第3章 类和对象

面向对象程序设计是软件系统设计与实现的新方法，这种新方法是通过增加软件的可扩充性和可重用性来改善并提高程序员的生产能力，并控制软件的复杂性和软件维护的开销，它的应用使软件开发的难度和费用大幅度降低，已为世界软件产业带来革命性的突破。

在面向过程程序设计中，数据和对数据的操作是分离的，因为对数据的操作需要把数据传递到特定的函数或过程中。而面向对象程序设计把数据和对数据的操作放在一个独立的实体当中，称其为对象(object)。客观世界中的每一个具体的事物或抽象的事件均可认为是对象。

3.1 概述

3.1.1 对象

对象是现实世界中一个实际存在的事物，它可以是有形的(如一支粉笔、一块橡皮等)，也可以是无形的或无法整体触及的抽象事件(如一项计划、一场球赛、一次借书等)。对象是构成世界的一个独立单位，它具有自己的静态特征和动态特征。静态特征可以用某种数据来描述，动态特征是对象所表现的行为或对象所具有的功能。

在面向对象系统中，对象是系统中用来描述客观事物的一个实体，它是构成系统的一个基本单位。一个对象由一组属性和对这组属性进行操作的一组服务构成。属性和服务是构成对象的两个主要因素：属性是一组数据结构的集合，表示对象的一种状态，对象的状态只供对象自身使用，用来描述静态特性；而服务是用来描述对象动态特征(行为)的一个操作序列，是对象一组功能的体现。

一个对象可以包含多个属性和多个服务，对象的属性值只能由这个对象的服务存取。对象是其自身所具有的状态特征及可以对这些状态施加的操作结合在一起所构成的独立实体，具有如下的主要特性：

1. 对象标识

对象标识即对象的名字，是用户和系统识别它的唯一标志。对象标识有“外部标识”和“内部标识”之分。外部标识供对象的定义者或使用者用，内部标识供系统内部唯一地识别每一个对象。

2. 属性

属性即一组数据，用来描述对象的静态特征。在 Java 程序中，把这一组数据称为数据成员。

3. 方法

方法也称为服务或操作，它是对象动态特征（行为）的描述。每一个方法确定对象的一种行为或功能，方法也称为成员方法或方法成员。

4. 模块独立性

从逻辑上看，一个对象是独立存在的模块。模块内部状态不因外界的干扰而改变，也不会涉及到其他模块；模块间的依赖性极小或几乎没有；各模块可独立地被系统组合选用，也可被程序员重用，不必担心破坏其他模块。

5. 动态连接性

客观世界中的对象之间是有联系的，在面向对象程序设计中，通过消息机制，把对象之间动态连接在一起，使整个机体运转起来，这称为对象的连接性。

6. 易维护性

由于对象的修改、完善功能及其实现的细节都被局限于该对象的内部，不会涉及到外部，这就使得对象和整个系统容易维护。

对象从形式上看是系统程序员、应用程序员或用户所定义的抽象数据类型的变量，当用户定义了一个对象，就创造出了具有丰富内涵的新的抽象数据类型的实体。

3.1.2 类

在面向对象的编程语言中，类（class）是一个独立的程序单位，是对具有相同属性和方法的一组对象的抽象。通过类可以对属于该类的全部对象进行统一的描述。因此，在定义对象之前应先定义类。描述一个类需要指明下述三个方面。

（1）类标识（即类名）：类的一个有别于其他类的名字，这是必不可少的。

（2）属性说明：用来描述相同对象的静态特征。

（3）类的方法：用来描述相同对象的动态特征。在面向对象系统中，并不是将各个具体的对象都进行描述，而是忽略其非本质的特性，找出其共性，将对象划分成不同的类，这一过程为抽象过程。

在 Java 程序中，类是创建对象的模板，对象是类的实例，任何一个对象都是隶属于某个类的。

3.1.3 消息

消息（message）是面向对象系统中实现对象间的通信和请求任务的操作，是要求某个

对象执行其中某个功能操作的规格说明。发送消息的对象称为发送者，接收消息的对象称为接收者。对象间的联系，只能通过消息来进行。对象在接收到消息时才被激活。

消息具有三个性质：

(1) 同一对象可接收不同形式的多个消息，产生不同的响应。

(2) 相同形式的消息可以送给不同对象，所做出的响应可以是截然不同的。

(3) 消息的发送可以不考虑具体的接收者，对象可以响应消息，也可以对消息不予理会，对消息的响应并不是必需的。

对象之间传送的消息一般由三部分组成：接收对象名、调用操作名和必要的参数。

3.1.4 面向对象系统的特性

1. 抽象性(abstract)

面向对象鼓励程序员以抽象的观点看待程序，可以将一组对象的共同特征进一步抽象出来，从而形成“类”的概念。抽象是一种从一般的观点看待事物的方法，它要求程序员集中于事物的本质特征，而不是具体细节或具体实现。类的概念来自人们认识自然、认识社会的过程。在这一过程中，人们主要使用两种方法：从特殊到一般的归纳法和从一般到特殊的演绎法。在归纳的过程中，从一个个具体的事物中把共同的特征抽取出来，形成一个一般的概念，这就是“归类”；在演绎的过程中，又把同类的事物根据不同的特征分成不同的小类，这就是“分类”。对于一个具体的类，它有许多具体的个体，这些个体称为“对象”。

2. 封装性(encapsulation)

所谓数据封装，就是指一组数据和与这组数据有关的操作组装在一起，形成一个能动的实体，也就是对象。数据封装给数据提供了与外界联系的标准接口，无论是谁，只有通过这些接口，使用规范的方式，才能访问这些数据。数据封装是软件工程发展的必然产物，使得程序员在设计程序时可以专注于自己的对象，同时也切断了不同模块之间数据的非法使用，减少了出错的可能性。

3. 继承性(inheritance)

从已有的对象类型出发建立一种新的对象类型，使它继承原对象的特点和功能，这种思想是面向对象设计方法的主要贡献。继承是对许多问题中分层特性的一种自然描述，因而也是类的具体化和被重新利用的一种手段，它所表达的就是一种对象类之间的相交关系。它使得某类对象可以继承另外一类对象的特征和能力。继承所具有的作用有两个方面：一方面可以减少代码冗余，另一方面可以通过协调性来减少相互之间的接口和界面。

从继承源上划分继承可分为单一继承(单继承)和多重继承(多继承)。子类对单个直接父类的继承称为单继承。子类对多于一个的直接父类的继承称为多重继承。父类也称为基类或超类，子类也称为派生类。单一继承如图 3.1 所示，多重继承如图 3.2 所示。

Java 中只支持类之间的单一继承，多重继承要通过接口来实现。

4. 多态性(polymorphism)

Java 支持两种多态性，即编译时的多态性和运行时的多态性。编译时的多态性通过方

法重载实现，而运行时的多态性通过类之间的继承、方法覆盖以及动态联编技术实现。多态性使程序具有良好的可扩展性，并使程序易于编写、易于理解。使用多态性可以大大提高程序员解决复杂问题的能力。

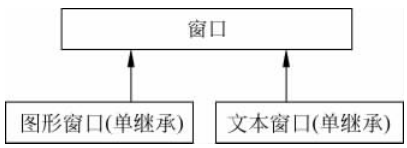

图 3.1 单一继承举例

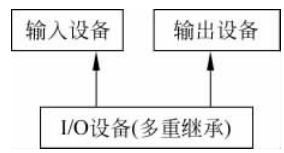

图 3.2 多重继承举例

3.2 类

一个 Java 源程序文件往往是由许多个类组成的。从用户的角度看，Java 源程序中的类分为两种：

(1) 系统定义的类，即 Java 类库。它是系统定义好的类。类库是 Java 的重要组成部分。Java 由语法规则和类库两部分组成，语法规则确定 Java 程序的书写规范；类库则提供了 Java 程序与运行它的系统软件(Java 虚拟机)之间的接口。Java 类库是一组由它的发明者 Sun 公司以及其他软件开发商编写好的 Java 程序模块，每个模块通常对应一种特定的基本功能和任务，且这些模块都是经过严格测试的，因而也总是正确有效的。

(2) 用户自己定义的类。系统定义的类虽然实现了许多常见的功能，但是用户程序仍然需要针对特定问题的特定逻辑来定义自己的类。

Java 中的类包括成员变量和成员方法两部分，成员变量又称为数据成员。类的成员变量可以是基本数据类型的数据或数组，也可以是一个类的实例；成员方法用于处理该类的成员变量。成员方法与其他语言中函数的区别在于：Java 语言中的成员方法只能是类的成员，只能在类中定义；调用一个类的成员方法，实际上是进行对象之间或用户与对象之间的消息传递。

3.2.1 类的定义

类的定义格式如下：

```
[类修饰符] class 类名 [extends 父类名称][implements 接口名称列表]{
    [成员变量定义]
    [成员方法声明或定义]
}
```

1. 类修饰符

在类的定义中，中括号[]中的内容可以省略，有关[extends 父类名称]和[implements 接口名称列表]选项的具体内容参见第 5 章。类修饰符为：

```
[public] [abstract | final]
```

其含义如表 3.1 所示。

表 3.1 类修饰符

修饰符	含 义
public	公共访问控制符。可以在任何类中访问该公共类
缺省	默认访问控制符。缺省修饰符时(即不加 public 访问控制修饰符),也称为包访问或默认访问权限,表示可被当前包中的其他类所访问
abstract	抽象类,不能创建抽象类的实例
final	最终类,最终类不能被其他类所继承

注意:

(1) 一个源文件仅可以有一个 public 类,且与文件同名。

(2) 如果一个类不加 public 修饰符,则该类为友好类,具有包访问属性。

(3) abstract 和 final 不同时修饰一个类。

(4) 不能用 private 和 protected 修饰符来修饰类。

2. 成员变量

当一个变量的定义出现在类定义体中而并不属于任何一个方法,则该变量称为所属类的成员变量,又称为数据成员。类成员变量的定义与一般变量的定义一样,必须包括数据类型与变量名,但增加了可选的变量修饰符。成员变量定义格式如下:

```
[成员变量修饰符]  变量数据类型  变量名 1,变量名 2[ = 变量初值]…;
```

成员变量修饰符为:

```
[public | protected | private] [static] [final] [transient] [volatile]
```

其含义如表 3.2 所示。

表 3.2 成员变量修饰符

修饰符	含 义
public	公共访问控制符。可以被所有的类访问
protected	保护访问控制符。可以被该类自身、派生类、同一包中的其他类所访问
缺省	默认访问控制符。不加 public、protected 和 private 访问控制修饰符,为默认修饰符。表示可被当前包中的类所访问,而其他包中的类不能访问该成员变量
private	私有访问控制符。仅可被该类自身访问,不能被其他任何类(包括派生类)访问
static	静态变量或类变量,该成员变量可被所有对象共享,可通过类名直接访问类变量
final	最终修饰符。常量,定义的同时应对其进行初始化,并且不能再改变常量的值
transient	与对象序列化有关。用来声明一个暂时性变量,表示该变量并不属于对象的永久状态,从而不能被永久存储
volatile	异步控制修饰符。表示多个并发线程共享的变量,这使得各线程对该变量的访问保持一致

注意：

(1) public、protected、private 说明了对该成员变量的访问控制权限，根据访问权限的级别排列，按访问权限从高到低的排列顺序是 public、protected、缺省的、private；static 修饰的变量称为静态变量或类变量，是该类的所有对象所共享的变量，而没有 static 修饰的变量为实例变量。

(2) 在定义类的成员变量时，可以同时赋初值，表明成员变量的初始状态，但对成员变量的操作只能放在成员方法中。

3. 成员方法

成员方法的定义包括两部分，分别是方法头部和方法体。方法头部主要包括方法修饰符、方法返回值类型、方法名称、一对小括号及形式参数(简称形参)列表等。方法体是用一对大括号括起来的语句序列。成员方法的定义格式如下：

```
[方法修饰符] 返回值类型  方法名称([参数 1,参数 2,…]) [throws 异常列表]{
    [语句序列;]    //方法体：实现特定的功能
}
```

成员方法修饰符为：

```
[public | protected | private] [static] [final | abstract] [native]
[synchronized]
```

其含义如表 3.3 所示。

表 3.3 成员方法修饰符

修饰符	含　义
public	可以被所有的类访问
protected	可以被该类自身、派生类、同一包中的其他类所访问
缺省	表示可被当前包中的类访问
private	仅可被该类自身访问，不能被其他任何类(包括派生类)访问
static	静态方法或类方法，不需要实例化对象就可以通过类名直接访问的方法，如 main 方法
final	最终方法，该方法不能被派生类所覆盖
abstract	抽象方法，只有方法声明而没有方法体，要在派生类中进行具体的实现
native	本地方法，此方法的方法体是用其他语言在程序外部编写的
synchronized	同步方法，在多线程程序设计中，用于对方法加锁，运行结束后解锁，以防止其他线程访问

注意：

(1) public、protected、private 说明了对该成员方法的访问控制权限。

(2) static 修饰的方法为类方法，而没有 static 修饰的方法为实例方法，通过类名不能直接调用实例方法，而经常通过类对象调用实例方法。

(3) 实例方法既能对类变量操作，也能对实例变量操作；类方法只能对类变量进行操作。

(4) 一个类中的实例方法可以互相调用,实例方法也可以调用该类中的类方法;类方法只能调用该类的类方法,不能调用实例方法。

进行方法调用时使用的参数称为实际参数(简称为实参)。如果方法没有形参,则该方法称为无参方法。方法的返回值可以为任意的 Java 数据类型,如果方法不返回任何值,则返回类型为 void。

方法体是对方法的具体实现。方法体中定义的变量和形式参数称为局部变量。局部变量的作用域在该方法内部,当方法返回时,局部变量不再存在。而成员变量在整个类内都有效,局部变量可以和类的成员变量同名,若同名,则类的成员变量在方法体内被隐藏,若想访问被隐藏的成员变量,则使用 this 关键字(this 关键字的使用请参见 3.5 节),参见下列 Circle 类中的 SetRadius 方法定义。

注意:方法的形参列表应分别给出每个参数的数据类型,如形参列表:(int a, int b)不能写成(int a, b)。而局部变量定义语句:

```
int a; int b;
```

等价于:

```
int a, b;
```

下列程序代码段定义了圆类(Circle 类),请注意 Circle 类中的 setRadius 方法定义。

```
//Circle.Java
public class Circle {
    final double PI = 3.14;                    //常量 PI
    private double radius = 1;                 //私有成员变量 radius,表示圆的半径
    private double area;                       //私有成员变量 area,表示圆的面积
    private static int numberOfObjects = 0;    //类变量,表示创建的圆的个数

    public Circle() {                          //以默认半径 1 创建一个圆,圆的个数加 1
        area = PI * radius * radius;           //设置圆的面积
        numberOfObjects++;
    }

    public Circle(double radius) {
        //以半径 radius(指形参 radius)创建一个圆,圆的个数加 1
        this.radius = radius;
        area = PI * radius * radius;           //设置圆的面积
        numberOfObjects++;
    }

    public double getRadius() {                //获取圆的半径
        return radius;
    }

    public void setRadius(double radius) {     //设置圆的半径
        this.radius = (radius >= 0) ? radius : 0;
        area = PI * radius * radius;           //修改圆的面积
    }
```

```
    public static int getNumberOfObjects() { //获取圆的个数
        return numberOfObjects;
    }

    public double getArea() {                    //获取圆的面积
        return area;
    }
}
```

【程序解析】

√ 该类中定义了私有成员变量 radius，而方法 setRadius 中的局部变量 radius 与其同名，在 setRadius 方法体内成员变量 radius 被隐藏，所以使用 this 关键字访问成员变量。setRadius 方法的功能是：若形参 radius 的值大于等于 0，则把形参 radius 的值赋给成员变量 radius，否则给成员变量 radius 赋值为 0。

√ 该类的定义中没有使用 setArea()方法设置圆的面积，而是在创建一个圆时就设置了圆的面积，符合自然规律。当使用 setRadius 方法重新设置圆的半径时，相应地修改圆的面积。

3.2.2 方法重载(method overloading)

如果几个方法的方法名相同，只是方法形参个数或形参类型有所不同，则称这几个方法为重载方法。

注意：*如果几个方法的方法名、方法的形参个数和形参类型均相同，只是方法的返回值类型不同，则不称其为重载方法，编译时会出错。*

如果几个方法的功能非常相似，则把它们定义为重载方法，如求两个整数的最大值、求三个整数的最大值、求两个 float 数的最大值等。Java 编译器会根据方法名和实参的个数及类型决定调用最合适的一个方法。上述圆的定义中定义了两个 Circle 方法，它们是重载方法。

使用重载方法可以使程序清晰，并增强可读性。但在使用重载方法时一定要注意避免二义性。例如有如下的两个重载方法说明：

```
int max(int num1, double num2);
double max(double num1, int num2);
```

当使用 max(1, 4)方法调用时，编译器不能确定调用哪一个方法更合适，这称为方法调用的二义性，有二义性的方法调用会引起编译时的错误。

方法重载是面向对象程序设计语言多态性的一种形式，它实现了 Java 编译时的多态性，即由编译器在编译时确定具体调用哪个被重载的方法。

3.2.3 构造方法

方法名和所在类的类名完全相同的方法称为构造方法。上述 Circle 类的定义中，有 Circle()和 Circle(double radius)两个构造方法，构造方法具有下列特性：

(1) 构造方法名必须和类名完全相同。

(2) 构造方法不具有任何返回值类型，如果写上任何一种返回值类型(包括关键字

void)则该方法不再是构造方法,而成为普通的成员方法。

(3) 构造方法也可以重载。重载构造方法的目的是使类对象具有不同的初始值,为类对象的初始化提供方便。

(4) 当使用 new 运算符创建对象时,系统会自动调用构造方法,构造方法起着初始化对象的作用。

(5) 当在一个类中没有显式定义构造方法时,系统会提供一个默认构造方法。默认构造方法没有任何形式参数,并且方法体为空。

(6) 不能使用 static、final、abstract、native 和 synchronized 修饰符修饰构造方法。

注意:只要用户定义了构造方法(不一定是无参构造方法),Java 语言就不再提供默认的构造方法。如果为类定义了一个带参数的构造方法,还想使用无参构造方法,则用户必须自己定义。建议用户自定义类的无参构造方法。

3.3 对象的定义和使用

3.3.1 创建对象

创建一个类,就创建了一种新的数据类型。创建对象也称为类的实例化。创建对象包括两个步骤:声明对象引用变量(即对象名)和实例化对象(即为对象分配存储空间)或者将两个步骤合二为一。

1. 声明对象

```
类名 对象名表;
```

其中,类名是指对象所属类的名字,它是在声明类时定义的;对象名表是指一个或多个对象名,若为多个对象名时,对象名之间用半角逗号进行分隔。如下述语句声明了 circleOne 和 circleTwo 两个 Circle 类的对象引用变量:

```
Circle circleOne, circleTwo;
```

注意:声明对象实质上是声明了对象引用变量,而并没有为对象开辟内存空间,对象引用变量存放对象的内存单元起始地址。严格地讲,对象和对象引用变量是有区别的,但是大部分时候它们的区别会被忽略。

2. 为对象分配内存空间

使用 new 运算符和类的构造方法为声明的对象分配内存空间,如果类中没有显式定义构造方法,系统会调用默认的无参构造方法。为对象分配内存空间的格式如下:

```
对象名 = new 构造方法([实参列表]);
```

例如:

```
circleOne = new Circle();      //此时 circleOne 引用被分配内存空间的真实地址
circleTwo  = new Circle(2.0);
```

建议读者在创建对象时采用下述格式一步完成：

```
类名 对象名 = new 构造方法([实参列表]);
```

例如下述语句：

```
Circle circleTwo  = new Circle(2.0);
```

在内存中开辟一块 Circle 对象的内存空间，circleTwo 对象引用变量存放该内存单元的起始地址。

3.3.2 对象的使用

在程序中创建对象的目的是使用对象。创建一个对象就要为对象的各个成员变量分配存储空间。可以通过引用对象的成员来使用对象，对象成员变量的引用方式如下：

```
对象名.成员变量名
```

对象成员方法的引用方式如下：

```
对象名.成员方法名(实参列表)
```

【例 3.1】 编写程序，测试上述 Circle 类。

源程序如下：

```
//TestCircle.Java
public class TestCircle {
    /* * Main method */
    public static void main(String[] args) {
        Circle myCircle = new Circle(5.0);  //以半径 5.0 创建一个 Circle 的实例
        //通过对象引用成员方法，输出圆的半径和面积
        System.out.println("The radius of the circle is  "
            + myCircle.getRadius());
        System.out.println("The area of the circle is "
            + myCircle.getArea());
        //增大圆的半径为原值的 1.1 倍
        myCircle.setRadius(myCircle.getRadius() * 1.1);
        System.out.println("The new radius of the circle is  "
            + myCircle.getRadius());
        System.out.println("The new area of the circle is "
            + myCircle.getArea());
    }
}
```

程序的运行结果如图 3.3 所示。

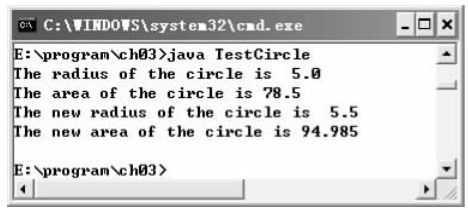

图 3.3 例 3.1 的运行结果

3.3.3 对象的清除

在 C++中使用 new 运算符动态分配的堆内存空间必须使用 delete 运算符来释放，这种内存管理方法需要跟踪内存的使用情况，不仅复杂而且容易造成系统内存资源利用率低。Java 采用垃圾自动回收机制进行内存管理，使程序员不需要跟踪每个对象，避免了上述问题的产生。Java 运行时系统通过垃圾自动回收机制周期性地释放无用对象所使用的内存，完成垃圾的自动回收。当一个对象的引用为空时（如语句“myCircle＝null;”将使得对象引用变量 myCircle 不再指向实例对象），该对象称为一个无用对象，它所占据的内存称为垃圾。垃圾收集器以较低优先级在系统空闲周期中执行，因此垃圾的收集速度比较慢。在某些情况下，也可以通过调用 System 类中的静态方法 gc()，即语句“System. gc();”可以向 Java 虚拟机申请尽快进行垃圾回收，但不能保证 Java 虚拟机会立即进行垃圾回收。gc()的方法头部如下：

```
public static void gc()
```

与 C++类似，Java 也提供了 finalize()方法，用来销毁一个不再使用的对象，该方法是类 java. lang. Object 的成员方法。在对象被回收前，Java 垃圾收集器通常会调用对象的 finalize()方法，释放当前对象所使用的资源。但是 Java 系统不保证在回收实例对象所占据的存储单元之前一定会调用 finalize()方法。

在类的定义中，除了定义一般的成员方法外，还可以重新定义用于对象清除的方法 finalize()，它的格式如下：

```
protected void finalize() throws Throwable{
    //撤销对象
}
```

虽然 Java 使用垃圾自动回收机制，可以自动完成对内存资源的回收。但在有些情况下，当一个对象被破坏后，需要执行一些垃圾收集器不能处理的特殊清理工作，要用到 finalize()方法。对于任意的类，用于对象撤销的方法要完成的功能基本上是一致的，例如，关闭已经打开的文件；保存对象实例中需要保存的信息；释放对象占用的内存空间等。

3.3.4 方法的参数传递

进行方法调用时，需要提供实参，它们必须与方法定义中所对应的形参次序相同，这叫

做参数顺序匹配。实参必须与方法头中的形参在次序上和数量上匹配，在类型上兼容。类型兼容是指不需要经过显式的类型转换，实参的值就可以传递给形参，例如将 int 型的实参值传递给 double 型形参。方法调用的格式有以下 3 种形式：

```
形式一：成员方法名(实参列表)
形式二：表达式.成员方法名(实参列表)
形式三：类名.静态成员方法名(实参列表)
```

在 Java 的方法调用中，方法中的参数是以传值的形式进行的，不管它是什么数据类型。如果是基本数据类型，则将实参的值传递给形参，改变方法内部的形参的值不会影响方法外部的实参值。

无论形参在方法中如何改变，实参均不受影响。如果是类类型，则传入的是引用的一个副本。

如同可以给方法传递基本数据类型一样，也可以向方法传递类类型的对象，称其为引用传递。引用传递传入的是引用的一个副本，归根结底还是传的值。

下面的例子说明了向方法传递基本数据类型和传递引用数据类型的区别。

【例 3.2】 编写程序，测试上述 Circle 类，方法调用时传递基本数据类型和引用数据类型。

源程序如下：

```
//TestPassParameter.Java
public class TestPassParameter {
    public static void main(String[] args) {
        Circle circle = new Circle(); //circle 为引用数据类型
        int n = 5;
        printAreas(circle, n);        //打印半径分别为 1、2、3、4、5 的圆的面积

        System.out.println("Radius is " + circle.getRadius());
        System.out.println("n is " + n);
    }

    public static void printAreas(Circle c, int times) {
        System.out.println("Radius\t\tArea");
        while (times >= 1) {
            System.out.println(c.getRadius() + "\t\t" + c.getArea());
            c.setRadius(c.getRadius() + 1);
            times -- ;
        }
    }
}
```

程序的运行结果如图 3.4 所示。

【程序解析】

√ 进行方法调用时，若传递基本数据类型的参数，则传递的是实参的值。本例中，当进行

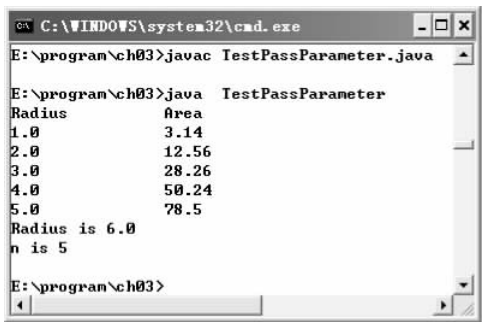

图 3.4 例 3.2 的运行结果

printAreas(circle, n)方法调用时,实参 n 的值 5 传递给形参 times,在 printAreas()方法内,times 的内容被改变,但这并不影响实参 n 的内容。所以 main()方法体中输出 n 的值仍然为 5。

√ 进行方法调用时,若传递的是引用数据类型的参数,则传递的是引用的副本。本例中,当进行 printAreas(circle, n)方法调用时,形参 c 与实参 circle 具有相同的引用值,即引用同一块内存空间。所以,要改变对象的属性,可以在 printAreas 方法内部使用 c,也可以在 printAreas 方法外部使用 circle,效果是一样的。在 printAreas 方法体内通过对象引用变量 c 调用 setRadius 方法修改了圆的半径,所以在 main()方法体中输出实参 circle 的半径时变为 6.0,而不再是最初的 1.0。

3.4 实例变量、实例方法和类变量、类方法

3.4.1 变量与方法

在类的定义中,用 static 关键字修饰的类的成员变量称为类变量或静态变量,用 static 关键字修饰的类的成员方法称为类方法或静态方法,非静态的变量和方法又分别称为实例变量和实例方法。

在生成每个类的对象时,Java 运行时系统为每个对象分配一块内存空间,然后可以通过对象引用变量来访问这些对象的实例变量。不同对象的实例变量值可能不同。在例 3.1 和例 3.2 中,Circle 类中的变量 radius 是一个实例变量,不能被同一个类的所有对象共享。

类变量只在系统加载其所在类时分配空间并初始化,并且在创建该类的对象时将不再分配空间,所有的对象将共享类变量。每个对象对类变量的改变都会直接影响到其他对象。类变量可用来在实例对象之间进行通信或跟踪该类实例的数目。在以上例子中,Circle 类中的变量 numberOfObjects 是一个类变量,定义时要用关键字 static 进行修饰,它被 Circle 类的所有对象共享。

static 类变量仍属于类的作用域,还可以使用 public、private 等进行修饰。修饰符不同,可访问的层次也不同。类变量可以通过类名直接访问,也可以通过对象名来访问,两种方法的结果是相同的。对于不是 private 类型的类变量,建议通过类名直接访问类变量,而

不像实例变量那样需要通过实例对象才能访问。

注意：

(1)Java 中没有全局变量，但类变量是在一个类的所有实例对象中都可以访问的变量，在一定程度上类似于其他语言中的全局变量。

(2) 实例方法可以对当前对象的实例变量进行操作，也可以对类变量进行操作，但类方法不能访问实例变量。

实例方法属于实例对象而且只能在实例对象创建后调用。实例方法的调用形式如下：

```
对象名.实例方法名(实参列表);
```

类方法由类名直接调用，其调用形式如下：

```
类名.类方法名(实参列表);
```

关于类方法的使用，有如下一些限制：

√ 在类方法中没有 this 关键字，不能访问所属类的实例变量和实例方法，只能访问方法体内定义的局部变量、该方法的形式参数和类变量；

√ 在类方法中不能使用 super 和 this 关键字；

√ main()方法是一个静态方法。因为它是程序的入口点，这可以使 JVM 不创建实例对象就可以运行该方法。因此，如果要在 main()方法中访问所在类的成员变量或方法，就必须首先创建相应的实例对象，否则会出现编译时错误。例如，下面的程序代码段，在静态 main()方法中不能直接访问实例变量 number：

```
public class Test {
    int number = 1;                                    //实例变量

    public static void main(String[] args) {           //静态方法 main()
        System.out.println("number is " + number);     //错误
    }
}
```

注意： 在编程时应该将常量定义为类的所有对象都能共享的 static 成员变量，即类变量，不能修改常量的值。描述对象共有属性的变量应该声明为类变量。

【例 3.3】 下面的代码在 3.2.1 节 Circle.java 文件的基础上，说明了类变量、实例变量和类方法、实例方法的使用。

源程序如下：

```
//TestClassAndInstance.Java
public class TestClassAndInstance {
    public static void main(String[] args) {
        Circle circle1 = new Circle();
        System.out.println("Before creating circle2:");
```

```
        System.out.println("    The radius of circle1 is: "
                + circle1.getRadius());              //通过对象名访问实例方法
        System.out.println("    The number of Circle object is: "
                + Circle.getNumberOfObjects());      //通过类名访问类方法

        Circle circle2 = new Circle(10);
        System.out.println("After creating circle2:");
        System.out.println("    The radius of circle2 is: "
                + circle2.getRadius());
        System.out.println("    The number of Circle object is: "
                + Circle.getNumberOfObjects());

        circle1.setRadius(3);
        System.out.println("Modify the radius of circle1:");
        System.out.println("    The radius of circle1 is: "
                + circle1.getRadius());
        System.out.println("    The radius of circle2 is: "
                + circle2.getRadius());
        System.out.println("    The number of Circle object is: "
                + Circle.getNumberOfObjects());
    }
}
```

程序的运行结果如图 3.5 所示。

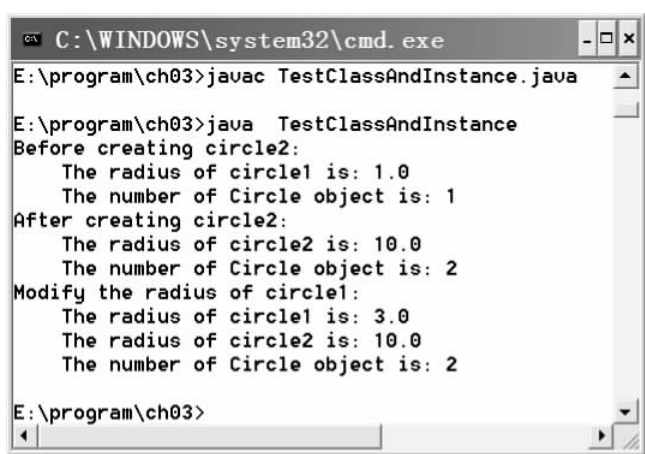

图 3.5　例 3.3 的运行结果

3.4.2　变量的作用域

实例变量用来描述对象的属性，类中的所有实例方法都可以访问这些变量。实例变量和类变量的作用域是整个类，它们可以在类中的任何位置说明，效果相同。可以把实例变量的说明放在类的末尾，而在前边定义使用该变量的方法。

类的数据成员和成员方法是类的成员，它们之间没有顺序。因此，在类中可以按任意顺序声明它们。

在方法中说明的变量称为局部变量，局部变量的作用域从声明它的位置开始到包含它的块尾。局部变量必须先说明后使用。

类的实例变量和类变量只能声明一次，但是局部变量可以在互不嵌套的块内多次声明，在同一嵌套块中只能声明一次，如下列代码段中的语句“int x=0;”是错误的。

```
public static void incorrectMethod() {
    int x = 1;
    int y = 1;
    for (int i = 1; i < 10; i++) {
        int x = 0;                          //错误，和上一个局部变量 x 在同一个嵌套的块中
        x += i;
    }
}
```

注意：实例变量和局部变量可以重名，但为了避免混淆，不要声明相同的变量名。

【例 3.4】 实例变量和局部变量重名。

源程序如下：

```
//CircleNew.java
public class CircleNew {
    private double radius = 1.0;                    //实例变量 radius
    int x = 1;

    void print() {
        double radius = 3.5;                        //局部变量 radius
        System.out.println("radius = " + radius);   //输出局部变量
        /* 通过 this 关键字调用实例变量 radius */
        System.out.println("this.radius = " + this.radius);
        System.out.println("x = " + x);
    }

    public static void main(String[] args) {
        CircleNew circle = new CircleNew();
        circle.print();
    }
}
```

程序的运行结果如图 3.6 所示。

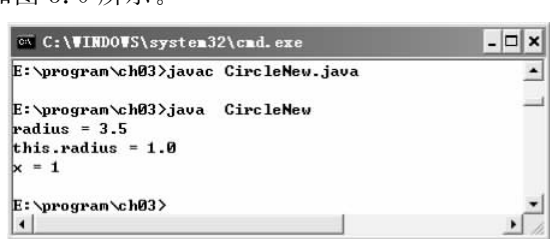

图 3.6 例 3.4 的运行结果

3.4.3 变量的初始值

如果没有对类的成员变量赋初值，则 Java 会对其赋默认值，对不同类型的成员变量赋以不同的默认值，如下所示。

引用类型：null；

数值类型：0；

boolean 类型：false；

char 类型：'\u0000'。

注意：对于方法体内的局部变量，Java 不提供初始值。程序员必须对其赋初值，否则会出现编译错误。

3.5 this 关键字

this 关键字用来指向当前对象或类的实例变量。如果局部变量与实例变量或类变量重名，则局部变量优先，同名的实例变量或类变量被隐藏。有时，需要引用方法中隐藏的实例变量或类变量。用“类名.类变量名”可以访问隐藏的类变量，用 this 关键字可以访问隐藏的实例变量。下列程序代码段说明了 this 关键字的使用：

```
class TestThis{
    int day = 1;                                    //实例变量 day
    void setDay(int day){
        this.day = day;
    }
}
```

其中，语句“this.day=day;”表示把形式参数 day 的值赋给当前对象的成员变量 day。如果 aa 为 TestThis 类的对象，当执行“aa.setDay(10);”时，系统会把 this 替换为当前对象 aa。

实际上，在一个对象的方法被调用执行时，Java 会自动给对象的实例变量和实例方法都加上 this 关键字，指向内存堆中的对象。

一个类的若干个构造方法之间可以相互调用。当一个构造方法需要调用本类的另一个构造方法时，可以使用 this 关键字，同时这个调用语句应该是整个构造方法的第一个可执行语句。使用 this 关键字来调用同类的其他构造函数，可以最大限度地提高对已有代码的利用程度(代码复用)，提高程序的抽象性和封装性，减少程序的维护工作量。

例如，可以重新定义 Circle 类如下：

```
public class Circle {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }
```

```
    public Circle() {
        this(1.0);
    }

    public double findArea() {
        return Math.PI * radius * radius;
    }
}
```

其中的语句“this(1.0);”调用类中参数为 double 类型的构造方法。

注意：

(1) Java 要求 this()语句出现在构造方法中任何其他语句之前。

(2) 如果一个类中有多个构造方法时，在无参的构造方法或参数少的构造方法中使用“this(实参列表);”来调用多参数的构造方法，可以增加类的可读性和可维护性。

3.6 包

对象复用是面向对象编程的主要优点之一，它是指同一对象在多个场合被反复使用。对象是类的实例，类是创建对象的模板，对象是以类的形式体现的。因此，对象复用也就体现在类的重用上。

为了便于管理大型软件系统中数目众多的类，解决类命名冲突问题和限定类的访问权限，可以将一组相关类和接口(参见第 5 章)“包裹”在一起形成包(package)。在 Java 程序中，如果要想使一个类在多个场合下反复使用，可以把它存放在一个称为“包”的程序组织单位中。一个包对应一个文件夹，一个包中可以包括许多类文件。包中还可以再有子包，称为包等级。

包的作用有四个：

(1) 定位类。具有相似功能的类可以放置在同一个包中，这样可以很容易地查找定位类。

(2) 避免命名冲突。在开发由其他程序员共享的可复用类时，会发生命名冲突，可以把类放在不同的包中，通过包名引用类可以避免命名冲突。

(3) 便于发布软件。包将相关的类组织到一起，可以方便地发布软件。

(4) 控制类之间的访问。包可以提供对类的保护，允许同一个包中的类访问类中被保护的成员，而外部类则无此权限。

注意：包是一个类名空间，同一个包中的类和接口(参见第 5 章)不能重名，不同包中的类可以重名。根据 Java 的命名规则，包名均为小写字母。类之间的访问控制是通过类修饰符来实现的，若类修饰符为 public，则表明该类不仅可供同一包中的类访问，也可以被其他包中的类访问。若类无修饰符，则表明该类仅供同一包中的类访问。Java 的包等级和 Windows 的文件组织方式完全相同，只是表示方法不同。

3.6.1 包的定义

包的定义就是将源程序文件中的接口和类纳入指定的包。一般情况下，Java 源程序由

四部分组成：

(1) 一个包定义语句(可选项)。其作用是将本源文件中的接口和类纳入指定包。源文件中若有包定义语句,必须是第一条语句。

(2) 若干个 import 语句(可选项)。其作用是引入本文件中所需要使用的包。

(3) 一个类声明。在一个源文件中只能有一个 public 类。

(4) 若干个属于本包的类声明(可选项)。

包的定义语句格式如下：

```
package 包名 1[.包名 2[.包名 3…]];
```

利用上述语句就可以定义一个具有指定名字的包,当前 Java 文件中的所有类和接口都被放在这个包中。习惯上,包名都用小写字母。Java 规定,如果一个 Java 文件中有 package 语句,那么 package 语句必须写在 Java 源程序的第一行,该行前只能有空格和注释行。package 语句在每个 Java 源程序中只能有一条,一个类只能属于一个包。例如下述语句：

```
package cn.edu.hebiace;
```

定义了包,语句中的包名分隔符“.”相当于目录分隔符。使用 package 语句指定一个源文件中的类属于一个特定的包。Java 要求包名与文件系统的目录结构一一对应。对于名为 cn.edu.hebiace 的包,必须创建一个如图 3.7 所示的目录结构。

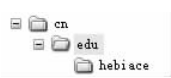

图 3.7　包 cn.edu.hebiace 对应文件系统的一个目录结构

图 3.7 中的目录 cn 不必是根目录。为使 Java 知道包在文件系统中的地址,必须修改环境变量 classpath,使它指向包所在的目录。classpath 中指定的目录顺序就是搜寻类的顺序。如果在不同的目录中有两个同名的类,Java 使用首先找到的类。

若源文件中未使用 package 语句创建包,则该源文件中的接口和类位于 Java 的无名包中(无名包又称默认包,指当前目录),会把源文件中的类存储在当前目录(即存放 Java 源文件的目录)下。无名包中的类不能被其他包中的类引用和复用,为此需要重建有名字的包。例如,在当前目录 D:\program\ch03 下编写下述程序代码段,Format 类中 format 方法的作用是返回 number 的值,要求四舍五入且小数点后保留 n 位数值。

```
//Format.Java
package pack1.pack2;

public class Format {
    public static double format(double number, int n) {
        return Math.round(number * Math.pow(10, n))/Math.pow(10, n);
    }
}
```

使用 javac -d . Format.java 命令编译该源文件,则自动在当前目录下创建 pack1 文件夹

(“.”表示当前目录),在 pack1 文件夹下创建 pack2 文件夹,并且自动把编译生成的 Format.class 字节码文件放入 pack2 文件夹中。

说明:如果编译源文件时,不使用-d 选项,则编译生成的字节码文件(即类文件)存放在当前目录下,不会创建 pack1 和 pack2 两个文件夹。

3.6.2 设置类路径

包是一种组织代码的有效手段,包名指出了程序中需要使用的.class 文件的所在之处。另一个能指明.class 文件所在的位置是系统环境变量 classpath,classpath 指明所有缺省的字节码文件.class 的路径。当一个程序找不到它所需使用的其他类的.class 文件时,系统会自动到 classpath 环境变量所指明的路径中去寻找。

在 Windows XP 或 Windows 2000 中可以通过设置“我的电脑”属性来设置系统的环境变量。

对于 Java Application 程序,还可以通过为 Java 解释器设置参数来指定类文件路径。例如,对于 JDK 中的 Java 解释器 java.exe,有开关参数-classpath;假设当需要解释执行的 test.class 文件不在当前目录而在 E 盘的 temp 目录下时,可以使用如下的命令行语句:

```
java -classpath E:\temp test
```

来运行这个程序。

3.6.3 包的使用

将类组织成包的目的是为了更好地使用包中的类。通常一个类只能引用与它在同一个包中的类。如果需要使用其他包中的 public 类,则可以使用如下的几种方法。

1. 在使用的类前加包名

一个类要引用其他类有两种方式:对于同一包中的其他类可直接引用;对于不同包中的其他类引用时需在类名前加包名,例如“pack1.pack2.Format.format(23.4533, 2);”,这种类名前加包名的引用方式适用于在源文件中使用较少的情况。

2. 单类型导入(single-type-import)

上述方法使用起来比较麻烦。使用 import 关键字可以加载需要使用的类到当前程序中,这样在程序中引用这个类的地方就不需要再使用包名作为前缀。例如上面的语句在程序开始处增加了“import pack1.pack2.Format;”语句之后,就可以直接写成:

```
Format.format(23.4533, 2);
```

3. 按需类型导入(type-import-on-demand)

按需类型导入,如“import java.util.*;”只会按需导入,也就是说,它并非导入整个包,而仅仅导入当前类需要使用的类。单类型导入和按需类型导入对类文件的定位算法是不一样的。对于单类型导入很简单,因为包名和文件名都已经确定,所以可以一次性查找定位。

对于按需类型导入则比较复杂，编译器会把包名和文件名进行排列组合，然后对所有的可能性进行类文件的查找定位。例如有下述语句：

```
package cn.edu;
import pack1.pack2.*;
import java.io.*;
```

如果类文件中用到了 Format 类，那么可能出现 Format 类的地方如下：

Format——Format 类属于无名包，就是说 Format 类没有 package 语句，编译器会首先搜索无名包。

cn.edu.Format——Format 类属于当前包。

java.lang.Format——编译器会自动导入 java.lang 包。

pack1.pack2.Format。

java.io.Format。

注意：

(1) 编译器找到 pack1.pack2.Format 类之后并不会停止下一步的寻找，而要把所有的可能性都查找完以便确定是否有类导入冲突。

(2) 如果在查找完成后编译器发现了两个同名的类，那么就会报错。需要程序员删除不用的那个类，然后再编译。

(3) java.lang 包总是被自动导入的。

(4) 用 import 关键字加载包如“import java.awt.*;”会按需导入 java.awt 包中除了子包之外的类。

(5) 一般情况下程序员只使用单类型导入，而很少使用按需类型导入。

按需类型导入绝对不会降低 Java 代码的执行效率，但会影响到 Java 代码的编译速度。使用单类型导入的两个好处是：

(1) 提高编译速度。

(2) 避免命名冲突。例如，如果有两条引入语句“import java.awt.*; import java.util.*;”使用 List 的时候编译器将会出编译错误。

【例 3.5】 按需类型导入的使用举例。

源程序如下：

```
//TestImport.java
import pack1.pack2.*;

public class TestImport {
    public static void main(String[]args){
        System.out.println(Format.format(23.4533, 2));
    }
}
```

【程序解析】

√ 磁盘的部分目录结构如图 3.8 所示，TestImport.java 文件存放在 ch03 文件夹下，

pack2 文件夹下存放有 Format.class 字节码文件。

√ 编译运行该程序，输出结果为 23.45。若在 ch03 文件夹下存放有 Format.java 文件或 Format.class 文件，则在编译 TestImport.java 文件时，会出现如图 3.9 所示的错误信息。删除掉 ch03 文件夹下的 Format.java 和 Format.class 文件后，编译通过，输出正确的运行结果。这是因为编译器查找 Format.class 字节码文件时，发现了两个同名的类，那么就会报错。需要程序员删除不用的那个类，然后再编译。

图 3.8　磁盘部分目录结构

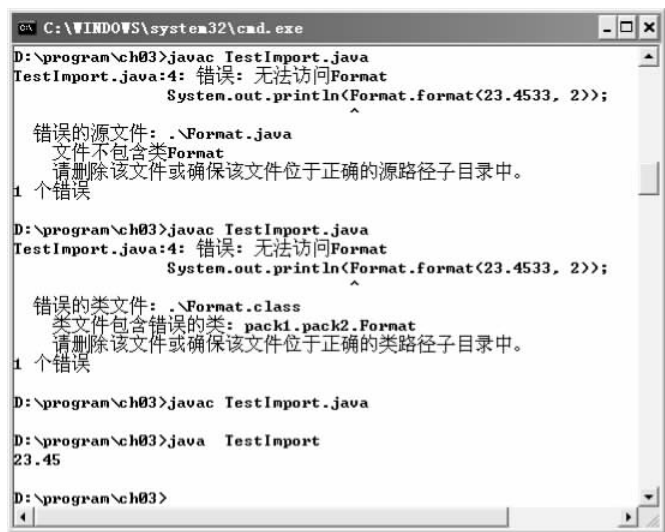

图 3.9　例 3.5 的编译运行情况

3.7　内部类和匿名类

3.7.1　内部类

嵌套定义于另一个类中的类称为内部类(Inner Classes)或内隐类，也称为嵌套类，包含内部类的类称为外部类。内部类可以将逻辑上相关的一组类组织起来，并由外部类来控制内部类的可见性。当建立一个内部类 InnerClass 时，其对象就拥有了与外部类(假设外部类的类名为 OuterClass)对象之间的一种关系，这是通过一个特殊的 this 引用形成的，使得内部类对象可以随意访问外部类中所有的成员。下面是一个内部类的例子。

【例 3.6】 设计一个人员类，要求自动生成人员的编号。

源程序如下：

```
class PeopleCount {
    private String Name;
    private int ID;
    private static int count = 0;      //类变量

    public class People {              //内部类 People 的定义
```

```
        public People() {
            count++;                    //访问外部类的私有成员变量
            ID = count;
        }

        public void output() {          //内部类的方法
            System.out.println(Name + "的 ID 为: " + ID);
        }
    }

    public PeopleCount(String sn) {  //外部类的构造方法
        Name = sn;
    }

    public void output() {           //外部类的方法
        People p = new People();     //建立内部类对象
        p.output();                  //调用内部类的方法
    }
}

public class TestInnerClass {
    public static void main(String[] args) {
        PeopleCount p1 = new PeopleCount("Person 1");
        p1.output();
        PeopleCount p2 = new PeopleCount("Person 2");
        p2.output();
        PeopleCount.People p3 = p1.new People();
        p3.output();
    }
}
```

程序的运行结果如图 3.10 所示。

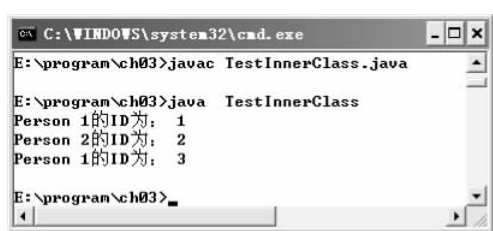

图 3.10　例 3.6 的运行结果

【程序解析】

√ 在外部类 PeopleCount 内嵌套定义了一个内部类 People，当定义外部类对象 p1 和 p2 后，调用 p1 或 p2 的 output 方法时，该方法将首先定义一个内部类对象，内部类的构造方法先将外部类的类变量 count 加 1，然后把 count 的值赋给成员变量 ID，然后调用内部类的 output 方法进行输出。

√ 此例中的内部类是非静态内部类，在创建内部类 People 类的对象时，必须已经创建好外部对象 p1 或 p2 了。注意非 static 型内部类的对象生成方式“PeopleCount. People p3＝p1. new People();”由先前创建的外部对象 p1 或 p2 来创建内部类对象 p3。

外部类与内部类的访问原则是：在外部类中，一般通过一个内部类的对象来访问内部类的成员变量或方法；在内部类中，可以直接访问外部类的所有成员变量和方法（包括静态成员变量和方法、实例成员变量和方法及私有成员变量和方法）。

内部类与其他常规类类似，且有下列特征：

（1）Java 将内部类作为外部类的一个成员，内部类可以调用包含它的外部类的成员变量和成员方法，所以不必把外部类的引用传递给内部类的构造方法。

（2）内部类的类名只能用在外部类和内部类自身中，内部类的类名不能与外部类的类名相同。当外部类引用内部类时，必须给出完整的名称（外部类名.内部类名）。

（3）内部类只是用来支持其外部类的工作，编译后，它的名称形如“外部类名称 $ 内部类名称.class”。例 3.6 编译后生成 PeopleCount $ People.class、PeopleCount.class 和 TestInner Class.class 三个字节码文件。

（4）内部类可以声明为 public、protected 或 private，其含义与用在类的其他成员上相同。

在实际的 Java 程序设计中，内部类主要用来实现接口和编写事件驱动程序。

和普通的类一样，内部类也可以有静态的。static 型内部类只能访问外部类中的 static 成员。若要访问非 static 成员，须先创建一个外部类对象，然后通过该对象进行访问。在任何非静态内部类中，都不能有静态数据成员、静态成员方法或者另外一个静态内部类（内部类的嵌套可以不止一层）。静态内部类中却可以拥有这一切。创建 static 内部类对象时无须先创建外部类对象，可以使用下列语句直接创建 static 内部类对象：

```
OutClass.StaticInnerClass nameOfInnerClass = new OutClass.StaticInnerClass();
```

【例 3.7】 静态内部类和非静态内部类举例。

源程序如下：

```
class Outer {                                          //外部类
    private int index = 100;
    private static int index2 = 200;

    class Inner {                                      //非静态内部类
        private int index = 50;

        void print() {
            int index = 30;
            System.out.print(index + "\t");            //输出局部变量 index 的值 30
            System.out.print(this.index + "\t");
                    //上述语句输出 Outer.Inner.index 的值 50
            System.out.println(Outer.this.index);
                    //上述语句输出 Outer.index 的值 100
        }
    }

    static class Inner2 {                              //静态内部类
        private int index = 50;

        void print() {
```

```
            int index = 30;
            System.out.print(index + "\t");              //输出局部变量 index 的值 30
            System.out.print(this.index + "\t");
                    //上述语句输出 Outer.Inner2.index 的值 50
            System.out.println(Outer.index2); /* 该语句输出 Outer.index2 的值 200,此方法
中不能输出非静态成员 index 的值 100 */
        }
    }

    Inner getInner() {
        return new Inner();
    }

    void print() {
        Inner inner = new Inner();
        inner.print();
    }
}

class TestOuterInnerClass {
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.print();
        Outer.Inner inner1 = outer.getInner();
        inner1.print();
        Outer.Inner inner2 = outer.new Inner();          //创建非静态内部类
        inner2.print();
        Outer.Inner2 staticInner2 = new Outer.Inner2(); //创建静态内部类
        staticInner2.print();
    }
}
```

程序的运行结果如图 3.11 所示。

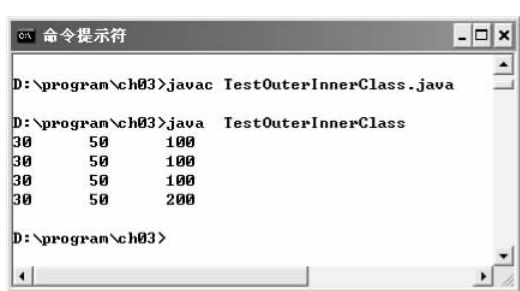

图 3.11 例 3.7 的运行结果

3.7.2 匿名类

匿名类又称为匿名内部类,是指可以利用内部类创建没有名称的对象,它一步完成了声明内部类和创建该类的一个对象,并利用该对象访问到类的成员。有时在定义事件处理代码时,由于代码短小,不必再明确定义一个类,可使用匿名内部类。匿名内部类是 final(最

终)类,而非 static 类,匿名内部类将类的声明和创建类的实例一步完成。下面的代码段是按钮注册事件处理代码:

```
JButton jb = new JButton("Exit");
jb.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.exit(-1);
    }
});
```

一般做法是定义一个类,实现 ActionListener 接口,重写 public void actionPerformed (ActionEvent e)方法,最后向按钮申请注册。现由于该代码只有一行:System.exit(-1);,故用匿名内部类将上述步骤合起来,直接写在 new 运算符的表达式中。

注意:在使用匿名内部类时,要记住以下几个原则:

(1) 匿名内部类不能有构造方法;

(2) 匿名内部类不能定义任何静态成员、方法和类。

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

(4) 只能创建匿名内部类的一个实例。一个匿名内部类一定是在 new 的后面,用其隐含实现一个接口或实现一个类。因匿名内部类为局部内部类,所以局部内部类的所有限制都对其生效。内部类只能访问外部类的静态变量或静态方法。

(5) 当在匿名类中用 this 时,这个 this 则指的是匿名类或内部类本身。这时如果要使用外部类的方法和变量,则应该加上外部类的类名。

3.8 访问控制和类的封装性

3.8.1 访问控制

Java 提供了用来控制对数据成员、成员方法和类进行访问的修饰符。修饰符包括 private、public、protected 和默认修饰符。protected 修饰符在第 5 章中介绍。四种修饰符的作用范围如表 3.4 所示。

表 3.4 四种修饰符的作用范围

修饰符	同一个类	同一个包	子类	全部
public	√	√	√	√
protected	√	√	√	
default	√	√		
private	√			

用 public 修饰的数据成员、成员方法和类可以被任何类所访问。

用 private 修饰的数据成员和成员方法只能被本类所访问。

如果数据成员、成员方法和类不用任何修饰符修饰,则称为默认访问属性,或包访问属

性，具有包访问属性的数据成员、成员方法和类可以被同一个包中的任何类所访问。

和其他面向对象程序设计语言相比，Java 增加了包的概念，这样，在同一个包中类之间的信息传递就比较方便。

注意：private 修饰符只能用来修饰类的数据成员和成员方法，而不能用来修饰类。修饰符 private、public、protected 都不能用来修饰方法中的局部变量。在大部分情况下，一个类的构造方法都是 public 的。但是，如果不想创建类的实例，可以定义 private 构造方法。

3.8.2 类的封装性

面向对象程序设计语言的一个重要特性是其封装性。Java 是按类划分程序模块的，Java 很好地实现了类的封装性。

保证大型软件设计正确性和高效性的一个重要原则是模块化软件设计。这需要设计许多可重复使用的模块，然后在需要使用这些模块的地方进行调用。但是，如何划分好模块的界限，以保证模块的正确性是非常重要的问题。因为如果随意修改了已经被其他人使用的模块，将使程序出错，并且这样的错误很难被发现和修改。

在面向对象程序设计中，保证模块正确性的基本方法是类的封装性。类的封装性是指类把数据成员和方法封装为一个整体，这就划分了模块的界限。

保证模块正确性的措施是由信息的隐藏性来实现的。类的成员包括数据成员和成员方法两部分。那些允许其他包访问和修改的成员可以定义为 public 类型；只允许同一个包中的其他类，以及该类的子类访问和修改的成员可以定义为 protected 类型；不允许其他类（内部类除外）访问和修改的成员可以定义为 private 类型。private 类型和 protected 类型的成员有效地隐藏了类的不能被随意修改的成员信息，从而保证了共享的类模块的正确性。类的封装性和信息的隐藏性是紧密结合在一起的。类方法不同访问控制权限的定义，给调用者提供了权限明确的接口。

习题

1. 设计一个复数类。要求复数类包括实部和虚部两个成员变量，同时类中应包含复数运算的各种方法。最后编写程序进行测试。

2. 设计一个长方形类。成员变量包括长度和宽度。类中除了包含计算周长和面积的方法外，还应该能够用 set 方法来设置长方形的长度和宽度，以及能够用 get 方法来获得长方形的长度和宽度。最后，编写一个测试程序来测试所定义的长方形类能否实现预定的功能。

要求：使用自定义包。

3. 设计一个分数类。要求分数类包括分子和分母两个成员变量，同时类中应包含分数运算的各种方法。编写一个测试程序进行测试。

4. 设计一个日期类，其输出格式是“月/日/年”或“月　日　年”，并编写一个测试程序来测试所定义的日期类能否实现预定的功能。

要求：把所设计的日期类作为测试类的内部类。