第3章 C#面向对象程序设计基础

教学目标

- 了解类和对象的定义和使用。
- 掌握类的构造函数和析构函数的定义及特点。
- 熟练掌握继承和多态性的实现方法。
- 熟悉接口的定义和特点。
- 掌握委托的性质和事件的处理方法。
- 了解常用集合类的使用。
- 掌握类库的创建和使用方法。

案例介绍

C#是面向对象语言,在C#中一切都是对象。本章通过一个简单的图形类继承层次结构的实现,介绍C#面向对象程序设计的核心知识点(见案例3-12)。在该案例中,定义了抽象基类Shape和抽象方法,在其所有派生类中实现了抽象方法,通过List集合统一管理Shape类的所有派生类对象,从而实现了多态性。本章综合案例的运行结果如图3.1所示。

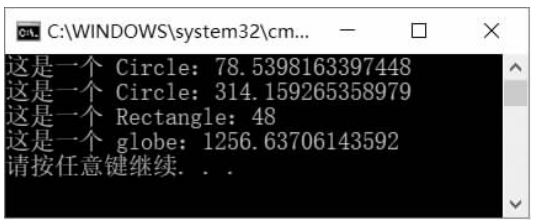

图3.1 本章综合案例运行结果

3.1 类和对象

面向对象编程(Object-Oriented Programming,OOP)是一种以对象为中心的编程方法。在面向对象方法中,对象封装了数据(属性)和操作(行为),对象之间通过发送消息进行通信和交互,合作完成系统的功能。具有共同特征的对象可以抽象为一个类,类在C#中就是一种数据类型,在类中既有数据(即类的字段和事件),又有操作(即类的方法)。前面介绍的每个基本数据类型都是一个类,控制台应用程序中的Main()函数就是类的一个方法。

面向对象方法与人类认识世界的思维方法是一致的,程序中的对象及其交互模拟了现实世界中实际存在的对象及其交互。在现实世界中,对象是一个实际存在的事物(实体),对象可以是有形的,也可以是抽象的概念或规则。复杂的对象可以由若干个简单对象构成。对象常用一组属性和一组行为描述。一组对象之间或多或少地存在一些共同点,可以提取出它们的共同点而忽略其不同点,这样就形成了一个类。类是对一组具有相同属性和行为的对象的抽象。从程序设计的角度看,类是对象的模板,对象是类的实例。类是面向对象思想的核心,类、对象、封装、继承、多态性和事件构成面向对象方法的特性。

3.1.1 类的定义

在C#中,使用class关键字定义类。类定义的格式如下:

```
类访问修饰符 class 类名:基类名
{
    定义类的成员
}
```

其中的类访问修饰符指定了类的可访问性。表3.1列出了类定义中可以使用的类访问修饰符及其含义。

表3.1 类定义中可以使用的类访问修饰符

类访问修饰符	含　义
不加修饰符或internal	只能在当前项目中访问类
public	可以在任何地方访问类
abstract或internal abstract	类只能在当前项目中访问,不能实例化,只能供继承使用
public abstract	类可以在任何地方访问,不能实例化,只能供继承使用
sealed或internal sealed	类只能在当前项目中访问,只能实例化,不能供继承使用
public sealed	类可以在任何地方访问,只能实例化,不能供继承使用

提示:

(1) 默认情况下,类声明为内部的(internal),即只能在当前项目中才能访问它。如果需要在其他项目中访问这个类,则要指定其为公共类(public)。abstract修饰符指定类是抽象的(即不能实例化,只能被继承)。sealed修饰符指定类是密封的(即不能被继承,只能

实例化)。

(2) 如果在定义类时没有显式地指定基类,则该类的基类默认为 System. Object。

1. 类的成员

类中的数据称为类的数据成员,数据成员包含了类的数据——字段和事件。字段是与类相关的变量,用于存储对象的属性。如果出现了某些行为,例如改变类的字段或属性或者进行了某种形式的用户交互操作,此时事件可以让对象通知调用程序,通过调用事件处理代码响应该事件。

数据成员可以是静态数据(与整个类相关)或实例数据(与类的每个实例相关,即每个对象都有自己的数据副本)。静态成员使用 static 关键字进行显式声明。与实例成员不同,静态成员只能通过类访问,而不能通过类的对象访问。

类中的函数称为类的函数成员,提供了操作类中数据的某些功能,包括方法、属性、构造函数和析构函数、运算符重载以及索引器。方法是与某个类相关的函数,可以是实例方法,也可以是静态方法。属性是对类的字段提供特定访问的函数组,其访问方式与访问类的公共字段类似。构造函数是在实例化对象时自动调用的函数,其作用是初始化字段的值。析构函数类似于构造函数,但它在系统删除对象时被调用。运算符重载允许用户在自己的类中对已有的运算符进行重新定义。索引器允许对象以数组或集合的方式进行索引。

类中的所有成员都有访问级别,可以使用下面的访问修饰符。当没有指定访问修饰符时,默认为 private。

- public:成员可以由任何代码访问。
- private:成员只能由类中的代码访问。
- protected:成员只能由类或其派生类中的代码访问。
- internal:成员只能由定义它的程序集(项目)内部的代码访问。
- protected internal:成员只能由项目(确切地讲是程序集)中的类及其派生类的代码访问。
- static:指定静态成员。静态成员只能通过类访问,不能通过类的对象访问。

2. 定义字段

字段的定义方式与普通变量的定义相同,并且可以进行初始化。可以使用关键字 readonly 定义只读字段。只读字段在初始化后不允许修改。只读字段既可以在声明时初始化,也可以在构造函数中初始化,因此,根据所使用的构造函数,不同对象的只读字段可能具有不同的值。

可以使用关键字 const 创建一个常量字段,常量字段必须在声明的同时赋值。常量字段也是静态的,但是不需要加 static 修饰符,所以一个类的所有实例的常量字段的值是相同的。与常量字段不同,只读字段是实例成员,所以不同的实例可以有不同的常量值,只读字段要想成为静态成员必须显式声明。

【案例 3-1】 类的字段和方法的定义。

定义 MyClass 类,在其中定义私有字段、只读字段和常量字段,在构造函数中完成私有字段和只读字段的初始化。

```
using System;
```

```
namespace 案例 3_1
{
    public class MyClass
    {
        private int x=1;                          //私有字段,只能通过公共方法访问
        public readonly int y=20;                 //公共只读私有字段,可以直接读取其值
        public const int MinValue=5;              //常量字段
        public MyClass()                          //无参构造函数
        {    }
        public MyClass(int a, int b)              //带参构造函数,初始化字段 x、y
        {  x=a; y=b;  }
        public void SetX(int value)               //修改字段 x 的值
        {  x=value;  }
        public int GetX()                         //读取私有字段的值
        {  return x;  }
    }
    class Program
    {
        static void Main(string[] args)
        {
            MyClass c1=new MyClass();             //调用无参构造函数
            Console.WriteLine("c1.x={0}   c1.y={1}   c1.MinValue={2}", c1.GetX
            (), c1.y, MyClass.MinValue);
            c1.SetX(15);
            Console.WriteLine("c1.x={0}   c1.y={1}   c1.MinValue={2}", c1.GetX
            (), c1.y, MyClass.MinValue);
            MyClass c2=new MyClass(66, 88);       //调用带参构造函数
            Console.WriteLine("c2.x={0}   c2.y={1}   c2.MinValue={2}", c2.GetX
            (), c2.y, MyClass.MinValue);
            MyClass c3=new MyClass(666, 888);
            Console.WriteLine("c3.x={0}   c3.y={1}   c3.MinValue={2}", c3.GetX
            (), c3.y, MyClass.MinValue);
            //Console.ReadKey();
        }
    }
}
```

案例 3-1 的运行结果见图 3.2。

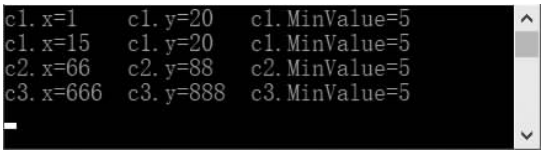

图 3.2　案例 3-1 的运行结果

由运行结果可以看出：

(1) 3 个对象的常量字段值是相同的,而私有字段和只读字段的值可以不同。

(2) 只读字段的值不允许修改,如果在 Main()函数中添加下面的语句,编译器将报告错误:

```
c1.y=50;                          //错误:不能对只读字段赋值(只能在构造函数中或定义时赋值)
```

(3) 常量字段在定义时必须赋值,而且只能在定义时赋值。如果在定义时没有为常量字段赋值,或者在其他地方为常量字段赋值,编译器都将报告错误。例如:

```
public const int MaxValue ;  //错误:常量字段必须提供一个值
MyClass.MinValue=15;         //错误,常量字段不能修改
```

(4) 在 Main()函数中不能直接访问私有字段 x 的值,读取和修改 x 的值必须通过公共方法 GetX()和 SetX()完成。

3. 定义方法

在 C# 中,方法是一种函数成员。类中的方法定义使用 C# 的标准函数定义格式,具体格式如下:

```
[修饰符] 返回类型 方法名([形参表])
{
    方法的实现代码
}
```

修饰符指定方法的访问级别,默认为 private。在 C# 中,每个方法都必须单独声明为 public 或 private。所有的 C# 方法都在类定义中声明和定义。

返回值类型指定方法需要返回的数据的类型,可以是任何已有的类型。当方法没有返回值时,就把返回类型指定为 void。方法名是用户命名的标识符。形参表规定了方法执行时需要调用者提供的信息。

可以在方法定义中使用下述关键字描述方法的特性。

- virtual:虚方法,该方法可以重载。
- abstract:抽象方法,该方法必须在非抽象的派生类中重载(只用于抽象类)。
- override:重载方法,该方法重载了一个基类方法(如果方法被重载,就必须使用该关键字)。

方法调用时需要指定方法名和实参(实际要处理的数据),并且要求实参的数量、类型和顺序要与形参表匹配。

方法定义和调用的示例代码见案例 3-1 和案例 3-2。

【案例 3-2】 方法中的不同参数的使用。

定义 3 个方法,分别使用值参数、引用参数和输出参数完成相应功能。

```
using System;
namespace 案例 3_2
{
    class Program
    {
        static void Fun1(int[] a, int i)              //值参数
```

```
        {
            i=10;
            a[0]=10;
        }
        static void Fun2(int[] a, ref int i)              //引用参数
        {
            i=10;
            a[0]=10;
        }
        static void Fun3(int[] a, out int c1, out int c2) //输出参数
        {
            c1=c2=0;                                  //必须初始化,否则编译时会报告错误
            foreach (int x in a)
            {
                if (x %2==0)
                    c1++;
                else
                    c2++;
            }
        }
        static void Main(string[] args)
        {
            Console.WriteLine("值参数:");
            int i=0;
            int[] a={1, 3, 5, 8};
            Console.WriteLine("i={0},\ta[0]={1}", i, a[0]);
            Fun1(a, i);                               //值参数方法调用
            Console.WriteLine("i={0},\ta[0]={1}", i, a[0]);
            Console.WriteLine("\n引用参数:");
            Fun2(a, ref i);                           //引用参数方法调用
            Console.WriteLine("i={0},\ta[0]={1}", i, a[0]);
            Fun1(a, i);
            Console.WriteLine("i={0},\ta[0]={1}", i, a[0]);
            Console.WriteLine("\n输出参数:");
            int[] intArray=new int[] { 1, 2, 3, 4, 5, 7 };
            int counter1;                             //保存奇数的个数,可以不用初始化
            int counter2;                             //保存偶数的个数,可以不用初始化
            Fun3(intArray, out counter1, out counter2);     //输出参数方法调用
            Console.WriteLine("偶数{0}个\n奇数{1}个", counter1, counter2);
        }
    }
}
```

在C#中,方法的参数有4种不同形式:值参数、引用参数、输出参数和参数数组。

1)值参数

在C#中,如果变量是通过值传送给方法的,被调用的方法得到的是变量的一个副本。

也就是说，当方法被调用时，编译器为值参数分配存储单元，然后将对应的实参的值复制到形参中；在方法退出后，对变量进行的修改会丢失。值参数对应的实参可以是变量、常量和表达式，但要求其值的类型必须与形参的类型相同或者兼容。

在 C# 中，如果没有特别说明，所有的参数都是通过值传递的。

2）引用参数

使用 ref 关键字定义的参数为引用参数。引用参数与方法调用中的实参变量共用一个存储单元，因此，在方法内对引用参数的任何修改都会影响对应的实参变量。

使用引用参数时应注意以下几点：

- ref 关键字仅对跟在它后面的一个参数有效，即每个引用参数都需要明确指定。
- 在调用方法时，也需要用 ref 修饰实参，而且实参必须是变量，不能是常量或表达式。
- 引用参数对应的实参在调用之前必须已经初始化，C# 不允许假定引用参数在使用它的函数中初始化（即把未赋值的变量用作引用参数是非法的）。

引用参数的示例代码见案例 3-2 中的 Fun2 方法，即在参数 i 前加上 ref。由运行结果可以看出，Fun2 方法调用后，变量 i 的值被修改了。

下面的函数实现交换两个整型变量的值，此时必须使用引用参数返回交换后的结果。

```
static void Swap(ref int x, ref int y)
{
    int temp=x;
    x=y;
    y=temp;
}
```

3）输出参数

使用 out 关键字指定输出参数。输出参数用于从方法返回结果。当在方法的参数前面加上 out 关键字时，传递给该方法的变量可以不用初始化。该变量通过引用传送，所以从被调用的方法返回时，方法对该变量进行的任何改变都会保留下来。输出参数与引用参数一样通过引用传递。两者的区别是：输出参数只能用于从方法中传出值，而不能接收实参数据，所以可以把未赋值的变量用作输出参数。在方法内输出参数被认为是未赋值的（即调用代码可以把已赋值的变量用作输出参数，存储在该变量中的值会在函数执行时丢失），所以在方法中应该对输出参数赋值。在调用该方法时，也需要使用 out 关键字，与在定义该方法时一样。

案例 3-2 中的函数 Fun3 统计一维整型数组中奇数的个数和偶数的个数，使用两个输出参数返回统计结果，c1 返回偶数的个数，c2 返回奇数的个数。

out 关键字是 C# 中的新增内容，该关键字的引入使 C# 程序更安全，更不容易出错。如果在函数体中没有给输出参数分配一个值，该方法就不能编译。例如，在 Fun3 函数中，如果去掉语句 c1＝c2＝0；，则编译时就会发生错误。

4）参数数组

一般而言，调用方法时实参必须与该方法声明的形参在类型和数量上相匹配。但有时开发者希望更灵活一些，能够给方法传递任意个数的参数。C# 提供了传递可变长度的参

数表的机制，即使用 params 关键字指定一个可变长度的参数表，称之为参数数组。

【案例 3-3】 使用参数数组查找若干个成绩中的最大值。

```
using System;
namespace 案例 3_3
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] score={86, 57, 92, 100, 78, 69, 48, 84, 88};
            int MaxScore;
            MaxValue(out MaxScore);                    //可变参数的个数可以是零个
            Console.WriteLine("MaxScore={0} ", MaxScore);
            MaxValue(out MaxScore, 46, 67, 98, 96); //在 4 个数中找最大值
            Console.WriteLine("MaxScore={0} ", MaxScore);
            MaxValue(out MaxScore, score);          //可变参数也可接收数组对象
            Console.WriteLine("MaxScore={0} ", MaxScore);
            Console.ReadKey();
        }
        static void MaxValue(out int max, params int[] a)
        {
            if (0==a.Length)                  //如果可变参数为零个，取一个事先设定的值
            {
                max=-1;
                return;
            }
            max=a[0];
            for (int i=1; i <a.Length; i++)
            {
                if (a[i] >max) max=a[i];
            }
        }
    }
}
```

4. 定义属性

属性是对类的字段提供特定访问方法的类成员，它是一个方法或一对方法，但是对使用类的客户看来，它是一个字段，所以属性可以使用简单的方法访问私有字段。因为属性在类中是按一种与方法类似的方式执行的，所以它不会危害类中受保护的和隐藏的私有数据。

属性的定义方式与字段类似，但属性拥有两个类似于函数的访问器：一个用于获取属性的值，称为 get 访问器；另一个用于设置属性的值，称为 set 访问器。同时包含 get 和 set 访问器的属性是读写属性，只包含 get 访问器的属性是只读属性，只包含 set 访问器的属性是只写属性。

get 访问器不带参数，但必须有一个属性类型的返回值，简单的属性一般与私有字段相关联，以控制对这个字段的访问，此时 get 访问器可以直接返回该字段的值。set 访问器可以把一个值赋给字段，但是 set 访问器没有任何显式参数，编译器假定它带一个类型与属性相同的参数，该参数使用关键字 value 表示用户提供的属性值。

属性可以像方法一样使用 virtual、abstract 和 override 关键字，而且访问器可以有自己的访问权限。C# 允许给属性的 get 和 set 访问器设置不同的访问修饰符，所以属性可以有公共的 get 访问器和私有的或受保护的 set 访问器。这有助于控制属性的设置方式。在 get 和 set 访问器中，必须有一个具备属性的访问级别，但是访问器的访问权限不能高于它所属的属性。

【案例 3-4】 属性的定义和访问。

定义一个类 Circle，通过公共属性 Radius 访问私有字段 radius。

```
using System;
namespace 案例 3_4
{
    public class Circle
    {
        double radius;                                  //半径
        public double GetRadius()
        {
            return radius;
        }
        public double Radius                            //公共属性，访问私有字段 radius
        {
            get  {  return radius;  }
            set  {  radius=value;  }
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Circle c1=new Circle();
            Console.WriteLine("初始时圆的半径为{0}", c1.GetRadius());
            c1.Radius=30;
            Console.WriteLine("修改后圆的半径为{0}", c1.GetRadius());
            Console.WriteLine("修改后圆的半径为{0}", c1.Radius);
        }
    }
}
```

5. 静态成员和实例成员

静态成员（又称共享成员）可以在类的实例之间共享，所以可以将它看作类中的全局对象。静态属性和静态字段可以访问独立于任何对象实例的数据，静态方法可以执行与对象

类型相关但与对象实例无关的命令。在使用静态成员时,不需要实例化对象。

使用 static 修饰符定义的成员称为静态成员。静态方法只能对类中的静态成员进行操作,不可以直接访问实例字段。实例方法可以直接访问静态字段和实例字段。

经常使用的 Console. WriteLine()和 Convert. ToString()方法就是静态的,因此不需要实例化 Console 或 Convert 类对象。

许多情况下,静态属性和方法有很好的效果。例如,可以使用静态属性跟踪给类创建了多少个实例。静态成员的使用示例见案例 3-5。

3.1.2 构造函数和析构函数

1. 构造函数

当定义了一个类之后,就可以通过 new 实例化对象。为了能安全地使用这个对象,C#提供了对对象进行初始化的方法,这就是构造函数。在 C#中,用 new 关键字调用构造函数。

构造函数名必须与类名相同,没有返回类型,并且没有任何返回值。构造函数可以没有参数,也可以有一个或多个参数。如果类中没有提供构造函数,编译器会自动产生一个默认的构造函数,这是一个非常基本的构造函数,它只能把所有的字段成员初始化为标准的默认值。例如,将引用类型的字段成员初始化为空引用,将值类型的字段成员初始化为 0,将 bool 类型的字段成员初始化为 false。这通常就足够了,否则就需要开发者编写构造函数。

构造函数可以重载,即可以为一个类定义多个构造函数,只要它们的参数个数或类型不同即可。

如果类仅用作某些静态成员或属性的容器,永远不会实例化,就可以把构造函数定义为 private 或 protected,这样不相关的类就不能访问它们。

构造函数也分为实例构造函数和静态构造函数。实例构造函数负责对非静态数据成员进行初始化,静态构造函数负责对静态数据成员进行初始化。实例构造函数在创建对象时自动调用,而静态构造函数只能在创建类的实例或者引用类的任何静态成员时执行。无论创建了多少个类实例,其静态构造函数都只调用一次。在 C#中,静态构造函数通常在第一次调用类的成员之前执行。

一个类只能有一个静态构造函数。静态构造函数不能有访问修饰符,也不能带任何参数。静态构造函数是不可继承的。如果类中包含静态成员,而没有声明静态构造函数,那么编译器会自动产生一个默认的静态构造函数。

构造函数的定义和调用示例见案例 3-5。

2. 析构函数

析构函数负责清理对象。一般情况下,不需要提供析构函数的代码,而是由默认的析构函数自动执行操作。但是如果对象删除时需要执行一些重要的操作,例如释放内存,就应提供特定的析构函数。析构函数名必须与类名相同,前面需要加~以表明它是析构函数。析构函数没有返回类型,也不能带参数,当然也不能被继承,所以一个类最多只能有一个析构函数。一个类如果没有显式地声明析构函数,则编译器将自动产生一个默认的析构函数。

析构函数不能由程序显式地调用，而是由系统在释放对象时自动调用。如果这个对象是一个派生类对象，那么在调用析构函数时将首先执行派生类的析构函数，然后执行基类的析构函数。如果这个基类还有自己的基类，这个过程就会不断重复，直到调用 Object 类的析构函数为止，其执行顺序正好与构造函数相反。

析构函数的定义和调用示例见案例 3-5。

3.1.3 对象的使用

定义了一个类后，就可以在项目中使用该类的对象，并通过 new 运算符将其实例化。

【案例 3-5】 类的定义和对象的使用。

本案例演示类的定义和对象的使用方法，以及构造函数和析构函数的定义和调用，并用静态字段跟踪类创建了多少个实例。

```
using System;
namespace 案例 3_5
{
    public class Point
    {
        private int x, y;                          //定义私有字段
        private static int NumofObject;            //定义静态字段
        public Point()                             //默认构造函数
        {
            x=y=0;
            NumofObject++;
            Console.WriteLine("默认构造函数被调用,点({0},{1})生成", x, y);
        }
        public Point(int xx, int yy)               //非默认构造函数
        {
            x=xx>0?xx:0;
            y=yy>0?yy:0;
            NumofObject++;
            Console.WriteLine("非默认构造函数被调用,点({0},{1})生成", x, y);
        }
        static Point()                             //静态构造函数
        {
            NumofObject=0;
            Console.WriteLine("静态构造函数被调用");
        }
        ~Point()                                   //析构函数
        {
            NumofObject--;
            Console.WriteLine("析构函数被调用,(点{0},{1})被删除", x, y);
            Console.WriteLine("ObjectNumber={0}\n", GetObjectNum());
        }
        public static int GetObjectNum()
```

```
        {
            return NumofObject;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("ObjectNumber={0}\n", Point.GetObjectNum());
            Point p1=new Point();
            Console.WriteLine("ObjectNumber={0}\n", Point.GetObjectNum());
            Point p2=new Point(3, 4);
            Console.WriteLine("ObjectNumber={0}\n", Point.GetObjectNum());
            Point p3=new Point(10, 20);
            Console.WriteLine("ObjectNumber={0}\n", Point.GetObjectNum());
        }
    }
}
```

3.1.4 类的继承与多态性

1. 继承

继承是面向对象程序设计中最重要的特性之一。继承表达的是类与类之间的关系，任何类都可以从另一个类中继承，被继承的类称为基类(或父类)，新定义的类称为派生类(或子类)。C＃中仅支持单一继承，即一个类仅能直接派生于一个基类。

继承具有重要的实际意义，它简化了人们对事物的认识和描述，只需要发现和描述派生类所独有的特征即可。派生类对象的特征由两部分组成：一是从基类继承的特征，二是派生类所独有的特征。在软件开发过程中，不需要把基类已经定义过的属性和操作重复地书写一遍，这就大大降低了软件开发工作的强度。除此之外，继承的作用还表现为通过增强一致性减少模块间的接口和界面。

在面向对象程序设计中，有两种不同的继承类型，即实现继承和接口继承。

实现继承是指一个类派生于一个基类，拥有该基类的所有成员字段和函数。在实现继承中，派生类的每个函数可以采用基类的实现代码，也可以根据需要重载该函数的实现代码。在需要给现有的类添加功能或许多相关的类共享一组重要的公共功能时，这种继承类型是非常有效的。

接口继承是指一个类只继承了函数的签名(接口)，没有继承任何实现代码。在需要指定该类具有某些可用的特性时，最好使用这种继承类型。接口继承常被看作提供了一种契约：让类派生于接口，以保证为客户提供某个功能。

在C＃中，既有实现继承，也有接口继承，这样便于为不同的情形选择最好的体系结构。

1) 派生类的定义

派生类的定义格式如下，其中的基类可以是任何已有的类。

```
public class 派生类名 : 基类名
{
    派生类的新成员
}
```

通过继承，派生类能够拥有基类的所有成员，同时还可以定义自己的新成员。派生类不能访问基类的私有成员，但是可以访问其公有成员和保护成员(protected)。另外，C# 不支持私有继承，因此在基类名前没有 public 或 private 修饰符。

派生类的定义示例见案例 3-6。

2）派生类的构造函数

在创建派生类的对象时，不仅要初始化在派生类中新定义的成员，还必须初始化从基类继承的成员，这将通过派生类的构造函数隐式或显式调用基类的构造函数完成。如果基类也是派生于其他类，构造函数的调用将如何执行呢？在 C# 中，构造函数的执行顺序是这样的：根据类的层次结构找到最顶级的基类，先调用基类的构造函数，再依次调用各级派生类的构造函数。析构函数的调用顺序正好相反。

由于在 C# 中，所有的类都是从 System. Object 类派生的，所以总是最先调用 System. Object 类的构造函数。

如果在派生类的构造函数中没有明确指定，总是调用基类的默认构造函数对基类成员进行初始化。可以使用 base 关键字指定使用基类中某个非默认构造函数完成基类成员的初始化。案例 3-6 演示了派生类的构造函数和析构函数的定义和调用顺序。

2. 多态性

多态性是指不同的对象收到相同的消息时产生多种不同的行为方式。换句话说，对象的多态性是指在基类中定义的属性和操作被派生类继承后可以具有不同的数据类型或表现出不同的行为，使得同一个属性或操作在基类及其各个派生类中具有不同的意义。多态性增加了程序的灵活性。

由于在基类和派生类中会存在相同的方法和属性，可以把某个派生类的对象赋给基类的对象，然后通过基类的对象调用基类的方法，结果是执行派生类中的实现代码。

在派生于同一个类的不同对象上执行任务时，多态性是一种极为有效的技巧，其使用的代码最少。

在 C# 中，通过虚方法实现多态性。在类的方法前加上关键字 virtual，则该方法就成为虚方法，它就可以在派生类中重载。C# 要求在派生类中重载另一个方法时要使用 override 关键字显式声明。通过在派生类中对虚方法进行重载，就可以实现多态性。

注意，静态函数成员不能声明为 virtual，因为这个概念只对类中的实例函数成员有意义。

【案例 3-6】 派生类的定义和使用。

本案例演示派生类的定义和对象的使用、派生类的构造函数和析构函数的定义和调用顺序以及多态性的实现。

```
using System;
namespace 案例 3_6
```

```
{
    public class Point                                          //定义基类
    {
        private int x, y;                                       //定义私有字段
        protected Point()                                       //保护的默认构造函数
        {
            x=y=0;
            Console.WriteLine("Point 类的默认构造函数被调用,点({0},{1})", x, y);
        }
        public Point(int xx, int yy)                            //公有的非默认构造函数
        {
            x=xx>0?xx:0;
            y=yy>0?yy:0;
            Console.WriteLine("Point 类的非默认构造函数被调用,点({0},{1})",x,y);
        }
        ~Point()                                                //析构函数
        {
            Console.WriteLine("Point 类的析构函数被调用,点({0},{1})",x,y);
        }
        public virtual double Area()                            //虚方法
        {
            return 0;
        }
    }
    public class Circle : Point                                 //定义派生类
    {
        private int radius;
        const double PI=3.14159;
        public int GetRadius()
        {
            return radius;
        }
        public void SetRadius(int r)
        {
            radius=r >0 ?r : 10;
        }
        public Circle()                                         //调用基类的默认构造函数
        {
            radius=10;
            Console.WriteLine("Circle 类的默认构造函数被调用, radius={0}", radius);
        }
        public Circle(int xx, int yy, int r) : base(xx, yy) //调用基类的非默认构造函数
        {
            SetRadius(r);
            Console.WriteLine("Circle 类的非默认构造函数被调用, radius={0}",radius);
```

```
        }
        ~Circle()
        {
            Console.WriteLine("Circle类的析构函数被调用, radius={0}",radius);
        }
        public override double Area()                    //虚方法
        {
            return PI * radius * radius;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Circle c1=new Circle();
            Circle c2=new Circle(10, 20, 100);
            Console.WriteLine("c1.radius={0}", c1.GetRadius());
            Console.WriteLine("c2.radius={0}", c2.GetRadius());
            Console.WriteLine("c1.Area={0}", c1.Area());
            Console.WriteLine("c2.Area={0}", c2.Area());
            c1.SetRadius(20);
            c2.SetRadius(200);
            Console.WriteLine("c1.radius={0}", c1.GetRadius());
            Console.WriteLine("c2.radius={0}", c2.GetRadius());
        }
    }
}
```

3. 抽象类与抽象方法

C#允许把类和函数声明为抽象的,抽象类不能实例化,而抽象函数没有执行代码,必须在非抽象的派生类中重载。显然,抽象函数也是虚函数,但是不需要使用 virtual 关键字,如果使用了该关键字,就会产生一个语法错误。如果一个类中包含抽象函数,则该类也是抽象的,也必须声明为抽象类。

抽象类是一种特殊的基类,并不与具体的事物联系。例如,并没有“图形”这样的具体事物,所以可以将“图形”定义为抽象类,派生出“圆形”和“四边形”这样一些可以产生具体实例的普通类。需要注意的是,抽象类不能被实例化,它只能作为其他类的基类使用。抽象类的定义使用关键字 abstract。下面的代码将 Shape 类定义为抽象类:

```
public abstract class Shape
{
    …
}
```

在抽象类中使用关键字 abstract 定义抽象方法,要求所有派生的非抽象类中都要使用 override 关键字重载该抽象方法。引入抽象方法的原因在于抽象类本身是一种抽象的概

念，有的方法并没有具体的实现，而是留下来让派生类来重载实现。Shape 类中 GetArea() 方法本身没有具体的意义，而只有到了派生类 Circle 和 Rectangle 中才可以计算具体的面积。抽象方法 GetArea()的定义如下：

```
public abstract double GetArea();
```

则派生类的 GetArea()方法的重载实现为

```
public override double GetArea()
{
    …
}
```

3.2 接口

接口是把非静态的公共实例方法和属性结合起来，以封装特定功能的一个集合。一旦定义了接口，就可以在类中实现它。这样，类就可以支持接口所指定的所有成员。一般情况下，接口中只能包含方法、属性、索引器和事件的声明。

注意，接口不能单独存在，不能实例化接口。另外，接口不能包含实现其成员的任何代码，而只能定义成员本身，实现过程必须在实现接口的类中完成。接口不能有构造函数或字段。接口定义也不允许包含运算符重载。在接口定义中不允许使用声明成员的修饰符，接口成员总是公共的，不能声明为虚的或静态的。

3.2.1 接口的定义

接口的定义方式与类的定义方式相似，但使用的关键字是 interface，语法格式如下：

```
interface 接口名
{
    接口的成员
}
```

与类一样，接口默认也是内部的，必须使用 public 关键字才能使其成为公共的。不能在接口中使用关键字 abstract 和 sealed，因为这两个修饰符在接口中没有意义。

接口的名称一般用大写字母 I 开头。接口的成员可以是方法、属性、索引器和事件。接口的成员默认是公共的，因此不允许成员加上修饰符。

接口成员的定义与类成员的定义相似，但有几个重要的区别：

(1) 接口成员不允许使用访问修饰符(public、private、protected 或 internal)，所有接口成员都是公共的。

(2) 接口成员不能包含代码实体。

(3) 接口不能定义字段成员。

(4) 接口成员不能用关键字 static、virtual、abstract 或 sealed 定义。

(5) 接口可以定义为类的成员，但不能定义为其他接口的成员。

在接口中定义的属性也可以定义 get 和(或)set 访问器。例如：

```
interface IMyInterface
{
    int MyInt { get; set; }
}
```

其中 int 属性 MyInt 有 get 和 set 访问器。也可以只定义其中一个访问器。

接口之间也有继承关系，其语法与类的继承相似，主要区别是可以使用多个基接口。例如：

```
publc interface ID: IA, IB
{
    …
}
```

类可以同时有一个基类和零个以上的接口，并要将基类写在前面。例如：

```
class ClassB: ClassA, IA, IB
{
    …
}
```

接口不是类，所以没有继承 System. Object。但是为了方便起见，System. Object 的成员可以通过接口类型的变量访问。

3.2.2 接口的实现

实现接口的类必须包含该接口所有成员的实现代码，且必须匹配指定的签名（包括匹配指定的 get 和 set 块），并且必须是公共的。可以使用关键字 virtual 或 abstract 实现接口成员，但不能使用 static 或 const。还可以在基类上实现接口成员。继承一个实现给定接口的基类，就意味着派生类隐式地支持这个接口。

尽管不能像实例化对象那样实例化接口，但是可以建立接口类型的变量，然后就可以在支持该接口的对象上使用这个变量访问该接口提供的方法和属性。

【案例 3-7】 接口的定义和实现。

本案例定义了两个接口和一个类。接口 IA 定义了方法 SetValueA()，接口 IB 定义了方法 SetValueB()。接口 IB 继承接口 IA。Test 类实现接口 IB，因此在 Test 类中既要实现 IB 的 SetValueB()方法，也要实现 IA 的 SetValueA()方法。

```
using System;
namespace 案例 3_7
{
    public interface IA                          //定义接口 IA
    {
        void SetValueA(int x);
    }
    public interface IB : IA                     //定义接口 IB 继承 IA
    {
```

```
        void SetValueB(int x);
    }
    public class Test : IB                          //定义类 Test,继承接口 IB
    {
        private int a;
        private int b;
        public void SetValueA( int x )              //实现接口 IA 的方法 SetValueA()
        {
            a=x;
            Console.WriteLine("SetValueA: a={0}", a);
        }
        public void SetValueB( int x )              //实现接口 IB 的方法 SetValueB()
        {
            b=x;
            Console.WriteLine("SetValueB: b={0}", b);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Test t=new Test();
            t.SetValueA(10);                        //通过对象访问接口成员
            IB ib=t;
            ib.SetValueB(200);                      //通过接口变量访问接口成员
        }
    }
}
```

3.3 委托与事件

3.3.1 委托

当需要把方法作为参数传递给其他方法时,就要使用委托(delegate)。委托可以创建“指向”某个方法的变量,只需通过该变量,就可以在需要的时候调用委托所指向的方法。使用委托可以编写适用于多种场合的高度灵活的代码。委托最重要的用途是处理事件,它是事件的基础。

C# 中的委托类似于 C/C++ 中的函数指针,但是两者是有区别的。函数指针通过指针获取一个函数的入口地址,实现对函数的操作;而委托是面向对象的,在进行面向对象编程时,方法很少是孤立存在的,在调用前,通常需要与类的实例相关联。所以,在. NET Framework 中,如果要传递方法,就必须把方法的细节封装在一种新类型的对象中,即委托。委托只是一种特殊的对象类型,其特殊之处在于,前面定义的所有对象都包含数据,而委托包含的只是方法的地址。

1. 定义委托

对委托的使用要先定义，然后实例化，最后再调用。委托的定义非常类似于 C++ 中的函数原型，但是委托只指定一个返回类型和一个参数列表，不带函数体。委托使用关键字 delegate 定义，例如：

```
delegate int MyDelegate(int nID, string sName);
```

在这个示例中，定义了一个委托 MyDelegate，并指定该委托的每个实例都包含一个方法的细节，该方法带有一个 int 类型的参数，一个 string 类型的参数，并返回 int 类型的值。由于委托的类型安全性要求非常高，所以在定义委托时，必须给出它所代表的方法签名和返回类型等全部细节。

委托实现为派生自基类 System. MulticastDelegate 的类，System. MulticastDelegate 又派生自基类 System. Delegate。所以定义委托基本上是定义一个新类，可以在定义类的任何地方定义委托，既可以在另一个类的内部定义，也可以在任何类的外部定义，还可以在命名空间中把委托定义为顶层对象。根据定义的可见性，可以在委托定义中添加一般的访问修饰符（如 public、private、protected 等）。

2. 使用委托

在定义了委托后，就可以声明该委托类型的变量，然后用与委托有相同返回值类型和参数列表的方法名初始化该委托变量，之后就可以使用委托变量调用这个方法，就像该变量是一个函数。也可以把委托变量作为参数传递给一个函数，这样，该函数就可以使用委托调用它引用的任何方法，而且在运行之前无须知道调用的是哪个方法。

以下是实例化委托的例子：

```
MyDelegate d1=new MyDelegate(wr.InstanceMethod);
```

在 C# 中，委托在语法上总是带有一个参数的构造函数，这个参数就是委托引用的方法。其中，方法 InstanceMethod()定义如下：

```
public int InstanceMethod(int nID, string sName)
```

委托的调用方法如下：

```
d1(5, "aaa");
```

通过委托变量 d1 实现对方法 InstanceMethod()的调用。调用还必须有一个前提条件：方法 InstanceMethod()的参数和委托 MyDelegate 的参数一致，并且返回值为 int 类型。

实例化委托的方法可以是任何类型的实例方法或静态方法，只要方法的签名与委托的签名匹配即可。使用委托一次可以调用一个方法，也可以调用多个方法（称为多播），通过＋和－运算符实现多播时调用方法的增加或减少。也可以使用委托数组完成对多个方法的调用。

【案例 3-8】 委托的定义和使用。

本案例演示了通过委托调用实例方法和静态方法以及多播的实现。

```
using System;
```

```
namespace 案例 3_8
{
    public class Test
    {
        public void MethodOne(int x)                    //委托引用的非静态方法
        {
            Console.WriteLine("调用方法 MethodOne,返回值={0}", x);
        }
        static public void MethodTwo(int x)             //委托引用的静态方法
        {
            Console.WriteLine("调用方法 MethodTwo,返回值={0}", x+x);
        }
    }
    public delegate void MyDelegate(int n);             //定义委托,参数与上面两个方法相同
    class Program
    {
        static void Main(string[] args)
        {
            Test t=new Test();
            MyDelegate d1=new MyDelegate(t.MethodOne);          //调用实例方法
            d1(100);
            MyDelegate d2=Test.MethodTwo;                       //调用静态方法
            d2(200);
            Console.WriteLine("使用委托数组…");
            MyDelegate[] delegateArray={ t.MethodOne, Test.MethodTwo, Test.MethodTwo};
            for (int i=0; i <delegateArray.Length; i++)
            {
                delegateArray[i](i);
            }
            Console.WriteLine("多播…");
            MyDelegate d3=d1+d2;
            d3(10);
            int MethodNum=d3.GetInvocationList().Length;        //委托中的方法个数
            Console.WriteLine("委托 d3 中的方法个数: {0}", MethodNum);
            d3=d3-d1;
            d3(20);
        }
    }
}
```

3.3.2 事件

基于 Windows 的应用程序也是基于消息的，Windows 使用预定义的消息与应用程序进行通信。消息是包含各种信息的结构，应用程序和 Windows 使用这些信息决定下一步的操作。.NET 把这些消息封装在事件中，如果需要响应某个消息，就应处理对应的事件。