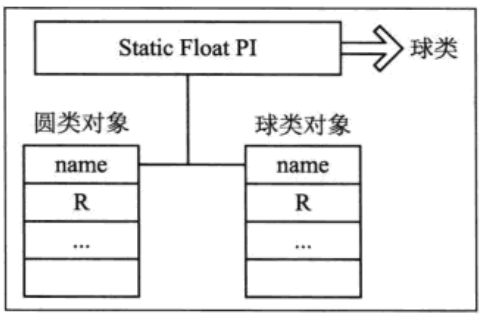

图 7.12 PI 常量在内存中被共享情况

被声明为 static 的变量、常量和方法被称为静态成员。静态成员是属于类所有的，区别于个别对象，可以在本类或其他类使用类名和“.”运算符调用静态成员。语法格式如下：

类名.静态类成员

【例 7.5】 在项目中创建 StaticTest 类，该类中的主方法调用静态成员并在控制台中输出。

```
public class StaticTest {
    static double PI=3.1415;                          //在类中定义静态常量
    static int id;                                    //在类中定义静态变量
    public static void method1(){                     //在类中定义静态方法
        //方法体
    }
    public void method2(){
    System.out.println(StaticTest.PI);                //调用静态常量
    System.out.println(StaticTest.id);                //调用静态变量
    StaticTest.method1();                             //调用静态方法
    }
}
```

在例 7.5 中设置了 3 个静态成员，分别为常量、变量和方法，然后在 method2()方法中分别调用这 3 个静态成员，直接使用“类名.静态成员”的形式进行调用即可。

注意 虽然静态成员也可以使用“对象.静态成员”的形式进行调用，但这样的形式通常不被鼓励使用，因为这样容易混淆静态成员和非静态成员。

静态数据与静态方法的作用通常是为了提供共享数据或方法，如数据计算公式等，以 static 声明并实现，这样当需要使用时，直接使用类名调用这些静态成员即可。尽管使用这种方式调用静态成员比较方便，但静态成员同样遵循着 public、private、protected 修饰符的约束。

【例 7.6】 在项目中创建 StaticTest 类，该类中的主方法调用静态成员并在控制台中输出。

```
public class StaticTest {
    static double PI=3.1415;                          //在类中定义静态常量
    static int id;                                    //在类中定义静态变量
    public static void method1(){                     //在类中定义静态方法
        //方法体
    }
    public void method2(){                            //在类中定义一个非静态方法
    System.out.println(StaticTest.PI);                //调用静态常量
    System.out.println(StaticTest.id);                //调用静态变量
    StaticTest.method1();                             //调用静态方法
    }
    public static StaticTest method3(){               //在类中定义一个静态方法
        method2();                                    //调用非静态方法
        return this;                                  //在 return 语句中使用 this 关键字
    }
}
```

读者也许会发现在 Eclipse 中输入上述代码后，编译器会发生错误，这是因为 method3()方法为一个静态方法，而在其方法体中调用了非静态方法和 this 关键字。在 Java 语言中对静态方法有以下两点规定：

- ☑ 在静态方法中不可以使用 this 关键字。
- ☑ 在静态方法中不可以直接调用非静态方法。

注意 在Java语言中规定不能将方法体内的局部变量声明为static。例如下述代码就是错误的：

```
public class example{
 public void method(){
      static int i=0;
 }
}
```

技巧 如果在执行类时，希望先执行类的初始化动作，可以使用static定义一个静态区域。例如：

```
public class example{
 static{
      //some
 }
}
```

当这段代码被执行时，首先执行static块中的程序，并且只会执行一次。

7.2.8 权限修饰符

Java语言中的权限修饰符主要包括private、public和protected，这些修饰符控制着对类和类的成员变量以及成员方法的访问。如果一个类的成员变量或成员方法被修饰为private，则该成员变量只能在本类中被使用，在子类中是不可见的，并且对其他包的类也是不可见的。如果将类的成员变量和成员方法的访问权限设置为public，则除了可以在本类使用这些数据之外，还可以在子类和其他包中的类中使用。如果一个类的访问权限被设置为private，这个类将隐藏其内的所有数据，以免用户直接访问它。如果需要使类中的数据被子类或其他包中的类使用，可以将这个类设置为public访问权限。如果一个类使用protected修饰符，那么只有本包内该类的子类或其他类可以访问此类中的成员变量和成员方法。

这么看来，public和protected修饰的类可以由子类访问，如果子类和父类不在同一包中，那么只有修饰符为public的类可以被子类进行访问。如果父类不允许通过继承产生的子类访问它的成员变量，那么必须使用private声明父类的这个成员变量。表7.1中描述了private、protected、public修饰符的修饰权限。

表7.1 Java语言中的修饰符

访问包位置	类修饰符		
	private	protected	public
本类	可见	可见	可见
同包其他类或子类	不可见	可见	可见
其他包的类或子类	不可见	不可见	可见

注意 当声明类时不使用 public、protected、private 修饰符设置类的权限，则这个类预设为包存取范围，即只有一个包中的类可以调用这个类的成员变量或成员方法。

【例 7.7】 在项目中的 com.wsy 包下创建 AnyClass 类，该类使用默认的访问权限。

```
package com.wsy;
class AnyClass {
    public void doString(){
        //...方法体
    }
}
```

在上述代码中，由于类的修饰符为默认修饰符，即只有一个包内的其他类和子类可以对该类进行访问，而 AnyClass 类中的 doString()方法却又被设置为 public 访问权限，即使这样，doString()方法的访问权限依然与 AnyClass 类的访问权限相同，因为 Java 语言规定，类的权限设定会约束类成员上的权限设定，所以上述代码等同于下面的代码。

【例 7.8】 本实例等同于例 7.7 的代码。

```
package com.wsy;
class AnyClass {
    void doString(){
        //...方法体
    }
}
```

7.2.9 this 关键字

【例 7.9】 在项目中创建一个类文件，该类中定义了 setName()方法，并将方法的参数值赋予类中的成员变量。

```
private void setName(String name){                    //定义一个 setName()方法
    this.name=name;                                   //将参数值赋予类中的成员变量
}
```

在上述代码中可以看到，成员变量与在 setName()方法中的形式参数的名称相同，都为 name，那么如何在类中区分使用的是哪一个变量？在 Java 语言中规定使用 this 关键字来指定。this 关键字被隐式地用于引用对象的成员变量和方法，如在上述代码中，this.name 指定的就是 Book 类中的 name 成员变量，而 this.name=name 语句中的第二个 name 则指定的是形参 name。实质上 setName()方法实现的功能就是将形参 name 的值赋予成员变量 name。

在这里读者明白了 this 可以调用成员变量和成员方法，但这并不是 Java 语言中的最常规调用方式——使用“对象.成员变量”或“对象.成员方法”进行调用。既然 this 关键字可以调用成员变量和成员方法，究竟 this 关键字与对象之间具有怎样的关系？事实上 this 引用就是对一个对象的引用。

在 setName()方法体中也可以使用 name=name 这种形式，可以在引用成员变量时省略 this 关键字，其结果与使用 this 关键字是一致的。既然使用 this 与没有使用 this 的效果相同，那么为什么要使用 this 关键字？其实 this 除了可以调用成员变量或成员方法之外，还可以作为方法的返回值。

【例 7.10】 在项目中创建一个类文件，该类中定义 Book 类型的方法，并通过 this 关键字进行返回。

```
public Book getBook(){
    return this;                                    //返回 Book 类引用
}
```

在 getBook()方法中，方法的返回值为 Book 类，所以方法体中使用 return this 这种形式将 Book 类的对象进行返回。

7.2.10 范例 1：自定义图书类

在使用 Java 语言进行开发时，时刻要以面向对象的思想考虑问题。面向对象的基础就是类，本范例将演示如何自定义类，它用于表示图书。运行结果如图 7.13 所示。（**实例位置：光盘\TM\sl\7\1**）

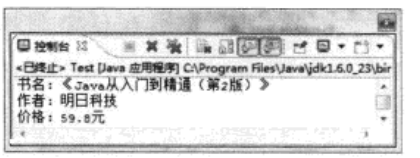

图 7.13 输出自定义的图书类对象

（1）在项目中创建 Book 类，在类中定义 3 个成员变量，分别表示书名、作者和价格，同时提供构造方法和成员方法来修改成员变量。代码如下：

```
public class Book {
    private String title;                                   //定义书名
    private String author;                                  //定义作者
    private double price;                                   //定义价格
    public Book(String title, String author, double price) {    //利用构造方法初始化域
        this.title = title;
        this.author = author;
        this.price = price;
    }
    public String getTitle() {                              //获得书名
        return title;
    }
    public String getAuthor() {                             //获得作者
        return author;
    }
    public double getPrice() {                              //获得价格
        return price;
    }
}
```

（2）在 com.mingrisoft 包中创建类文件，名称为 Test。在该类的 main()方法中，创建了一个 Book

对象并输出了其属性。代码如下：

```
public class Test {
    public static void main(String[] args) {
        Book book = new Book("《Java 从入门到精通（第 2 版）》", "明日科技", 59.8); //创建对象
        System.out.println("书名：" + book.getTitle());                          //输出书名
        System.out.println("作者：" + book.getAuthor());                         //输出作者
        System.out.println("价格：" + book.getPrice() + "元");                    //输出价格
    }
}
```

7.2.11 范例 2：温度单位转换工具

目前有两种广泛使用的温度单位，即摄氏度和华氏度。在标准大气压下，沸腾的水可以表示成 100 摄氏度和 212 华氏度。本范例将编写一个简单的温度单位转换工具，它可以将用户输入的摄氏度转换成华氏度，其运行结果如图 7.14 所示。（**实例位置：光盘\TM\sl\7\2**）

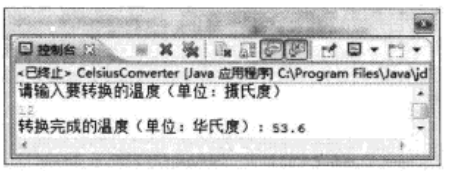

图 7.14 将输入的摄氏温度转换成华氏温度

在项目中创建 CelsiusConverter 类，在类中定义了两个方法，getFahrenheit()方法用于将摄氏温度转换成华氏温度，main()方法用于测试。代码如下：

```
public class CelsiusConverter {
    public double getFahrenheit(double celsius) {
        double fahrenheit = 1.8 * celsius + 32;                                  //计算华氏温度
        return fahrenheit;                                                       //返回华氏温度
    }
    public static void main(String[] args) {
        System.out.println("请输入要转换的温度（单位：摄氏度）");
        Scanner in = new Scanner(System.in);                                     //获得控制台输入
        double celsius = in.nextDouble();                                        //获得用户输入的摄氏温度
        CelsiusConverter converter = new CelsiusConverter();                     //创建类的对象
        double fahrenheit = converter.getFahrenheit(celsius);                    //转换温度为华氏度
        System.out.println("转换完成的温度（单位：华氏度）：" + fahrenheit);      //输出转换结果
    }
}
```

7.3 对 象

视频讲解：光盘\TM\lx\7\对象.exe

Java 是一门面向对象的程序设计语言，对象是由类抽象出来的，所有的问题都是通过对象来处理，

对象可以操作类的属性和方法解决相应的问题，所以了解对象的产生、操作和消亡对学习 Java 语言是十分必要的。本节就来讲解对象在 Java 语言中的应用。

7.3.1 对象的创建

在 7.1 节中曾经介绍过对象，对象可以认为是在一类事物中抽象出某一个特例，通过这个特例来处理这类事物出现的问题，在 Java 语言中通过 new 操作符来创建对象。以前曾经在讲解构造方法中介绍过每实例化一个对象就会自动调用一次构造方法，实质上这个过程就是创建对象的过程。准确地说，可以在 Java 语言中使用 new 操作符调用构造方法创建对象。语法格式如下：

```
Test test=new Test();
Test test=new Test("a");
```

其中参数说明如表 7.2 所示。

表 7.2 创建对象语法中的参数说明

设 置 值	描 述
Test	类名
test	创建 Test 类对象
new	创建对象操作符
"a"	构造方法的参数

test 对象被创建出来时，test 对象就是一个对象的引用，这个引用在内存中为对象分配了存储空间，可以在构造方法中初始化成员变量，当创建对象时，自动调用构造方法，也就是说在 Java 语言中初始化与创建是被捆绑在一起的。

每个对象都是相互独立的，在内存中占据独立的内存地址，并且每个对象都具有自己的生命周期，当一个对象的生命周期结束时，对象变成了垃圾，由 Java 虚拟机自带的垃圾回收机制处理，不能再被使用（对于垃圾回收机制的知识会在 7.3.5 小节中进行介绍）。

注意

在 Java 语言中对象和实例事实上可以通用。

下面来看一个创建对象的实例。

【例 7.11】 在项目中创建 CreateObject 类，在该类中创建对象并在主方法中创建对象。（实例位置：光盘\TM\sl\7\3）

```
public class CreateObject {
    public CreateObject(){                          //构造方法
        System.out.println("创建对象");
    }
    public static void main(String args[]){         //主方法
```

```
        new CreateObject();                                    //创建对象
    }
}
```

运行结果如图 7.15 所示。

在上述实例的主方法中使用 new 操作符创建对象，在创建对象的同时，自动调用构造方法中的代码。

图 7.15　创建对象

7.3.2　访问对象的属性和行为

当用户使用 new 操作符创建一个对象后，可以使用“对象.类成员”来获取对象的属性和行为。前文已经提到过，对象的属性和行为在类中是通过类成员变量和成员方法的形式来表示的，所以当对象获取类成员，也就相应地获取了对象的属性和行为。

【例 7.12】　在项目中创建 TransferProperty 类，该类中说明对象是如何调用类成员的。（**实例位置：光盘\TM\sl\7\4**）

```
public class TransferProperty {
    int i=47;                                                  //定义成员变量
    public void call(){                                        //定义成员方法
        System.out.println("调用 call()方法");
        for(i=0;i<3;i++){
            System.out.print(i+" ");
            if(i==2){
                System.out.println("\n");
            }
        }
    }
    public TransferProperty(){                                 //定义构造方法
    }
    public static void main(String[] args) {
        TransferProperty t1=new TransferProperty();            //创建一个对象
        TransferProperty t2=new TransferProperty();            //创建另一个对象
        t2.i=60;                                               //将类成员变量赋值为 60
        //使用第一个对象调用类成员变量
        System.out.println("第一个实例对象调用变量 i 的结果："+t1.i++);
        t1.call();                                             //使用第一个对象调用类成员方法
        //使用第二个对象调用类成员变量
        System.out.println("第二个实例对象调用变量 i 的结果："+t2.i);
        t2.call();                                             //使用第二个对象调用类成员方法
    }
}
```

运行结果如图 7.16 所示。

在上述代码的主方法中首先实例化一个对象，然后使用“.”操作符调用类的成员变量和成员方法。

但是在运行结果中可以看到，虽然使用两个对象调用同一个成员变量，结果却不相同，因为在打印这个成员变量的值之前将该值重新赋值为 60，但在赋值时使用的是第二个对象 t2 调用成员变量，所以在第一个对象 t1 调用成员变量打印该值时仍然是成员变量的初始值，由此可见，两个对象的产生是相互独立的，改变了 t2 的 i 值，不会影响到 t1 的 i 值。在内存中这两个对象的布局如图 7.17 所示。

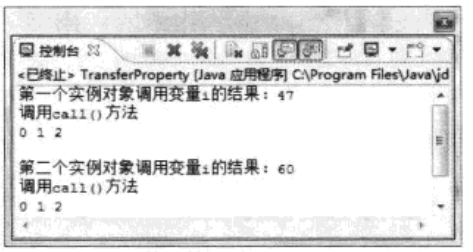

图 7.16 使用对象调用类成员

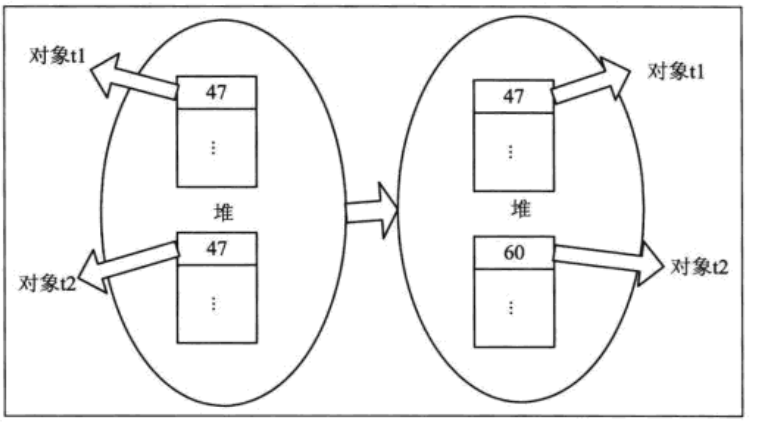

图 7.17 内存中 t1、t2 两个对象的布局

如果希望成员变量不被其中任何一个对象改变，可以使用 static 关键字（前文曾经介绍过一个被声明为 static 的成员变量的值可以被本类或其他类的对象共享）。可以将上述代码改写为以下形式。

【例 7.13】 在项目中创建 CopyOfTransferProperty 类，该类举例说明对象调用静态成员变量。（**实例位置：光盘\TM\sl\7\5**）

```
public class CopyOfTransferProperty {
    static int i=47;                                          //定义静态成员变量
    public void call(){                                       //定义成员方法
        System.out.println("调用 call()方法");
        for(i=0;i<3;i++){
            System.out.print(i+" ");
            if(i==2){
                System.out.println("\n");
            }
        }
    }
    public CopyOfTransferProperty(){                          //定义构造方法
    }
    public static void main(String[] args) {                  //定义主方法
        CopyOfTransferProperty t1=new CopyOfTransferProperty();  //创建一个对象
        CopyOfTransferProperty t2=new CopyOfTransferProperty();  //创建另一个对象
        t2.i=60;                                              //将类成员变量赋值为 60
        //使用第一个对象调用类成员变量
        System.out.println("第一个实例对象调用变量 i 的结果："+t1.i++);
        t1.call();                                            //使用第一个对象调用类成员方法
        //使用第二个对象调用类成员变量
        System.out.println("第二个实例对象调用变量 i 的结果："+t2.i);
```

```
        t2.call();                                          //使用第二个对象调用类成员方法
    }
}
```

运行结果如图 7.18 所示。

从上述运行结果中可以看到，由于使用“t2.i=60;”语句改变了静态成员变量的值，使用对象 t1 调用成员变量的值也为 60，这正是 i 值被定义为静态成员变量的功效，即使使用两个对象对同一个静态成员变量进行操作，依然可以改变静态成员变量的值，因为在内存中两个对象同时指向同一块内存区域。“t1.i++;”语句执行完毕后，i 值变为 3。当再次调用 call()方法时又被重新赋值为 0，做循环打印操作。

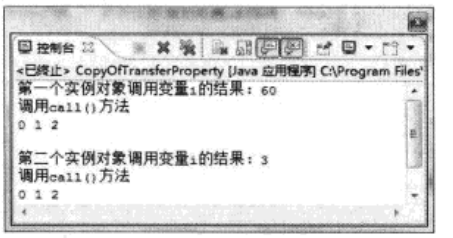

图 7.18　对象调用静态成员变量运行结果

7.3.3　对象的引用

在 Java 语言中尽管一切都可以看作对象，但真正操作标识符实质上是一个引用，那么引用究竟在 Java 语言中是如何体现的？来看下面的语法。

语法格式如下：

```
类名 对象引用名称
```

例如一个 Book 类的引用可以使用的代码如下：

```
Book book;
```

通常一个引用不一定需要有一个对象相关联，引用与对象相关联的语法如下：

```
Book book=new Book();
```

- ☑ Book：类名。
- ☑ book：对象。
- ☑ new：创建对象操作符。

注意　引用只是存放一个对象的内存地址，并非存放一个对象，严格说引用和对象是不同的，但是可以将这种区别忽略，如可以简单地说 book 是 Book 类的一个对象，而事实上应该说 book 是包含 Book 对象的一个引用。

7.3.4　对象的比较

在 Java 语言中有两种对象的比较方式，分别为“==”运算符与 equals()方法。实质上这两种方式有着本质区别，下面举例说明。

【例 7.14】 在项目中创建 Compare 类，该类说明了“==”运算符与 equals()方法的区别。（实例位置：光盘\TM\sl\7\6）

```
public class Compare {
    public static void main(String[] args) {
        String c1=new String("abc");                                    //创建两个 String 型对象引用
        String c2=new String("abc");
        String c3=c1;                                                   //将 c1 对象引用赋予 c3
        System.out.println("c2==c3 的运算结果为："+(c2==c3));              //使用“==”运算符比较 c2 与 c3
        System.out.println("c2.equals(c3)的运算结果为："+(c2.equals(c3))); //使用 equals()方法比较 c2 与 c3
    }
}
```

运行结果如图 7.19 所示。

从上述运行结果中可以看出，“==”运算符和 equals()方法比较的内容是不相同的。equals()方法是 String 类中的方法，它用于比较两个对象引用所指的内容是否相等；而“==”运算符比较的是两个对象引用的地址是否相等。由于 c1 与 c2 是两个不同的对象引用，两者在内存中的位置不同，而“String c3=c1;”语句将 c1 的引用赋给 c3，所以 c1 与 c3 这两个对象引用是相等的，也就是打印 c1==c3 这样的语句将返回 true 值。对象 c1、c2、c3 在内存中的布局如图 7.20 所示。

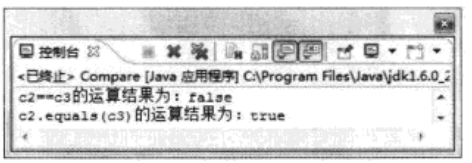

图 7.19 比较两个对象引用

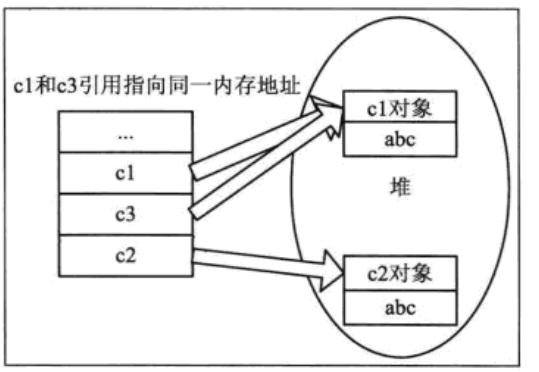

图 7.20 对象 c1、c2、c3 在内存中的布局

7.3.5 对象的销毁

每个对象都有生命周期，当对象的生命周期结束时，分配给该对象的内存地址将会被回收，在其他语言中需要手动回收废弃的对象，但是 Java 语言拥有一套完整的垃圾回收机制，用户不必担心废弃的对象占用内存，垃圾回收器将回收无用的占用内存的资源。

在谈到垃圾回收机制之前，读者首先需要了解何种对象会被 Java 虚拟机视为垃圾。主要包括以下两种情况：

（1）对象引用超过其作用范围，则这个对象将被视为垃圾，如图 7.21 所示。

（2）将对象赋值为 null，如图 7.22 所示。

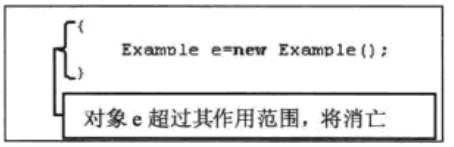

图 7.21　对象超过作用范围将消亡

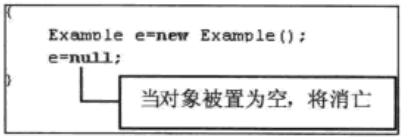

图 7.22　对象被置为 null 值时将消亡

虽然垃圾回收机制已经很完善，但垃圾回收器只能回收那些由 new 操作符创建的对象，如果某些对象不是通过 new 操作符在内存中获取一块内存区域，这种对象可能不被垃圾回收机制所识别，所以在 Java 语言中提供了一个 finalize()方法，这个方法是 Object 类的方法，它被声明为 protected，用户可以在自己的类中定义这个方法，如果用户在类中定义了 finalize()方法，在垃圾回收时首先调用该方法，并且在下一次垃圾回收动作发生时，才能真正回收对象占用的内存。

说明　垃圾回收或是 finalize()方法不保证一定会发生，如 Java 虚拟机面临内存损耗待尽的情形，它是不会执行垃圾回收的。

由于垃圾回收不受人为控制，具体执行时间也不确定，所以 finalize()方法也就无法执行，为此，Java 提供了 System.gc()方法强制启动垃圾回收器，这与给 120 打电话通知医院来救护的道理一样，告知垃圾回收器来清理。

7.3.6　范例 3：统计图书销量

在商品（类的实例）的销售过程中，需要对销量进行统计。本范例将在类的构造方法增加计数器来实现该功能，其运行结果如图 7.23 所示。（**实例位置：光盘\TM\sl\7\7**）

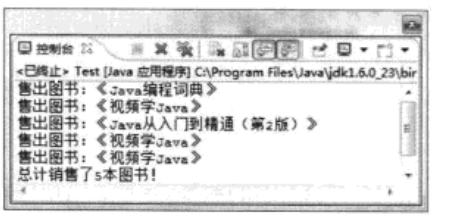

图 7.23　输出图书销售信息

（1）在项目中创建 Book 类，在类中定义了一个静态的成员变量用于保存实例化的次数。在构造方法中实现了计数器的功能，其代码如下：

```
public class Book {
    private static int counter = 0;                          //定义一个计数器
    public Book(String title) {
        System.out.println("售出图书：" + title);             //输出书名
        counter++;                                           //计数器加 1
    }
    public static int getCounter() {                         //获得计数器的结果
        return counter;
    }
}
```

（2）在项目中创建 Test 类，在类的 main()方法中创建 Book 类对象并输出创建对象的个数，代码如下：

```
public class Test {
    public static void main(String[] args) {
        //创建书名数组
        String[] titles = { "《Java 从入门到精通（第 2 版）》", "《Java 编程词典》", "《视频学 Java》" };
        for (int i = 0; i < 5; i++) {
            new Book(titles[new Random().nextInt(3)]);                    //利用书名数组创建 Book 对象
        }
        System.out.println("总计销售了" + Book.getCounter() + "本图书！ ");   //输出创建对象的个数
    }
}
```

7.3.7 范例 4：重新计算对象的哈希码

Java 语言中创建的对象是保存在堆中的，为了提高查找的速度而使用了散列查找。散列查找的基本思想是定义一个键来映射对象所在的内存地址。当需要查找对象时，直接查找键就可以了，这样就不用遍历整个堆来查找对象了。本范例将查看不同对象的散列值。运行结果如图 7.24 所示。（**实例位置：光盘\TM\sl\7\8**）

（1）在项目中创建 Cat 类，在类中定义 4 个成员变量分别表示猫咪的名字、年龄、重量和颜色，并提供构造方法来设置这些属性值。本类的重点内容在于重写 equals()方法和 hashCode()方法。重写 equals()方法可以比较两个对象是否相同，重写 hashCode()方法可以让相同的对象保存在相同的位置。代码如下：

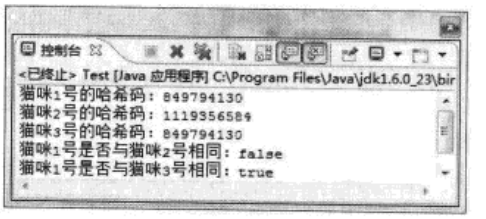

图 7.24 输出对象哈希码和比较结果

```
public class Cat {
    private String name;                                                  //表示猫咪的名字
    private int age;                                                      //表示猫咪的年龄
    private double weight;                                                //表示猫咪的重量
    private Color color;                                                  //表示猫咪的颜色
    public Cat(String name, int age, double weight, Color color) {       //初始化猫咪的属性
        this.name = name;
        this.age = age;
        this.weight = weight;
        this.color = color;
    }
    @Override
    public boolean equals(Object obj) {                                   //利用属性来判断猫咪是否相同
        if (this == obj) {                                                //如果两个猫咪是同一个对象则相同
            return true;
        }
        if (obj == null) {                                                //如果两个猫咪有一个为 null 则不同
            return false;
        }
```

```
        if (getClass() != obj.getClass()) {                                    //如果两个猫咪的类型不同则不同
            return false;
        }
        Cat cat = (Cat) obj;
        return name.equals(cat.name) && (age == cat.age)
                && (weight == cat.weight) && (color.equals(cat.color)); //比较猫咪的属性
    }
    @Override
    public int hashCode() {                                                    //重写 hashCode()方法
        return 7 * name.hashCode() + 11 * new Integer(age).hashCode() + 13
                * new Double(weight).hashCode() + 17 * color.hashCode();
    }
}
```

（2）在 com.mingrisoft 包中创建类文件，名称为 Test。在该类的 main()方法中创建了 3 只猫咪，并为其初始化，然后输出猫咪的哈希码和比较的结果。代码如下：

```
public class Test {
    public static void main(String[] args) {
        Cat cat1 = new Cat("Java", 12, 21, Color.BLACK);                       //创建猫咪 1 号
        Cat cat2 = new Cat("C++", 12, 21, Color.WHITE);                        //创建猫咪 2 号
        Cat cat3 = new Cat("Java", 12, 21, Color.BLACK);                       //创建猫咪 3 号
        System.out.println("猫咪 1 号的哈希码：" + cat1.hashCode());            //输出猫咪 1 号的哈希码
        System.out.println("猫咪 2 号的哈希码：" + cat2.hashCode());            //输出猫咪 2 号的哈希码
        System.out.println("猫咪 3 号的哈希码：" + cat3.hashCode());            //输出猫咪 3 号的哈希码
        System.out.println("猫咪 1 号是否与猫咪 2 号相同：" + cat1.equals(cat2)); //比较是否相同
        System.out.println("猫咪 1 号是否与猫咪 3 号相同：" + cat1.equals(cat3)); //比较是否相同
    }
}
```

7.4 经典范例

7.4.1 经典范例 1：汉诺塔问题求解

视频讲解：光盘\TM\lx\7\汉诺塔问题求解.exe

汉诺塔问题的描述如下：有 A、B 和 C 3 根柱子，在 A 上从下往上按照从小到大的顺序放着 64 个圆盘，以 B 为中介，把盘子全部移动到 C 上。移动过程中，要求任意盘子的下面要么没有盘子，要么只能有比它大的盘子。本范例将演示如何求解 3 阶汉诺塔问题。运行结果如图 7.25 所示。（**实例位置：光盘\TM\sl\7\9**）

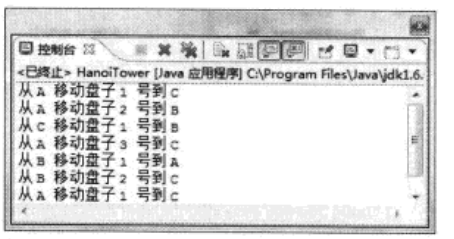

图 7.25 三阶汉诺塔解决步骤

在项目中创建 HanoiTower 类，在类中定义了一个 moveDish()方法，它使用递归算法完成汉诺塔问题的求解；一个 main()方法用于测试。代码如下：

```
public class HanoiTower {
    public static void moveDish(int level, char from, char inter, char to) {
        if (level == 1) {                                          //如果只有一个盘子就退出迭代
            System.out.println("从 " + from + " 移动盘子 1 号到 " + to);
        } else {                                                   //如果有大于一个盘子就继续迭代
            moveDish(level - 1, from, to, inter);
            System.out.println("从 " + from + " 移动盘子 " + level + " 号到 " + to);
            moveDish(level - 1, inter, from, to);
        }
    }
    public static void main(String[] args) {
        int nDisks = 3;                                            //设置汉诺塔为三阶
        moveDish(nDisks, 'A', 'B', 'C');                           //实现移动算法
    }
}
```

7.4.2 经典范例 2：单例模式的应用

视频讲解：光盘\TM\lx\7\单例模式的应用.exe

中国历史上的皇帝通常仅有一人。为了保障其唯一性，古人采用增加“防伪标识”的办法，如玉玺。更简单的方法是限制皇帝的创建。本范例使用单例模式来保证皇帝的唯一性。运行结果如图 7.26 所示。（**实例位置：光盘\TM\ sl\7\10**）

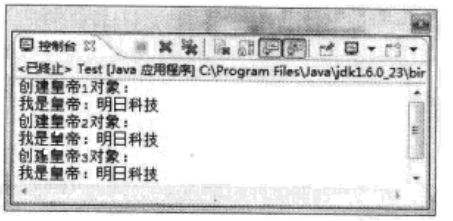

图 7.26 使用单例模式保证了皇帝的唯一性

（1）在项目中创建 Emperor 类，在类中将构造方法设置成私有，并提供了一个静态方法用于获得该类的实例。代码如下：

```
public class Emperor {
    private static Emperor emperor = null;                         //声明一个 Emperor 类的引用
    private Emperor() {                                            //将构造方法私有
    }
    public static Emperor getInstance() {                          //实例化引用
        if (emperor == null) {
            emperor = new Emperor();
        }
        return emperor;
    }
    public void getName() {                                        //使用普通方法输出皇帝的名字
        System.out.println("我是皇帝：明日科技");
    }
}
```

（2）在项目中创建 Test 类，在类的 main()方法中创建了 3 个 Emperor 对象并输出了其名字。代码如下：

```
public class Test {
    public static void main(String[] args) {
        System.out.println("创建皇帝 1 对象：");
        Emperor emperor1 = Emperor.getInstance();          //创建皇帝对象
        emperor1.getName();                                //输出皇帝的名字
        System.out.println("创建皇帝 2 对象：");
        Emperor emperor2 = Emperor.getInstance();          //创建皇帝对象
        emperor2.getName();                                //输出皇帝的名字
        System.out.println("创建皇帝 3 对象：");
        Emperor emperor3 = Emperor.getInstance();          //创建皇帝对象
        emperor3.getName();                                //输出皇帝的名字
    }
}
```

7.5 本章小结

通过本章的学习，读者应该掌握面向对象的编程思想，这样对 Java 的学习十分有帮助，同时在此基础上读者可以编写类，定义类成员、构造方法、主方法以解决一些实际问题，类以及类成员可以使用权限修饰符进行修饰，读者应该了解这些修饰符的具体范围。由于在 Java 中通过对象来处理问题，所以对象的创建、比较、销毁的应用就显得非常重要。作为初学者应该反复揣摩这些基本概念和面向对象的编程思想，为以后 Java 语言的学习打下坚实的基础。

7.6 实战练习

1．尝试编写一个类，定义一个修饰权限为 private 的成员变量，定义两个成员方法，一个方法实现为此成员变量赋值，另一个成员方法获取这个成员变量的值，保证其他类继承该类时能获取该类的成员变量的值。（**答案位置：光盘\TM\sl\7\11**）

2．尝试编写一个矩形类，将长与宽作为矩形类的属性，在构造方法中将长、宽初始化，定义一个成员方法求此矩形的面积。（**答案位置：光盘\TM\sl\7\12**）

3．根据运行参数的个数决定循环打印变量 i 值的次数。（**答案位置：光盘\TM\sl\7\13**）

技术篇

- 第 8 章　接口、继承与多态
- 第 9 章　类的高级特性
- 第 10 章　Java 集合类
- 第 11 章　异常处理
- 第 12 章　输入/输出
- 第 13 章　Swing 程序设计

本篇介绍了接口、继承与多态，类的高级特性，Java 集合类，异常处理，输入/输出，Swing 程序设计等。学习完这一部分后，能够开发一些小型应用程序。