第 7 章　面向对象编程技术

本章要点

- 面向对象编程基本思想的介绍
- 类与对象的建立，构造函数与析构函数的使用
- 继承与多态的使用
- 接口的使用方式

在.NET 语言产生以前，我们使用的语言大多是 C 语言和 C++。然而 C 语言是个面向过程的程序设计语言，在使用中有很多的缺点，例如：

- 功能与数据分离，不符合人们对现实世界的认识规律，要保持功能与数据的相容也十分困难。
- 基于模块的设计方式，导致软件修改困难。
- 自顶向下的设计方法，限制了软件的可重用性，降低了开发效率，也导致最后开发出来的系统难以维护。

为了解决结构化程序设计的诸多问题，面向对象编程技术就被提出，20 世纪 80 年代初美国 AT&T 贝尔实验室设计并实现了 C++语言，增加了面向对象程序设计的支持。

Visual C#语言秉承了 C++语言面向对象的特性，支持面向对象的所有关键概念：封装、继承和多态。

7.1 面向对象编程基本思想

面向对象编程(OOP)与面向过程编程(如 C、Pascal 中)有几方面不同之处，任何东西在 OOP 中都是通过对象组织起来的。面向对象编程从最纯粹的观念上定义就是：通过向对象发送消息来完成任务。可以这样认为：“面向对象=对象+类+继承+通信”。如果一个软件系统是使用这样 4 个概念来设计和实现的，那么我们就认为这个软件系统是面向对象的。为了了解这些概念，首先需要知道对象是什么。

1．什么是对象(Object)

对象是问题域或实现域中某些事物的一个抽象，它反映此事物在系统中需要保存的信息和发挥的作用；它是一组属性和有权对这些属性进行操作的一组服务的封装体。关于对象，要从两方面去理解：一方面指系统所要处理的现实世界中的对象；另一方面是计算机不直接处理对象，而是处理相应的计算机表示，这种计算机表示也称为对象。简单地说，一个人就是一个对象，

一个尺子也可以说是个对象。当这些对象可以用数据直接表示时，就称为属性，尺子的度量单位可以是厘米、公尺或英尺，这些度量单位就是尺子的属性。

对象的接口由一组消息通过传递组成，每个命令执行一个特定的动作。一个对象通过发送一个消息来要求另一个对象执行一个动作。把请求(发送)的对象称为发送方而把接收的对象称为接收方，如图 7.1 所示。

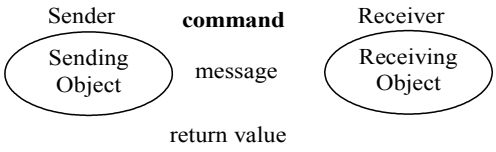

图 7.1 发送方与接收方

控制权交给了接收对象直到它完成这个命令为止；然后控制权返回给发送对象，如图 7.2 所示。比如，一个 School 对象通过给 Student 对象发送一个请求其名字的消息来获取他的名字。接收对象 Student 返回其名字给发送对象。

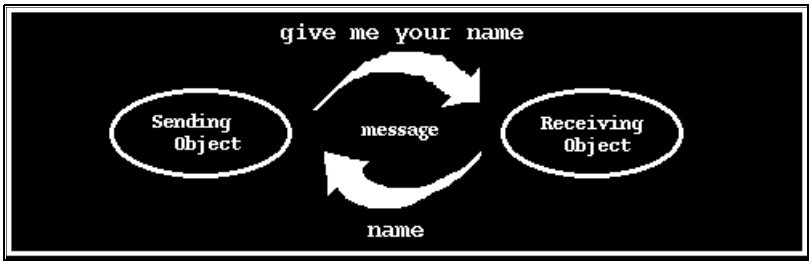

图 7.2 发送对象与接收对象

一个消息当然能够包含发送对象需要传递给接收对象的信息，这些信息称为参数。接收对象总是给发送对象一个返回值。这个返回值对发送对象可能有用也可能没用，如图 7.3 所示。比如, School 对象现在想改变学生的名字。它通过发送一个消息给 Student 对象来将它的名字改为新名字。新的地址也作为一个参数包含在消息中被传递。这种情况下，这个 School 对象并不关心从消息中返回的值。

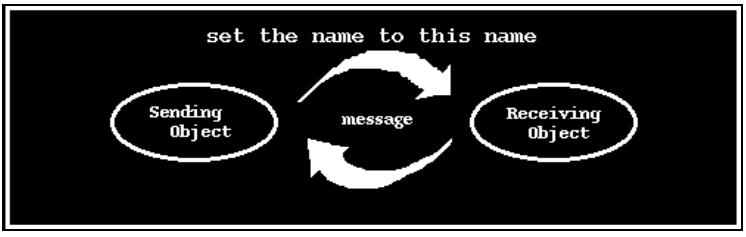

图 7.3 发送对象与接收对象之间的消息传送

2．什么是类(Class)

类是一组具有相同数据结构和相同操作的对象的集合。类是对一系列具

有相同性质的对象的抽象，是对对象共同特征的描述。比如每一辆汽车都是一个对象的话，所有的汽车可以作为一个模板，我们就定义汽车这个类。

在一个类中，每个对象都是类的实例，可以使用类中提供的方法。从类中产生对象时，必须有建立实例的操作，C++和 C#中的 New 操作符可用于建立一个类的实例。C#为我们提供的方法则更加安全。

3．什么是继承(Inheritance)

继承是使用已存在的定义作为基础建立新定义的技术。新类的定义可以是现有类所声明的数据和新类所增加的声明的组合。新类可以复用现有类的定义，而不要求修改现有类，现有类可以作为基类来引用，而新类可以作为派生类来引用。这种复用技术大大降低了软件开发的费用，例如，动物作为一个类已经存在，作为具有自身特征的狗就可以从动物类中继承。它同动物一样，具有眼睛、耳朵这些特征，可以执行奔跑和饮食方法。它还具有一般动物所不具备的犬吠。

7.2 类与对象的建立

类是面向对象的程序设计的基本构成模块。从定义上讲，类是一种数据结构，这种数据结构可能包含数据成员、函数成员以及其他的嵌套类型。其中数据成员类型有常量、字段和事件；函数成员类型有方法、属性、索引器、操作符、构造函数和析构函数。

在 C#中使用 class{...}来定义一个类，对于类，其实在前面我们已经多次使用了，只是没有专门说明。要注意的是类的定义无论放在哪儿都可以，但是不可以放在 namespace{}之外或者函数之内，也就是说，类定义是全局性的声明。

类的声明格式如下：

```
class-modifers class classname
{
    //...
}
```

其中 class-modifers 为类的修饰符，classname 为类的类名。

类的修饰符可以是以下几种之一或者是它们的组合(在类的声明中同一修饰符不允许出现多次)。

- new：仅允许在嵌套类声明时使用，表明类中隐藏了由基类中继承而来的、与基类名相同的成员。
- public：表示不限制对该类的访问。
- protected：表示只能从所在类和所在类派生的子类进行访问。
- internal：此成员只在当前编译单元中可见，internal 访问修饰符是根

据代码所在的位置而不是类在层次结构中的位置决定可见性。

- private：不能在定义此成员的类之外访问它。因此，即使是派生类也不能访问。
- abstract：抽象类，不允许建立类的实例。
- sealed：密封类，不允许被继承。

下面先来声明一个空白的类，并且使用此类来创建一个对象。

例 7.1 创建一个类，并使用此类创建一个对象。

程序代码：

```
namespace ch07_1
{
    class Program
    {
        static void Main(string[] args)
        {
            student st = new student();
            st.name = "张三";
            st.age = 18;
            st.print();
        }
    }
   public  class student
    {
        public string name;
        public int age;
        public void print()
        {
            Console.WriteLine("我叫{0}，今年{1}岁",name,age);
        }
    }
}
```

运行结果如图 7.4 所示。

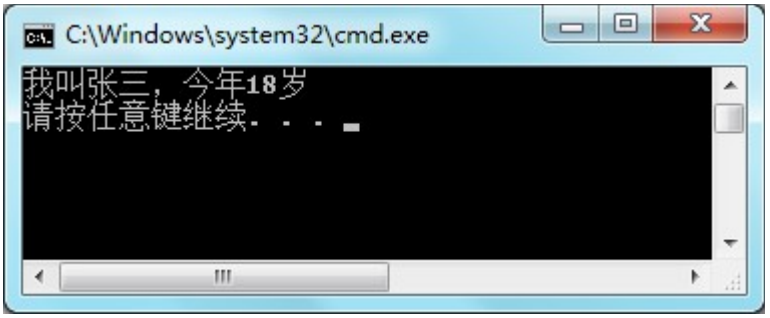

图 7.4 例 7.1 的运行结果

以上例子声明了一个名为 student 的类，并且使用代码 student st = new student();建立一个新的对象 st。

7.3 构造函数和析构函数

关于类的成员，其中函数、字段以及属性我们在第 5 章已经讲解过，本节我们将讲解类中的构造函数和析构函数。

7.3.1 构造函数

构造函数用于执行类的实例的初始化。每个类都有构造函数，即使我们没有声明它，编译器也会自动地为我们提供一个默认的构造函数。构造函数的名称与类名相同，而且在语法上类似于函数。但是，构造函数没有明确的返回类型。一般构造函数总是 public 类型的，若是 private 类型的，表明类不可以被实例化，这通常用于只含有静态成员的类。在构造函数中不要做对类的实例进行初始化以外的事情，也不要尝试显式地调用构造函数。

例 7.2 修改例 7.1，创建构造函数。

程序代码：

```
namespace ch07_2
{
    class Program
    {
        static void Main(string[] args)
        {
            student st = new student("张三",18);
            st.print();
        }
    }
    public class student
    {
        public student(string nam, int ag)
        {
            this.name = nam;
            this.age = ag;
        }
        public string name;
        public int age;
        public void print()
```

```
        {
            Console.WriteLine("我叫{0}，今年{1}岁", name, age);
        }
    }
}
```

程序运行结果如图 7.4 所示。

在上面例子中的构造函数有两个参数，可以在一个类中定义多个构造函数，这些构造函数的参数个数和类型必须有一定的差异，以便于区分。在类实例化的时候，会自动根据参数的类型和个数去寻找匹配的构造函数。下面举个具体的例子。

例 7.3 创建一个有 3 个构造函数的类，用于计算 1、2、3 个数的和。

程序代码：

```
namespace ch07_3
{
    class Program
    {
        static void Main(string[] args)
        {
            Area ar1 = new Area(1);
            ar1.print();
            Area ar2 = new Area(2, 3);
            ar2.print();
            Area ar3 = new Area(4, 5, 6);
            ar3.print();
        }
    }
    public class Area
    {
        private int result;
        public Area(int x)
        {
            result = x;
        }
        public Area(int x,int y)
        {
            result = x + y;
        }
        public Area(int x,int y,int z)
        {
```

```
            result = x + y + z;
        }
        public void print()
        {
            Console.WriteLine(result);
        }
    }
}
```

程序运行结果如图 7.5 所示。

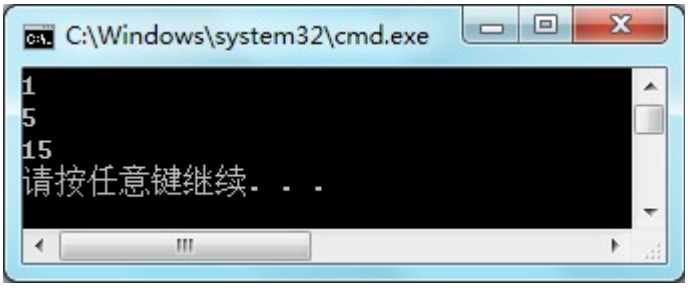

图 7.5 例 7.3 的运行结果

从上面的例子可以看到，类 Area 有 3 个构造函数。若函数的签名仅由函数名组成，那么编译器就不知道该调用哪个构造函数来分别创建各个实例了。正是由于构造函数签名包括了函数的参数类型以及参数个数，所以编译器可以识别并且匹配，以至于能够帮我们正确地解决问题。

7.3.2 析构函数

在使用 new 运算符给对象分配动态空间的时候，由于内存也是有限的，可能会被用完，所以在类的实例超出范围时，我们希望确保它所占的内存能被收回。C#中提供了析构函数，专门用于释放被占用的系统资源。

定义一个在无用单元收集程序进行对象的最后消除之前调用的方法，这个方法就称为析构函数。析构函数的名称与类名也相同，只是在前面加了一个符号“~”。析构函数不接受任何参数，也不返回任何值。如果试图声明其他任何一个以符号“~”开头而不与类名相同的方法，编译器就会产生一个错误。

析构函数的基本形式为：

```
~classname
{
    //code;
}
```

析构函数不能是继承而来的，也不能显式地调用。当某个类的实例被认

为不再有效，符合析构的条件时，析构函数就可能在某个时刻被执行。

例 7.4 创建一个析构函数实例

程序代码：

```
namespace ch07_4
{
    class Program
    {
        static void Main(string[] args)
        {
            student st = new student("王菲");
            st.print();
        }
    }
    public class student
    {
        private string name;
        public student(string nam)
        {
            name = nam;
        }
        public void print()
        {
            Console.WriteLine(name);
        }
        ~student()
        {
            Console.WriteLine("bye");
        }
    }
}
```

程序运行结果如图 7.6 所示。

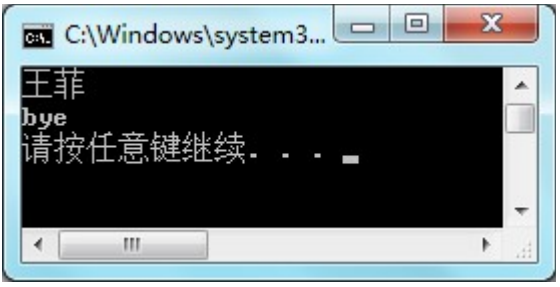

图 7.6 例 7.4 的运行结果

7.4 继承与多态

了解了类的基本定义以及对象的使用之后，接下来就要进一步讲解Visual C#面向对象中的两个最为有力的机制——继承与多态。

如果所有的类都处在同一级别上，这种没有相互关系的平坦结构就会限制系统面向对象的特性。继承的引入，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，建立起类的层次。

同一操作作用于不同的对象，可以有不同的解释，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。

7.4.1 继承

类继承的基本语法是：

```
class A
{
    //Acode;
}
class B : A
{
    //Bcode;
}
```

上述代码就是类继承的基本样式，类 B 继承于类 A，我们把类 A 称作父类(也叫基类)，类 B 称作子类(也叫派生类)。需要注意的是，不同于 C++，在 Visual C#中，派生类只可以从一个类中继承。派生类从基类中继承除了构造函数和析构函数以外的所有成员，如函数、字段、属性、事件和索引器等。

在下面的例子中，首先定义一个 Person 类，而后定义了一个继承自 Person 类的新类 Woman。

例 7.5 关于类继承的一个小例子。

程序代码：

```
namespace ch07_5
{
    class Program
    {
        static void Main(string[] args)
        {
            Woman wm = new Woman("江苏太仓",170,"long");
            wm.Description();
```

```
        }
    }
    class Person
    {
        public string jiguan;   //籍贯
        public int Height;          //身高
    }
    class Woman : Person
    {
        public string hair;     //头发
        public Woman(string jg,int Hei,string hr)
        {
            jiguan = jg;
            Height = Hei;
            hair = hr;
        }
        public void Description()
        {
                Console.WriteLine("籍贯={0}, 身高={1}cm, 头发
={2}",jiguan, Height,hair);
        }
    }
}
```

程序运行结果如图 7.7 所示。

图 7.7 例 7.5 的运行结果

在这个例子中，Woman 类继承自类 Person，我们在类 Woman 使用了类 Person 的字段，并且增加了 Description()这个方法。

C#中的继承应遵循下列规则：

- 继承是可以传递的。如果类 C 继承于类 B，同时类 B 又继承于类 A，那么类 C 不仅继承了类 B 中声明的成员，同样也继承了类 A 中的成员。Object 类是所有类的基类。
- 派生类应该是对基类的扩展。派生类可以添加新的成员，但是不能除去已经继承的成员的定义。

- 构造函数和析构函数不可以被继承。除此以外的其他成员，不论对它们定义了怎样的访问方式，都能被继承。基类中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，就可以覆盖已继承的成员。但这并不意味着派生类删除了这些成员，只是不能再访问这些成员。
- 类可以定义虚方法、虚属性以及虚索引器，它的派生类能够重载这些成员，从而实现类可以展示出多态性。

在例 7.5 中，我们注意到 Woman 类中的成员都声明为 public 类型的，关于类中成员的保护等级，在前面的章节提到过，这里再强调一下。

(1) public：使用 public 所声明出来的成员，就会变成类的一个接口，也就是允许任何来自外界的直接访问。

(2) private：使用 private 所声明出来的成员，只允许类中的程序来引用，外界不可以使用。

(3) protected：使用 protected 所声明的成员，不仅仅可以在本类中使用，在本类的派生类中也可以使用。

7.4.2 多态

在面向对象编程思想中，多态是一个非常重要的概念。在程序的编写过程中，如果子类需要对父类中定义的方法进行修正或者增加新功能，就可以在子类中重新定义原本继承自父类中的方法。

下面先看一个简单的多态的例子。

例 7.6 将例 7.5 简单修改为一个多态的例子。

程序代码：

```
namespace ch07_6
{
    class Program
    {
        static void Main(string[] args)
        {
            Person ps = new Person("江苏南京",180);
            ps.Description();
            Woman wm = new Woman("江苏太仓", 170, "long");
            wm.Description();
            ps = wm;
            ps.Description();
            Man mn = new Man("江苏南京", 175, false);
            mn.Description();
```

```
            ps = mn;
            ps.Description();
        }
    }
    class Person
    {
        public string jiguan;   //籍贯
        public int Height;
        public Person(string pjiguan,int pHeight)
        {
            jiguan = pjiguan;
            Height = pHeight;
        }
        public virtual void Description()
        {
            Console.WriteLine("这是基类(Person类）");
        }
    }
    class Woman : Person
    {
        public string hair;     //头发
            public Woman(string pjiguan, int pHeight, string
Phr):base(pjiguan,pHeight)
        {
            jiguan = pjiguan;
            Height = pHeight;
            hair = Phr;
        }
        public override void Description()
        {
            Console.WriteLine("这是派生类（Woman类)");
              Console.WriteLine("籍贯={0},身高={1}cm,头发={2}",
jiguan, Height, hair);
        }
    }
    class Man : Person
    {
        public bool zy;
          public Man(string pjiguan, int pHeight, bool pzy) :
```

```
base(pjiguan, pHeight)
            {
                jiguan = pjiguan;
                Height = pHeight;
                zy = pzy;
            }
            public override void Description()
            {
                Console.WriteLine("这是派生类（Man 类）");
                  Console.WriteLine("籍贯={0},身高={1}cm,是否抽烟
={2}", jiguan, Height, zy);
            }
        }
    }
```

程序运行结果如图 7.8 所示。

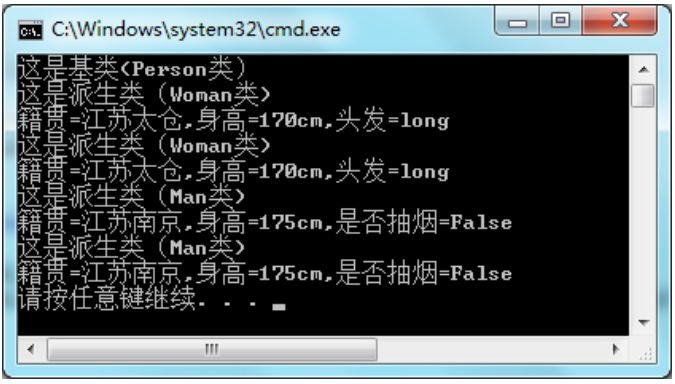

图 7.8　例 7.6 的运行结果

在上面的例子中，我们在基类 Person 中的 Description()方法声明前添加了 virtual 修饰符，使之成为虚方法。使用了 virtual 修饰符后，就不允许再使用 static、abstract 或者 override 修饰符。大家应该注意到，在派生类中，我们也定义了 Description()方法，且在 Description()方法之前添加了 override 修饰符，这样就完成了派生类中声明对虚方法的重载，添加 override 修饰符之后，就不能有 new、static 或者 virtual 修饰符了。

另外，还可以从上面的例子看出，Person 类的实例 ps 先后被赋予 Woman 类的实例 wm 以及 Man 类的实例 mn。在执行的过程中，ps 先后指代不同的类的实例，从而调用不同的版本。这里 ps 的 Description()方法实现了多态性，并且 ps.Description()究竟执行哪个版本，不是在程序编译时确定的，而是在程序动态运行时，根据 ps 某一时刻的指代类型来确定的，所以体现了动态的多态性。

在 C#中，编译时的多态主要通过函数的重载以及操作符重载来实现，而

运行时的多态主要通过虚成员来实现。

7.4.3 抽象与密封

在创建一个基类的时候，有时候我们只想定义派生类的一般化形式，而后让派生类具体实现内容。这种类决定了方法(派生类必须实现这些方法，而此类本身不提供这些方法的实现)，我们把这种类称之为抽象类。抽象类使用 abstract 关键字作为修饰符。

对于抽象类的使用，有以下几点规定：

- 抽象类只可以作为其他类的基类，它不能直接被实例化，而且抽象类不能使用 new 操作符。如果抽象类中含有抽象的变量或值，则它们要么是 Null 类型，要么包含了对非抽象类的实例的引用。
- 抽象类可以包含抽象成员。
- 抽象类不可以又是封装的。即一个类中不可以 abstract 关键字与 sealed 关键字共存(关键字为 sealed 的类称之为封装类，在此内容后面我们就讨论它)。

如果一个类 A 派生于一个抽象类 B，那么这个类 A 就要通过重载方法实现所有这个抽象类 B 中的抽象成员。下面举一个具体的简单例子。

例 7.7 一个简单抽象类的例子。

程序代码：

```
namespace ch07_7
{
    class Program
    {
        static void Main(string[] args)
        {
            Redcar rd = new Redcar();
            rd.Run();
        }
    }
    abstract class car
    {
        public abstract void Run();
    }
    class Redcar : car
    {
        public override void Run()
        {
            Console.WriteLine("Red Car No:999 is running!");
```

```
        }
    }
}
```

程序运行结果如图 7.9 所示。

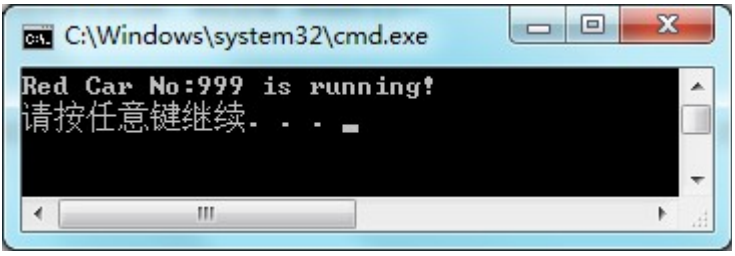

图 7.9 例 7.7 的运行结果

上面的例子就是一个简单的抽象类的例子，先是定义了抽象类 car 以及抽象类 car 中的抽象函数 Run，而后又定义了类 Redcar 作为抽象类 car 的派生类，正如前面所述，在类 Redcar 中，我们必须对函数 Run 进行重载，因为函数 Run 是作为一个存在于抽象类中的抽象方法。

需要注意的是，并非抽象类中的所有函数和变量都必须是抽象的，也可以不是抽象的。若不是抽象的，则用途跟我们前面所讲的类的函数以及变量的用法一样，但是要切记，不可以对抽象类进行实例化。抽象方法必须定义在抽象类中，不可以在派生类中使用 base 关键字对抽象方法进行访问。例如，下面的代码编译的时候就会报错：

```
abstract class A
{
    public abstract void F();
}
class B : A
{
    public override void F()
    {
        base.F();  //错误!!!
    }
}
```

我们还可以使用抽象方法重载基类的虚方法，这时基类中的虚方法的执行代码就会被拦截，具体的见下面的例子。

例 7.8 使用抽象方法重载基类的虚方法。

程序代码：

```
namespace ch07_8
{
    class Program
```

```
    {
        static void Main(string[] args)
        {
            Redcar rd = new Redcar();
            rd.Run();
        }
    }
    class vehicle
    {
        public virtual void Run()
        {
            Console.WriteLine("这是基类（vehicle类）");
        }
    }
    abstract class car:vehicle
    {
        public abstract override void Run();
    }
    class Redcar : car
    {
        public override void Run()
        {
            Console.WriteLine("Red Car No:999 is running!");
        }
    }
}
```

程序运行结果如图 7.10 所示。

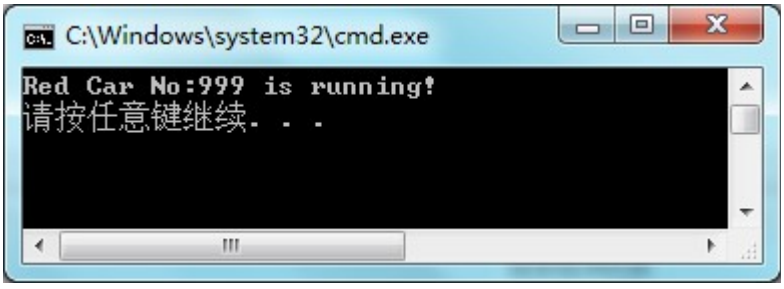

图 7.10 例 7.8 的运行结果

在上面的例子中，我们在类 vehicle 中声明了一个虚函数 Run，派生类 car 使用抽象方法重载了函数 Run，这样类 car 的派生类 Redcar 就可以重载 Run 并提供自己的实现。

密封与抽象相比在某种程度上可以看作是两个相反的概念。上面讲到，

抽象类必须被继承，而密封类恰恰相反，密封类不允许建立其派生类。密封类的好处是，不允许类被随意地继承，可以避免类的层次结构变得复杂庞大。

密封类在声明时使用关键字 sealed，这样就可以防止该类被其余类所继承。

密封函数与密封类类似，使用关键字 sealed 修饰函数，这样，可以防止在类的派生类中实现对函数的重载。不是类的每个成员函数都可以作为密封函数，密封函数必须对基类的虚方法进行重载，提供具体的实现方法。所以，在函数的声明中，sealed 修饰符总是和 override 修饰符同时使用的。

例 7.9 一个简单密封函数的例子。

程序代码：

```
namespace ch07_9
{
    class Program
    {
        static void Main(string[] args)
        {
            car cr = new car();
            cr.Run();
            cr.wheels();
            Redcar rd = new Redcar();
            rd.Run();
            rd.wheels();
        }
    }
   abstract class vehicle
    {
        public  virtual void Run()
        {
            Console.WriteLine("这是 vehicle 中的 Run 方法");
        }
       public  abstract void wheels();
    }
     class car : vehicle
    {
        public sealed override void Run()
        {
            Console.WriteLine("这是 car 中的 Run 方法");
        }
        public override void wheels()
        {
```

```
            Console.WriteLine("这是 car 中的 wheels 方法");
        }
    }
    class Redcar : car
    {
        public override void wheels()
        {
            Console.WriteLine("这是 Redcar 中的 wheels 方法");
        }
    }
}
```

程序的运行结果如图 7.11 所示。

图 7.11 例 7.9 的运行结果

在以上的例子中，先是定义了抽象类 vehicle，在抽象类 vehicle 中我们定义了两个成员：虚函数 Run 以及抽象函数 wheels。而后又定义了类 vehicle 的派生类 car，car 的成员也有 2 个，其中 Run 函数是一个密封函数，是对类 vehicle 中的 Run 函数的重写；wheels 函数是对类 vehicle 的 wheels 函数的重写，这个是必需的，因为类 vehicle 中的 wheels 函数是个抽象函数。类 Redcar 派生于类 car，Redcar 中也重写了 wheels 函数，但是 Redcar 中不可以重载 Run 函数，否则编译器会报错。

7.5 接　　口

在面向对象程序设计中，定义类必须完成的内容而不是完成的方法有时是很有帮助的。大家已经见过使用抽象方法的示例。抽象方法定义函数的签名而不提供实现，需要每个派生类必须提供基类定义的所有抽象函数的实现。因此，抽象函数指定了函数的接口而不是实现。

7.5.1 接口的声明以及实现

接口在语法上与抽象类类似。接口是把隐式公共方法和属性组合起来，

以封装特定功能的一个集合。一旦定义了一个接口，就可以在类中执行它。这样，类就可以支持接口所指定的所有属性和成员。

值得注意的是，接口是不可以单独存在的，不能像实例化一个类那样实例化一个接口。另外，接口不能包含执行其成员的任何代码，而只能定义成员本身。执行过程必须在执行接口的类中实现。另外，一个类可以实现多个接口，若要实现多个接口，必须将这些接口之间用逗号(,)隔开。

接口的声明格式：

```
interface-modifiers interface interfacename
{
    type method-name1(param-list);
    type method-name2(param-list);
    type method-name3(param-list);
    ...
}
```

interface-modifiers 为接口修饰符，接口仅可使用以下这些修饰符：new、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，表示覆盖了继承而来的同名成员 public、protected、internal、private 这几个修饰符定义了对接口的访问权限。

下面举一个接口的小例子。

例 7.10 使用接口实现一组票的打印问题。

程序代码：

```
namespace ch07_10
{
    class Program
    {
        static void Main(string[] args)
        {
            A  a1=new A();
            a1.print();
            B b1 = new B();
            b1.print();
            C c1 = new C();
            c1.print();
        }
    }
    public interface Iprint
    {
```

```
        void print();
    }

    class A : Iprint
    {
       public  void print()
        {
            Console.WriteLine("这是学生票");
        }
    }
    class B : Iprint
    {
       public  void print()
        {
            Console.WriteLine("这是成人票");
        }
    }
    class C : Iprint
    {
       public  void print()
        {
            Console.WriteLine("这是折扣票");
        }
    }
}
```

运行结果如图 7.12 所示。

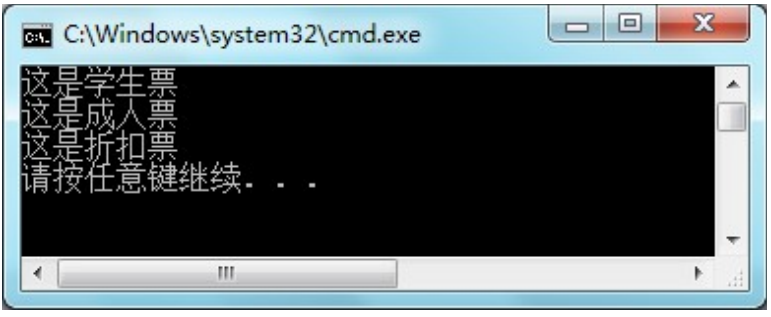

图 7.12 例 7.10 的运行结果

在上面的这个例子中，先是定义了接口 Iprint，而后分别定义了类 A、B、C，这三个类继承于接口 Iprint。可以看到，这三个实现了接口 Iprint 的所有函数成员。这样，我们就可以在 Main 函数中对这三个类进行实例化，最终实现程序的最初设计目的。

7.5.2 通过使用 is 实现查询

用户可以使用 is 关键字检测运行时对象的类型是否与某一给定的类型兼容，使用的语法为：

表达式 is 类型

其中，“表达式”为一种引用类型表达式，而“类型”则为一种引用类型。这个 is 运算符运算之后产生的结果为一个布尔值，因此可以使用条件语句来判断。当表达式不为 Null 且表达式可以被强制转换为引用类型时，is 表达式就返回 true，否则就返回 false。

看看下面的代码：

```
static void Main(string[] args)
        {
            string str="";
            if(str is string)
                Console.WriteLine("true");
            else
                Console.WriteLine("false");
        }
```

程序运行结果如图 7.13 所示。

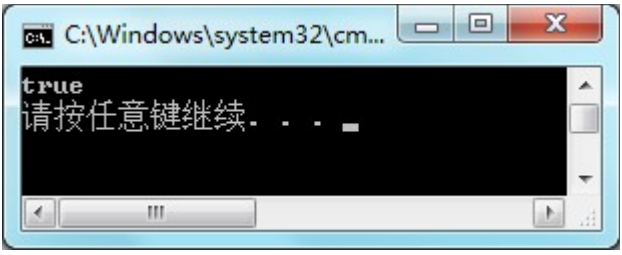

图 7.13 使用 is 运算符例子的运行结果

7.5.3 通过使用 as 实现查询

用户可以使用 as 运算符在兼容类型之间实现转换，其使用语法为：

对象 = 表达式 as 类型

其中，表达式为任何引用类型，可以把 as 运算符看作是 is 运算符的组合，而如果在问题中两个类型是兼容的，就转换类型。as 运算符和 is 运算符之间的主要不同是，如果表达式和类型不兼容，as 运算符设置对象为 Null，代替返回一个值。

看看下面的代码：

```
using System;
using System.Collections.Generic;
```

```
using System.Text;
namespace InterfaceAs
{
    public interface Interface1
    { }
    public interface Interface2
    { }
    public class classTest : Interface1
    { }
    class Program
    {
        static void Main(string[] args)
        {
           classTest ct = new classTest();
            Interface2 testInterface = ct as Interface2;
            if (testInterface!=null)
            {
                Console.WriteLine("the Interface2 Interface is
implemented");
            }
            else
            {
                Console.WriteLine("the Interface2 Interface is
not implemented");
            }
            Console.ReadLine();
        }
    }
}
```

程序运行结果如图 7.14 所示。

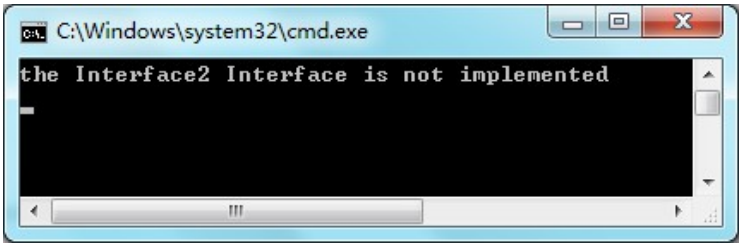

图 7.14 使用 as 运算符例子的运行结果

7.6 代理(delegate)

在程序开发中，回调函数是一个比较重要的机制。在 Windows 操作系统中就大量地使用回调函数，如窗口过程、挂钩函数以及异步过程调用等。同样在 Microsoft .NET Framework 中，也要大量地使用回调函数，例如用户可以注册回调方法来获取程序状态改变的通知和文件系统的改变通知等。

熟悉 C 或者 C++语言的读者，可以对 C#中的代理这样理解：代理很类似于 C/C++中的指针。

在程序运行中，同一个代理能够用来调用不同的方法，只要改变它引用的函数即可。因此，代理调用的函数不是在编译时决定的，而是在运行时决定。这就是代理的主要优点。代理声明的语法为：

delegate-modifiers delegate return-type delegate-name(param-list);

delegate-modifiers 是代理修饰符，delegate-name 是代理名，param-list 是参数列表，return-type 是被代理函数的返回类型。

例 7.11 创建一个简单代理实例。

程序代码：

```
namespace ch07_11
{
    class Program
    {
        static void DelegateMethod(string message)
        {
            Console.WriteLine(message);
        }
        static void Main(string[] args)
        {
            Delegate d1 = new Delegate(DelegateMethod);
            d1("Hello");
        }
    }
    delegate void Delegate(string message);
}
```

程序运行结果如图 7.15 所示。

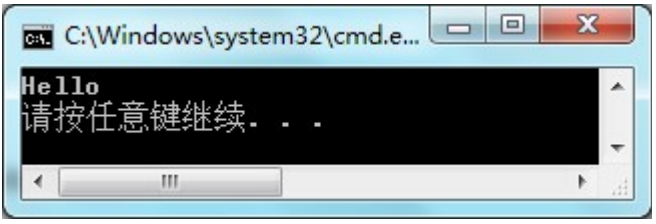

图 7.15 例 7.11 的运行结果

代理三步曲：

- 生成自定义代理类： delegate void Delegate(string message);
- 然后实例化代理类：Delegate d1 = new Delegate(DelegateMethod);
- 最后通过实例对象调用方法：d1("Hello");

7.7 案 例 实 训

1. 案例说明

本例子是关于继承时构造函数设置的一个小例子，类 Vehicle 是一个父类，Car 类是其子类，Train 类有两个构造函数，一个是没有参数的，另外一个有多个参数。通过此例仔细推敲一下构造函数中参数的传递过程。

2. 编程思路

子类与父类之间的继承关系可以参照前面部分讲解的内容。

3. 步骤

【步骤 1】新建一个 Windows 窗体应用程序，取名为 ch07-12.

【步骤 2】拖动三个 Label、四个 TextBox、一个 Button 到窗体上，界面如图 7.16 所示，属性设置如表 7.1 所示。

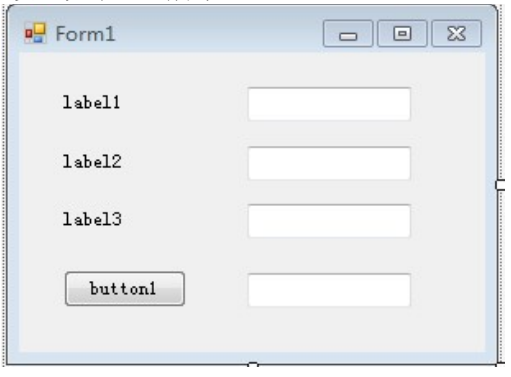

图 7.16 程序界面

表 7.1 控件属性设置

控件类型	控件名称	属性	设置结果

Form	Form1	Text	求总分平均数
Label	Label1	Text	请输入姓名：
	Label2	Text	请输入英语成绩：
	Label3	Text	请输入数学成绩：
TextBox	TextBox1	Name	txtName
	TextBox2	Name	txtEnglish
	TextBox3	Name	txtMath
	TextBox4	Name	txtResult
		ReadOnly	True
Button	Button1	Name	btnCal
		Text	求值

【步骤 3】双击 Button 按钮，打开代码窗口，编写代码。

```
namespace ch07_12
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void btnCal_Click(object sender, EventArgs e)
        {
            int Math = int.Parse(txtMath.Text);
            int Eng = int.Parse(txtEnglish.Text);
            Stu s1 = new Stu(txtName.Text,Eng,Math);
               txtResult.Text = s1.Show() + "\r\n 平均分：" +
s1.Avarage() + "，总分：" + s1.Total();
```

```
        }
    }
    class Student
    {
        protected string Name;
        public Student(string name)
        {
            Name = name;
        }
        public string Show()
        {
            return "姓名：" + Name;
        }
    }
    //创建派生类
    class Stu : Student
    {
        private int Score1, Score2;
        public Stu(string name, int score1, int score2)
            : base(name)
        {
            this.Score1 = score1;
            this.Score2 = score2;
        }
        public int Total()
        {
            return Score1 + Score2;
        }
        public float Avarage()
        {
            return (float)(Score1 + Score2) / 2;
        }
    }
}
```

【步骤 4】调试运行结果如图 7.17 所示。

图 7.17　案例的运行结果

7.8 小　　结

本章主要介绍了面向对象编程思想在 Visual C#中的应用，并依次讲解了类与对象的建立，构造函数、析构函数以及继承、多态、代理等面向对象编程常用的手段。

类的概念以及与对象的关系是基于对象的程序设计的思想基础。C#的面向对象的特点是 C#继承的，并通过用.NET 构架的特点进行构造和加强的。此外，C#的继承与 C++是不同的，C#只支持单一继承，而需求多重继承的时候，必须借助于实现多重接口来达到最终目的。

在 Visual C#中，代理是类型安全、操作可靠的对象，起着与 C++的函数指针一样的作用，用来管理对象。代理与类以及接口不一样，代理是在编译时定义的，它一般用于执行异步处理，并能把用户代码加到一个类的代码路径中去。代理可以用于许多目的，包括使用它们作为 callback(回调)方法，定义静态方法以及使用它们来定义事件。

7.9 习　　题

1. 选择题

(1) 在 Visual C#中，接口与类的主要不同在于_____。

A. 类不可以多重继承而接口可以

B. 类可以继承而接口不可以

C. 类不可以继承而接口可以

D. 类可以多重继承而接口不可以

(2) is 运算符的作用是_____。

A. 检测对象类型

B. 检测运行时对象的类型是否与某一给定的类型兼容

C. 强制类型转换

D. 检测表达式是否正确

(3) as 运算符的作用是_____。

A. 在兼容类型之间进行转换

B. 检测类型兼容性

C. 检测表达式是否正确

D. 检测逻辑运算结果

(4) 在声明接口时，不可为接口成员指定任何修饰符，且不需要为接口成员指定任何代码，故不可为接口成员实例化代码，只需在声明接口的时指定_____即可。

A. 接口成员的名称和参数

B. 成员的名称

C. 成员的类型

D. 成员的参数

2. 填空题

(1) sealed 类的作用是_____。

(2) 构造方法实例化对象的形式是_____。

(3) 代理是一种用来引用静态方法或者 的数据类型，同类接口、字符串和对象一样，代理也是 C#的一种 数据类型。

3. 编程题

(1)计算圆的面积和周长：定义一个圆类，使用类方法实现计算。

(2)编一个程序，定义类 student 和它的成员（学号,姓名,年龄和 c_sharp 程序设计成绩）,使用构造函数实现对数据的输入，使用成员函数实现对数据的输出。

(3) 编一个程序，定义类(有姓名，年龄，手机号码三个字段)，再定义一个一维数组，使数组元素为类，存入数据，然后依次输出,使用 for 循环语句进行输入输出操作。