第3章 面向对象编程基础

面向对象程序设计是在面向过程程序设计的基础上发展而来的，它将数据和对数据的操作看作是一个不可分割的整体。通过采用数据抽象和信息隐藏技术，将现实世界问题的求解简单化，从而获得更好的应用架构和开发效率，同时提高程序的可维护性。本章将对面向对象程序设计中的基础知识进行详细讲解。

3.0 问题导入

【导入问题】 我们需要创建一个程序来描述多辆汽车的信息，每辆车的基本信息包括：车辆型号、发动机型号、车轮个数、车轮型号，假定车辆分为两大类：轿车类和公共汽车类，两类汽车都有鸣笛行为，但鸣笛发出的声音是不同的，该如何实现呢？

【初步分析】 按照前面所学的知识，需要定义多个数组，每个数组存储车辆的一种信息，要获取一辆车的完整信息需要访问多个数组。对两类汽车的鸣笛行为分别调用不同的方法实现。这种操作方式是比较烦琐的，操作也不够灵活，那么利用面向对象的编程思想来创建应用程序会使问题简化吗？

3.1 类的定义

类是封装数据的基本单位，是一组具有相同数据结构和相同操作的对象的集合，用来定义对象的组成及可执行的操作。我们将类的实例称为对象，一旦创建了一个对象，就可以调用对象的属性、方法和事件来访问对象的功能。

对象是现实中一个具体的事物（可以是具体的也可以是抽象的），类是对对象的概括。比如一个苹果、一个梨等都是一个对象，水果类就是对所有对象的抽象。

3.1.1 类的声明与成员组织

C#语言中，声明类的一般形式为：

```
【访问修饰符】class 类名称【:【基类】【,接口序列】】
{
    【字段声明】
    【构造函数】
    【方法】
    【事件】
}
```

关于声明类的说明：

(1)【】中的内容为可选项，冒号(:)后面的内容表示被继承的类或接口。

(2) 当一个类从另一个类继承时，被继承的类叫作基类或父类，继承的类叫作派生类或子类。

(3) 基类只能有一个，但一个类可以继承多个接口，继承的内容多于一项时，各项之间用逗号分开。

(4) 如果一个类同时继承基类和接口，则要把基类放在冒号后面的第一项，然后才是接口。

【例 3-1】 类的声明举例。

```
01. namespace Example3_1
02. {
03.     public class Apple
04.     {
05.         private string productId;
06.         private double productPrice;
07.         //带参数的构造函数
08.         public Apple(string id, double price)
09.         {
10.             productId = id;
11.             productPrice = price;
12.         }
13.         //方法
14.         public void OutPut()
15.         {
16.             Console.WriteLine("{0}_apple's price is:{1}",productId, productPrice);
17.         }
18.     }
19.     class Example3_1
20.     {
21.         static void Main(string[] args)
22.         {
23.             Apple apple1 = new Apple("a001",2.3);
24.             Apple apple2 = new Apple("a002",3.3);
25.             Apple apple3 = new Apple("a003",4.6);
26.             apple1.OutPut();
27.             apple2.OutPut();
28.             apple3.OutPut();
29.             Console.Read();
30.         }
31.     }
32. }
```

【运行结果】

单击工具栏中的“开始”按钮，即可在控制台中输出如图 3-1 所示的结果。

【程序说明】

- 在这个实例中，声明了一个 Apple 类，第 5、6 行声明了 Apple 类的两个私有字段，分

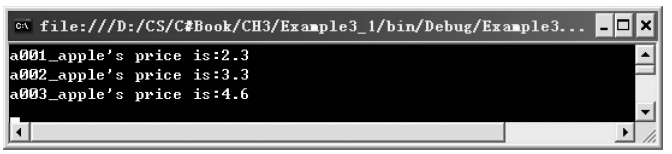

图 3-1　例 3-1 运行结果

别代表产品的编号和价格。

- 第 8 行定义了一个 Apple 类的带参构造函数，通过该构造函数在创建对象时，可以定义产品的编号和价格。
- 第 14 行定义了一个 Apple 类的公有输出方法，通过该方法在外部可以读取产品的信息。
- Example3_1 类中主函数的前三行，使用 new 创建了三个 Apple 类的对象，在创建对象的同时定义了产品的编号和价格。
- 主函数中第 26～28 行对每个 Apple 类对象调用其 OutPut 方法输出产品信息。

说 明

如果不使用类去创建三个 Apple 对象，你会怎么做呢？按以前的方法需要对三个苹果的信息分别创建一次，而在 Apple 类中通过构造函数可以轻松创建多个对象，Apple 类中定义的方法可以为所有对象共同使用。与面向过程的编程方式比较一下，可以感觉到面向对象编程方式的好处。

3.1.2　字段和局部变量

字段是在类或结构中声明的任何类型的“类级别”变量，它是类或结构的直接下属，是整个类内部所有方法和事件都可以访问的变量。

局部变量是相对于字段来说的，可以将它理解为“块”一级的变量。例如，在某个方法或循环体内定义的变量等，其作用域仅局限于定义它的语句块内。

两者在使用上要注意：对于字段如果没有初始化，C＃会自动将其初始化为默认值；但对于局部变量，C＃不会自动为其进行初始化，如果局部变量在使用前未被赋值，在编译时会报错。

【例 3-2】　字段和局部变量的使用举例。

```
01. namespace Example3_2
02. {
03.     class Example3_2
04.     {
05.         public static int m;
06.         static void Main(string[] args)
07.         {
08.             int n = 1;
09.             Console.WriteLine("静态字段 m 的值是: {0}",m);
10.             Console.WriteLine("局部变量 n 的值是: {0}",n);
11.             Console.Read();
```

```
12.         }
13.     }
14. }
```


【运行结果】

单击工具栏中的“开始”按钮，即可在控制台中输出如图 3-2 所示的结果。

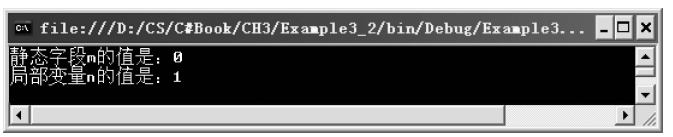

图 3-2　例 3-2 运行结果

【程序说明】

- 第 5 行定义了一个静态整型字段 m，未对其赋初始值，则其默认值为 0。
- 第 8 行在 Main 函数中定义了一个局部整型变量 n，如果对 n 未进行初始化，编译时就会报错。
- 局部变量没有与静态字段 m 重名的，所以在 m 的前面没加类名也可以。

当字段和局部变量名相同时（程序设计时应尽量避免），如果要引用静态字段，可以使用下面的形式：

```
类名.字段名
```

如果是实例字段，则使用下面的形式：

```
this.字段名
```

这里的 this 指当前实例。

3.1.3　静态成员和实例成员

在类中定义的数据称为类的数据成员，如字段、常量等。而函数成员则提供类的某些操作功能，如方法、属性等。默认情况下这些成员都是实例成员，这些成员属于类的实例所有，每创建一个对象时这些成员就会被创建一次。有一些成员是所有对象共用的，如果把这样的成员作为实例成员，当创建了多个对象时，在堆中就会出现很多相同的内容，这样会导致系统资源的浪费。

解决这个问题的办法是将成员定义为静态的，静态成员属于类所有，为这个类的所有实例所共享，无论为这个类创建了多少对象，一个静态成员在内存中只占一块区域。

实例成员通过类的实例进行调用，静态成员由类本身进行调用。

【例 3-3】 实例成员与静态成员的使用。

```
01. namespace Example3_3
02. {
03.     public class Payment
04.     {
05.         int productNums;                          //商品数量
06.         double productPrice;                      //商品单价
```

```
07.         static double totalPrice;                    //所有商品总价
08.         public Payment(int nums, double price)       //实例构造函数
09.         {
10.             productNums = nums;
11.             productPrice = price;
12.         }
13.         static Payment()                             //静态构造函数
14.         {
15.             totalPrice = 0.0;
16.         }
17.         public void Calculate()                      //实例方法
18.         {
19.             totalPrice += productNums * productPrice;
20.         }
21.         public static void PrintTotal()              //静态方法
22.         {
23.             Console.WriteLine("所有商品总价为: {0}",totalPrice);
24.         }
25.     }
26.     class Example3_3
27.     {
28.         static void Main(string[] args)
29.         {
30.             Payment apple = new Payment(5, 1.2);
31.             Payment banana = new Payment(6, 1.7);
32.             apple.Calculate();
33.             banana.Calculate();
34.             Payment.PrintTotal();
35.             Console.Read();
36.         }
37.     }
38. }
```

【运行结果】

单击工具栏中的“开始”按钮，即可在控制台中输出如图 3-3 所示的结果。

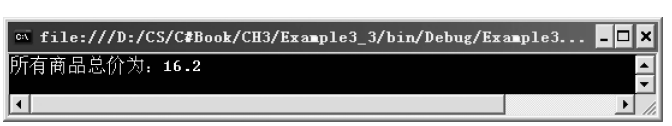

图 3-3　例 3-3 运行结果

【程序说明】

- 在这个实例中，Payment 类中首先定义了三个字段，其中 totalPrice 是静态字段。
- Payment 类中第 8～12 行定义了一个带参数的构造函数，通过该构造函数可以指定商品的数量和单价；第 13～16 行定义了一个静态的构造函数，该函数完成对静态字段 totalPrice 的初始化。
- Payment 类中第 17～20 行定义了一个实例计算方法，计算当前商品的总价后加到静态字段 totalPrice 上；第 21～24 行定义了一个静态方法用于输出所有商品的总

价，即输出 totalPrice 的值。

- 在类 Example3_3 的 Main 函数中实例化了两个 Payment 类的对象，并调用相应的方法。静态构造函数只被调用一次，所以静态字段 totalPrice 也只被初始化一次；而实例构造函数被调用两次分别生成 apple 和 banana 对象。
- 实例方法 Calculate 为每个对象所有，在第 32 行和第 33 行通过对象自身的调用分别计算出 apple 和 banana 的总价。
- 静态方法 PrintTotal 为 Payment 类本身所有，在 34 行通过类本身进行调用，输出所有商品的总价。

3.1.4 访问修饰符

访问修饰符用于控制类及类的成员的访问权限，便于对数据进行控制和修改，下面分别进行介绍。

1. 类的访问修饰符

类的访问修饰符及每个修饰符的含义如下。

(1) public：公有类，不限制对该类的访问。

(2) internal：项目内类，在当前项目内可以被自由访问，而对其他程序集来说无法访问。

(3) partial：分部类型，类的定义和实现可以分布在多个文件中，但都需要使用 partial 标注。

2. 类成员访问修饰符

类成员的访问修饰符及每个修饰符的含义如下。

(1) public：公有访问，外部类可以不受限制地存取这个类的数据成员或访问其方法。

(2) private：私有访问，类的数据成员和方法只能在此类中使用，外部无法存取。

(3) protected：保护访问，类及派生类中的成员可以访问，无法从类的外部进行访问。

(4) internal：内部访问，在当前项目内可以被自由访问，而对其他程序集来说无法访问。

(5) protected internal：在当前项目内，只有类及派生类中的成员可以访问。

【例 3-4】 访问修饰符的使用。(改写例 3-3)

```
01. namespace Example3_4
02. {
03.     class Payment
04.     {
05.         int productNums;
06.         double productPrice;
07.         static double totalPrice;
08.         public Payment(int nums, double price)
09.         {
10.             productNums = nums;
11.             productPrice = price;
12.         }
13.         static Payment()
```

```
14.         {
15.             totalPrice = 0.0;
16.         }
17.         internal void Calculate()
18.         {
19.             totalPrice += productNums * productPrice;
20.         }
21.         internal static void PrintTotal()
22.         {
23.             Console.WriteLine("所有商品总价为: {0}", totalPrice);
24.         }
25.     }
26.     class Example3_4
27.     {
28.         static void Main(string[] args)
29.         {
30.             Payment apple = new Payment(5, 1.2);
31.             Payment banana = new Payment(6, 1.7);
32.             apple.Calculate();
33.             banana.Calculate();
34.             Payment.PrintTotal();
35.             Console.Read();
36.         }
37.     }
38. }
```

【运行结果】

单击工具栏中的“开始”按钮，即可在控制台中输出如图 3-4 所示的结果。

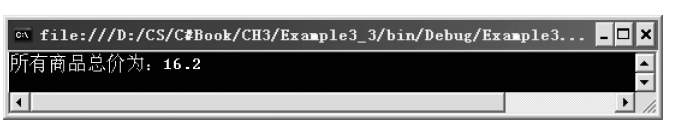

图 3-4　例 3-4 运行结果

【程序说明】

- 本例中未对 Payment 类加访问修饰符，默认为 internal；第 5～7 行定义的三个字段默认访问修饰符为 private，在类的外部无法被访问。
- 实例构造函数的访问修饰符一般为 public，静态构造函数不允许使用访问修饰符，对于构造函数将在 3.2 节进行介绍。
- 第 17 行和第 21 行的 Calculate 和 PrintTotal 方法的访问修饰符定义为 internal，在此项目内(namespace Example3_4 命名空间下)可以被访问，项目外无法被访问。如果将这两个方法的访问修饰符改为 private 或 protected，在类 Example3_4 的 Main 函数中就无法访问了。
- 由此例可以看出，类的默认访问修饰符为 internal；类中的数据成员和函数成员的默认访问修饰符为 private。

3.2 构造函数和析构函数

对象和客观世界中的事物一样，从创建到消亡都有一个生命周期，对象的创建和销毁是通过类的构造函数和析构函数来完成的。

3.2.1 构造函数

构造函数是在创建给定类型的对象时执行的类方法，通常用于初始化新对象的数据成员。

构造函数具有以下特点。

(1) 构造函数的名称与类名相同。

(2) 构造函数不包含任何返回值。

(3) 每个类至少有一个构造函数，如果没有显式地定义构造函数，系统会自动为该类提供一个默认的构造函数。

(4) 构造函数在创建对象时被自动调用，不能被显式地调用。

(5) 一般使用访问修饰符 public 定义构造函数，以便在其他函数中可以创建该类的实例。如果使用访问修饰符 private 定义构造函数则是私有构造函数，私有构造函数是一种特殊的构造函数，通常用在只包含静态成员的类中，用来阻止该类被实例化。

1. 默认构造函数

如果在类中没有显式定义构造函数，系统会自动提供一个默认的构造函数，默认构造函数没有参数。提供默认构造函数的目的是为了保证能够在使用对象前先对未初始化的非静态类成员进行初始化。具体如下：

(1) 对数值型，如 int、double 等，初始化为 0。

(2) 对 bool 类型，初始化为 false。

(3) 对引用类型，初始化为 null。

2. 静态构造函数

静态构造函数主要用于对类的静态字段进行初始化，它不能使用任何访问限制修饰符。在程序中第一次用到某个类时，类的静态构造函数自动被调用，而且是仅此一次。

3. 带参构造函数

不带任何参数的构造函数称为“默认构造函数”。有时希望通过传递不同的数据来创建不同的对象，这时可以使用带参数的构造函数。根据传递参数的不同，一个类可以有多个构造函数，以便根据不同的需要来创建不同的对象。

3.2.2 析构函数

析构函数是对象销毁前释放所占用系统资源的类成员。.NET Framework 类库具有垃圾回收功能，当对象使用完毕后，并符合析构条件时，.NET Framework 类库的垃圾回收功能就会调用对应类的析构函数实现垃圾回收。

析构函数具有以下特点。

(1) 析构函数的名称与类名相同，但在名称前面需加一个符号“～”。

(2) 析构函数不带任何参数，也不返回任何值。

(3) 析构函数不能使用任何访问限制修饰符。

(4) 析构函数的代码中通常只进行销毁对象的工作,而不应执行其他的操作。

(5) 析构函数不能被继承,也不能被显式地调用,一个类只能有一个析构函数。

【例 3-5】 构造函数和析构函数使用示例。

```
01. namespace Example3_5
02. {
03.     public class Destructor
04.     {
05.         public Destructor()
06.         {
07.             Console.WriteLine(" --- 创建对象时自动调用默认构造函数! --- ");
08.         }
09.         ~Destructor()
10.         {
11.             Console.WriteLine(" --- 程序运行结束时自动调用析构函数! --- ");
12.             Console.Read();
13.         }
14.     }
15.     class Example3_5
16.     {
17.         static void Main(string[] args)
18.         {
19.             Destructor test = new Destructor();
20.         }
21.     }
22. }
```

【运行结果】

单击工具栏中的“开始”按钮,即可在控制台中输出如图 3-5 所示的结果。

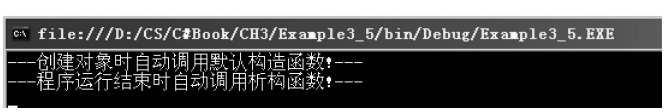

图 3-5 例 3-5 运行结果

【程序说明】

- 第 3 行定义了一个 Destructor 类,其中包含一个无参的默认构造函数和一个析构函数。
- 第 19 行在 Main 函数中创建一个 Destructor 类对象 test 时,首先调用了该类的默认构造函数,当程序运行结束时自动调用析构函数。

说明

如果类中没有显式地定义析构函数,编译器会为其生成一个默认的析构函数,其执行代码为空。事实上,在 C# 语言中使用析构函数的机会很少,通常只用于一些需要释放资源的场合,如删除临时文件、断开与数据库的连接等。

3.3 类的方法

方法是一组程序代码的集合，用于完成指定的功能。每个方法都有一个方法名，便于识别和让其他方法调用。方法实际上就是函数，在面向过程的程序设计语言中称为函数，在面向对象的程序设计语言中，除了构造函数以外，其他的函数可以是方法或事件。

3.3.1 方法的声明

C#程序中方法必须包含在类或结构中。声明方法的一般形式为：

```
【访问修饰符】返回值类型 方法名称(【参数序列】)
{
    【语句序列】
}
```

一个方法的名称和参数列表定义了该方法的签名，即一个方法的签名由方法的名称、参数的个数、参数的修饰符及参数的类型组成。

在定义方法时，需要注意以下几点：

(1) 如果参数序列中的参数有多个，用逗号分隔开；如果没有任何参数，方法名后面的小括号必须保留。

(2) 如果声明一个非 void 类型的方法，则方法中至少要有一个 return 语句；如果方法的返回类型为 void，则可以没有 return 语句。

(3) 方法的名称必须与在同一个类中声明的所有其他非方法成员的名称都不相同。

例 3-4 中的 Calculate()和 PrintTotal()方法都是不带参数的，关于带参数的方法下面进行详细介绍。

3.3.2 方法中的参数传递

C#中的许多方法成员是有参数的，定义方法时声明的参数是形式参数(或叫虚拟参数)，调用方法时要给形式参数传值，传递的值是实在参数。C#有“值传递”和“引用传递”两种参数传递方式。具体的传递形式有以下几种：

1. 传递值类型的参数

值传递是 C#默认的传递方式，使用值传递方式时，向形式参数传递的是实在参数的副本，方法内对形式参数的更改对实在参数本身没有任何影响，就像文件的复印件一样，无论如何修改复印件，原件没有任何改变。

【例 3-6】 以值传递的方式传递值类型数据。

```
01. namespace Example3_6
02. {
03.     class Swap
04.     {
05.         public static void SwapInt(int x, int y)
06.         {
07.             int temp;
```

```
08.             temp = x;
09.             x = y;
10.             y = temp;
11.         }
12.     }
13.     class Example3_6
14.     {
15.         static void Main(string[] args)
16.         {
17.             int m = 5, n = 10;
18.             Console.WriteLine("交换前: m = {0}, n = {1}", m, n);
19.             Swap.SwapInt(m, n);
20.             Console.WriteLine("交换后: m = {0}, n = {1}", m, n);
21.             Console.Read();
22.         }
23.     }
24. }
```

【运行结果】

单击工具栏中的“开始”按钮，即可在控制台中输出如图 3-6 所示的结果。

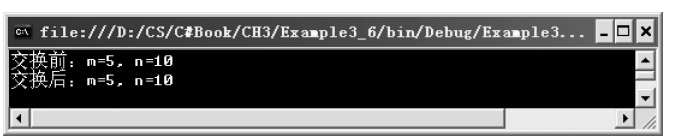

图 3-6　例 3-6 运行结果

【程序说明】

- 在类 Swap 中定义了一个静态的 SwapInt 方法，该方法的功能是完成两个整数的交换。
- Main 函数中第 17 行定义了两个整型变量 m 和 n，并赋予初始值；第 19 行在 Main 函数中调用 Swap 类的 SwapInt 方法，m、n 作为实参将值传递给形参 x、y，从运行结果可以看到 m、n 的值并未发生改变。
- 出现上述情况的原因在于 m、n 的值分别传给 x、y 时，x 和 y 开辟了新的存储空间，它们是 m 和 n 的复印件，所以在 SwapInt 方法内对 x 和 y 的交换并不能改变 m 和 n 的值。

【例 3-7】 以值传递的方式传递引用类型数据。

```
01. namespace Example3_7
02. {
03.     class Swap
04.     {
05.         public static void AddOne(int[] a)
06.         {
07.             for (int i = 0; i < a.Length; i++ )
08.                 a[i]++ ;
09.         }
10.     }
```

```
11.     class Example3_7
12.     {
13.         static void Main(string[] args)
14.         {
15.             int[] array1 = { 1, 2, 3, 4, 5 };
16.             Console.Write("处理前 array1 中的值依次为: ");
17.             for (int i = 0; i < array1.Length; i++ )
18.                 Console.Write("{0} ",array1[i]);
19.             Console.WriteLine();
20.             Swap.AddOne(array1);
21.             Console.Write("处理后 array1 中的值依次为: ");
22.             for (int i = 0; i < array1.Length; i++ )
23.                 Console.Write("{0} ", array1[i]);
24.             Console.WriteLine();
25.             Console.Read();
26.         }
27.     }
28. }
```

【运行结果】

单击工具栏中的“开始”按钮,即可在控制台中输出如图 3-7 所示的结果。

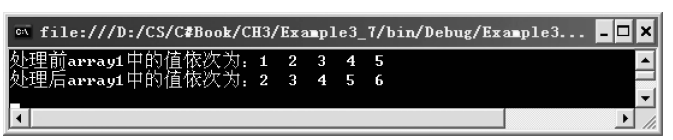

图 3-7　例 3-7 运行结果

【程序说明】

- 在类 Swap 中定义了一个静态的 AddOne 方法,该方法的功能是将给定的整型数组中的每一个元素值加 1。
- Example3_7 类的 Main 函数中,程序的第 15 行定义了一个整型数组 array1,第 20 行调用 Swap.AddOne 方法,实参为数组 array1,最终结果 array1 中每一个元素的值都加 1 了。
- 值传递不是不改变实参的值吗? 原因在于数组是引用类型,array1 保存的是数组的引用地址,传给形参 a 的值也是该数组的引用地址,即形参 a 和实参 array1 是指向同一段数据存储区,所以对 a 的操作就是对 array1 的操作。

2. 传递引用类型的参数

与传递值类型参数不同,引用传递时形式参数并不创建新的存储单元,形参与方法调用中的实参变量同处一个存储单元,方法内对形式参数的改变就是对实在参数的改变。为了和传递值类型参数区分,需要在参数前面加上 ref 关键字。

【例 3-8】 传递引用类型参数举例。(改写例 3-6)

```
01. namespace Example3_8
02. {
03.     class Swap
04.     {
```

```
05.         public static void SwapInt(ref int x, ref int y)
06.         {
07.             int temp;
08.             temp = x;
09.             x = y;
10.             y = temp;
11.         }
12.     }
13.     class Example3_8
14.     {
15.         static void Main(string[] args)
16.         {
17.             int m = 5, n = 10;
18.             Console.WriteLine("交换前：m={0}, n={1}", m, n);
19.             Swap.SwapInt(ref m, ref n);
20.             Console.WriteLine("交换后：m={0}, n={1}", m, n);
21.             Console.Read();
22.         }
23.     }
24. }
```

【运行结果】

单击工具栏中的“开始”按钮，即可在控制台中输出如图 3-8 所示的结果。

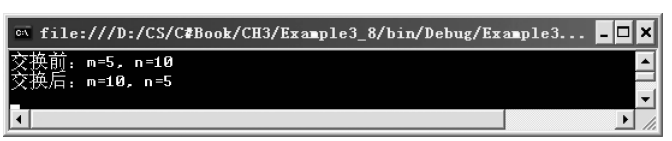

图 3-8　例 3-8 运行结果

【程序说明】

- 相比例 3-6，在 Swap 类的 SwapInt 静态方法中，形参前面加上了 ref，方法调用时实参前面也加上了 ref。
- 从运行结果可以看到 m、n 的值进行了交换。原因在于，采用引用参数传递时，形参 x 与实参 m 共用同一存储单元，同样形参 y 与实参 n 共用同一存储单元。所以当在 SwapInt 方法内交换 x 和 y 的值时，相当于交换 m 和 n 的值。

说明

使用 ref 关键字时请注意：①ref 关键字仅对跟在它后面的参数有效，而不能应用于整个参数表，例如，SwapInt 方法中的参数 x、y 都要加 ref 修饰；②在调用方法时，实参变量也用 ref 修饰，因为是引用参数，所以要求实参与形参的数据类型必须完全匹配，而且实参必须是变量，不能是常量或表达式；③在方法外，ref 修饰的实在参数在调用前必须明确赋值。

3. 输出多个引用类型的参数

有时候一个方法的计算结果有多个，而 return 语句一次只能返回一个结果，此时可以使用 out 关键字达到输出多个返回值的目的，使用 out 关键字表明该引用参数是用于输出

的。out 传递与 ref 类似，二者的区别是：ref 要求参数在传递之前必须初始化，out 则不要求初始化。

【例 3-9】 传递 out 引用类型参数举例。

```
01. namespace Example3_9
02. {
03.     class OutParameter
04.     {
05.         public static void OpArray(int[] a, out int max, out int min)
06.         {
07.             max = min = a[0];
08.             for (int i = 1; i < a.Length; i++ )
09.             {
10.                 if (a[i] > max) max = a[i];
11.                 if (a[i] < min) min = a[i];
12.             }
13.         }
14.     }
15.     class Example3_9
16.     {
17.         static void Main(string[] args)
18.         {
19.             int[] score = { 65, 78, 95, 76, 86, 59, 82 };
20.             int smax, smin;
21.             OutParameter.OpArray(score,out smax,out smin);
22.             Console.WriteLine("最高分：{0},最低分：{1}",smax ,smin);
23.             Console.Read();
24.         }
25.     }
26. }
```

【运行结果】

单击工具栏中的“开始”按钮，即可在控制台中输出如图 3-9 所示的结果。

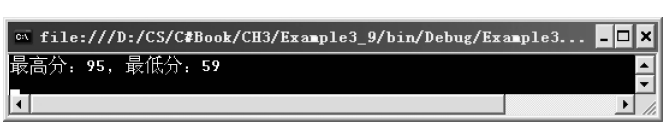

图 3-9　例 3-9 运行结果

【程序说明】

- OutParameter 类中定义了一个 OpArray 静态方法，该方法用于得到一个整型数组中的最大值和最小值。其中，形参 max 和 min 前面都加了 out 关键字，表明是输出类型引用参数。
- 第 21 行在 Main 函数中调用 OutParameter.OpArray 方法时，实参 smax 和 smin 前面也要加 out 关键字，它们分别接收形参输出的最大值和最小值。相比于 ref 关键字，实参 smax 和 smin 在调用前不必赋初始值。

4. 传递个数不确定的参数

当需要传递的参数个数不确定时，如求几个数的平均值，由于没有规定是几个数，运行程序时每次输入的数值个数很难确定。为了解决这个问题，C＃语言采用 params 关键字声明数组型参数，此时方法能够接收不同数量的参数。

【例 3-10】 传递 params 类型参数举例。

```
01. namespace Example3_10
02. {
03.     class Example3_10
04.     {
05.         public static void Average(out double avg, params int[] a)
06.         {
07.             int i;
08.             if (a.Length == 0)
09.             {
10.                 avg = 0.0;
11.                 return;
12.             }
13.             for (i = 0,avg = 0; i < a.Length; i++ )
14.                 avg += a[i];
15.             avg = avg / a.Length;
16.         }
17.         static void Main(string[] args)
18.         {
19.             double mavg;
20.             int[] array1 = { 1, 2, 3, 4, 5 };
21.             Average(out mavg);
22.             Console.WriteLine("0 个参数的平均值为：{0}",mavg);
23.             Average(out mavg, 5, 6, 7, 8);
24.             Console.WriteLine("5、6、7、8 的平均值为：{0}", mavg);
25.             Average(out mavg, array1);
26.             Console.WriteLine("array1 中所有元素的平均值为：{0}", mavg);
27.             Console.Read();
28.         }
29.     }
30. }
```

【运行结果】

单击工具栏中的“开始”按钮，即可在控制台中输出如图 3-10 所示的结果。

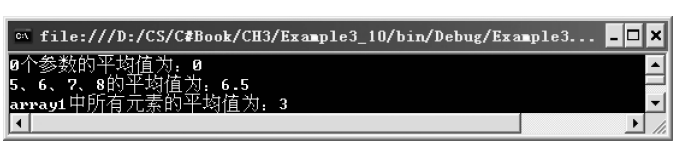

图 3-10 例 3-10 运行结果

【程序说明】

- 第 5 行定义了一个静态 Average 方法，该方法中第一个参数为 out 类型的，第二个参数为 params 数组类型。在此方法中，首先判断数组参数的长度是否为 0，如果是

直接返回平均值为0,否则先求所有元素的总和再求出平均值。

- 第21行调用Average方法时,params参数的个数为0个;第23行调用Average方法时,params参数为4个整数;第25行调用Average方法时,params参数为一个一维数组。可见params类型的参数在使用时的灵活。

说明

使用参数数组时请注意:①一个方法中只能声明一个params参数,如果还有其他常规参数,则params参数应放在参数列表的最后;②用params修饰符声明的参数是一个一维数组类型,不可以是多维数组;③由于params参数其实是一个数组,所以在调用时可以为参数数组指定零个或多个参数,其中每个参数的类型都应与参数数组的元素类型相同或能隐式转换;④params参数在方法内被作为一个数组处理,所以可以使用数组的长度属性来确定在每次调用中所传递参数的个数;⑤params参数在内部会进行数据的复制,所以方法内对参数数组元素的修改不会使方法外的数值发生变化;⑥不能将params修饰符与ref和out修饰符组合使用。

3.3.3 方法重载

方法重载是指调用同一方法名,但各方法中参数的数据类型、个数或顺序不同。只要类中有两个以上的同名方法,但是使用的参数类型、个数或顺序不同,调用时编译器就可以判断在哪种情况下调用哪种方法。这种技术非常有用,在项目开发过程中,我们会发现很多方法都需要使用重载技术。

【例3-11】 重载方法的使用。

```
01. namespace Example3_11
02. {
03.     class Example3_11
04.     {
05.         public static void Swap(ref int m, ref int n)
06.         {
07.             int temp = m;
08.             m = n;
09.             n = temp;
10.         }
11.         public static void Swap(ref double m, ref double n)
12.         {
13.             double temp = m;
14.             m = n;
15.             n = temp;
16.         }
17.         static void Main(string[] args)
18.         {
19.             int i1 = 1, i2 = 2;
20.             double d1 = 1.0, d2 = 2.0;
21.             Console.WriteLine("交换前: i1 = {0}, i2 = {1}", i1, i2);
22.             Swap(ref i1, ref i2);
23.             Console.WriteLine("交换后: i1 = {0}, i2 = {1}", i1, i2);
```

```
24.             Console.WriteLine("交换前: d1 = {0}, d2 = {1}", d1, d2);
25.             Swap(ref d1, ref d2);
26.             Console.WriteLine("交换后: d1 = {0}, d2 = {1}", d1, d2);
27.             Console.Read();
28.         }
29.     }
30. }
```

【运行结果】

单击工具栏中的“开始”按钮,即可在控制台中输出如图 3-11 所示的结果。

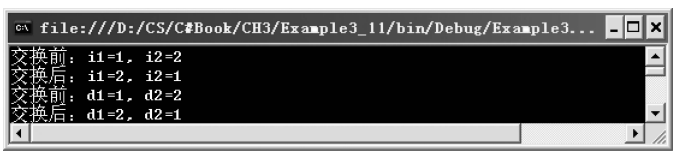

图 3-11　例 3-11 运行结果

【程序说明】

- 第 5 行和第 11 行分别定义了 Swap 方法,这两个方法只有参数类型不同,前者是 int 类型后者是 double 类型。
- 第 22 行和第 25 行进行方法调用时,实参的类型分别是 int 和 double,系统在调用时会自动找到最匹配的方法。

说 明

方法重载需要注意:①在.NET 公共语言规范中,方法的返回类型不足以对方法进行标识;②ref 和 out 的参数类型区别不足以标识方法,除此之外,不同的参数类型也能标识不同的方法。例如:

```
//合法的重载形式
int f1(int x, int y) {}与 int f1(ref int x, ref int y) {}可以
//ref 和 out 的参数类型区别不能标识方法
int f1(ref int x, ref int y) {}与 int f1(out int x, out int y) {}不可以
```

3.4　属性与索引器

为了实现良好的数据封装和数据隐藏,类的字段成员的访问属性一般设置成 private 或 protected,这样在类的外部就不能直接读/写这些字段成员了,通常的办法是提供 public 级的方法来访问私有的或受保护的字段。

但 C# 中提供了属性这个更好的方法,把字段域和访问它们的方法相结合。对类的用户而言,属性值的读/写与字段域语法相同;对编译器来说,属性值的读/写是通过类中封装的特别方法 get 访问器和 set 访问器实现的。

3.4.1　属性

在类中,属性的声明形式如下:

```
访问修饰符 属性类型 属性名
{
    get
    { 【读访问器语句块】 }
    set
    { 【写访问器语句块】 }
}
```

get 访问器：用于返回字段值，或用于计算并返回计算结果。例如：

```
class Student
{
    private int age;                                    //字段
    public int Age                                      //属性
    {
        get
        {
            return age;
        }
    }
}
```

set 访问器：没有返回值，它使用称为 value 的隐式参数，此参数的类型与属性的类型相同。例如：

```
class Student
{
    private int age;                                    //字段
    public int Age                                      //属性
    {
        set
        {
            age = value;
        }
    }
}
```

关于属性的说明如下：

(1) 当读取属性时，执行 get 访问器的代码块；当向属性分配一个新值时，执行 set 访问器的代码块。

(2) 只包含 get 访问器的属性称为只读属性，只包含 set 访问器的属性称为只写属性，同时具有这两个访问器的属性称为读/写属性。

(3) get 访问器的返回值类型与属性类型相同，所以在其语句块中 return 语句的返回值必须与属性类型相同或能隐式转换为属性类型。

(4) 属性也可以使用 static 关键字声明为静态属性。

当属性访问器中不需要其他逻辑时，也可以用简单的方式声明属性，这种方式称为自动实现的属性。自动实现的属性必须同时声明 get 和 set 访问器。如果希望声明只读属性，必须将 set 访问器声明为 private；如果希望声明只写属性，必须将 get 访问器声明为 private。

使用自动实现的属性可使属性声明变得更简单，因为这种方式不再需要声明对应的私有字段。例如：

```
class Student
{
    public int Age                                    //只读属性
    { get; private set;}
    public sting Name                                 //读写属性
    { get; set; }
}
```

上面这段代码中声明了两个自动实现的属性，其中 Age 是只读属性，Name 是读写属性。

【例 3-12】 属性的声明与使用。

```
01. namespace Example3_12
02. {
03.     public class Student
04.     {
05.         public string Name { get; set; }
06.         private DateTime birthday;
07.         private int grade = 0;
08.         public int Age
09.         {
10.             get { return DateTime.Now.Year - birthday.Year; }
11.         }
12.         public int Grade
13.         {
14.             get { return grade; }
15.             set
16.             {
17.                 if (value >= 0 && value <= 100)
18.                     grade = value;
19.             }
20.         }
21.         public Student()
22.         {
23.             Name = "李四";
24.             birthday = new DateTime(1988, 8, 8);
25.         }
26.     }
27.     class Example3_12
28.     {
29.         static void Main(string[] args)
30.         {
31.             Student stu = new Student();
32.             Console.WriteLine("姓名：{0},年龄：{1},考试成绩：{2}", stu.Name, stu.Age,
    stu.Grade);
33.             stu.Grade = 88;
34.             Console.WriteLine("姓名：{0},年龄：{1},考试成绩：{2}", stu.Name, stu.Age,
```

```
stu.Grade);
35.             Console.Read();
36.         }
37.     }
38. }
```


【运行结果】

单击工具栏中的“开始”按钮，即可在控制台中输出如图 3-12 所示的结果。

图 3-12　例 3-12 运行结果

【程序说明】

- Student 类中，首先定义了一个自动实现的属性 Name 表示姓名，定义了一个日期类型的私有字段 birthday 表示出生日期，还有一个 int 类型私有字段 grade 用来保存成绩。
- 第 8～11 行在 Student 类中定义了整型属性 Age，该属性为只读的，通过当前的年份与出生年份之差得到学生的年龄，此属性不与字段对应，只完成计算功能。
- 第 12～20 行定义了 Student 类中的 Grade 属性，该属性为可读可写，Grade 属性与私有字段 grade 相关联，通过该属性完成对成绩的存取。
- 类 Example3_12 的 Main 函数中，对学生信息的读取均通过属性来实现。

3.4.2　索引器

索引器是对属性的进一步扩展，用于封装内部集合或数组。索引器在语法上方便程序员将类、结构或接口作为数组进行访问。和属性一样，索引器也可以被看作是 get 和 set 访问器的组合体，它同样使用 return 语句为 get 访问器返回结果，使用 value 关键字为 set 访问器传递值。与属性的不同之处在于：

(1) 索引器以 this 关键字加数组下标[]进行定义，并通过数组下标的形式进行访问。

(2) 索引器的 get 访问器和 set 访问器带有参数(通常为整数类型或字符串类型)。

(3) 索引器不能是静态的。

【例 3-13】　索引器的声明与使用。

```
01. namespace Example3_13
02. {
03.     public class Student
04.     {
05.         public string Name { get; set; }
06.         private DateTime birthday;
07.         public int Age
08.         {
09.             get { return DateTime.Now.Year - birthday.Year; }
10.         }
```

```
11.         public Student(string sname, int year, int month, int day)
12.         {
13.             Name = sname;
14.             birthday = new DateTime(year, month, day);
15.         }
16.     }
17.     public class Students
18.     {
19.         private Student[] stuArray;
20.         public int Length
21.         {
22.             get { return stuArray.Length; }
23.         }
24.         public Student this[int index]
25.         {
26.             get { return stuArray[index]; }
27.             set { stuArray[index] = value; }
28.         }
29.         public Students(int length)
30.         {
31.             stuArray = new Student[length];
32.         }
33.     }
34.     class Example3_13
35.     {
36.         static void Main(string[] args)
37.         {
38.             Students stuList = new Students(2);
39.             stuList[0] = new Student("李四", 1988, 8, 8);
40.             stuList[1] = new Student("王五", 1986, 6, 6);
41.             for(int i = 0;i < stuList .Length ;i++ )
42.                 Console.WriteLine("姓名: {0},年龄: {1}", stuList[i].Name, stuList[i].
Age);
43.             Console.Read();
44.         }
45.     }
46. }
```

【运行结果】

单击工具栏中的“开始”按钮,即可在控制台中输出如图 3-13 所示的结果。

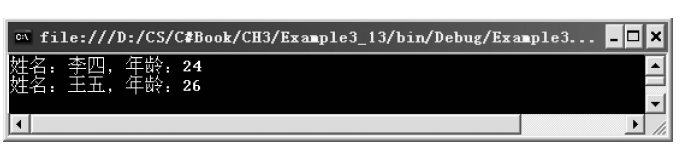

图 3-13　例 3-13 运行结果

【程序说明】

- 第 17～33 行定义了一个 Students 类,第 19 行在该类中定义了一个私有字段 stuArray,该字段是 Student 类型的数组,用来保存多个学生的信息。

- 第 24～28 行在 Students 类中定义了一个索引器，该索引器的数据类型为 Student 类类型，访问下标为 int 类型，可以读取一个学生的信息，也可以将一个学生的信息放到数组 stuArray 的指定位置中。
- 通过 Students 类中索引器的使用，在 Main 函数中就可以通过下标对类对象进行存取。

3.5 结　　构

结构是由一系列相关的、但类型不一定相同的变量组织在一起而构成的数据表示形式。结构中可以包含构造函数、常量、字段、方法、属性、事件和嵌套类型等，但如果要同时包括上述几种成员，则应考虑使用类。凡是定义为结构的，都可以用类来定义。

结构和类的主要区别在于：结构是值类型，而类是引用类型。

3.5.1 结构的定义及特点

在 C# 中，使用 struct 关键字定义结构。定义结构的一般形式为：

```
访问修饰符 struct 结构名
{
    【结构体】
}
```

访问修饰符与类的访问修饰符相同。

结构具有以下特点：

(1) 结构是值类型。

(2) 结构的实例化可以不使用 new 运算符。

(3) 结构可以声明构造函数，但它们必须带参数。

(4) 一个结构不能从另一个结构或类继承，所有结构都直接继承自 System.ValueType，类继承自 System.Object。

(5) 结构可以实现接口。

(6) 在结构中初始化实例字段是错误的。

3.5.2 结构的使用

用结构实现的都可以用类来实现，为什么还要区分类和结构呢？这是因为，对于一些简单的数据类型，在程序执行上使用结构能够得到比类高得多的执行效率。

【例 3-14】 结构使用示例。

```
01. namespace Example3_14
02. {
03.     public struct StructRect
04.     {
05.         public double width;
06.         public double height;
07.         public StructRect(double w, double h)
```

```
08.         {
09.             width = w;
10.             height = h;
11.         }
12.         public double Area()
13.         {
14.             return width * height;
15.         }
16.     }
17.     class Example3_14
18.     {
19.         static void Main(string[] args)
20.         {
21.             StructRect sRect1 = new StructRect(3, 3);
22.             StructRect sRect2 = new StructRect();
23.             StructRect sRect3;
24.             sRect3.width = 4.0;
25.             sRect3.height = 4.0;
26.             Console.WriteLine("Rectangle1's area is:{0}", sRect1.Area());
27.             Console.WriteLine("Rectangle2's area is:{0}", sRect2.Area());
28.             Console.WriteLine("Rectangle3's area is:{0}", sRect3.Area());
29.             Console.Read();
30.         }
31.     }
32. }
```

【运行结果】

单击工具栏中的“开始”按钮，即可在控制台中输出如图 3-14 所示的结果。

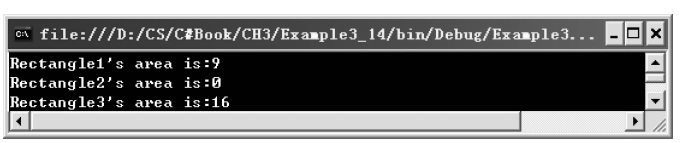

图 3-14　例 3-14 运行结果

【程序说明】

- 第 3～16 行定义了结构 StructRect，该结构内包含两个字段、一个构造函数和一个求面积的方法。
- 第 21～23 行在 Main 函数中，分别声明了 StructRect 类型的三个对象，第一个对象调用了带参构造函数，第二个对象调用了无参构造函数，第三个对象没有使用 new 进行实例化。虽然结构中没有默认的无参构造函数，但可以调用无参构造函数，并且将数据成员设置为对应类型的默认值。
- 如果再声明一个 StructRect 类型的对象 sRect4，执行 sRect4 = sRect1；sRect4.width = 4.0；两条语句后，sRect1.Area()和 sRect4.Area()的值分别是多少呢？如果将 StructRect 的类型由结构变为类，上面的两个结果又是怎样的呢？
- 值类型在堆栈上分配地址，引用类型在堆上分配地址。堆栈的执行效率要比堆高，因此结构的效率高于类。原因在于，堆用完后由 .NET 的垃圾收集器自动回收，程

序大量使用堆，将导致程序性能的下降。

3.6 操作符重载

C#中操作符重载是指允许用户使用用户自定义的类型编写表达式的能力，这样做的好处是使用自定义的数据类型就像使用基本数据类型一样自然、合理。

例如，通常需要编写类似于以下内容的代码，以将两个数字相加。很明显，sum 是两个数字之和。

```
int i = 5, j = 6; int sum = i + j;
```

如果可以使用代表复数的自定义类型来编写相同类型的表达式，那当然是最好不过了：

```
Complex i = 5, j = 6; Complex sum = i + j;
```

运算符重载允许为用户定义的类型重载诸如“+”这样的运算符。如果不进行重载，则用户需要编写以下代码：

```
Complex i = new Complex(5); Complex j = new Complex(6);
Complex sum = Complex.Add(i, j);
```

此代码可以很好地运行，但 Complex 类型并不能像语言中的预定义类型那样发挥作用。

通过操作符重载可以让 struct、class、Interface 等能够进行运算。

操作符重载语法：

```
public static 返回值类型 operator 操作符(操作参数)
```

例如：

```
public static Hour operator + (Hour lhs, Hour rhs){...}
```

C#中操作符重载和 C++比较起来，有很大的不同。定义的时候重载操作符方法必须是 static，而且至少有一个参数（一目和二目分别是一个和两个），C#和 C++比起来，最重要的特征是：<、>；==、!=；true、false 必须成对出现，即重载了“<”就必须重载“>”，重载了“==”就必须重载“！=”，重载了“true”就必须重载“false”。

【例 3-15】 操作符重载示例。

```
01. namespace Example3_15
02. {
03.     struct Hour
04.     {
05.         private int hvalue;
06.         public Hour(int ivalue)
07.         {
08.             this.hvalue = ivalue;
09.         }
10.         public int HValue
11.         {
```

```
12.             get
13.             { return hvalue; }
14.             set
15.             { hvalue = value; }
16.         }
17.         public static Hour operator +(Hour h1, Hour h2)
18.         {
19.             Hour h3 = new Hour();
20.             h3.HValue = h1.HValue + h2.HValue;
21.             return h3;
22.         }
23.         public static Hour operator +(Hour h1, int h2)
24.         {
25.             return new Hour(h2) + h1;
26.         }
27.         public static Hour operator ++ (Hour hrValue)
28.         {
29.             hrValue.HValue++ ;
30.             return hrValue;
31.         }
32.         public static bool operator == (Hour hr1, Hour hr2)
33.         {
34.             return hr1.HValue == hr2.HValue;
35.         }
36.         public static bool operator != (Hour hr1, Hour hr2)
37.         {
38.             return hr1.HValue != hr2.HValue;
39.         }
40.     }
41.     class Example3_15
42.     {
43.         static void Main(string[] args)
44.         {
45.             Hour hrValue1 = new Hour(10);
46.             Hour hrValue2 = new Hour(20);
47.             Hour hrSum = hrValue1 + hrValue2;
48.             Console.WriteLine("hrValue1 + hrValue2 = {0}",hrSum.HValue);
49.             Hour hrSumInt = hrValue1 + 10;
50.             Console.WriteLine("hrValue1 + 10 = {0}", hrSumInt.HValue);
51.             Console.ReadLine();
52.         }
53.     }
54. }
```

【运行结果】

单击工具栏中的“开始”按钮，即可在控制台中输出如图 3-15 所示的结果。

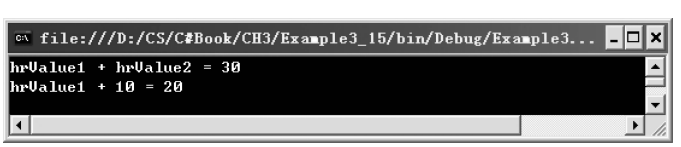

图 3-15　例 3-15 运行结果

【程序说明】

- 结构 Hour 中包含一个字段和一个属性，实现了 4 种操作符的重载。
- 第 17～23 行实现对两个 Hour 类型的变量进行相加的重载；第 24～27 行实现一个 Hour 类型的变量和一个整型变量相加的重载。
- 第 28～32 行实现对一元操作符“++”的重载，使 Hour 类型的变量的 hvalue 字段值加 1。
- 第 33～36 行实现对二元操作符“==”的重载，用于判定两个 Hour 类型的变量是否相等。按照 C# 中的操作符重载规则，重载“==”就需要重载“!=”，所以 37～40 行实现了对“!=”运算符的重载。

3.7 问 题 解 决

经过对类的基本知识的学习，使我们对导入问题的解决方法有了另一种思考。可以采取下面的步骤来加以解决：

(1) 声明轿车类，使用成员方法定义其鸣笛行为。

(2) 声明公共汽车类，使用成员方法定义其鸣笛行为。

根据以上思路，解决问题的完整代码如下：

【例 3-16】 解决导入问题。

```
01. namespace Example3_16
02. {
03.     public class Car
04.     {
05.         private string id;                          //汽车型号
06.         private double engine;                      //发动机型号
07.         private int wheels;                         //车轮个数
08.         private int whlType;                        //车轮型号
09.         public Car(string cid, double e, int whls, int wtype)
10.         {
11.             id = cid;
12.             engine = e;
13.             wheels = whls;
14.             whlType = wtype;
15.         }
16.         public void PrintInfo()
17.         {
18.              Console.WriteLine("{0}:排气量{1},车轮数{2},车轮型号{3}", id, engine,
    wheels, whlType);
19.         }
20.         public void Whistle()
21.         {
22.             Console.WriteLine("小轿车: 嘀嘀!");
23.         }
24.     }
25.     public class Bus
26.     {
```

```
27.         private string id;
28.         private double engine;
29.         private int wheels;
30.         private int whlType;
31.         public Bus(string bid, double e, int whls, int wtype)
32.         {
33.             id = bid;
34.             engine = e;
35.             wheels = whls;
36.             whlType = wtype;
37.         }
38.         public void PrintInfo()
39.         {
40.              Console.WriteLine("{0}:排气量{1},车轮数{2},车轮型号{3}", id, engine,
    wheels, whlType);
41.         }
42.         public void Whistle()
43.         {
44.             Console.WriteLine("大公交: 嘟嘟!");
45.         }
46.     }
47.     class Example3_16
48.     {
49.         static void Main(string[] args)
50.         {
51.             Car c1 = new Car("红旗 2012", 1.6, 4, 20);
52.             Car c2 = new Car("中华 2012", 1.4, 4, 20);
53.             Bus b1 = new Bus("黄海 2012", 4.2, 6, 50);
54.             c1.PrintInfo();
55.             c1.Whistle();
56.             c2.PrintInfo();
57.             c2.Whistle();
58.             b1.PrintInfo();
59.             b1.Whistle();
60.             Console.Read();
61.         }
62.     }
63. }
```

【运行结果】

单击工具栏中的“开始”按钮,即可在控制台中输出如图 3-16 所示的结果。

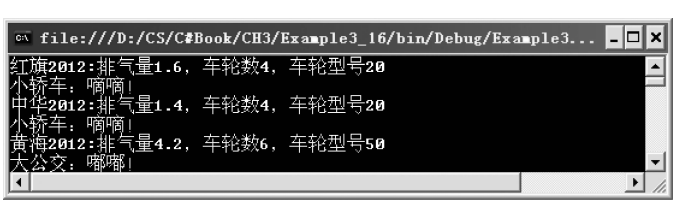

图 3-16　例 3-16 运行结果

【程序说明】

- 程序中定义了轿车类 Car 和公共汽车类 Bus，通过类的构造函数完成车辆信息的初始化，PrintInfo()方法输出车辆信息，Whistle()方法输出鸣笛信息。
- 在主程序中实例化了两个 Car 对象和一个 Bus 对象，通过调用对象的方法输出各自的信息和实现各自的鸣笛行为。
- 通过本例可以看到，通过类对车辆对象进行封装后，大大提高了车辆的“生产效率”，操作上非常方便、快捷。

小　　结

类是面向对象程序设计的基本元素，是 C# 中最重要的一种数据类型。类可以包含字段成员、方法成员以及其他的嵌套类型。构造函数、析构函数、属性、索引器、事件和操作符都可以视为特殊的方法成员，它们在使用中都有着各自的特点。

对象的生命周期从构造函数开始，到析构函数结束；类的构造函数用于对象的初始化，而析构函数用于对象的销毁。利用事件和索引函数提供的访问方法，可以隐藏数据处理的细节，更好地实现对象的封装性。通过操作符重载，C# 预定义的操作符就能直接作用于各种自定义类型。

类的大部分成员用法也适用于结构，唯一的区别在于类是引用类型，而结构是值类型。对于方法来说，学习的重点在于掌握形参和实参的区别，以及不同类型参数的使用方法。

习　　题

一、选择题

1. C# 语言的核心是面向对象编程(OOP)，所有 OOP 语言都应至少具有三个特性：(　　)。

A. 类、对象和方法　　B. 封装、继承和多态

C. 封装、继承和派生　　D. 封装、继承和接口

2. 在类的定义中，类的(　　)描述了该类的对象的行为特征。

A. 方法　　B. 类名　　C. 所属的命名空间　　D. 私有域

3. C# 可以采用下列(　　)技术来进行对象内部数据的隐藏。

A. 静态成员　　B. 变量　　C. 属性　　D. 装箱技术

4. C# 的构造函数分为实例构造函数和静态构造函数，实例构造函数可以对(　　)进行初始化，静态构造函数只能对(　　)进行初始化。

A. 静态成员　　B. 非静态成员

C. 静态成员或非静态成员　　D. 常量成员

5. C# 实现了完全意义上的面向对象，所以它没有(　　)，任何数据域和方法都必须封装在类体中。

A. 全局变量　　B. 全局常数

C. 全局方法　　D. 全局变量、全局常数和全局方法

6. 方法中的值参数是(　　)的参数。

A. 按引用传递　　B. 按值传递

C. 按地址传递　　D. 不传递任何值

7. 假设 class Myclass 类的一个方法的签名为：public void Max (out int max, params int[ ]a),m1 是 Myclass 类的一个对象,maxval 是一个 int 型的值类型变量,arrayA 是一个 int 型的数组对象,则下列调用该方法有错的是(　　)。

A. m1. Max (out maxval)　　B. m1. Max (out maxval,4,5,3)

C. m1. Max (out maxval,ref arrayA)　　D. m1. Max (out maxval,3,3.5)

8. 类 Myclass 中有下列方法定义：

```
public void testParams(params int[ ]a)
{  Console.Write("使用 Params 参数!"); }
public void testParams(int x, int y)
{  Console.Write("使用两个整型参数!"); }
```

请问上述方法重载有无二义性？若没有,则下列语句的输出为(　　)。

```
Myclass m1 = new Myclass();
m1. testParams(1);
m1. testParams(1,2);
m1. testParams(1,2,3);
```

A. 有语义二义性

B. 使用 Params 参数!使用两个整型参数!使用 Params 参数!

C. 使用 Params 参数!使用 Params 参数!使用 Params 参数!

D. 使用 Params 参数!使用两个整型参数!使用两个整型参数!

9. 下面有关属性的说法,不正确的是(　　)。

A. 属性可以有默认值

B. 属性可以不和任何字段相关联

C. 属性的 get 访问函数是不带参数的特殊方法

D. 属性的 set 访问函数是没有返回值的特殊方法

10. 分析下列程序：

```
public class Myclass
{
    private string _sData = "";
    public string sData {set {_sData = value;}}
}
```

在 Main 函数中,成功创建该类的对象 m1 后,下列哪些语句是合法的？(　　)

A. Console. Write(m1. sData);　　B. m1. _sData = "Good!";

C. m1. set(m1. sData);　　D. m1. sData = "Good!";

11. 下面有关结构的说法,正确的是(　　)。

A. 结构是轻量级引用类型　　B. 结构有默认的构造函数

C. 结构有析构函数　　D. 结构的效率低于类

12. 以下不能作为复合赋值操作符被重载的是(　　)。

A. *=　　B. +=　　C. &=　　D. ~=

二、简答题

1. 对比面向对象程序设计和面向过程程序设计之间的优劣性。

2. 写出下面程序的运行结果。

```
namespace Lx3_0202
{
    class Myclass
    {
        public void SortArray(int[] a)
        {
            int i, j, pos, tmp;
            for (i = 0; i < a.Length - 1; i++ )
            {
                for (pos = j = i; j < a.Length; j++ )
                    if (a[pos] > a[j]) pos = j;
                if (pos != i)
                {
                    tmp = a[i];
                    a[i] = a[pos];
                    a[pos] = tmp;
                }
            }
        }
    }
    class Test
    {
        static void Main()
        {
            Myclass m = new Myclass();
            int[] score = { 87, 89, 56, 90, 100, 75, 64, 45, 80, 84 };
            m.SortArray(score);
            for (int i = 0; i < score.Length; i++ )
            {
                Console.Write("score[{0}] = {1}, ", i, score[i]);
                if (i == 4) Console.WriteLine();
            }
            Console.Read();
        }
    }
}
```

3. 写出下面程序的运行结果。

```
namespace Lx3_0203
{
    class Myclass
    {
        public void Swap1(string s, string t)
```

```
        {
            string tmp;
            tmp = s;
            s = t;
            t = tmp;
        }
        public void Swap2(ref string s, ref string t)
        {
            string tmp;
            tmp = s;
            s = t;
            t = tmp;
        }
    }
    class Test
    {
        static void Main()
        {
            Myclass m = new Myclass();
            string s1 = "ABCDEFG", s2 = "134567";
            m.Swap1(s1, s2);
            Console.WriteLine("s1 = {0}", s1);
            Console.WriteLine("s2 = {0}", s2);
            m.Swap2(ref s1, ref s2);
            Console.WriteLine("s1 = {0}", s1);
            Console.WriteLine("s2 = {0}", s2);
            Console.Read();
        }
    }
}
```

4. 写出下面程序的运行结果。

```
namespace Lx3_0204
{
    class Myclass
    {
        public void Change(string s)
        {
            s = s + "_ch01";
        }
        public void Change(ref string s)
        {
            s = s + "_ch02";
        }
        public void Change(string s1, out string s2)
        {
            s1 = s1 + "_ch03";
            s2 = s1;
        }
    }
```

```
    class Test
    {
        static void Main()
        {
            Myclass m = new Myclass();
            string s1, s2;
            s1 = "Good";
            m.Change(s1);
            Console.WriteLine("s1 is {0}", s1);
            m.Change(ref s1);
            Console.WriteLine("s1 is {0}", s1);
            m.Change(s1, out s2);
            Console.WriteLine("s1 is {0}, s2 is {1}", s1, s2);
            Console.Read();
        }
    }
}
```

5. 找出下面代码中的错误。

```
class Program
{
    int x = 3;
    static int y = 4;
    const int z = 5;
    public Program()
    {
        x = 10; y = 20;
    }
    static Program()
    {
        x = 5; y = 10;
    }
    static void Main()
    {
        Program p = new Program();
        Console.WriteLine(p.x);
        Console.WriteLine(p.y);
        Console.WriteLine(p.z);
    }
}
```

三、编程题

1. 定义一个描述复数的类，并实现复数的输入和输出。设计两个方法分别完成复数的加法和减法运算。

2. 定义一个描述学生基本情况的类，数据成员包括学号、姓名、数学成绩和C#成绩，成员函数包括输入学生基本信息和成绩、求出每门课程的平均成绩和输出数据。

3. 设有一个描述坐标点的MyPoint类，其私有变量x和y代表一个点的横纵坐标值。编写程序实现以下功能：利用属性完成对坐标值的读写，利用成员函数输出点的坐标值，利

用成员函数实现点的水平和垂直移动，并用成员函数输出修改后的坐标值。

4. 定义一个描述复数的类，通过重载运算符：＋、－、＊、/，直接实现两个复数的4种运算，在主程序中进行测试。

5. 定义一个描述顾客信息的结构体，包括顾客的会员卡号、姓名、消费积分，当顾客购买商品时根据消费额增加积分，当顾客结账时根据积分进行相应折扣的打折。