第3章

chapter 3

面向对象程序设计

面向对象(OO,Object Oriented)是当前计算机界关心的重点,它是20世纪90年代软件开发方法的主流。起初,面向对象是专指在程序设计中采用封装、继承、抽象等设计方法。可是,这个定义显然已不再适合现在的情况。面向对象的思想已经涉及软件开发的各个方面。如面向对象的分析(OOA,Object Oriented Analysis)、面向对象的设计(OOD,Object Oriented Design)以及我们经常说的面向对象的编程实现(OOP,Object Oriented Programming)。面向对象的概念和应用已经超越了程序设计和软件开发的限制,扩展到了很宽的范围,如数据库系统、交互式界面、应用结构、应用平台、分布式系统、网络管理结构、CAD技术、人工智能等领域。

本章主要介绍Visual C#.NET面向对象程序设计基础知识,包括类的定义、继承与多态、集合、委托与事件等内容。通过本章的学习,读者能够掌握面向对象程序设计的基本理念。这些理念是目前程序开发技术的必备要求,应实现以下目标。

- 理解面向对象程序设计的思想和概念。
- 理解类、对象、继承、多态、方法等概念。
- 掌握类的定义方法。
- 掌握继承、接口、委托与事件的使用方法。
- 掌握运算符重载的方法。

3.1 面向对象编程简介

传统的程序设计是面向过程的,其核心是功能的分解。面向过程的程序设计,首先采用自顶向下、逐步细化的方法将问题分成若干模块,然后再根据功能模块设计数据结构,最后编写对数据进行操作的过程或函数。在这种方法中,数据和施加在数据上的操作是分开的。在大型的结构化程序中,一个数据结构可能被多个过程处理,数据结构的改变将导致相关模块的修改,程序可重用性(重用性是指同一事物不经修改或稍加修改就可多次重复使用的性质。软件重用性是软件工程追求的目标之一)差,开发和维护代价高。这时候,面向对象语言作为一种降低复杂性的工具产生了,面向对象程序设计也随之产生。现在,面向对象的程序设计已经发展得相当完善了。

软件工程学家Coad和Youydon曾给出面向对象的一个简单定义:

面向对象=对象+类+继承+通信

如果一个软件系统使用上述 4 个概念进行设计并加以实现，则认为这个软件系统是面向对象的。

面向对象技术的基本观点可以概括如下。

（1）客观世界由对象组成。任何客观实体都是对象，复杂对象可以由简单对象组成。

（2）具有相同数据和操作的对象可以归纳成类。对象是一个类的实例。

（3）类可以派生出子类。子类除了继承父类的全部特性外，还可以有自己的特性。

（4）对象之间的联系通过消息传递来维系。由于类的封装性，使它具有某些对外界不可见的数据。这些数据只能通过消息请求调用可见方法来访问。

1. 类和对象

在面向对象程序设计中，对象（Object）是系统中的基本运行实体。它是有特殊属性（数据）和行为方式（方法）的实体。对象由两个元素构成：第一个元素是一组包含数据的属性，第二个元素是允许对属性中包含的数据进行操作的方法。也可以说，对象是将某些数据代码和对该数据的操作代码封装起来的模块。对象是有特殊属性（数据）和行为方式（方法）的逻辑实体。

在一个面向对象的系统中，对象是运行期的基本实体。它可以用来表示一个人、一个银行账户或者一张数据表格，以及其他需要被程序处理的东西。它也可以用来表示用户定义的数据，如一个向量、时间或者列表。在面向对象程序设计中，问题的分析一般以对象及对象间的自然联系为依据。如前所述，对象在内存中占有一定空间，并且具有一个与之关联的地址，就像 C 语言中的结构一样。

当一个程序运行时，对象之间会通过互发消息来相互作用。例如，程序中包含一个 customer 对象和一个 account 对象，而 customer 对象可能会向 account 对象发送一个消息去查询其银行账目。每个对象都包含数据以及操作这些数据的代码。即使不了解彼此的数据和代码的细节，对象之间依然可以相互作用，需要了解的只是对象所能接受的消息的类型以及对象返回的响应的类型，虽然不同的人会以不同的方法来实现它们。

类（Class）是对具有公共的方法和一般特殊性的一组基本相同的对象的描述。一个类实质上定义的是一种对象类型。它由数据和方法构成，用来描述属于该类型的所有对象的性质。对象在执行过程中，由其所属的类动态生成。一个类可以生成不同的对象。在面向对象的程序设计中，对象是构成程序的基本单位。每个对象都属于某一个类。对象也可称为类的一个实例（Instance）。

从理论上讲，类是一个 ADT（抽象数据类型）的实现。信息隐蔽原则表明，类中的所有数据都是私有的。类的公共接口是由两种类型的类方法组成的：一种是返回有关实例状态的抽象辅助函数，另一种是用于改变实例状态的变换过程。

一个类可以由其他已存在的类派生出来。类与类之间按具体情况以层次结构组织起来。在这种层次结构中，处于上层的类称为父类或基类，处于下层的类称为子类或派生类。

抽象类（Abstract Class）是一种不能建立实例的类。抽象类将有关的类组织在一起，由它提供一个公共的根，其他一系列的子类都从这个根派生出来。抽象类刻画出了公共行为的特征，并将这些特征传给它的子类。通常，一个抽象类只描述与这个类有关的操作接口或这些操作的部分实现，完整的实现留给一个或几个子类，即可用抽象类作为派生类的基类。抽象类的常见用途是，用来定义一种协议（或概念）。

类与对象的关系：集合—成员，抽象描述—具体实例。

事实上，对象就是类的类型变量。定义了一个类，就可以创建这个类的多个对象。每个对象都与一组数据相关，而这组数据的类型在类中进行定义。因此，一个类就是一组具有相同类型的对象的抽象。例如，芒果、苹果和橘子都是 fruit 类的对象。类是用户定义的数据类型，但在程序设计语言中，它和内建的数据类型行为相同。比如，创建一个类对象的语法和创建一个整数对象的语法一模一样。如果 fruit 被定义为一个类，那么语句：

```
fruit apple;
```

就创建了一个 fruit 类的对象 apple。

2. 方法和消息

程序语句操纵一个对象来完成相应的操作。与对象有关的完成相应操作的程序语句称为方法(Method)。方法是对象本身内含的执行特定操作的函数或过程。方法的内容是不可见的，用户不必过问，只要执行它就可以了。方法的操作范围只能是对象内部的数据或对象可以访问的数据。

消息(Message)用来请求对象进行某些处理或回答某些请求。消息统一了数据流和信息流。在面向对象的程序设计中，通过消息来请求对对象进行操作。对象间的联系(或称相互作用)也是通过消息来完成的。消息只包括发送者的请求，它不指示接收者具体该如何去处理这些消息。对象接收一个消息后，由该对象所含的方法决定该对象如何处理消息，即对象由消息控制操作。

一个对象可以接收不同形式、不同内容的多个消息，相同形式的消息可以送给不同的对象，不同的对象对于形式相同的消息可以有不同的解释，做出不同的反映。因此，只要给出对象的所有消息模式及对应于每个消息的处理方法，也就是定义了一个对象的外部特征。

3. 继承性

继承性(Inheritance)指的是一个新类可以从现有的类中派生出来。新类具有父类中所有的特性，它直接继承了父类的方法和数据。新类的对象可以调用该类及父类的成员变量和成员函数。继承是可以让某个类型的对象获得另一个类型对象的属性的方法，它支持按级分类的概念。例如，灰喜鹊属于飞鸟类，也属于鸟类。继承性是自动共享类、子类和对象中的方法和数据的机制。合理使用继承可以减少很多的重复劳动。

在 OOP 中，继承的概念很好地支持了代码的重用性(reusability)。也就是说，我们可以向一个已经存在的类添加新的特性，而不必改变这个类。这可以通过从这个已存在的类中派生一个新类来实现。这个新类将具有原来那个类的特性，同时兼具新的特性。而继承机制的魅力和强大就在于它允许程序员利用已经存在的类，并且可以以某种方式修改这个类，而不会影响其他的东西。注意，每个子类都只定义这个类所有的特性。如果没有按级分类，每个类都必须显式地定义它所有的特性。

4. 封装性

任何程序都包含两个部分：代码和数据。在 SP 模式中，数据在内存中进行分配，并由子程序和函数代码处理；而在 OOP 模式中则是将处理数据的代码、数据的声明和存储封装在一起的。一个对象中的数据和代码相对于程序的其余部分是不可见的。它能防止那些非期望的交互和非法的访问。

封装(Encapsulation)就是将对象的属性和方法封装到具有适当定义接口的容器中。对象接口提供的方法和属性应使对象能够如期使用。

封装的功能取决于两个重要的概念：模块化和信息隐藏。模块化是对象自给自足的特性，它不会访问定义接口以外的其他对象。信息隐藏是指将对象的信息限制在对象接口使用范围内，删除对象中仅供对象内部操作的信息。封装是一种信息隐蔽技术。用户只能见到对象封装界面上的信息，对象内部对用户是隐蔽的。封装的目的在于将对象的使用者和设计者分开，使用者不必知道行为实现的细节，只需用设计者提供的消息来访问该对象即可。

5．多态性

多态性(Polymorphism)是指同一个消息为不同的对象所接收时，可导致完全不同的行为的现象。所谓多态是指事物具有不同形式的能力。对于不同的实例，某个操作可能会有不同的行为，这个行为依赖于所要操作的数据的类型。比如说加法操作，如果操作的数据是数，则对两个数求和；如果操作的数据是字符串，则连接两个字符串。

多态性可使公共的信息传送给基类对象及所有的派生类的对象，允许每一个基类的对象按适合于其定义的方式响应信息格式。

多态性有时也指方法的重载。方法的重载是指同一个方法名在上下文中有不同的含义，它是该类以统一的方式处理不同数据类型的一种手段。方法的重载是静态的，这是因为，在实现类和编写方法之前，需要考虑将要遇到的所有数据类型。子类在动态运行时提供了更丰富的多态性。多态性的表现为编译时的多态性——重载，如函数重载、运算符重载；运行时的多态性——虚函数。

从对象接收消息后的处理方式看，多态性指的是同一个消息被不同的对象接收时解释为不同含义的能力。也就是说，同样的消息被不同的类对象接收时，会产生完全不同的行为。利用多态性，用户能发送一般形式的消息，而将其所有实现的细节留给接收消息的对象去解决。

多态机制使具有不同内部结构的对象可以共享相同的外部接口。这意味着，虽然针对不同对象的具体操作不同，但通过一个公共的类，可以通过相同的方式予以调用。多态在实现继承的过程中被广泛应用。面向对象程序设计语言支持多态，术语称之为 one interface multiple method（ 一个接口，多个实现）。简单来说，多态机制允许通过相同的接口引发一组相关但不相同的动作。通过这种方式，可以减少代码的复杂度。在某个特定的情况下应该做出怎样的动作，这由编译器决定，而不需要程序员手动干预。

6．抽象

从许多事物中舍弃个别的、非本质性的特征，抽取共同的、本质性的特征，就叫作抽象(abstraction)。抽象是形成概念的必要手段。抽象的原则具有两方面的意义：第一，尽管问题域中的事物很复杂，但是分析员并不需要了解和描述它们的一切，只需要分析研究其中与系统目标有关的事物及其本质性特征即可；第二，通过舍弃个体事物在细节方面的差异，抽取其共同特征而得到一批事物的抽象概念。抽象的原则是面向对象方法中使用最为广泛的原则。在软件开发领域中，早在面向对象方法出现之前就已经开始运用抽象的原则。那时抽象的原则主要是过程抽象和数据抽象。

抽象指仅表现核心的特性而不描述背景细节的行为。类使用了抽象的概念，并且被定

义为一系列抽象的属性(如尺寸、重量和价格)以及操作这些属性的函数。类封装了将要被创建的对象的所有核心属性。因为类使用了数据抽象的概念,所以它们被称为抽象数据类型(ADT)。

3.2 类

类是面向对象编程的基本单位,它是一种包含数据成员、函数成员和嵌套类型的数据结构。类的数据成员有常量、域和事件。函数成员包括方法、属性、索引指示器、运算符、构造函数和析构函数。类和结构同样都包含了自己的成员,它们之间最主要的区别在于:类是引用类型,而结构是值类型。类支持继承机制,通过继承,派生类可以扩展基类的数据成员和函数方法,进而达到代码重用和设计重用的目的。

3.2.1 类的声明

类的定义形式为:

```
Class classname
{
    类成员;
}
```

下面的示例定义了一个名为 SampleClass 的类,并创建了该类的一个实例,即一个称为 sampleClass1 的对象。因为 C#要求在类中定义 Main 函数,所以下面的代码还定义了一个 Program 类,但是该类并不用于创建对象。

```
class SampleClass
{
    public void SayHello()
    {
        Console.WriteLine("Hello, World!");
    }
}
class Program
{
    static void Main(string[] args)
    {
        SampleClass sampleClass1 = new SampleClass();  // 创建一个对象
        sampleClass1.SayHello();                        // 调用方法
    }
}
```

从根本上而言,类就是自定义数据类型的蓝图。定义类之后,便可通过将类加载到内存中来加以使用。已加载到内存中的类称为对象或实例。可以通过使用 C#关键字 new 来创建类的实例。

上面的类只定义了一个方法 SayHello()。类的成员除了方法之外,还有属性和字段。属性和字段的区别在于:属性是逻辑字段,是字段的扩展,它源于字段,属性并不占用实际

的内存；字段则占用内存位置及空间。最直接地说，属性是被外部使用的，字段是被内部使用的，例如：

```
public class Computer
{
    private string name;
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
    public string 主板 = "技嘉主板";
    public string 硬盘 = "希捷硬盘";
    public string 内存 = "IBM";
    public string 显卡 = "影驰";
    //其他组成元素
    public Computer()
    { }
    public Computer(string name)
    {
        this.Name = name;
    }
}
```

C#用多种修饰符来表达类的不同性质。根据其保护级，C#的类有以下5种不同的限制修饰符。

（1）public：可以被任意类存取。

（2）protected：只可以被本类和其继承子类存取。

（3）internal：只可以被本组合体(Assembly)内所有的类存取。组合体是C#语言中类被组合后的逻辑单位和物理单位，其编译后的文件扩展名往往是.dll或.exe。

（4）protected internal：唯一的一种组合限制修饰符，它只可以被本组合体内所有的类和这些类的继承子类所存取。

（5）private：只可以被本类所存取。

如果不是嵌套的类，命名空间或编译单元内的类只有public和internal两种修饰。

new修饰符只能用于嵌套的类，它表示对继承父类同名类型的隐藏。

abstract用来修饰抽象类，它表示该类只能作为父类被用于继承，而不能进行对象实例化。抽象类可以包含抽象的成员，但这并不是必须的。abstract不能和new同时使用。下面是抽象类用法的伪代码。

```
abstract class A
{
    public abstract void F( );
}
abstract class B: A
{
    public void G( ) {   }
}
```

```
class C: B
{
    public override void F( )
    {
        //方法 F 的实现
    }
}
```

抽象类 A 内含一个抽象方法 F()，它不能被实例化。类 B 继承自类 A，其内包含了一个实例方法 G()，但并没有实现抽象方法 F( )。所以，类 B 仍然要声明为抽象类。类 C 继承自类 B，实现类抽象方法 F()，于是它可以进行对象实例化。

sealed 用来将类修饰为密封类，以阻止该类被继承。同时对一个类做 abstract 和 sealed 的修饰是没有意义的，也是被禁止的。

3.2.2　构造函数

另一个比较重要的概念就是构造函数。每一个类都有一个默认的构造函数来初始化对象的一些数据。

下面介绍一个 Person 类。

```
class Person                               //定义一个类为 Person
{
    public String name;                    //字段
    public String sex;                     //字段
    public int age;                        //字段
    public double weight;                  //字段
    public Person()                        //构造函数，初始化对象
    {
        name = "Wang";                     //
        sex = "man";                       //
        age = 30;                          //
        weight = 100;                      //初始化 4 个字段;
    }
    public Person(String name, String sex, int age, double weight)
    {                                      //构造函数，初始化指定的对象
        this.name = name;
        this.age = age;
        this.weight = weight;
        this.sex = sex;
    }                                      //关于 this，以后再详细说明。在这里理解为“这个”
    public void eatFood(double quanity)    //类中的方法
    {
        double temp = this.weight;         //关于 this，以后再详细说明。在这里理解为“这个”
        this.weight = temp + quanity;
    }

}
class Program
{
```

```
        static void Main(string[] args)
        {
            Person firstman = new Person();  //创建一个 Person 类的对象
            Console.WriteLine("Person()构造函数");
    Console.WriteLine("name = {0}, sex = {1}, age = {2}, weight = {3}", firstman.name, firstman.sex,
    firstman.age, firstman.weight);
            Person secondman = new Person();
            Console.WriteLine("构造函数 Person(string name, string sex, int age, double weight)");
    Console.WriteLine("name = {0}, sex = {1}, age = {2}, weight = {3}", secondman .name , secondman .
    sex , secondman .age , secondman .weight);
            Person thirdwoman = new Person("zhang", "woman", 25, 85);
            Console.WriteLine("构造函数 Person(string name, string sex, int age, double weight)");
            Console.WriteLine("name = {0}, sex = {1}, age = {2}, weight = {3}", thirdwoman.name,
    thirdwoman.sex, thirdwoman.age, thirdwoman.weight);

            Console.Read();
        }
    }
```

运行结果如图 3.1 所示。

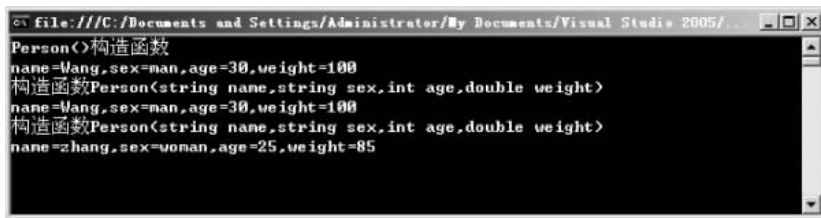

图 3.1 构造函数运行结果

同时还要注意构造函数的形式。构造函数的函数名必须与类的名字相同，而且是没有任何返回值的，也不允许用 void 来修饰。但是，构造函数允许重载。一个类中可以用多个不同的构造函数来满足创建对象。例如，在 Person 类中有两个构造函数：public Person(){ }和 public Person(String name，String sex，int age，double weight)。当我们在定义一个 Person 类的变量的时候，如“Person man＝new Person();”，同时也可以用另一个构造函数来初始化一个对象，如“Person niu＝new Person("name","man",22,99);”。但是对象 niu 和对象 man 是不一样的，这两个对象的属性也是不一样的。

3.2.3 析构函数

创建对象时要用构造函数，与此相对，释放对象时要用析构方法(destructor)(也称析构函数)。析构函数是用符号～开始的，并且其方法是与类同名的方法。该方法不带参数，不能写返回类型，也不能有修饰符。析构函数的形式如下。

```
～ 类名(){ … }
```

例如，在 Person 类中定义析构方法如下。

```
class Person
{
     ⋮
     ~ Person()
     {
     ⋮
     }
}
```

一个类的析构函数最多只有一个。如果没有提供析构函数，则系统会自动生成一个。由于对象的释放是由系统自动进行的，不能由程序控制，所以析构函数不能由程序显式调用，而是由系统在释放对象时自动调用。从这个意义上来说，普通对象的析构函数并不是特别重要的。

3.2.4 this的引用

在方法中，可以使用一个关键字 this 来表示这个对象本身。具体地说，在普通方法中，this 表示调用这个方法的对象；在构造方法中，this 表示新创建的对象。

在上面的 Person 类的定义中，两个构造函数的参数是一样的。在第二个构造函数中多了个 this。this 的作用是用来指定"这个"的，也就是用来指定当前这个对象的。

当然，我们可以把 public Person(string name, string sex,int age,double weight)的参数换成其他的名字，如 public Person(string myname, string mysex, int myage, double myweight)。这样并不会影响程序的结果。但是，如果把上面构造函数中的 this 去掉，来看看有什么影响。

```
public Person(String name, String sex, int age, double weight)
{                                         //构造函数,初始化指定的对象
      name = name;
      age = age;
      weight = weight;
      sex = sex;
}                                         //去掉 this 后
```

程序运行结果如图 3.2 所示。

图 3.2 没有 this 的运行结果

this 关键字引用被访问成员所在的当前实例。静态成员函数中没有 this 指针。this 关键字可以用来构造函数，在实例方法和实例化访问器中访问成员。不能在静态方法、静态属性访问器或者域声明的变量初始化程序中使用 this 关键字，否则将会产生错误。

- 在类的构造函数中出现的 this 作为一个值类型，表示对正在构造的对象本身的引用。
- 在类的方法中出现的 this 作为一个值类型，表示对调用该方法的对象的引用。
- 在结构的构造函数中出现的 this 作为一个变量类型，表示对正在构造的结构的引用。
- 在结构的方法中出现的 this 作为一个变量类型，表示对调用该方法的结构引用。

3.3 方法

方法又称成员函数(Member Function)，它集中体现了类或对象的行为。方法同样分为静态方法和实例方法。静态方法只可以操作静态域，而实例方法既可以操作实例域，也可以操作静态域(虽然这不被推荐)。操作静态域在某些特殊的情况下是很有用的。方法有 5 种存取修饰符：public、protected、internal、protected internal 和 private，它们的意义如前所述。

3.3.1 方法参数

方法的参数是个值得特别注意的地方。方法的参数传递有 4 种类型：传值(by value)、传址(by reference)、输出参数(by output)和数组参数(by array)。传值参数无需额外的修饰符，传址参数需要修饰符 ref，输出参数需要修饰符 out，数组参数需要修饰符 params。传值参数如果在方法调用过程中改变了参数的值，那么传入方法的参数在方法调用完成以后并不会因此而改变，而是保留原来传入时的值。传址参数恰恰相反。如果方法调用过程改变了参数的值，那么传入方法的参数在调用完成以后也将随之改变。实际上，从名称上可以清楚地看出两者的含义的不同：传值参数传递的是调用参数的一份拷贝，而传址参数传递的是调用参数的内存地址，该参数在方法内外指向的是同一个存储位置。请看下面的例子及其输出。

```
class Test
{
    static void Swap(ref int x, ref int y)
    {
        int temp = x;
        x = y;
        y = temp;
    }
    static void Swap(int x, int y)
    {
        int temp = x;
        x = y;
        y = temp;
    }
    static void Main()
    {
        int i = 1, j = 2;
```

```
            Swap(i, j);
            Console.WriteLine("i = {0}, j = {1}", i, j);
            Swap(ref i, ref j);
            Console.WriteLine("i = {0}, j = {1}", i, j);
        }
    }
```

程序运行结果如图 3.3 所示。

从上面的例子中可以清楚地看到两个交换函数 Swap()由于参数的差别(传值与传址),而得到了不同的调用结果。注意,传址参数的方法调用无论是在声明时还是在调用时都要加上 ref 修饰符。

笼统地说传值不会改变参数的值在有些情况下是错误的。例如:

```
class Element
{
    public int Number = 50;
}
class Test
{
    static void Change(Element s)
    {
        s.Number = 100;
    }
    static void Main()
    {
        Element e = new Element();
        Console.WriteLine(e.Number);
        Change(e);
        Console.WriteLine(e.Number);
    }
}
```

程序运行结果如图 3.4 所示。

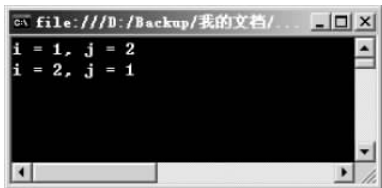

图 3.3 传值与传址的运行结果

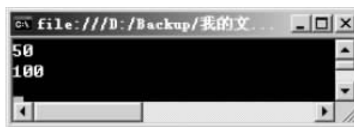

图 3.4 传值的运行结果

从上面的例子可以看到,传值方式改变了类型为 Element 类的对象 t。但从严格意义上讲,传值方式实际上是改变了对象 t 的域,而非对象 t 本身。例如:

```
class Element
{
    public int Number = 10;
}
class Test
```

```
{
    static void Change(Element s)
    {
        Element t = new Element();
        t.Number = 100;
        s = r;
    }
    static void Main()
    {
        Element e = new Element();
        Console.WriteLine(e.Number);
        Change(e);
        Console.WriteLine(e.Number);
    }
}
```

图 3.5 传值的运行结果

程序运行结果如图 3.5 所示。

传值方式根本没有改变类型为 Element 类的对象 t。实际上，如果能够理解类这一 C# 中的引用类型(reference type)的特性，便能看出上面两个例子的差别。在传值过程中，引用类型本身不会改变(t 不会改变)，但引用类型内含的域却会改变(t.Number 改变了)。C# 语言的引用类型有 object 类型(包括系统内建的 class 类型和用户自建的 class 类型——继承自 object 类型)、string 类型、interface 类型、array 类型和 delegate 类型。它们在传值调用中都具有上面两个例子中所展示的特性。

在传值和传址过程中，C# 强制要求参数在传入之前要由用户进行明确的初始化，否则编译器将报错。但如果有一个并不依赖于参数初值的函数，在函数返回时只需要得到它的值，这时该怎么办呢？往往在函数返回值不止一个的情况下特别需要这种技巧。答案是：用 out 修饰输出参数。但需要记住，输出参数与通常的函数返回值有一定的区别：函数返回值往往存在堆栈里，在返回时弹出；而输出参数需要用户预先分配存储位置，也就是说，用户需要提前声明变量，当然也可以进行初始化。例如：

```
class Test
{
    static void ResoluteName(string fullname, out string firstname, out string lastname)
    {
        string[] strArray = fullname.Split(new char[] { ' ' });
        firstname = strArray[0];
        lastname = strArray[1];
    }
    public static void Main()
    {
        string MyName = "黄 小娟";
        string MyFirstName, MyLastName;
        ResoluteName(MyName, out MyFirstName, out MyLastName);
        Console.WriteLine("My first name: {0}",MyFirstName);
```

```
            Console.WriteLine("My last name: {0}",MyLastName);
            Console.Read();
        }
    }
```

程序运行结果如图 3.6 所示。

图 3.6　用 out 修饰输出参数的运行结果

在函数体内，所有输出参数必须均被赋值，否则编译器会报错。out 修饰符同样应该应用在函数声明和调用两个地方。除了充当返回值这一特殊的功能外，out 修饰符与 ref 修饰符有个很相似的地方，即传址。可以看出 C＃完全抛弃了传统 C/C++语言赋予程序员的莫大的自由度，毕竟 C＃是用来开发高效的下一代网络平台的，安全性——包括系统安全（系统结构的设计）和工程安全（避免出现程序员经常犯的错误），是它设计时需要考虑的重要因素。当然 C＃并没有因为安全性而影响性能。

数组参数也是经常用到的一个参数，它用来传递大量的数组集合参数。先来看看下面的例子。

```
class Test
{
    static int Sum(params int[] args)
    {
        int s = 0;
        foreach (int n in args)
        {
            s += n;
        }
        return s;
    }
    static void Main()
    {
        int[] var = new int[] { 1, 2, 3, 4, 5 };
        Console.WriteLine("The Sum:" + Sum(var));
        Console.WriteLine("The Sum:" + Sum(10, 20, 30, 40, 50));
        Console.Read();
    }
}
```

程序运行结果如图 3.7 所示。

图 3.7　数组参数的运行结果

从上面的例子中可以看出，数组参数可以是数组，如 var，也可以是能够隐式转化为数组的参数，如{10，20，30，40，50}。这为程序提供了很高的扩展性。

同名方法参数的不同会导致方法出现多态现象，这种现象称为重载（overloading）方法。需要指出的是，编译器在编译时便绑定了方法和方法调用。只能通过参数的不同来重载方

法，其他方面的不同(如返回值)不能为编译器提供有效的重载信息。

3.3.2 方法继承

面向对象机制为 C＃的方法引入了 virtual、override、sealed 和 abstract 4 种修饰符，来提供不同的继承需求。类的虚方法是可以在该类的继承子类中改变其实现的方法。当然，这种改变仅限于方法体的改变，而非方法头(方法声明)的改变。被子类改变的虚方法必须在方法头加上 override。当调用虚方法时，该类的实例即对象运行时的类型(run-time type)将决定哪个方法体被调用。例如：

```
class Parent
{
    public void F() { Console.WriteLine("Parent.F"); }
    public virtual void G() { Console.WriteLine("Parent.G"); }
}
class Child : Parent
{
    new public void F() { Console.WriteLine("Child.F"); }
    public override void G() { Console.WriteLine("Child.G"); }
}
class Test
{
    static void Main()
    {
        Child b = new Child();
        Parent a = b;
        a.F();
        b.F();
        a.G();
        b.G();
        Console.Read();
    }
}
```

图 3.8 虚方法的运行结果

程序运行结果如图 3.8 所示。

从上面的例子中可以看到 class Child 中 F()方法的声明采取了重写(new)的办法来屏蔽 class Parent 中的非虚方法 F()的声明。而 G()方法则采用了覆盖(override)的办法来提供方法的多态机制。需要注意重写(new)方法和覆盖(override)方法的不同。从本质上讲，重写方法是编译时绑定的，而覆盖方法则是运行时绑定的。值得指出的是，虚方法不可以是静态方法，也就是说，不可以用 static 和 virtual 同时修饰一个方法，这是由它运行时的类型辨析机制所决定的。override 必须和 virtual 配合使用，当然，它不能和 static 同时使用。

如果在一个类的继承体系中不想再使一个虚方法被覆盖，该怎样做呢？答案是 sealed override (密封覆盖)，使用 sealed 和 override 同时修饰一个虚方法便可以达到这种目的，如 sealed override public void F()。注意，这里一定是 sealed 和 override 同时使用，也一定是

密封覆盖一个虚方法,或者一个被覆盖(而不是密封覆盖)了的虚方法。密封一个非虚方法是没有意义的,也是错误的。例如:

```
class Parent
{
    public virtual void F()
    {
        Console.WriteLine("Parent.F");
    }
    public virtual void G()
    {
        Console.WriteLine("Parent.G");
    }
}
class Child : Parent
{
    sealed override public void F()
    {
        Console.WriteLine("Child.F");
    }
    override public void G()
    {
        Console.WriteLine("Child.G");
    }
}
class Grandson : Child
{
    override public void G()
    {
        Console.WriteLine("Grandson.G");
    }
}
```

抽象(abstract)方法在逻辑上类似于虚方法,只是它不能像虚方法那样被调用,它只是一个接口的声明而非实现。抽象方法没有类似于{...}这样的方法实现,也不允许这样做。抽象方法同样不能是静态的。含有抽象方法的类一定是抽象类,同时一定要加 abstract 类修饰符。但抽象类并不一定要含有抽象方法。继承自含有抽象方法的抽象类的子类必须覆盖并实现(直接使用 override)该方法,或者组合使用 abstract override,使之继续抽象,或者不提供任何覆盖和实现。后两者的行为是一样的。例如:

```
abstract class Parent
{
    public abstract void F();
    public abstract void G();
}
abstract class Child : Parent
{
    public abstract override void F();
}
```

```
abstract class Grandson : Child
{
    public override void F()
    {
        Console.WriteLine("Grandson.F");
    }
    public override void G()
    {
        Console.WriteLine("Grandson.G");
    }
}
```

抽象方法可以抽象一个继承来的虚方法，例如：

```
class Parent
{
    public virtual void Method()
    {
        Console.WriteLine("Parent.Method");
    }
}
abstract class Child : Parent
{
    public abstract override void Method();
}
abstract class Grandson : Child
{
    public override void Method()
    {
        Console.WriteLine("Grandson.Method");
    }
}
```

归根结底，掌握了运行时绑定和编译时绑定的基本机理便能看透方法所呈现出的种种形态，如 overload、virtual、override、sealed、abstract 等形态。

3.4 属性

属性可以说是 C＃语言的一个创新。理解属性的设计初衷是用好属性这一工具的根本。C＃不提倡将域的保护级别设为 public 以使用户在类外可以任意操作。对于所有有必要在类外可见的域，C＃都推荐采用属性来表达。属性不表示存储位置。这是属性和域的根本性的区别。下面是一个典型的属性设计。

```
class MyClass
{
   int integer;
   public int Integer
   {
        get { return integer; }
```

```
            set { integer = value; }
        }
    }
    class Test
    {
        public static void Main()
        {
            MyClass MyObject = new MyClass();
            Console.WriteLine(MyObject.Integer);
            MyObject.Integer++;
            Console.WriteLine (MyObject.Integer);
            Console.Read();
        }
    }
```

程序输出 0 1。从上面的程序中可以看到，属性通过对方法的包装，向程序员提供了一个友好的域成员的存取界面。这里的 value 是 C# 的关键字，它是进行属性操作时的 set 的隐含参数，也就是在执行属性写操作时的右值。

属性提供了只读(get)、只写(set)和读写(get 和 set)3 种接口操作。域的这 3 种操作必须在同一个属性名下声明，不可以将它们分离。来看下面的实现方法。

```
class MyClass
{
    private string name;
    public string Name
    {
        get { return name; }
    }
    public string Name
    {
        set { name = value; }
    }
}
```

上面的这种分离 Name 属性实现的方法是错误的。应该像前面的例子一样将只读和只写放在一起声明。

当然，属性远远不仅仅限于域的接口操作。属性的本质还是方法，可以根据程序逻辑在属性的提取或赋值过程中进行某些检查、警告等额外操作。例如：

```
class MyClass
{
    private string name;
    public string Name
    {
        get { return name; }
        set
        {
            if (value == null)
                name = "Microsoft";
            else
```

```
                    name = value;
                }
            }
        }
```

由于属性的方法的本质，属性当然也有方法的种种修饰。属性有5种存取修饰符，但属性的存取修饰往往为public，否则它就失去了属性作为类的公共接口的意义。除了不具备方法的多参数带来的方法重载等特性外，virtual、sealed、override和abstract等修饰符对属性与方法有同样的行为。但由于属性在本质上被实现为两个方法，它的某些行为需要加以注意。例如：

```
abstract class A
{
    int y;
    public virtual int X
    {
        get { return 0; }
    }
    public virtual int Y
    {
        get { return y; }
        set { y = value; }
    }
    public abstract int Z { get; set; }
}
class B : A
{
    int z;
    public override int X
    {
        get { return base.X + 1; }
    }
    public override int Y
    {
        set { base.Y = value < 0 ? 0 : value; }
    }
    public override int Z
    {
        get { return z; }
        set { z = value; }
    }
}
```

这个例子集中地展示了属性在继承上下文中的某些典型行为。这里，类A由于抽象属性Z的存在而必须声明为abstract。子类B中通过base关键字来引用父类A的属性。类B中可以只通过Y-set来覆盖类A中的虚属性。

3.5 继承

为了提高软件模块的可复用性和可扩充性，来提高软件的开发效率，开发人员总是希望能够利用前人或自己以前的开发成果，同时又希望在开发过程中能够有足够的灵活性，不拘

泥于复用的模块。C#这种完全面向对象的程序设计语言提供了两个重要的特性——继承性(inheritance)和多态性(polymorphism)。

继承是面向对象程序设计的主要特征之一,它可以通过重用代码来节省程序设计的时间。任何类都可以从另一个类中继承,但是每一个类都只能继承几个基本类。被继承的类称为父类,继承的类称为子类。继承就是在类之间建立一种相交关系,使得新定义的派生类的实例可以继承已有的基类的特征和能力,而且可以加入新的特性,或者是修改已有的特性,建立起类的新层次。

现实世界中的许多实体之间并不是相互孤立的,它们往往具有共同的特征,但也存在内在的差别。可以采用层次结构来描述这些实体之间的相似之处和不同之处。

3.5.1 继承的使用

一个类从一个基类继承后,就得到基类的所有成员、属性以及字段。现在来看 Student 类继承 Person 类后的情况。class Student :Person 表示 Student 类继承 Person 类。例如:

```
class Person                              //Person类
{
    public string name;
    public string sex;
    public int age;
    public double weight;
    public Person()                       //构造函数
    {
        name = "ZhangSan";
        sex = "man";
        age = 22;
        weight = 99;
    }
    public Person(string name, string sex, int age, double weight)//构造函数
    {
        this.name = name;
        this.sex = sex;
        this.age = age;
        this.weight = weight;
    }
    public void Eat(double food)          //方法,吃东西后体重增加
    {
        this.weight += food;
    }
}
class Student : Person
{
    public string school_name;
    public double score;
    public int grade;
    public Student()
    {
        school_name = "cqu";
```

```
            score = 99;
            grade = 3;
        }
        public void study(double hours)
        {
            this.weight - = hours;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Person man = new Person("ZhaoWei", "woman", 31, 95);
            Console.WriteLine(" Person 方法和属性:");
            Console.WriteLine("name = {0},sex = {1},age = {2},weight = {3}",
            man.name, man.sex, man.age, man.weight); man.Eat(2);
            Console.WriteLine("man.Eat(2);", man.weight);
            Student zhangsan = new Student();
            Console.WriteLine("Student 方法和属性:");
             Console.WriteLine("zhangsan.name = {0}, zhangsan.sex = {1}, zhangsan.age = {2},
    zhangsan.weight = {3}", zhangsan.name, zhangsan.sex, zhangsan.age, zhangsan.weight);
            //调用父类的属性
            Console.WriteLine("zhangsan.school_name = {0},zhangsan.score = {1},zhangsan.grade =
    {2}", zhangsan.school_name, zhangsan.score, zhangsan.grade);  //自己的属性
            Console.WriteLine("zhangsan.weight = {0}", zhangsan.weight);
            zhangsan.Eat(3);                //调用父类的方法
            Console.WriteLine("zhangsan.weight = {0}", zhangsan.weight);
            zhangsan.study(5);              //调用自己的方法
            Console.WriteLine("zhangsan.weight = {0}", zhangsan.weight);
        }
    }
```

程序运行结果如图 3.9 所示。

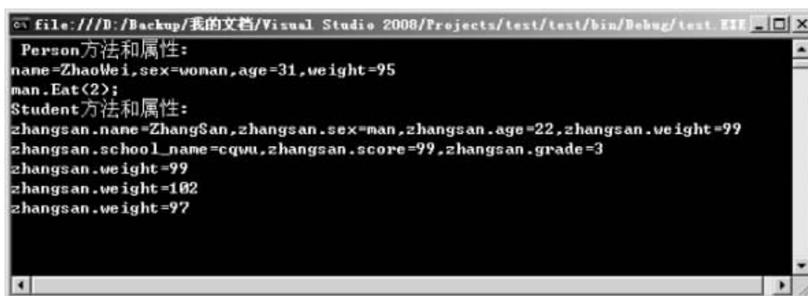

图 3.9 继承的运行结果

3.5.2 隐藏基类成员

想想看，如果所有的类都可以被继承，继承的滥用会带来什么后果？有时候，开发人员并不希望自己编写的类被继承。还有一些时候，有的类已经没有必要再被继承。C♯提出了一个密封类(sealed class)的概念，来帮助开发人员解决这一问题。密封类在声明中使用

sealed 修饰符，这样就可以防止该类被其他类继承。如果试图将一个密封类作为其他类的基类，C＃将提示出错。理所当然，密封类不能同时又是抽象类，因为抽象类总是希望被继承的。在哪些场合下使用密封类呢？密封类可以阻止其他程序员在无意中继承该类。而且密封类可以起到运行时优化程序的效果。实际上，密封类中不可能有派生类。如果密封类实例中存在虚成员函数，则该成员函数可以转化为非虚的，函数修饰符 virtual 便不再生效。

3.5.3 密封方法

C＃还提出了密封方法(sealedmethod)的概念，以防止在方法所在类的派生类中对该方法的重载。可以对方法使用 sealed 修饰符，这种方法称为密封方法。不是类的每个成员方法都可以作为密封方法，必须对基类的虚方法进行重载，提供具体的实现方法的方法才称为密封方法。所以，在方法的声明中，sealed 修饰符总是和 override 修饰符同时使用。例如：

```
class A
{
    public virtual void F( )
    {
        Console.WriteLine("A.F") ;
    }
    public virtual void G( )
    {
           Console.WriteLine("A.G") ;
    }
}
class B: A
{
    sealed override public void F( )
    {
        Console.WriteLine("B.F") ;
    }
    override public void G( )
    {
        Console.WriteLine("B.G") ;
    }
}
class C: B
{
    override public void G( )
    {
        Console.WriteLine("C.G") ;
    }
}
```

C＃中的继承符合下列规则。

- 继承是可传递的。如果 C 从 B 中派生，B 又从 A 中派生，那么 C 不仅继承了 B 中声明的成员，同样也继承了 A 中的成员。Object 类是所有类的基类。
- 派生类应当是对基类的扩展。派生类可以添加新的成员，但不能除去已经继承的成员的定义。

- 构造函数和析构函数都不能被继承。除此之外的其他成员，不论对它们定义了怎样的访问方式，它们都能被继承。基类中的成员的访问方式只能决定派生类能否访问它们。
- 派生类如果定义了与继承而来的成员同名的新成员，则新成员就可以覆盖已继承的成员。但这并不表示该派生类删除了这些成员，只是说明该派生类不能再访问这些成员。
- 类可以定义虚方法、虚属性以及虚索引指示器。它的派生类能够重载这些成员，从而实现类的多态性。
- 派生类只能从一个类中继承，它可以通过接口实现多重继承。

3.6 多态

同一操作作用于不同的对象，能够有不同的解释，产生不同的执行结果，这就是多态性。多态性通过派生类重载基类中的虚函数型方法来实现。在面向对象的系统中，多态性是个很重要的概念。它允许客户对一个对象进行操作，然后由对象来完成一系列的动作。对象具体实现哪个动作、如何实现都由系统负责解释。

在C＃中，多态性的定义是：同一操作作用于不同的类的实例，不同的类将进行不同的解释，最后产生不同的执行结果。

C＃支持以下两种类型的多态性。

- 编译时的多态性。编译时的多态性是通过重载来实现的。对于非虚的成员来说，系统在编译时根据传递的参数、返回的类型等信息决定实现何种操作。
- 运行时的多态性。运行时的多态性就是指直到系统运行时，才根据实际情况决定实现何种操作。在C＃中，运行时的多态性通过虚成员实现。

编译时的多态性提供了运行速度快的特点，而运行时的多态性则带来了高度灵活和抽象的特点。

3.6.1 方法覆盖与多态

多个类能够从单个基类继承。通过继承，类可以在基类所在的同一实现中接收基类的任何方法、属性和事件。这样，类便可以根据需要来实现附加成员，而且能够重写基成员，以提供不同的实现。请注意，继承类也能够实现接口，这两种技术不是互斥的。

C＃通过继承提供多态性。对于小规模的研发任务而言，多态是个功能强大的机制，但对于大规模系统，它通常会存在问题。过分强调继承驱动的多态性一般会导致资源大规模地从编码转移到设计，这对于缩短总的研发时间没有任何帮助。例如：

```
class myFirst
{
    int value_myFirst;
    public myFirst(int f)
    {
        value_myFirst = f;
```

```
        }
        public void f1()
        {
            System.Console.WriteLine("myFirst.f1()!");
        }
        public virtual void f2()                //virtual 也可以提到最前面
        {
            System.Console.WriteLine("myFirst.f2()!");
        }
    }
    class mySecond : myFirst
    {
        int value_mySecond;
        public mySecond(int s) : base(s)
        {
            value_mySecond = s;
        }
        //使用关键字 new 覆盖基类中的同名方法
        public new void f1()                    //new 也可以提到最前面
        {
            System.Console.WriteLine("mySeconde.f1()!");
        }
        //基类函数中声明是 virtual,必须用 override 覆盖
        public override void f2()               //override 也可以提到最前面
        {
            System.Console.WriteLine("mySeconde.f2()!");
        }
        }
    class Program
    {
        static void Main(string[] args)
        {
            myFirst mf = new myFirst(1);
            mySecond ms = new mySecond(2);
            mf.f1();                            //myFirst.f1()!
            mf.f2();                            //myFirst.f2()!
            ms.f1();                            //mySeconde.f1()!
            ms.f2();                            //mySeconde.f2()!
            mf = ms;                            //向上转型之后
            mf.f1();                            //myFirst.f1()!
            mf.f2();                            //mySeconde.f2()! 这是用 override 的运行结果
            //如果是 new,那么结果是 myFirst.f2( )!
            Console.Read();
        }
    }
```

运行结果如图 3.10 所示。

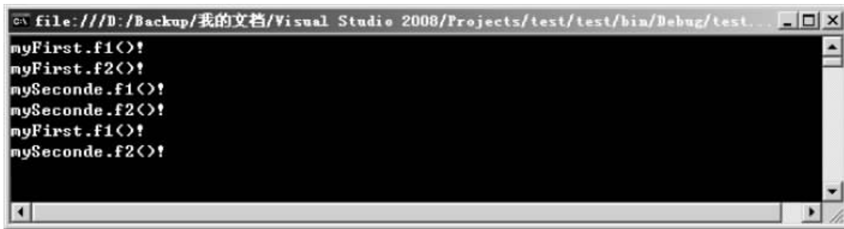

图 3.10　多态的运行结果

3.6.2　抽象类

抽象类可以同时提供继承和接口的元素。抽象类本身不能实例化，它必须被继承。该类的部分或全部成员都可能未实现，该实现由继承类提供。已实现的成员仍可被重写，并且继承类仍能够实现附加接口或其他功能。抽象类提供继承和接口实现的功能。抽象类不能实例化，它必须在继承类中实现。它能够包含已实现的方法和属性，但也能够包含未实现的过程。这些未实现的过程必须在继承类中实现。这样，可以在类的某些方法中提供不变级功能，同时为其他的过程保持灵活性。抽象类的另一个好处是：当需要组件的新版本时，抽象类可以根据需要，将附加方法添加到基类，但要求接口必须保持不变。

上面已经说明，虽然基类方法声明为 virtual，以便派生类用 override 覆盖，但是派生类仍然可以用 new 关键字覆盖(不具有多态性)。可以强制让派生类覆盖基类的方法，将基类方法声明为抽象的。方法是，采用 abstract 关键字。抽象方法没有方法体，它由派生类来提供。如果派生类不实现基类的抽象方法，则派生类也需要声明为 abstract 类。

类中只要存在抽象方法，就必须将抽象方法声明为抽象类。例如：

```
abstract class myFirst
{
    int value_myFirst;
    public myFirst(int f)
    {
        value_myFirst = f;
    }
    //抽象方法没有方法体,以分号结尾
    public abstract void f1();
    public void f2()
    {
        System.Console.WriteLine("myFirst.f2()!");
    }
    public virtual void f3()
    {
        System.Console.WriteLine("myFirst.f3()!");
    }
}
class mySecond : myFirst
{
    int value_mySecond;
```

```
    public mySecond(int s): base(s)
    {
        value_mySecond = s;
    }
    //覆盖基类的抽象方法
    public override  void f1()
    {
        System.Console.WriteLine("mySeconde.f1()!");
    }
    //覆盖基类的一般方法
    public new void f2()
    {
        System.Console.WriteLine("mySeconde.f2()!");
    }
    //覆盖基类的虚拟方法
    public override void f3()
    {
        System.Console.WriteLine("mySecond.f3()!");
    }
}
class Program
{
    static void Main(string[] args)
    {
        //抽象类和接口不能声明对象
        //myFirst mf = new myFirst(1);
        mySecond ms = new mySecond(2);
        ms.f1();                        //mySeconde.f1()!
        ms.f2();                        //mySeconde.f2()!
        ms.f3();                        //mySecond.f3()!
        //这里向上转型,采用强类型转换的方式
        ((myFirst)ms).f1();             //mySeconde.f1()!
        ((myFirst)ms).f2();             //myFirst.f2()!
        ((myFirst)ms).f3();             //mySecond.f3()!
    }
}
```

运行结果如图 3.11 所示。

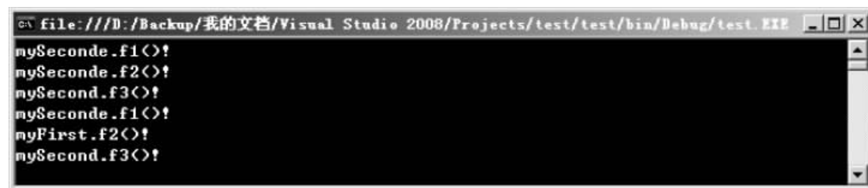

图 3.11 抽象类的运行结果

3.6.3 接口多态性

多个类可以实现相同的接口,而单个类能够实现一个或多个接口。接口描述类需要实

现的方法、属性和事件以及每个成员需要接收和返回的参数类型，但将这些成员的特定实现将留给实现类去完成。

组件编程中的一项强大技术是能够在一个对象上实现多个接口。每个接口都由一小部分紧密联系的方法、属性和事件组成。通过实现接口，组件能够为需要该接口的任何其他组件提供功能，而无需考虑其中所包含的特定功能。这使后续组件的版本可以包含不同的功能，这样就不会干扰核心功能。研发人员最常使用的组件功能是组件类本身的成员。然而，对于包含大量成员的组件，使用起来可能会比较困难。可以考虑将组件的某些功能分解出来，作为私下实现的单独接口。

3.7 接口

3.7.1 接口定义

接口（interface）用来定义一种程序的协定，即实现接口的类或结构，要与接口的定义严格一致。有了这个协定，就可以抛开编程语言的限制（理论上）。接口可以从多个基接口继承，而类或结构可以实现多个接口。接口包含方法、属性、事件和索引器。接口本身不提供它所定义的成员的实现，它只指定实现该接口的类或接口必须提供的成员。

定义接口的一般形式为：

```
[attributes] [modifiers] interface identifier [:base - list] {interface - body}[;]
```

对接口的定义有如下说明：

（1）attributes（可选）。附加的定义性信息。

（2）modifiers（可选）。允许使用的修饰符有 new 和 4 个访问修饰符，分别是 public、protected、internal、private。在一个接口定义中，同一修饰符不允许出现多次。new 修饰符只能出现在嵌套接口中，它表示覆盖了继承而来的同名成员。

（3）identifier。接口名称。

（4）base-list（可选）。包含一个或多个显式基接口的列表，接口间由逗号分隔。

（5）interface-body。对接口成员的定义。

（6）接口可以是命名空间或类的成员，并且可以包含下列成员的签名：方法、属性及索引器。

（7）一个接口可从一个或多个基接口继承。

例如：

```
interface IMyExample
{
    string this[int index] { get ; set ; }
    event EventHandler Even ;
    void Find(int value) ;
    string Point { get ; set ; }
}
public delegate void EventHandler(object sender, Event e) ;
```

上述例子中的接口包含一个索引（this）、一个事件（Even）、一个方法（Find）和一个属性

(Point)。

接口可以支持多重继承。下例中接口 IcomboBox 可以同时从 ItextBox 和 IlistBox 中继承。

```
interface IControl
{
    void Paint( ) ;
}
interface ITextBox: IControl
{
    void SetText(string text) ;
}
interface IListBox: Icontrol
{
    void SetItems(string[ ] items) ;
}
interface IComboBox: ITextBox, IListBox { }
```

类和结构可以多重实例化接口。下例中类 EditBox 继承了类 Control,同时还从 IdataBound 和 Icontrol 继承。

```
interface IdataBound
{
    void Bind(Binder b) ;
}
public class EditBox: Control, IControl, IDataBound
{
    public void Paint( ) ;
    public void Bind(Binder b) {...}
}
```

在上面的代码中,Paint 方法从 Icontrol 接口继承而来;Bind 方法从 IdataBound 接口继承而来,它们都以 public 的身份在 EditBox 类中实现。

3.7.2 定义接口成员

接口可以包含一个或多个成员。

对接口成员的定义有如下说明。

(1) 接口的成员由从基接口继承的成员和由接口本身定义的成员组成。

(2) 接口定义可以定义零个或多个成员。接口的成员必须是方法、属性、事件或索引器。接口不能包含常数、字段、运算符、实例构造函数、析构函数或类型,也不能包含任何种类的静态成员。

(3) 定义一个接口,该接口对于每种可能种类的成员都包含一个方法、属性、事件或索引器。

(4) 接口成员的默认访问方式是 public。接口成员定义不能包含任何修饰符,比如,成员定义前不能加 abstract、public、protected、internal、private、virtual、override 或 static 修饰符。

(5) 接口的不同成员之间不能同名。继承而来的成员不用再定义,但接口可以定义与继承而来的成员同名的成员,这时接口成员覆盖了继承而来的成员。这不会导致错误,但编

译器会给出一个警告。关闭警告提示的方式是在成员定义前加上一个 new 关键字。但如果没有覆盖父接口中的成员，使用 new 关键字会导致编译器发出警告。

（6）方法的名称必须与同一接口中定义的所有属性和事件的名称不同。此外，方法的签名必须与同一接口中定义的其他方法的签名不同。

（7）属性或事件的名称必须与同一接口中定义的其他成员的名称不同。

（8）一个索引器的签名必须有别于在同一接口中定义的其他所有索引器的签名。

（9）接口方法声明中的属性（attributes）、返回类型（return-type）、标识符（identifier）和形式参数列表（formal-parameter-list）与一个类的方法声明中的相应内容的意义相同。一个接口方法声明不允许指定一个方法主体，而声明通常用一个分号结束。

（10）接口属性声明的访问符与类属性声明的访问符相对应，除了访问符主体之外，通常必须用分号。因此，无论属性是读写、只读或只写，其访问符都完全确定。

3.7.3 访问接口

对接口方法的调用和采用索引指示器访问的规则与类中的情况也是相同的。如果底层成员的命名与继承而来的高层成员一致，那么底层成员将覆盖同名的高层成员。但由于接口支持多继承，在多继承中，如果两个父接口含有同名的成员，这就产生了二义性（这也正是 C# 中取消了类的多继承机制的原因之一），这时需要进行显式的定义。例如：

```
interface ISequence
{
    int Count { get; set; }
}
interface IRing
{
    void Count(int i);
}
interface IRingSequence : ISequence, IRing { }
class CTest
{
    void Test(IRingSequence rs)
    {
        //rs.Count(1) ; 错误, Count 有二义性
        //rs.Count = 1; 错误, Count 有二义性
        ((ISequence)rs).Count = 1;      // 正确
        ((IRing)rs).Count(1);           // 正确调用 IRing.Count
    }
}
```

在上面的例子中，前两条语句“rs . Count(1);”和“rs . Count＝1;”会产生二义性，从而导致编译错误。因此，必须显式地给 rs 指派父接口类型，这种指派在运行时不会带来额外的开销。

再看下面的例子。

```
interface IInteger
{
    void Add(int i) ;
```

```
}
interface IDouble
{
    void Add(double d) ;
}
interface INumber: IInteger, IDouble {}
class CMyTest {
void Test(INumber Num)
{
    // Num.Add(1) ; 错误
    Num.Add(1.0) ;                          // 正确
    ((IInteger)n).Add(1) ;                  // 正确
    ((IDouble)n).Add(1) ;                   // 正确
    }
}
```

调用 Num. Add(1) 会导致二义性,因为候选的重载方法的参数类型均适用。但是,调用 Num. Add(1.0) 是允许的,因为 1.0 是浮点数,参数类型与方法 IInteger. Add()的参数类型不一致,这时只有 IDouble. Add 才是适用的。不过,只要加入了显式的指派,就绝不会产生二义性。

接口的多重继承的问题也会带来成员访问上的问题。例如:

```
interface IBase
{
    void FWay(int i) ;
}
interface ILeft: IBase
{
    new void FWay (int i) ;
}
interface IRight: IBase
{
    void G( ) ;
}
interface IDerived: ILeft, IRight { }
class CTest
{
    void Test(IDerived d)
    {
        d. FWay (1) ;                       // 调用 ILeft. FWay
        ((IBase)d). FWay (1) ;              // 调用 IBase. FWay
        ((ILeft)d). FWay (1) ;              // 调用 ILeft. FWay
        ((IRight)d). FWay (1) ;             // 调用 IBase. FWay
    }
}
```

在上例中,方法 IBase. FWay 在派生的接口 ILeft 中被 Ileft 的成员方法 FWay 覆盖了。所以,对 d. FWay (1)的调用实际上调用了 ILeft. FWay()方法。虽然从 IBase→IRight→IDerived 这条继承路径上来看,ILeft. FWay 方法是没有被覆盖的。注意,成员一旦被覆盖

以后，所有对它的访问都将被覆盖以后的成员拦截。

3.7.4　实现接口

为了实现接口，类可以定义显式接口成员执行体（Explicit Interface Member Implementations）。显式接口成员执行体可以是一个方法、一个属性、一个事件或者是一个索引指示器的定义。定义与该成员对应的全权名接口的全权名（fully qualified name）是这样构成的：接口名加小圆点“.”。例如，下面的 ICloneable. Clone()应保持一致。

```
interface ICloneable
{
    object Clone( ) ;
}
interface IComparable
{
    int CompareTo(object other) ;
}
class ListEntry: ICloneable, IComparable
{
    object ICloneable.Clone( ) {...}
    int IComparable.CompareTo(object other) {...}
}
```

上面的代码中 ICloneable. Clone 和 IComparable. CompareTo 就是显式接口成员执行体。

对显式接口成员执行体的定义有如下说明。

（1）不能在方法调用、属性访问以及索引指示器访问中通过全权名访问显式接口成员执行体。事实上，显式接口成员执行体只能通过接口的实例，仅仅引用接口的成员名称来访问。

（2）显式接口成员执行体不能使用任何访问限制符，也不能加上 abstract、virtual、override 或 static 修饰符。

（3）显式接口成员执行体和其他成员有着不同的访问方式。因为不能在方法调用、属性访问以及索引指示器访问中通过全权名访问，显式接口成员执行体在某种意义上来看是私有的。但它们又可以通过接口的实例访问，所以也具有一定的公有性质。

（4）只有类在定义时，才把接口名写在基类列表中，而且类中定义的全权名、类型和返回类型都与显式接口成员执行体完全一致时，显式接口成员执行体才是有效的。例如：

```
class Shape: ICloneable
{
    object ICloneable.Clone( ) {...}
    int IComparable.CompareTo(object other) {...}
}
```

使用显式接口成员执行体通常有以下两个目的。

（1）因为显式接口成员执行体不能通过类的实例进行访问，这就可以从公有接口中把接口的实现部分单独分离开。如果一个类只在内部使用该接口，而类的使用者不会直接使用到该接口，这时显式接口成员执行体就可以起到作用。

（2）显式接口成员执行体避免了接口成员之间因为同名而发生混淆的问题。如果一个类希望对名称和返回类型相同的接口成员采用不同的实现方式，这就必须使用显式接口成员执行体。如果没有显式接口成员执行体，那么对于名称和返回类型不同的接口成员，也无法用类进行实现。

3.8　索引器与集合

3.8.1　索引器

索引器（Indexer）是C＃引入的一个新型的类成员，它使得对象可以像数组那样被方便、直观地引用。索引器类似于我们前面讲到的属性，但索引器可以有参数列表，且其参数列表只能作用在实例对象上，不能直接作用在类上。

索引的定义方法如下。

```
修饰符 类型名 this [ 参数列表 ]
{
    set
    {
        // 取数据
    }
    get
    {
        // 存数据
    }
}
```

其中，索引的定义具有set及get方法，这一点与属性的定义相似。在set方法中，也可以使用一个特殊变量value，用以表示用户指定的值。在get方法中，使用return返回所得到的索引值。但与属性的定义不同的是，索引的定义没有名字，要用this表示索引。

索引器没有像属性和方法那样的名字，关键字this清楚地表达了索引器引用对象的特征。和属性一样，value关键字在set后的语句块里有参数传递的意义。实际上，从编译后的IL中间语言代码来看，可以这样实现一个索引器。

```
class MyClass
{
    public object get_Item(int index)
    {
        // 取数据
    }
    public void set_Item(int index, object value)
    {
        //存数据
    }
}
```

由于索引器被编译成get_Item(int index)和set_Item(int index, object value)两个方法，不能再在声明实现索引器的类里面声明实现这两个方法，所以编译器会对这样的行为报

错。这样，隐含实现的方法同样也可以被开发人员进行调用、继承等操作，这和开发人员自己实现的方法一样。通晓 C# 语言底层的编译实现，可以为下面理解 C# 索引器的行为提供一个很好的基础。

同方法一样，索引器有 5 种存取保护级别和 4 种继承行为修饰以及外部索引器。这些行为同方法没有任何差别，这里不再赘述。唯一不同的是，索引器不能是静态(static)的，这在对象引用的语义下很容易理解。值得注意的是，在覆盖(override)实现索引器时，应该用 base[E]来存取父类的索引器。同属性的实现一样，索引器的数据类型同时为 get 语句块的返回类型和 set 语句块中 value 关键字的类型。

索引器的参数列表也是值得注意的地方。索引的特征使得索引器必须具备至少一个参数，该参数位于 this 关键字之后的方括号内。索引器的参数也只能是传值类型，不可以由 ref(引用)和 out(输出)修饰。参数的数据类型可以是 C# 中的任何数据类型，C# 会根据不同的参数签名来进行索引器的多态辨析。方括号内的所有参数在 get 和 set 下都可以引用，而 value 关键字只能在 set 下作为传递参数。

下面是一个索引器的具体应用例子，它对读者理解索引器的设计和应用很有帮助。

```
class IndexerRecord
{
    private string[] data = new string[6];
    private string[] keys = { "Author", "Publisher", "Title", "Subject", "ISBN", "Comments"};
    public string this[int idx]
    {
        set
        {
            if (idx >= 0 && idx < data.Length)
                data[idx] = value;
        }
        get
        {
            if (idx >= 0 && idx < data.Length)
                return data[idx];
            return null;
        }
    }
    public string this[string key]
    {
        set
        {
            int idx = FindKey(key);
            this[idx] = value;
        }
        get
        {
            return this[FindKey(key)];
        }
    }
    private int FindKey(string key)
    {
```

```
            for (int i = 0; i < keys.Length; i++)
                if (keys[i] == key) return i;
            return -1;
        }
        static void Main()
        {
            IndexerRecord record = new IndexerRecord();
            record[0] = "谭浩强";
            record[1] = "清华大学出版社";
            record[2] = "C语言程序设计教程";
            Console.WriteLine(record["Title"]);
            Console.WriteLine(record["Author"]);
            Console.WriteLine(record["Publisher"]);
            Console.Read();
        }
    }
```

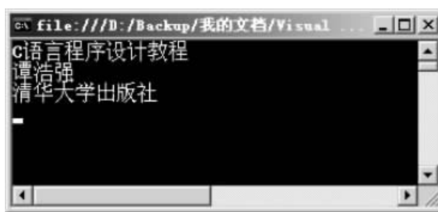

图 3.12 索引器使用的运行结果

运行结果如图 3.12 所示。

在上面的示例中，通过对索引器的使用，为用户提供了一个界面友好的字符数组，同时又大大降低了程序的存储空间代价。

3.8.2 集合

集合就如同数组，它用来存储和管理一组特定类型的数据对象。除了基本的数据处理功能外，集合还直接提供了各种数据结构及算法的实现，如队列、链表、排序等，可以让用户轻易地完成复杂的数据操作。命名空间 System.Collections 中包含了一些集合操作的接口和类，如 IList、ICollection、ArrayList、Stack、Queue、BitArray 等常用数据结构。下面讲述 System.Collections 命名空间中各个类的继承关系。

ICollection 接口是 System.Collections 命名空间中类的基接口。ICollection 接口扩展 IEnumerable、IDictionary 和 IList 则是扩展 ICollection 的更为专用的接口。IDictionary 实现是键/值对的集合，如 Hashtable 类。IList 实现是值的集合，其成员可通过索引访问，如 ArrayList 类。某些集合(如 Queue 类和 Stack 类)会限制对其元素的访问，它们直接实现 ICollection 接口。如果 IDictionary 接口和 IList 接口都不能满足所需集合的要求，则可以从 ICollection 接口派生新集合类以提高灵活性。

System.Collections 命名空间一共有 21 个。

(1) 1 个结构：DictionaryEntry。

(2) 9 个接口：IEnumerator、IEnumerable、ICollection、IList、IDictionary、IComparer、IEqualityComparer、IDictionaryEnumerator 和 IHashCodeProvider。

(3) 2 个抽象类：ReadOnlyCollectionBase 和 DictionaryBase。

(4) 2 个密封类：BitArray 和 Comparer。

(5) 7 个普通类：Stack、Queue、ArrayList、SortedList、Hashtable、CaseInsensitiveComparer 和CaseInsensitiveHashCodeProvider。

C#中 IEnumerable 与 IEnumerator 接口定义了对集合的简单迭代。IEnumerable 是

一个声明式的接口，声明实现该接口的类是可迭代(enumerable)的，但并没有说明如何实现迭代器(iterator)。IEnumerator 是一个实现式的接口，实现 IEnumerator 接口的类就是一个迭代器。

IEnumerable 和 IEnumerator 通过 IEnumerable 的 GetEnumerator()方法建立连接，可以通过该方法得到一个迭代器对象。从这个意义上，可以将 GetEnumerator()看做 IEnumerator 的工厂方法。一般将迭代器作为内部类实现，这样可以尽量减少向外暴露无关的类。一个 Collection 要支持 foreach 方式的遍历，必须实现 IEnumerable 接口（即必须以某种方式返回迭代器对象 IEnumerator）。迭代器可用于读取集合中的数据，但不能用于修改基础集合。

下面先来看一个示例。

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;
namespace test
{
    public abstract  class Animal
     {
        protected string name;
        public string Name
        {
            get { return name; }
            set { name = value; }
        }
       public Animal()
       {
           name = "The animal with no name!";
       }
       public Animal(string newName)
       {
           name = newName;
       }
       public void Feed()
       {
           Console.WriteLine("{0}has been Fed",name);
       }
    }
    public class Cow : Animal
    {
        public void Milk()
        {
            Console.WriteLine("{0}has been milked",name);
        }
        public Cow(string newName)
             : base(newName)
        {
        }
```

```
    }
    public class Chicken : Animal
    {
        public void LayEgg()
        {
            Console.WriteLine("{0}has laid an egg",name);
        }
        public Chicken(string newName)
            : base(newName)
        {
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("create an array type collection of animal " + "object and use it");
            Animal[] animalArray = new Animal[2];
            Cow mycow1 = new Cow("Ddirde");
            animalArray[0] = mycow1;
            animalArray[1] = new Chicken("ken");
            foreach (Animal myAnimal in animalArray)
            {
                Console.WriteLine("New{0}object added to Array collecton " + "NAME = {1}",
myAnimal.ToString(),myAnimal.Name);
            }
            Console.WriteLine("Array collection contains {0} object",animalArray.Length);
            animalArray[0].Feed();
            ((Chicken)animalArray[1]).LayEgg();
            Console.WriteLine();
            Console.WriteLine("Create an arraylist type collection of Animal  " + "object in
use it!");
            ArrayList animalArrarylist = new ArrayList();
            Cow mycow2 = new Cow("Hayley");
            animalArrarylist.Add(mycow2);
            animalArrarylist.Add(new Chicken("roy"));
            foreach (Animal myanimal in animalArrarylist)
            {
                Console.WriteLine("new {0} object added to Arraylist collection ," + "Name =
{1}",myanimal.ToString(),myanimal.Name);
            }
            Console.WriteLine("arraylist contains {0} objects",animalArrarylist.Count);
            ((Animal)animalArrarylist[0]).Feed();
            ((Chicken)animalArrarylist[1]).LayEgg();
            Console.WriteLine();
            Console.WriteLine("additional manipulation of animallist: ");
            animalArrarylist.RemoveAt(0);
            ((Animal)animalArrarylist[0]).Feed();
            animalArrarylist.AddRange(animalArray);
            ((Chicken)animalArrarylist[2]).LayEgg();
              Console.WriteLine("the animal called{0} is at index {1}", mycow1.Name,
```

```
animalArrarylist.IndexOf(mycow1));
            mycow1.Name = "jane";
            Console.WriteLine("the animal is now called {0} ",((Animal)animalArrarylist[1]).Name);
            Console.Read();
        }
    }
}
```

运行结果如图 3.13 所示。

```
file:///D:/Backup/我的文档/Visual Studio 2008/Projects/test/test/bin/Debug/test.EXE
create an array type collection of animal object and use it
Newtest.Cowobject added to Array collecton NAME=Ddirde
Newtest.Chickenobject added to Array collecton NAME=ken
Array collection contains 2 object
Ddirdehas been Fed
kenhas laid an egg

Create an arraylist type collection of Animal  object in use it!
new test.Cow object added to Arraylist collection ,Name =Hayley
new test.Chicken object added to Arraylist collection ,Name =roy
arraylist contains 2 objects
Hayleyhas been Fed
royhas laid an egg

additional manipulation of animallist:
royhas been Fed
kenhas laid an egg
the animal called Ddirde is at index 1
the animal is now  called jane
```

图 3.13　集合的运行结果

在这个示例中，创建了两个对象集合，第一个集合使用 System. Arrat 类(是一个简单的数组)，第二个集合使用 System. Arraylist 类。这两个集合都是 Animal 对象，都在 Animal. cs 中被定义。Animal 类是抽象类，所以它不能进行实例化。但通过多态性，可以使集合的项目成为派生于 Animal 类的 cow 和 chicken 的实例。这些数组在 program. cs 的 main()方法中创建好后，就可以显示其特性和功能了。有几个处理操作可以应用到 array 和 arraylist 集合上，但是它们的语法略有区别，也有一些操作可使用更高级的 arraylist 类型。

下面通过比较这两种集合的代码和结果，讨论这两种集合相类似的操作。

对于简单的数组来说，集合的创建必须用固定的大小来初始化数组，这样才能使用它。使用标准语法创建数组 animalarray 为：

```
Animal[] animalArray = new Animal[2];
```

另一方面，arraylist 集合不需要初始化其大小，所以可以使用下面的代码创建列表节。

```
ArrayList animalArrarylist = new ArrayList();
```

这个类还有另外两个构造函数可以使用。第一个构造函数把现有的集合作为参数，把现有的集合的内容复制到新的实例中；另外一个构造函数则通过一个参数设置集合的容量(capacity)。这个容量使用一个 int 值来指定，用于设置集合中可以包含的项目数。但是，

这并不是真实的容量,因为,如果集合的项目个数超过了这个值,容量就会自动增大一倍。

因为数组是引用类型的数组(如 Animal 和 Animal 派生的对象),所以用一个长度初始化数组,并没有初始化它所包含的项目。要使用一个指定的项目,该项目还需要进行初始化,即需要给这个项目赋初始化了的对象。例如:

```
Cow mycow1 = new Cow("Ddirde");
animalArray[0] = mycow1;
animalArray[1] = new Chicken("ken");
```

这段代码以两种方式完成了该初始化任务:用现有的 cow 对象来赋值,或者通过创建新的 chicken 对象来赋值。这两种方式的主要区别是:前者引用了数组的对象,本书中后面的代码使用了这种方式。

3.9 委托与事件

3.9.1 委托

委托的定义和方法的定义类似,只是委托的定义在前面加了一个 delegate。但委托不是方法,它是一种类型,是一种特殊的类型。常常把委托看成是一种新的对象类型,用于对与该委托有相同签名的方法调用。委托相当于 C++ 中的函数指针,但它是类型安全的。委托是从 System.Delegate 中派生出来的。但定义委托不能像定义常规类型一样,直接从 System.Delegate 派生,对委托的声明只能通过上面的声明格式进行定义。关键字 delegate 通知编译器,这是一个委托类型,从而会在编译的时候对该类进行封装。C# 定义了专门的语法来处理这一过程。

1. 委托的声明

委托是引用类型,声明一个委托的方式如下。

```
修饰符 delegate 返回类型委托名(参数列表);
```

形式参数列表指定了委托的签名,而结果类型指定了委托的返回类型。

委托的声明与方法的声明有些相似,这是因为,委托就是为了进行方法的引用。但要注意的是,委托是一种类型,而方法是类的成员,如"public delegate double MyDelegate (double x);"。

声明一个委托类型的变量与声明一个普通变量的方式一样,为:

```
委托类型名 委托变量名;
```

如:

```
MyDelegate d;
```

2. 委托的实例化

对委托进行实例化,即创建一个委托的实例的方法如下。

```
new 委托类型名( 方法名 );
```

其中,方法名可以是某个类的静态方法名,也可以是某个对象实例的实例方法名。方法的签

名及返回值类型必须与代理类型所声明的一致。例如：

```
MyDelegated d = new MyDelegate( System.Math.Sqrt );
MyDelegated d2 = new MyDelegate( obj.myMethod );
```

3. 委托的调用

委托的调用方式与方法的调用方式一样，都是传入参数，并获得返回值。形式如下：

```
委托变量名(参数列表 );
```

委托的一个重要的特点是，委托在调用方法时，不必关心方法所属的对象的类型。它只要求所提供的方法的签名和委托的签名相匹配即可。

下面通过一个委托的使用实例来学习它的用法。

下面的例子实现的是李小四委托张小三买票。首先定义一个 ZhangXiaoSan 类。

```
public class ZhangXiaoSan
{
    //其实买票的是张小三
    public static void BuyTicket()
    {
        Console.WriteLine("买车票!");
    }
    public static void BuyMovieTicket()
    {
        Console.WriteLine("买电影票!");
    }
    public static void BuyStocksTicket()
    {
        Console.WriteLine("买股票!");
    }
}
```

如果李小四只想买一种票，可以这样写：

```
class LiXiaoSi
{
    //声明一个委托
    public delegate void BTEvent();
    public static void Main(string[] args)
    {
        BTEvent myDelegate = new BTEvent(ZhangXiaoSan.BuyTicket);
        myDelegate();
        Console.ReadKey();
    }
}
```

程序运行结果如图 3.14 所示。

图 3.14 使用委托的运行结果

委托是可合并的，委托的合并称为多播（multicast）。合并的委托实际上是对多个函数的包装。对这样的委托的调用，实际上是对所包装的各个函数全部调用。其中各个

函数称为该委托的调用列表。对于多个相同类型的委托,可以用加号运算符(+)来进行调用列表的合并,可以用减号运算符(-)来移除其调用列表中的函数。同样,也可以使用“+=”,“-=”运算符。

若想实现买3种票的目的,只需在下面增加调用即可。例如:

```
class LiXiaoSi
{
    public delegate void BTEvent();
    public static void Main(string[] args)
    {
        BTEvent myDelegate = new BTEvent(ZhangXiaoSan.BuyTicket);
        myDelegate += ZhangXiaoSan.BuyMovieTicket;
        myDelegate += ZhangXiaoSan.BuyStocksTicket;
        myDelegate();
        Console.ReadKey();
    }
}
```

程序运行结果如图3.15所示。

其实,只是在程序中加了“myDelegate += ZhangXiaoSan.BuyMovieTicket;”、“myDelegate += ZhangXiaoSan.BuyStocksTicket;”这两条语句,就实现了买3种票的目的。

如果取消以前的委托,例如取消买电影票,那又如何实现呢?同理,可以再增加一条语句,如:

```
public static void Main(string[] args)
{
    BTEvent myDelegate = new BTEvent(ZhangXiaoSan.BuyTicket);
    myDelegate += ZhangXiaoSan.BuyMovieTicket;
    myDelegate += ZhangXiaoSan.BuyStocksTicket;
myDelegate -= ZhangXiaoSan.BuyMovieTicket;
    myDelegate();
    Console.ReadKey();
}
```

程序运行结果如图3.16所示。

图3.15 使用多播的运行结果

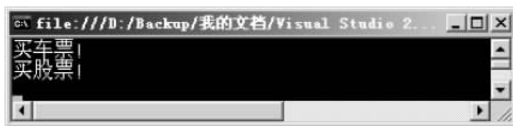

图3.16 取消委托的运行结果

3.9.2 事件

事件就是指当对象或类的状态发生改变时,对象或类发出的信息或通知。发出信息的对象或类称为事件源,对事件进行处理的方法称为接收者。通常,事件源在发出状态改变信息时,它并不知道由哪个事件接收者来处理,这就需要一种管理机制来协调事件源和接收

者。在 C++ 中，这是通过函数指针来完成的。在 C# 中，事件使用委托来为触发时将调用的方法提供类型安全的封装。

1. 声明一个委托

声明一个委托，如：

```
public delegate void EventHandler(object sender, System.EventArgs e);
```

2. 声明一个事件

声明一个事件的方式如下：

```
修饰符 event 指代类型名事件名;
```

其中，修饰符可以为访问控制符（public、protected、internal、private 或 protected internal）以及其他修饰符，如 static、new、virtual、abstract、override 或 sealed 等。声明一个事件，如“public event EventHandler Changed;”等。

3. 引发一个事件

```
public OnChanged(EnventArgs e)
{
    if ( Changed != null)
    {
        Changed(this,e);
    }
}
```

4. 定义事件处理程序

```
public MyText_OnChanged(Object sender,EventArgs e)
{
    …
}
```

5. 订阅事件（将事件处理程序添加到事件的调用列表中）

```
myText.Changed += EventHandler(MyText_OnChanged);
```

下面的一个小例子说明了怎样定义一个完整的事件机制。

```
class Program
{
    static void Main(string[ ] args)
    {
        MyText myText = new MyText();
        // 将事件处理程序添加到事件的调用列表中(即事件布线)
        myText.Changed += new MyText.ChangedEventHandler(myText_Changed);
        string str = "";
        while (str != "quit")
        {
            Console.WriteLine("please enter a string:");
            str = Console.ReadLine();
            myText.Text = str;
        }
```

```
            Console.Read();
        }
        //处理 Change 事件的程序
        private static void myText_Changed(object sender, EventArgs e)
        {
            Console.WriteLine("text has been changed  :{0}\n", ((MyText)sender).Text);
        }
    }
    public class MyText
    {
        private string _text = "";
        // 定义事件的委托
        public delegate void ChangedEventHandler(object sender, EventArgs e);
        // 定义一个事件
        public event ChangedEventHandler Changed;
        // 用以触发 Change 事件
        protected virtual void OnChanged(EventArgs e)
        {
            if (this.Changed != null)
                this.Changed(this, e);
        }
        // Text 属性
        public string Text
        {
            get { return this._text; }
            set
            {
                this._text = value;
                // 文本改变时触发 Change 事件
                this.OnChanged(new EventArgs());
            }
        }
    }
```

程序运行结果如图 3.17 所示。

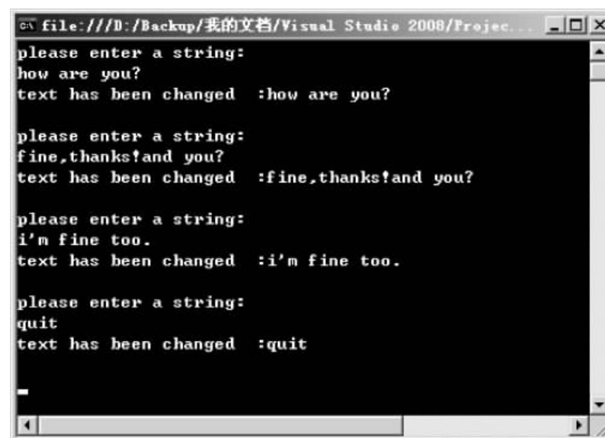

图 3.17 事件的运行结果

3.10 重载

重载，简单说，就是函数或者方法有同样的名称，但是参数列表不相同的情形，这样的同名不同参数的函数或者方法之间，互相称为重载函数或者重载方法。重载是多态性的表现，本质上是在相同的函数名或操作符下由于引用函数的实参个数和类别的不同及操作数的类型不同会运行不同的函数代码。下面是整数和浮点数的 add 函数及相应的引用代码例子。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _3_10
{
    class MyAdd
    {
        public int add(int a, int b)
        {
            int s;
            s = a + b;
            return s;
        }
        public float add(float a, float b)
        {
            float s;
            s = a + b;
            return s;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            int x = 2;
            float y = 10.3f;
            MyAdd myadd = new MyAdd();
            Console.WriteLine("调用整数运算,结果为:" + myadd.add(x, 10).ToString());
            Console.WriteLine("调用浮点数运算,结果为:" + myadd.add(x, 10.2f).ToString());
            Console.ReadLine();
        }
    }
}
```

运算结果如图 3.18 所示。

运算符重载是函数重载的一种特殊情况，下面详细介绍操作符重载。

图 3.18 函数重载运行结果

操作符是 C# 中用于定义类的实例对象间表达式操作的一种成员。和索引器类似，操作符仍然是对方法实现的一种逻辑界面的抽象，也就是说，在编译成的 IL 中间语言代码中，操作符仍然是以方法的形式调用的。C# 中的重载操作符共有 3 种：一元操作符、二元操作符和转换操作符。并不是所有的操作符都可以重载的，可以重载的运算符如表 3.1 所示。

表 3.1 可重载运算符

<table>
<tr><th>类 别</th><th>运 算 符</th><th>限 制</th></tr>
<tr><td>算术二元运算符</td><td>+,*,/,-,%</td><td>无</td></tr>
<tr><td>算术一元运算符</td><td>+,-,++,--</td><td>无</td></tr>
<tr><td>按位二元运算符</td><td>&,|,^,<<,>></td><td>无</td></tr>
<tr><td>按位一元运算符</td><td>!,~,true,false</td><td>true 和 false 运算符必须成对重载</td></tr>
<tr><td>比较运算符</td><td>==,!=,>=,<=,<,></td><td>必须成对重载</td></tr>
<tr><td>赋值运算符</td><td>+=,-=,*=,/=,>>=,<<=,%=,&=,|=,^=</td><td>不能显式重载这些运算符，在重写单个运算符如“+”，“-”，“%”等时，它们会被隐式重写</td></tr>
<tr><td>索引运算符</td><td>[]</td><td>不能直接重载索引运算符。索引器成员类型允许在类和结构上支持索引运算符</td></tr>
<tr><td>数据类型转换运算符</td><td>()</td><td>不能直接重载数据类型转换运算符。允许定义定制的数据类型转换</td></tr>
</table>

一元操作符声明的形式如下：

```
public static 类型 operator 一元操作符 ( 类型参数名 ){ … }
```

二元操作符声明的形式如下：

```
public static 类型 operator 二元操作符 ( 类型参数名,类型 参数名) … }
```

重载操作符必须是由 public 和 static 修饰的，否则会引起编译错误，父类的重载操作符会被子类继承，但这种继承没有覆盖、隐藏、抽象等行为，所以不能对重载操作符进行 virtual、sealed、override、abstract 修饰。操作符的参数必须为传值参数。例如：

```
class Complex
{
    double r, v; //r+ v i
    public Complex(double r, double v)
```

```
        {
            this.r = r;
            this.v = v;
        }
        public static Complex operator +(Complex a, Complex b)
        {
            return new Complex(a.r + b.r, a.v + b.v);
        }
        public static Complex operator -(Complex a)
        {
            return new Complex(-a.r, -a.v);
        }
        public static Complex operator ++(Complex a)
        {
            double r = a.r + 1;
            double v = a.v + 1;
            return new Complex(r, v);
        }
        public void Print()
        {
            Console.WriteLine (r + " + " + v + "i");
        }
    }
    class Test
    {
        public static void Main()
        {
            Complex a = new Complex(1, 3);
            Complex b = new Complex(4, 6);
            Complex c = -a;
            c.Print();
            Complex d = a + b;
            d.Print();
            a.Print();
            Complex e = a++;
            a.Print();
            e.Print();
            Complex f = ++a;
            a.Print();
            f.Print();
            Console.Read();
        }
    }
```

运行结果如图 3.19 所示。

图 3.19　重载操作符的运行结果

上面的例子实现了一个“＋”号二元操作符、一个“－”号一元操作符（取负值）和一个“＋＋”一元操作符。注意，这里没有对传进来的参数作任何改变。这一点在参数是引用类型的变量时尤其重要，虽然重载操作符的参数只能是传值方式。而在返回值时，往往需要一个新的变量，true 和 false 操作符除外。这在重载“＋＋”和“－－”操作符时尤其显得重要。也就

是说，在做 a++时，将丢弃原来的 a 值，把新的 new 出来的值给 a。值得注意的是，e=a++或 f=++a 中 e 的值或 f 的值与重载的操作符返回值没有一点联系，它们的值仅仅是在前置和后置的情况下获得 a 的旧值或新值而已。前置和后置的行为不难理解。

关系运算符(如“==”或“<”)也可以重载，其重载过程非常直观。重载关系运算符一般返回 true 或 false。重载关系运算符保持了这些运算符的正常用法，同时，它允许用在条件表达式中。如果返回其他类型的结果，就会极大地限制运算符的可用性。

下面是一个重载“<”和“>”运算符的 ThreeD 类版本。在下面的例子中，这些运算符都基于它们离开原点的距离来比较 ThreeD 对象。如果一个对象离开原点的距离大于另一个对象离开原点的距离，则前者就是较大的对象。如果一个对象离开原点的距离小于另一个对象离开原点的距离，则前者就是较小的对象。对于给出的两个点，可以使用这样的实现规则来确定哪一点位于较大的球面上。如果两个运算符都没有返回真值，那么这两个点就位于相同的球面上。当然，也可能有其他排序模式。例如：

```
namespace test
{
    class ThreeD
    {
        int x, y, z; // 3 - D coordinates
        public ThreeD() { x = y = z = 0; }
        public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }
        public static bool operator <(ThreeD op1, ThreeD op2)
        {
            if (Math.Sqrt(op1.x * op1.x + op1.y * op1.y + op1.z * op1.z) <
                Math.Sqrt(op2.x * op2.x + op2.y * op2.y + op2.z * op2.z))
                return true;
            else
                return false;
        }
        public static bool operator >(ThreeD op1, ThreeD op2)
        {
            if (Math.Sqrt(op1.x * op1.x + op1.y * op1.y + op1.z * op1.z) >
                Math.Sqrt(op2.x * op2.x + op2.y * op2.y + op2.z * op2.z))
                return true;
            else
                return false;
        }
        public void Show()
        {
            Console.WriteLine(x + "," + y + "," + z);
        }
    }
    class ThreeDDemo
    {
        static void Main()
        {
            ThreeD a = new ThreeD(3, 4, 5);
            ThreeD b = new ThreeD(7, 8, 9);
```

```
            ThreeD c = new ThreeD(1, 2, 3);
            ThreeD d = new ThreeD(5, 4, 3);
            Console.Write("Here is a: ");
            a.Show();
            Console.Write("Here is b: ");
            b.Show();
            Console.Write("Here is c: ");
            c.Show();
            Console.Write("Here is d: ");
            d.Show();
            Console.WriteLine();
            if (a > c) Console.WriteLine("a > c is true");
            if (a < c) Console.WriteLine("a < c is true");
            if (a > b) Console.WriteLine("a > b is true");
            if (a < b) Console.WriteLine("a < b is true");
            if (a > d) Console.WriteLine("a > d is true");
            else if (a < d) Console.WriteLine("a < d is true");
            else Console.WriteLine("a and d are same distance from origin");
            Console.Read();
        }
    }
}
```

运行结果如图 3.20 所示。

```
file:///D:/Backup/我的文档/Visual Studio 2008/Projects/test/test/bin/Debu...
Here is a: 3,4,5
Here is b: 7,8,9
Here is c: 1,2,3
Here is d: 5,4,3

a > c is true
a < b is true
a and d are same distance from origin
```

图 3.20　重载关系运算符的运行结果

重载关系运算符的应用有一个重要限制，即必须成对地重载关系运算符。例如，如果重载“＜”，就必须也重载“＞”，反之亦然。另外重载运算符时不能更改任何运算符的优先级，也不能更改运算符所需操作数的个数（尽管运算符方法可以选择忽略一个操作数）。有些运算符不能重载，如赋值运算符，包括复合赋值运算符（如“＋＝”）。表 3.2 列出了一些不能重载的运算符。

表 3.2　不能重载的运算符

&&	()	.	?
??	[]	||	=
=>	->	As	checked
default	is	new	sizeof
typeof	unchecked		

习题 3

1. 简述 private、protected、public 和 internal 修饰符的访问权限。
2. 结构和类的区别是什么?
3. 错误和异常有什么区别?为什么要进行异常处理?用于异常处理的语句有哪些?
4. 简要回答抽象类和接口的主要区别。
5. 使用委托的优点是什么?委托和事件有哪些区别和联系?