# 第3章 C#面向对象编程

面向对象编程技术中最重要的概念就是类和对象。类（class）表示对现实生活中一类具有共同特征的事物的抽象，是面向对象编程的基础。简单地说，类是一种抽象的数据类型，是对一类对象的统一描述。类是对某个对象的定义，它包含有关对象动作方式的信息，包括名称、方法、属性和事件。类本身不是对象，因为它不存在于内存中。

面向对象的程序设计具有 3 个特征：封装、继承和多态。可以大大地增加程序的可靠性、代码的可重用性和程序的维护性。

封装：将抽象得到的数据和行为（或功能）形成一个有机的整体，即将数据与操作数据的源代码进行有机地结合，形成“类”，其中数据和函数都是类的成员。

继承：通过继承可以创建父类和子类（派生类）之间的层次关系。子类可以使用现有类的所有功能（实现“代码重用”），并对现有的类的功能进行扩展。通过继承创建的新类称为“子类”或“派生类”。 被继承的类称为“基类”“父类”或“超类”。派生的新类既有基类的特点，又有其本身的新特征。

多态是指不同的类进行同一操作可以有不同的方法。实现多态，有两种方式：覆盖和重载。

覆盖：是指子类重新定义父类的虚函数的做法。

重载：是指允许存在多个同名函数，而这些函数的参数表不同（或许参数个数不同，或许参数类型不同，或许两者都不同）。

## 3.1 类与对象

C#的类是一种对包括数据成员、函数成员和嵌套类型进行封装的数据结构。其中数据成员可以是常量。函数成员可以是方法、属性、索引器、事件、操作符、实例构建器、静态构建器、析构器。

类是面向对象的程序设计的核心，实际上是一种新的数据类型。

### 3.1.1 类的结构与定义

类的定义一般分为声明部分和实现部分。声明部分用于声明该类的成员，包括数据成

员、成员函数的声明。成员函数用于对数据进行操作，又称之为“方法”。实现部分用于成员函数的定义。

类声明的一般格式如下：

```
[类访问修饰符] class 类名称：[基类名称]
{
    类体
}
```

【说明】 被方括号括起来的选项为可选项。

（1）类访问修饰符，可选项，用于定义类及其成员的可访问性，包括以下内容。

① public：公共的，表示可以被任意访问，访问不受限制。

② private：只可以被本类所访问。

③ protected：表示受保护的，只能被本类和其继承的子类访问。

④ internal：只限于此程序（类所在的程序内，即同一个编译单元：.DLL 或.EXE）访问。

⑤ abstract：抽象类，不允许建立类的实例。

⑥ sealed：密封类，不允许该类被继承。

声明每一个类时，如果省略了访问修饰符，则默认的访问权限为 internal。

（2）关键字 class，表示类的定义，只能是小写字母。

（3）类名称，是 C#中的一个合法的标识符。由用户指定，只要满足命名规则即可；基类名称，可选项，用于声明要继承的类或接口。

（4）类体，用大括号（{}）括起来，称之为类的主体。其中，一般包含两类成员：数据成员和成员函数。数据成员的数据类型可以是任意合法的数据类型，也可以是另外一个类的对象。例如，例 2.47 中的类 Student 的定义。

## 3.1.2　对象的定义

类是对同类对象的一种抽象，对象是类的实例。一个类定义之后，就可以定义该类的对象。对象是具有数据、行为、标识的编程结构，是面向对象应用程序的一个组成部分。这个组成部分封装了部分应用程序。该应用程序可以是一个过程、数据或一些抽象的实体。

### 1．对象的定义

类的对象声明与创建格式如下：

```
类名 对象名;                              //类的声明未实例化
类名 对象名 = new 类名称([参数表]);       //类的声明并实例化对象
```

### 2．对象的使用（成员的引用）

类的对象，通过运算符“.”引用类的成员。格式如下：

```
类名.成员名称
```

当然类的成员能否允许访问要受成员的访问修饰符的控制。在使用“对象名.成员名”来访问对象成员时，一定要确认该对象引用不能为空（null），否则会引起异常。例如：

```
Student st;          //参见例 2.47 中的类 Student 的定义,声明了类的对象,未实例化
```

```
st.sName = "赵颖";//将引发错误,因为 st 对象为 null。st 未实例化,即未指向任何存储空间
```

类和对象既有区别也有联系。类是一个相对抽象的概念，对象是一个具体的概念。类为对象的生成提供模板，利用 new 生成类的一个对象。

## 3.2 类的成员

定义在类体中的元素都是类的成员。类的成员分为数据成员（描述状态）和函数成员（描述操作）。类的成员要么是静态（static）成员，要么是实例（instance）成员。静态成员可以通过类名称直接引用，实例成员要通过类的实例（对象）引用。类中一些常用的成员如下。

（1）常量：与类关联的常量值，包括常成员函数和常数据成员。

（2）字段：类中定义的变量。

（3）方法：类可执行的计算或操作，主要是函数（方法）。

（4）属性：定义一些特性以及与读写这些特性相关的操作。

（5）事件：可由类生成的通知。

（6）索引器：能以数组方式索引类的实例的操作。

（7）运算符：类所支持的运算符。

（8）委托：本质也是类，可以引用一个或多个方法。

（9）构造函数：特殊的成员函数，初始化类的实例或类本身所需要的操作，名称与类名相同。

（10）析构函数：特殊的成员函数，在撤销类实例之前执行的操作。

（11）嵌套类型：在类中声明的类型。

### 3.2.1 数据成员与函数成员

**1．数据成员**

数据成员用于描述类的状态，包括字段（类中定义的变量）、常量（其值不可改变）、事件。数据成员可以是静态数据（与整个类相关）或实例数据（类的每个实例都有它自己的数据副本）。

（1）字段：字段是类中最常见的数据成员，用于表示在类中定义的与类或对象相关的变量。

字段的声明格式如下：

**[访问修饰符] 数据类型 字段名 [=初始值]**

若省略了访问修饰符，则类的成员的默认访问权限为 private；若声明字段时，省略了“=初始值”，则字段根据其数据类型的不同被自动初始化为相应的默认值。类的 public 字段称为类的公共字段。字段有静态字段与实例字段两种。静态字段是属于类的，实例字段是属于对象的。

（2）只读字段：定义字段时，用关键字 readonly 修饰声明的字段。readonly 声明的字段是不允许被改写的。不过有个例外，在构造函数中，可以改写其只读字段。

**【例 3.1】** 类的只读数据成员的修改方法实例。

源代码如下：

```
class Program
{   private readonly int i = 0 , j=0;   //声名只读字段 i、j
    Program()  //构造函数,与类的名称相同
    {
        this.j = 20; this.i = 100;       //构造函数中,可以改写只读字段 i、j 的值
    }
    static void Main(string[] args)
    {  Program a = new Program();        //建立对象引用,并实例化
       //a.i=10;                         //如果这样改写 i 的值,将会报错
       Console.Write("j={0}\t",a.j);     //通过类的实例对象输出只读型数据成员的值
       Console.WriteLine("i={0}",a.i);   //通过类的实例对象输出只读型数据成员的值
       Console.ReadKey();
    }
}
```

程序运行结果如下：

```
j=20  i=100
```

（3）静态字段和实例字段：使用关键字 static 修饰声明的字段就是静态字段（又称为静态变量）。静态字段是属于类的数据成员，因此，可以通过类名称直接访问静态字段。格式如下：

**类名称. 静态字段名称**

不用关键字 static 修饰声明的字段就是实例字段（又称之为实例变量）。类的每个实例都包含了该类的所有实例字段的一个独立副本。实例字段通过类的实例（对象）名称访问。

**【例 3.2】** 类的静态和实例数据成员的访问方法题例解析。

源代码如下：

```
class Program
{   private static int i = 10;          //声名一个静态字段(变量)
    private int j = 100;                //声名一个实例字段(变量)
    static void Main(string[] args)
    {
       Program a = new Program();       //建立对象引用,并实例化
       Console.WriteLine("instance member j=" + a.j); //用对象来访问字段 j
       Console.WriteLine("static member i=" + Program.i);
                                        //静态字段需要用类名来访问
       Console.ReadKey();               //暂停程序运行
    }
}
```

程序运行结果如下：

```
instance member j=100
static member i=10
```

（4）常数据成员：const 关键字用来声明常量，指定该成员的值只读，不允许修改。常量有别于字段，常量的值是在任何时候都不会被改变的，常量必须是基元类型。常量被默认为 static 的，但并不声明。例如：

```
public const int c=10;                    //用 const 关键字来声明一个常量
public const double pi=3.1415926;         //用 const 关键字来声明一个常量
```

（5）事件：事件是使对象或类能够提供通知的成员。事件的声明与字段类似，不同之处在于事件声明包含一个 event 关键字，并且事件声明的类型必须是委托类型。

在包含事件声明的类中，事件可以像委托类型的字段一样使用。该字段保存了一个委托的引用，表示事件处理程序已经被添加到事件上。如果尚未添加任何事件处理程序，则该字段为 null。

**2．函数成员**

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、析构函数、运算符以及索引器等。

（1）方法：是与某个类相关的函数，它们可以是实例方法，也可以是静态方法。实例方法处理类的某个实例，静态方法提供了更一般的功能，不需要实例化一个类。

（2）属性：是可以在客户机上访问的函数组，其访问方式与访问类的公共字段类似。

（3）构造函数：是在实例化对象时自动调用的函数。它们必须与所属类同名，且不能有返回类型。构造函数用于初始化字段的值。

（4）析构函数：类似于构造函数，它们的名称与类相同，但前面有一个“～”符号，析构函数在 C#语言中用得少，因为 CLR 会自动进行垃圾收集。

（5）运算符：用于定义类的实例的运算操作，运算符可以重载。

（6）索引器：允许对象以数组或集合的方式进行索引。

在 C#语言中，每个函数成员都必须与类相关。C#实际上区分了函数和方法：“函数成员”不仅包含方法，而且也包含类的一些非数据成员，例如索引器、运算符、构造函数和析构函数等，甚至还有属性。

在 C#语言中，所有的方法都在类定义中声明和定义（为 public 或 private）。方法的定义包括方法的修饰符、返回值的类型、方法名、输入参数的列表和方法体。

## 3.2.2 静态成员与实例成员

在 C#语言中，类可以有两类成员：静态成员和实例成员。非静态类可以包含静态的方法、字段、属性或事件。即使没有创建类的实例，始终通过类名而不是实例名称访问静态成员。

类的每一个实例都会复制这些实例成员，静态成员只会被复制一次，被复制后在整个应用程序的声明周期内都存在。默认情况下，成员是被定义为实例成员的。

**1．静态成员**

要将一个成员声明为静态成员，需要在成员名前面使用关键字 static。静态成员的特点如下。

（1）静态成员属于类，能被该类的所有实例共享。

（2）一个静态变量，在创建类的多个实例时，在内存中占用同一存储区域（永远只有一个副本）。

（3）静态函数成员（方法、属性、事件、构造函数）属于类的成员，在其代码体内不能直接引用实例成员，否则出现编译错误。

（4）静态成员只能通过类名称引用。

静态成员的引用格式如下：

```
类名称. 静态成员名
```

**2．实例成员**

实例成员属于具体的对象（类的实例），即每个对象都有实例成员的不同副本。实例成员的特点如下。

（1）实例成员必须通过类的对象来引用。

（2）类的变量属于类的对象，即每一个类的实例（对象）都有这些实例成员的副本。

（3）实例函数成员（方法、属性、构造函数、析构函数）作用于类给定的实例（对象），在实例函数体代码内可以直接使用实例成员，也可以直接使用类的静态成员。

实例成员的引用格式如下：

```
对象名称. 实例成员名
```

## 3.2.3　属性

字段和常量用于描述类的数据，通过访问修饰符 public、private、protected 设置对数据的访问权限。若需要分别限制对某些字段的读写操作，增强类的安全性和灵活性，C#利用属性来读取、修改或计算字段的值。定义属性的一般语法格式如下：

```
[访问修饰符] 数据类型 属性表
{
    get{  取得属性的代码，用 return 语句返回值  }
    set{  设置属性的代码，用 value 赋值    }
}
```

一个属性内可以包含一个 get 和一个 set 代码段，分别称之为 get 访问器和 set 访问器。get 访问器相当于一个具有属性类型返回值的无参数的方法，当在表达式中引用属性时，将调用该属性的 get 访问器计算属性的值。get 访问器必须用 return 语句返回值，并且所有的 return 语句都必须返回一个可隐式转换为属性类型的值。

set 访问器相当于一个具有单个属性类型值参数和 void 返回类型的方法。set 访问器的隐式参数始终命名为 value。当一个属性作为赋值的目标被引用时，就会调用 set 访问器，所传递的参数将提供新值。因为，set 访问器存在隐式的参数 value，所以，在 set 访问器中

不能自定义（使用）名称为 value 的局部变量或常量。

根据属性中是否存在 get 访问器、set 访问器，将属性分为以下 3 种。

（1）可读、可写属性：包含 get 访问器和 set 访问器。

（2）只读属性：只包含 get 访问器。

（3）只写属性：只包含 set 访问器。

**【例 3.3】** 类的属性及其 get 访问器、set 访问器实例解析。

源代码如下：

```
namespace ConsoleApp0303
{   class Student                          //自定义项目的类
    {   private string sname;              //学生姓名
        public string sName                //定义读、写型属性
        {
           get { return sname; }
           set { sname = value; }          //value 为 set 访问器的隐式参数名称
        }
        private string sex;                //学生性别
        public string sSex                 //定义读、写型属性
        {
            get { return sex; }            //定义 get 访问器
            set { sex = value; }
        }
        private int age;                   //学生年龄
        public int sAge                    //定义读、写型属性
        {
            get { return age; }            //定义 get 访问器
            set                            //定义 set 访问器
            {  if (value >= 17 && value <= 35) { age = value; }
                 else { age = 20;  }
          }
        }
       public Student() { }                //默认构造函数
       public Student(string xh,string xm, string xb, int vage){
                                           //重载构造函数
              this.sname = xm; this.sex = xb; this.age = vage;  }
        public void show(){
            Console.WriteLine("姓名：{0},性别：{1},年龄：{2}", sname, sex,
            age)   }
}//end for class Student
 class Program
 {
    static void Main(string[] args)
    {
        Student st = new Student();  /*创建类 student 的对象,调用默认的构造函
```

```
                                        数给对象 st 的每个字段赋值（年龄为赋值）*/
            st.sName = "赵颖"; st.sSex = "男";
            Console.Write("输入学生的年龄：");
            //ReadLine()输入一个年龄数字值,函数 Convert.ToInt32()将数字串转化为数值
            st.sAge = Convert.ToInt32(Console.ReadLine());
            st.show();   //调用类的成员函数,输出类的成员的值
            Console.ReadKey();
        }
    }
}
```

程序运行结果如下：

```
输入学生的年龄：15                    //输入的年龄值不满足条件,参见属性 sAge
姓名：赵颖,性别：男,年龄：20
```

**【说明】**

（1）在类的某个字段的任一位置处，按组合键 Ctrl+R+E，弹出“封装字段”对话框，在“属性名”输入框处输入属性名称（其他选项采用默认值），单击“确定”按钮，即可创建字段相关的属性。

（2）选择类的某个字段，然后右击，在弹出的快捷菜单中选择“重构”菜单选项，在弹出的级联菜单中，选择“封装字段”菜单选项，弹出“封装字段”对话框，在“属性名”输入框处输入属性名称（其他选项采用默认值），单击“确定”按钮，也可创建指定字段的属性。

以上方法都能创建与“字段”相关联的“属性”，将字段和属性封装在一起。封装用于隐藏内部实现，对外只是展示对类的基本操作，而不影响类的内部实现。

## 3.2.4　构造函数与析构函数

### 1．构造函数

类的定义中不能对数据成员进行初始化，为了能给数据成员赋某些初值，就要用到构造函数。构造函数的最大特点是在对象建立时自动执行。因此，对数据成员的初始化一般放在构造函数中完成。

C#规定构造函数必须与相应的类同名，它可以有参数，也可以没有参数，而且可以重载。

在定义类时，如果没有显式的定义构造函数，编译器自动为该类生成一个不带任何参数的默认构造函数。在定义类时，如果已定义了构造函数，则默认的构造函数将不再显式地调用，如果要显式地调用构造函数，在类中得显式地给出默认构造函数的定义。

构造函数和普通函数的区别如下。

（1）构造函数的名称必须与类名称相同，且一个类可以有一个或多个构造函数。

（2）构造函数不能有返回类型（包括 void 类型）。

### 2．析构函数

析构函数名与对应的类名相同，析构函数（destructor）与构造函数相反，只是在函数名前面加一个波浪符（～），以区别于构造函数。它不能带任何参数，也没有返回值（包括 void 类型）。只能有一个析构函数，且不能重载。如果用户未显式定义析构函数，编译系统会自动生成一个缺省的析构函数。

当对象脱离其作用域时（例如对象所在的函数已调用完毕），系统自动执行析构函数做“清理性”工作。当程序运行结束，需要回收对象所占用的内存资源。在对象被销毁之前，.NET 的公共语言运行时会自动调用析构函数，并使用垃圾回收器回收对象所占用的内存资源。

### 3．对象成员的初始化

在类的定义中，不能直接对数据成员进行初始化，可借助于构造函数对数据成员进行初始化。

**【例 3.4】** 类的构建、类的构造函数、析构函数实例解析。

源代码如下：

```
namespace ConsoleApp0304
{   class MyCalcs                  //自定义类
    {   private int a;             //定义类的数据成员（字段变量）
        public int vA              //定义类的字段（变量）相关的属性
        {   get { return a; }
            set { a = value; }     //value 为 set 访问器隐式的参数变量
        }
        //默认的构造函数
        public MyCalcs() { Console.WriteLine("成员,a：{0}",this.a); }
        //构造函数重载,参数的个数不同
        public MyCalcs(int a,int b) {
               Console.WriteLine("输出形参变量：a：{0};b：{1}", a, b); }
        //构造函数重载,参数的个数不同
        public  MyCalcs(int b) { Console.WriteLine("成员,a：{0};参数,b：{1}",
        this.a,b); }
        //构造函数重载,参数的个数和数据类型不同
        public  MyCalcs(string a,string b) { Console.WriteLine("输出形参,a+b={0}",
        a+b); }
        ～MyCalcs() { Console.WriteLine("这是析构函数,信息处理完毕！"); }
    }
    class Program
    {   static void Main(string[] args)
        {   MyCalcs c0 = new MyCalcs();        //创建类的对象,调用默认的构造函数
            MyCalcs c1 = new MyCalcs(10);      //创建类的对象,调用重载的构造函数
            MyCalcs c2 = new MyCalcs(10, 20); //创建类的对象,调用重载的构造函数
            MyCalcs c3 = new MyCalcs("10", "20");
                                               //创建类的对象,调用重载的构造函数
            Console.ReadKey();
```

```
        }
    }
}
```

程序运行结果（其中的注释语句不是程序运行结果）如下：

```
成员,a：0                  //调用默认的构造函数（输出的结果）
成员,a：0;参数,b：10       //调用重载的构造函数（输出的结果）
输出形参变量：a：10;b：20  //调用重载的构造函数（输出的结果）
输出形参,a+b=1020          //调用重载的构造函数（输出的结果）
```

**【程序解析】** 调用默认的构造函数时，数据成员 a 没有任何值，输入结果为 0；调用重载的构造函数，没有初始化数据成员 a 时，输入结果为 0；输出结果之后，按任一键时，调用析构函数，逐一释放（类的）对象及其相关所有的内存空间。

# 3.3 类的方法与函数

C#语言中方法就是与类相关的函数。在 VB、C++语言中，可以定义与类完全无关的全局函数，但在 C#语言中，每个函数都必须与类或结构相关。方法用于实现由类执行的计算和操作，是以函数的形式来定义的。方法是由 C#语句组成的可以完成特定功能的例程。C#语言中的类库中的各个类都提供了很多方法。

## 3.3.1 方法和函数的定义、调用

方法（或函数）由首部和方法体组成。

**1．方法（函数）的基本结构**

方法（函数）的基本语法结构如下：

```
[访问属性] 返回值类型 方法名称（[形式参数]）
{
    方法（函数体）
}
```

（1）方法的首部：方法的首部包括访问属性、类别、方法（函数）名称、形式参数。

① 访问属性：可以省略，主要是指 public、private、protected 这 3 种。默认访问权限为 private。

② 返回值类型：表示方法执行相应操作后返回值的数据类型，主要有整型、字符、字符串、空类、布尔型、对象型等。如果方法（函数）是空类型（void）时，即函数无返回值要求的情况下，此时函数类型符须写为 void，函数体中不能有返回表达式值的语句“return<表达式>”。如果方法有返回值，则通过语句“return<表达式>”返回。

③ 方法的名称：遵循标识符的命名规则。方法名最好“见名知意”，以增加代码的可读性。

④ 方法的参数：包括实际参数（简称实参）和形式参数（简称形参）。定义时指定的参数为形式参数，调用时指定的参数为实际参数。参数列表可以省略，表示该方法没有参数，但小括号不能省略。若有多个参数时，参数之间用逗号分隔，方法定义时每个参数的数据类型也不能省略。

（2）方法体部分：即方法要执行的语句序列，须用成对的大括号（{}）括起来。方法体包括说明语句、可执行语句部分。对其他方法的调用，方法体可以为空，但大括号不能省略。

① 声明语句部分：一般放在可执行语句之前，用于声明变量、外部变量或函数等。该语句可以省略。

② 执行语句部分：由若干语句构成。这些语句的基本结构包括顺序结构、选择结构、循环结构，还包括 C#的其他语句，如 using 语句、try 语句、catch 语句、finally 语句。函数体可以省略，但最外层的大括号不能省略。

**2．方法的定义**

方法的定义，就是根据方法的结构给出方法的首部和方法体部分。定义时，一般要指定方法的访问属性、数据类型、名称和形式参数。当方法无形式参数时，函数名称之后的小括号不能省略。方法体中可以没有任何语句（即方法体可以是空的），但最外层的大括号不能省略。

**3．方法的调用**

类的方法的调用，分为两种情形：被类的函数成员调用；被外部访问。当方法的访问属性不是 public 时，该方法只能被类的函数成员直接访问（调用）。当方法的访问属性为 public 时，可以被外部访问。

## 3.3.2 静态方法和实例方法

方法也是所属类的成员，和数据成员相似，方法也有静态方法和实例方法。

**1．静态方法及其定义**

若类的方法前加了 static 关键字，则该方法称为静态方法，反之称为实例方法。静态方法为类所有，可以通过类的对象来使用，也可以通过类的名称直接使用。但一般提倡通过类名来使用，因为静态方法只要定义了类，不必建立类的实例就可使用。静态方法只能用类的静态成员。

静态方法（函数）的结构和定义格式如下：

```
[访问属性] static 返回值类型 方法名称（[形式参数]）
  {
      方法（函数体）
  }
```

**2．静态方法调用**

静态方法的调用通过类名直接调用，调用格式如下：

```
类名.方法名
```

【使用规则】

（1）静态方法只能访问类的静态成员，不能访问类的非静态成员。

（2）非静态方法可以访问类的静态成员，也可以访问类的非静态成员。

（3）静态方法既可以用实例来调用，也可以用类名来调用。

（4）.NET Framework 类库中提供了很多包含静态方法的类，如 Math 类、String 类、StringBuild 类、Convert 类。

### 3．实例方法

不使用 static 修饰声明的方法为实例（instance）方法，实例方法对类的某个特定的实例进行操作，且能够访问静态成员和实例成员。在调用实例方法时，可以通过 this 指针显式地访问实例。

### 4．实例方法的调用

实例方法的调用通过类的实例对象调用，调用格式如下：

```
对象名.方法名
```

【例 3.5】 静态方法与实例方法实例解析。输入一个正整数，通过调用静态方法和实例方法计算其阶乘值，最后输出结果。

源代码如下：

```
namespace ConsoleApp0305
{   class MyFactClass   //自定义类 MyFactClass：其中定义了静态方法与实例方法
    {
        //定义静态方法,计算参数 n 的阶乘值;并返回阶乘值 n!
        //调用：通过类的实例调用;或者直接通过类名称调用;参数 n 表示任意正整数
        public static double staticfact(int n)
        {
            double p=1.0;
            for(int k=0;++k<=n;) p*=k;
            return p;
        }
        //实例方法,计算参数 n 的阶乘值;调用：通过类的实例调用
        //参数 n 表示任意正整数;函数返回：double 类型值,即参数 n 的阶乘值 n!
        public double instancefact(int n)
        {   double p=1.0;
            for(int k=0;++k<=n;)p*=k;
            return p;
        }
    }//END class MyFactClass
    //系统自定生成的类 Program：静态方法与实例调用实例
    class Program
    {
        private static int n;                    //静态数据成员
        static void Main(string[]args)
        {
            Console.Write("输一个正整数[>5]:");
```

```
            //从键盘读入数字串,通过类 int 的方法 Parse()转化为整数赋值给 n
            n=int.Parse(Console.ReadLine());
            MyFactClass a=new MyFactClass();//定义类 MyFactClass 的实例(对象)名称
            double s1=a.instancefact(n);//通过类的对象引用类的方法 instancefact
            //通过类名称(MyFactClass)引用静态方法 staticfact
            double s2=MyFactClass.staticfact(n);
            //调用类 Program 的静态成员函数 outvalue()输出结果
            outvalue(s1,s2);
            Console.ReadKey();//暂停程序运行等待输入
        }
        //方法: outvalue(double s1,double s2); 作用:输出参数 s1、s2 的值
        private static void outvalue(double s1,double s2)
        {
            Console.WriteLine("\t 实例方法计算结果: {0}",s1);
            Console.WriteLine("\t 静态方法计算结果: {0}",s2);
        }
    }//end class program
}
```

程序运行结果如下：

```
输一个正整数[>5]:6
    实例方法计算结果: 720
    静态方法计算结果: 720
```

## 3.3.3 方法的参数与传递机制

方法的参数包括实际参数（简称为实参）和形式参数（简称为形参）。方法定义时指定的参数称为形式参数，方法调用时指定的参数称为实际参数。

方法的调用一般伴随着函数之间的参数传递，参数实现了方法（函数）之间的数据传递或数据交换。方法中有 4 种类型的参数。

（1）值形式参数（values parameter）：定义时不带修饰符。

（2）引用参数（reference parameter）：定义时用 ref 加以修饰。

（3）输出参数（output parameter）：定义时用 out 加以修饰。

（4）形参数组（parameter array）：定义时用 parameter 加以修饰。

调用方法时，实际参数的类型、个数应与定义时指定的形式参数的类型、个数一一对应。即实际参数与形式参数在数量、数据类型、顺序上应严格一致，否则会发生“类型不匹配”的错误。

**1．值形式参数**

方法定义时，如果形式参数只是指定了参数的类型而未加其他修饰说明，则该参数为值形式参数。其初始值为对应实参的值（是实参值的副本，实参值的类型隐式的转化为形参的类型），与实际参数占用不同的存储空间。形式参数值的改变不影响对应的实际参数的值。

### 2．引用参数

定义方法时，用 ref 修饰的形式参数为引用形式参数，用于输入参数的传递。当形式参数为引用参数时，实参和形参都必须显式地使用 ref 关键字加以说明，且实参应为变量，并具有初值（否则出错）。引用参数传递的不是实参的值，而是实参变量所在存储空间的地址。在调用方法时，并不为形参重新分配存储空间，而是占用实参的存储空间。因此在方法调用的过程中，形参和实参实际上是“同一个变量”，如果形参变量值发生了变化，那么对应的实参变量值也会发生同样的变化。

### 3．输出（out）参数及应用实例

使用输出参数可以让一个函数返回多个值。定义输出参数的方法是在定义方法的参数时，在形参前面加上 out 关键字。方法定义和调用时，形式参数指定为输出参数，则形式参数都须显式的使用 out 关键字加以说明。输出参数传递的不是实参的值，而是实参变量所在存储空间的地址。

指定输出参数的方法调用时，并不为形参重新分配存储空间，而是与实参占用同一存储空间。因此，形参变量值的改变导致实参变量值同样的变化。在函数返回时，out 类形参应该有明确的值，否则出错。

**【例 3.6】** 函数的参数及其传递机制，值传递和引用传递的区别。

源代码如下：

```
namespace ConsoleApp0306
{
    class Program
    {
        static void Main(string[] args)    //主函数,程序入口
        {   double s=0,t;
            int n=getinteger();         //调用类的成员函数以输入一个整数存储于 n 中
            Console.WriteLine("n={0}",n);  //输出 n 本身的值
            getfact(ref s,n);           //s 作为引用类的输入参数,应该有初始值
            getjc(out t,n);            //t 作为引用类的输出参数,可以不初始化
            Console.WriteLine("n={0},{1}!={2}",n,n,s);
            Console.WriteLine("n={0},{1}!={2}",n,n,t);
            Console.WriteLine("n={0},{1}!={2}",n,n,Math.Round(s,0).ToString());
            Console.ReadKey();
        }
        private static int getinteger() //自定义函数,用于输入一个整数并返回。函
                                        //数无参数
        {
            int n;
            do  //do…while()循环的使用
            {   Console.Write("输入一个十进制整数[>5]：");
                //从键盘输入一个整数,通过类 int 的方法 Parse()转化为整数赋值给变量 n
                n=int.Parse(Console.ReadLine());
            }while(n<5);
            return n;
```

```
        }
        /*自定义函数 getfact(ref double p,int n),计算值参数 n 的阶乘值,通过引用
        参数 p 返回;参数 p 为引用类输入参数,应有初始值;n 为值参数,接受实际参数的值*/
        private static void getfact(ref double p,int n)
        {   int k=0;
            Console.WriteLine("ref double p={0}",p);
            p=1;                    //给引用参数 p 另外赋一个初始值
            while(++k<=n) p=p*k;    //计算 n! 存储于变量 p 中
        }
        //自定义函数,getjc(out double p,int n);参数 p 为引用类输出参数,可以无初始值
        //作用：计算值参数 n 的阶乘值,通过输出参数 p 返回给实际参数(变量)
        //p 值的改变直接影响对应的实际参数的值;通过参数 p“返回”n!
        private static void getjc(out double p,int n)//p 输出类参数可以无初始值
        {
            int k=0;
            p = 1;                  //给输出类参数 p 赋一个初始值,否则出错
            while(++k<=n) p=p*k;  //函数返回时,out 类参数变量须有明确的值,否则出错
        }
    }
}
```

**【程序解析】**

（1）方法 getinteger()没有参数，实现输入一个整型数，并返回输入值。函数 getfact(ref double p,int n)和函数 getjc(out double p,int n)都没有“返回值”，数据交换都是通过引用参数传递实现的。

（2）函数 getfact(ref s,n)调用时，s 作为引用类输入参数，应初始化；函数 getjc(out t,n)调用时，t 作为输出类参数，可以不初始化。这是 ref 参数与 out 参数的区别。两者都进行“引用”传递，通过引用参数实现函数之间的数据交换。

（3）参数进行引用传递时，形式参数值的改变将影响对应的实际参数变量的值（以此实现函数之间的数据交换）。

**【例 3.7】** ref、out 类参数应用实例解析。

源代码如下：

```
namespace ConsoleApp0307
{
   class Program
    {
        //说明：静态方法,分解文件所在的路径和文件名称,并通过输出参数返回
        //参数：vdir,vname：为输出(out)参数,形式参数值的改变将返回给对应的实际参数
        private static void SplitPath(
                    ref string path, out string vdir, out string vname)
        {   //获取文件所在的目录名称及其文件名称本身
            int n = path.Length;
            int k = path.LastIndexOf(@"\");//得到路径中最后一个字符\在路径串中的位置
```

```
            vdir = path.Substring(0, k + 1);          //得到目录名称
            vname = path.Substring(k + 1, n - k - 1);//得到文件名称
        }
        static void Main(string[] args)
        {
            string dir="",name="",path="";
            //得到可执行文件及其所在的完整路径
            path=Process.GetCurrentProcess().MainModule.FileName;
            Console.WriteLine("执行文件及其所在路径：");
            Console.WriteLine("{0}",path);
            //形参明确为 ref（或者 out）时,方法调用时,实参也应该明确为 ref（或者 out）
            SplitPath(ref path,out dir,out name);   //实参为输出(out)参数
            Console.WriteLine("当前目录：{0}",dir);   //输出返回的路径
            Console.WriteLine("文件名称：{0}",name);  //输出返回的文件名称
            Console.ReadKey();
        }
    }//end for class Program
}
```

程序运行结果在此省略，请读者自行分析并上机验证。

**【程序解析】** 方法 SplitPath(ref string path,out string vdir,out string vname)的类型是 void，因此没有返回值。但是通过输出 out 参数，返回了两个 string 类的值。方法 SplitPath 调用时，实际参数 vdir、vname 为空串，调用结束之后，实际参数 vdir、vname 有明确的值；在方法定义时，方法的形式参数明确为 ref（或者 out）时，则方法调用时，实际参数也应该明确为 ref（或者 out）。实际参数为 ref 时，应该有明确的值；形式参数为 out 时，函数返回（或结束）时应有明确的值。

C#获取执行程序所在的当前路径的方法有以下几种。

（1）获取和设置当前目录的完全路径，代码如下：

```
string str=System.Environment.CurrentDirectory;
```

（2）获取启动了应用程序的可执行文件的路径，不包括可执行文件的名称，代码如下：

```
string str=System.Windows.Forms.Application.StartupPath;
```

（3）获取新的 Process 组件并将其与当前活动的进程关联的主模块的完整路径及文件名。代码如下：

```
string str=Process.GetCurrentProcess().MainModule.FileName;
```

（4）获取 Thread 的当前应用程序域的基目录，代码如下：

```
string str=System.AppDomain.CurrentDomain.BaseDirectory;
```

（5）获取应用程序的当前工作目录，代码如下：

```
string str=System.IO.Directory.GetCurrentDirectory();
```

（6）获取和设置包含应用程序的目录名称，代码如下：

```
string s=System.AppDomain.CurrentDomain.SetupInformation.ApplicationBase;
```

（7）获取当前进程的完整路径及文件名，代码如下：

```
string s=this.GetType().Assembly.Location;
```

（8）获取启动了应用程序的可执行文件的路径及其可执行文件名，代码如下：

```
string s=System.Windows.Forms.Application.ExecutablePath;
```

（9）获取启动应用程序的可执行文件的路径及其可执行文件名，代码如下：

```
string s=System.Environment.CommandLine;
```

**4．形参数组**

在定义方法时，可以将函数的最后一个参数定义为参数数组，需要使用 params 关键字修饰。参数数组必须是形式参数表的最后一个参数，且只能是一维数组。params 关键字不能与 ref、out 组合使用。参数数组主要用于传递个数可变的参数。

**【例 3.8】** 参数数组即 params 关键字应用实例解析。建立控制台应用程序，系统生成的 using 语句在此省略。

源代码如下：

```
namespace ConsoleApp0308
{
    class Program
    {   //函数 void  showelem(),用于显示参数数组的元素个数及元素值
        //a 为形参数组,用 params 修饰
        static private void showelem(params int[] a)
        {
            Console.WriteLine("形参数组元素个数：{0}", a.Length);
                                                    //a.Length 为数组长度
            Console.Write("形参数组各元素的值：");
            foreach (int k in a) Console.Write("{0,6}", k);
                                                    //输出各元素,每位占 6 位宽度
            Console.WriteLine("");
        }
        /*方法重载：str 为 string 类的形参数组,用 params 修饰;显示数组元素个数及其各
        元素值*/
        static private void showelem(params string[] str)
        {
            //str.Length 为数组的长度,即元素(字符串)的个数
            Console.WriteLine("形参数组元素个数：{0}", str.Length);
            Console.Write("形参数组元素(城市)值：");
            //输出形参数组的元素值,{0,-6}表示每个输出项占 6 位宽度,左对齐输出
            foreach (string ss in str) Console.Write("{0,-6}", ss);
            Console.WriteLine("");
```

```
        }
        static void Main(string[] args)
        {   int[] ar = { 11, 12, 13, 14, 15 };
            showelem(ar);     //调用方法,输出数组 ar 的大小及各个元素的值
            string[] citys = { "深圳", "北京", "上海", "大连", "重庆", "成都" };
            showelem(citys); //调用重载方法,输出数组 citys 的大小及其各个元素的值
            Console.ReadKey();
        }
    }
}
```

运行结果显示在此省略，请读者自行分析并上机验证。

## 3.3.4　方法的返回值

方法可以有返回值，也可以没有返回值。当方法的类型指定为非 void 类型时，就应该有返回值。返回值通过“return 表达式”实现。return 语句的一般形式如下：

```
return 表达式;
```

或者

```
return (表达式);
```

**【说明】**

① 方法的值是指方法被调用之后返回的值。

② 在方法中允许有多个 return 语句。调用方法时只有一个 return 语句被执行。第一个被执行的“return 表达式”语句起作用。

③ 方法的值的类型和定义方法时指定的类型应保持一致，如果两者不一致，返回值的类型由方法定义的类型决定，并将返回值的类型转化为定义的类型，不能自动转化时将出现语法性错误。

④ 当方法的类型指定为 void 时，函数就没有返回值。不返回值的函数，应明确定义为“空类型（void）”。

⑤ 方法为 void 类型时，方法体中可以有 return 语句，但不能是返回“表达式”值的 return 语句，也不能在程序中使用被调函数的方法值，否则出现语法性错误。

⑥ return 语句有终止方法并返回给调用的作用。因此要注意 return 语句的位置关系。

## 3.3.5　方法重载

方法重载就是同一个方法名可以对应多个方法的实现，即允许多个同名的方法同时存在。方法重载须满足以下条件。

（1）方法名称必须相同。

（2）形式参数的个数不同。

（3）相同位置上的参数的数据类型不同。

**【例 3.9】** 函数的重载实例解析。

源代码如下：

```
namespace ConsoleApp0309
{
    class Program
    {    static void Main(string[]args)       //程序入口
         {
             Console.WriteLine("{0}+{1}={2}",10,20,add(10,20));
             Console.WriteLine("{0}+{1}={2}","中国","重庆",add("中国","重庆"));
             Console.ReadKey();
         }
         //函数：add,参数的类型为 int;功能：实现两个形式参数变量求和;返回：两个变量之和
         private static int add(int a,int b){  return a+b;  }
        //重载函数 add,参数类型不同（为 string）;功能：实现字符串相加;返回连接之后的字符串
         private static string add(string a,string b)  {  return a+b;  }
    }
}
```

程序运行结果如下：

```
10+20=30
中国+重庆=中国重庆
```

## 3.3.6 C#的 Main 函数

在 C#语言中，无论是控制台应用程序，还是 winform 程序，其程序框架中有一个重要的文件 program.cs。该文件的结构参见例 3.8。

**【例 3.10】** C#的 Main 函数及其结构。以下框架由系统自动生成，导入名称空间，程序将以新建项目名称自动创建一个命名空间。

源代码如下：

```
namespace ConsoleApp0310
{
    class Program   //系统自动生成的默认的类 program
    {   static void Main(string[] args){ …… }
                    //C#的主函数 Main 是程序入口
        自定义类的函数（或方法）
    }
}
```

在此只给出了 Main()函数及其结构（框架）。具体的实例请参阅之前的所有源代码中的 Main()。

## 3.3.7 C#消息对话框类 MessageBox 及方法 show

要引用此类，首先通过引用该方法的项目的属性选项，手动添加引用 System.Windows.

Forms.dll，同时，在代码中添加 using System.Windows.Forms。原型如下：

```
DialogResult MessageBox.Show（Text,Caption,buttons,icon,DefaultButtons,
Option）;
```

该类的方法的作用就是弹出一个如图 3.1 所示的 MessageBox 消息框对话框，其中显示的文本、标题、按钮、图标可以根据需要具体设置。

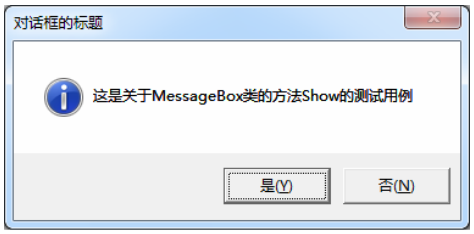


图 3.1　MessageBox 消息对话框

**【参数及其作用】**

（1）Text：字符串（string）类型，消息框中显示的正文，如图 3.1 所示。

（2）Caption：字符串（string）类型，消息框的标题。

（3）buttons：用于设置对话框上显示的按钮类型和个数，是一个枚举值，集成在枚举类型（MessageBoxButtons）中，有以下几种按钮。

① MessageBoxButtons.OK：只有“确定”按钮，返回值为 DialogResult.OK。

② MessageBoxButtons.OKCancel：按钮类型，包括“确定”“取消”。

③ MessageBoxButtons.AbortRetryIgnore：按钮类型，包括“终止”“重试”“忽略”。

④ MessageBoxButtons.YesNoCancel：按钮类型，包括“是”“否”“取消”。

⑤ MessageBoxButtons.YesNo：按钮类型，包括“是”“否”。

⑥ MessageBoxButtons.RetryCancel：按钮类型，包括“重试”“取消”。

⑦ 设置“确定”按钮时，对话框的返回值为 DialogResult.OK；设置“取消”按钮时，对话框的返回值为 DialogResult.Cancel；设置“终止”按钮，对话框的返回值为 DialogResult.Abort；设置“重试”按钮时，对话框的返回值为 DialogResult.Retry；设置“忽略”按钮时，对话框的返回值为 DialogResult.Ignore；设置“是”按钮时，对话框的返回值为 DialogResult.Yes；设置“否”按钮时，对话框的返回值为 DialogResult.No。

（4）icon：设置对话框上显示的图标样式，也是一个枚举值，集成在枚举类型 MessageBoxIcon 中，其值的含义如下。

① MessageBoxIcon.Information：该图符由一个圆圈加其中的一个字母 i 组成。

② MessageBoxIcon.Question：该图符由一个圆圈加其中的一个？组成。

③ MessageBoxIcon.Asterisk：该图符由一个圆圈加其中的一个字母 i 组成。

④ MessageBoxIcon.Error：该图符由一个红色背景的圆圈加其中的一个白色的字母 X 组成。

⑤ MessageBoxIcon.Exclamation：该图符由一个黄色背景的三角形加其中的一个感叹号(！)组成。

⑥ MessageBoxIcon.Hand：该图符由一个红色背景的圆圈加其中的一个白色的字母 X 组成。

（5）option：设置对话框的默认操作，也是一个枚举值，集成在枚举类型 MessageBoxOptions 中，其值的含义如下。

① MessageBoxOptions.DefaultDesktopOnly：消息框显示在活动桌面上，此乃默认操作设置。

② MessageBoxOptions.RightAlign：消息框的文本右对齐。

③ MessageBoxOptions.RtlReading：消息框显示在活动桌面上，此乃默认操作设置。

④ MessageBoxOptions.ServiceNotification：消息框显示在活动桌面上，此乃默认操作设置。

**【例 3.11】** MessageBox 类的 Show 实例解析。

源代码如下：

```
using System.Windows;              //手工方式添加的引用
using System.Windows.Forms;        //手工方式添加的引用
namespace ConsoleApp0311           //项目名称
{
   class Program
   {
      static void Main(string[] args)
      {   string t = "这是关于 MessageBox 类的方法 Show 的测试用例";
          MessageBoxButtons a = MessageBoxButtons.YesNo;//定义按钮的类型和个数
          MessageBoxIcon b = MessageBoxIcon.Information; //定义图标的类型
          MessageBoxDefaultButton d = MessageBoxDefaultButton.Button1;
                                      //设置默认按钮
          MessageBoxOptions op1 = MessageBoxOptions.DefaultDesktopOnly;
                                      //设置默认操作
          MessageBoxOptions op2 = MessageBoxOptions.RightAlign;
          MessageBoxOptions op3 = MessageBoxOptions.RtlReading;
          MessageBoxOptions op4 = MessageBoxOptions.ServiceNotification;
          /*MessageBox.Show(t, "对话框的标题", a, b, d, op1);用 show 方法弹出
          对话框*/
          DialogResult res = MessageBox.Show(t,"对话框的标题",a,b,d,op1);
          //以下 if 语句测试对话框的“返回值”(保存在 res 变量中),并加以利用
          if (res == DialogResult.Yes)
               {  Console.WriteLine("单击了对话框上的'是(Y)'按钮");  }
          else { Console.WriteLine("单击了对话框上的'否(N)'按钮"); }
          Console.ReadKey();
       }
    }
}
```

## 3.4 this 指针

this 可以理解为引用类的对象。在 C#语言中，this 指代当前运行所在的类。this 关键

字将引用类的当前实例，通过 this 调用类自身的变量、属性和方法。特别是在方法或函数的参数变量和类的变量或属性同名时能用 this 指针指代加以区分。静态成员函数没有 this 指针，在静态的方法中不能使用 this，如 Main 方法就是一个静态的方法，因此，不能在 Main 方法中使用 this。

在 C#语言中，关于 this 总结如下。

（1）this 关键字用于引用被访问成员所在的当前实例，静态成员函数没有 this 指针。this 关键字可以用来在构造函数、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，否则将产生错误。

（2）在类的构造函数中出现 this，作为一个值类型，表示对正在构造的对象本身的引用。

（3）在类的方法中出现 this，作为一个值类型，表示对调用该方法的对象的引用。

（4）在结构的构造函数中出现 this，作为一个变量类型，表示对正在构造的结构的引用。

（5）在结构的方法中出现 this，作为一个变量类型，表示对调用该方法的结构。

**【例 3.12】** this 指针应用实例解析。

源代码如下：

```
namespace ConsoleApp0312
{
    class Program
    {
        private string name;   private int age;     //定义类的私有成员
        static void Main(string[] args)
        {
            Program t = new Program(); //定义类 Program 的对象（实例）
            t.SetLocalValues();         //通过类的实例 t 引用类的方法
            t.GetLocalValues();         //通过类的实例 t 引用类的方法
            Console.WriteLine("在类的主函数 Main 中：");
            Console.WriteLine("输出类 Program 的私有成员：this.age={0}",
            t.getage1(100));
            Console.WriteLine("输出类成员函数 getage2(int age)的形参变量：
            age={0}", t.getage2(100));
            Console.ReadKey();
        }
        private void SetLocalValues()
        {
            //以下语句通过 this 指针引用类的私有数据成员,并为其数据成员赋值
            this.name = "张三丰"; //this.name 表示类 Program 中定义的成员变量 name
            this.age = 45;        //this.age 表示类 Program 中定义的成员变量 age
        }
        public void GetLocalValues()
        {
            string name; int age;      //定义本函数的局部变量 name、age
```

```
            name = this.name; /*通过 this 引用类的数据成员 name,赋值给本函数的局部
                              变量 name*/
            age = this.age;   /*通过 this 引用类的数据成员 age,赋值给本函数的局部
                              变量 age*/
            Console.WriteLine("在成员函数 GetLocalValues 中: ");
            Console.WriteLine("成员函数的局部变量 name 的值为:" + name);
            Console.WriteLine("成员函数的局部变量 age 的值为:" + age);
        }
        public int getage1(int age)
         //函数的形式参数 age 和类 Program 的私有成员 age 同名
        {   //虽然形参变量 age 接收了实参的值 100,但返回的是类 Program 的私有成员 age 的值
            return this.age;//返回类 Program 的私有成员 age 的值,而非形参 age 的值
        }
        public int getage2(int age) //函数的参数 age 和类 program 的私有成员 age 重名
        {  return age;   }          // 返回本函数的形式参数变量 age 的值
    }
}
```

程序运行结果如下：

```
在成员函数 GetLocalValues 中:
  成员函数的局部变量 name 的值为:张三丰
  成员函数的局部变量 age 的值为:45
在类的主函数 Main 中:
  输出类 Program 的私有成员: this.age=45
  输出类成员函数 getage2(int age)的形参变量: age=100
```

## 3.5 类的继承性与多态性

类的三大特征为封装性、继承性和多态性。本节介绍类的继承性、多态性及其实现方法。

### 3.5.1 类的继承性

在大型软件的设计与开发过程中，程序的可维护性和可扩展性是亟待解决的问题。类的可继承性，不仅可以扩展类的功能，使程序的扩展性得以极大的发挥，并能很好的实现代码的重用。

在 C#语言中，一个类可以继承另外一个类。被继承的类称之为基类（或父类），继承其他类的类简称为子类，又称之为派生类。在具有继承关系的两个类中，子类不仅具有自己独有的成员和特征，还具有父类的成员和特点。

### 1．子类的派生

C#语言中派生子类的语法格式如下：

```
[访问修饰符] class 子类名称：基类名称
{
   子类的成员（包括数据成员、函数成员）
}
```

【说明】 被方括号括起来的选项为可选项。

（1）类访问修饰符，参见 3.1.1 节的介绍。继承一个类时，父类成员的可访问性是一个主要问题。子类不能访问父类的私有（private）成员，可以访问父类的公有（public）成员。但是任何类都能访问类的公有（public）成员，这给程序和数据带来安全隐患。为了解决这一问题，C#提供了一种保护性访问修饰机制，类的继承时，通过访问修饰符 protected，使得被访问修饰符 protected 修饰的成员允许被其子类访问，而不被其他非子类访问。为此，通过 this 指针访问本类的成员，通过关键字 base 访问父类的成员。

（2）关键字 class，表示类的定义，只能是小写字母。

（3）子类名称，是派生新生成的类的名称，父类名称是被继承的类的名称。

（4）类体，用大括号（{}）括起来，称之为类的主体，其成员包括数据成员、函数成员。成员均可省略，但大括号不能省略。

【例 3.13】 类的继承与派生实例解析。

源代码如下：

```
namespace ConsoleApp0313          //ConsoleApp0313 为控制台应用程序项目名称
{
    //自定义类,封装个人基本信息
    public class Person
    {   protected string name, sex;
        protected int age;
        public string Sex         //定义类的属性
        {  get { return sex; }
           set { sex = value; }
        }
        public string Name        //定义类的属性
        {  get { return name; }
           set { name = value; }
        }
        protected int Age         //定义类的属性
        {  get { return age; }
           set { age = value; }
        }
       public Person(){}                                    //默认构造函数
       public Person(string name,string sex,int age)    //重载构造函数
       {
           this.name = name;   //将参数变量的 name 赋值给类的私有成员 name
```

```
        this.sex = sex;      //将参数变量的 sex 赋值给类的私有成员 sex
        this.age = age;      //将参数变量的 age 赋值给类的私有成员 age
    }
}//end class Person
class Students : Person          //定义派生类,继承于基类 Person
{
    private string university;   //毕业学院
    public string University
    {   get { return university; }
        set { university = value; }
    }
    public Students() { }        //子类 Students 的默认构造函数
    //重载子类 Students 的构造函数;成员 name,sex,age 继承自父类 Person 的属性
    //该子类的构造函数中的：base(参数)中的参数与父类的构造函数的参数类型、个数应该相同
    public Students(string name,string sex,int age,string xy):base(name,
    sex,age)
      {  //基类的参数 base(name,sex,age),表示继承自父类的属性（name,sex,age）
        this.university = xy;     //个人身份：学生,子类 Students 的扩展属性
    }
    public void show()
    {
      Console.Write("学生：{0},性别：{1},年龄：{2},",base.Name,base.Sex,
      base.Age);
      Console.WriteLine("毕业学院：{0}", this.university);
    }
}//end class Student
class Teachers : Person
{
      Teachers() { }  //子类 Teacher 默认构造函数
    //重载子类 Teachers 的构造函数;成员 name,sex,age 继承自父类 Person 的属性
    //该子类的构造函数中的：base(参数)中声明的参数与父类的构造函数的参数类型、个
    //数应该相同
    public Teachers(string name,string sex,int age,string zc):base(name,
    sex,age)
    {   //基类参数 base(name,sex,age),表示继承自父类的属性(name,sex,age)
        this.job_title = zc;        //教师职称,子类 Teachers 的扩展属性
    }
    private string job_title;      //职称
    public string Job_title
    {
        get { return job_title; }
        set { job_title = value; }
    }
    public string getmessage()    //返回个人信息字符串
    {   //格式化生成字符串,并连接到变量 xx 中
```

```
            string xx = string.Format("教师:{0},性别:{1},", base.Name, base.Sex);
            xx += string.Format("年龄:{0},职称:{1}", base.Age, this.job_title);
            return xx;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {   Students s = new Students("李嘉","女",20,"清华大学");
            Teachers t = new Teachers("张婧", "女", 33, "副教授");
            s.show();              //调用类 Student 的方法 show()输出学生信息
            Console.WriteLine(t.getmessage());
                                   //调用类 Teacher 的方法返回信息字符串
            Console.ReadKey();
        }
    }
}
```

程序运行结果如下：

```
学生：李嘉,性别：女,年龄：20,毕业学院：清华大学
教师：张婧,性别：女,年龄：33,职称：副教授
```

**【程序解析】** 首先定义了基类（Person），以此作为基类，定义派生（子）类 Students 和 Teachers。两个子类中未定义字段 name、sex、age，而是继承了（具有）基类的该属性。在子类中使用关键字 base 直接引用基类的该属性，通过在子类的构造函数之后添加“：base（参数变量表）”。

**【说明】**

① 在子类中可以使用关键字 base 直接引用基类的属性、方法、构造函数，方法是在子类的构造函数之后添加“：base（参数变量表）”，以用于指定该子类的构造函数调用哪个基类的构造函数，这样便可以初始化被继承的父类的属性。如本例中的重载构造函数 Teacher(string name, string sex, int age, string zc): base(name,sex,age)、Students(string name, string sex, int age, string xy): base(name, sex, age)。

② 子类可以具有基类的特征，也可以具有自己的特征。例如，子类 Student 中没有定义父类的同名字段（name、sex、age），但可以具有父类的这些字段（特征），但子类 Student 中有自己的字段（university），这是父类中没有的。

③ 派生类中，如果声明了与基类同名的成员，可以使用 new 对基类中的同名成员进行覆盖。

④ 当用关键字 sealed 修饰一个类时，该类不能被继承。

⑤ 在 C#语言中，一个子类不能同时继承多个父类，即只能从一个父类中继承，这称之为继承的单根性。

⑥ 继承实现了代码重用。合理的使用继承，使代码更加简洁。继承使得程序的结构简单，父类与子类的层次结构清晰。使得子类只关注自己的相关行为和状态，不需要关注

父类的行为与状态。

**2．子类对基类方法的访问**

**【例 3.14】** 以下几个实例，简要介绍了 C#语言中子类调用父类的方法及其具体实现。通过实例解析也说明了类中初始化构造函数的执行顺序。

（1）通过子类无参数的构造函数创建子类实例。

创建父类 Person 和子类 Student。

```
namespace ConsoleApp031401          //控制台应用程序的项目名称
{
    public class Person
    {
        public Person() {   Console.WriteLine("我是学生");  } //基类的默认构造函数
    }
    public class Student : Person   //定义派生类,基类为 Person
    {
        public Student(){ Console.WriteLine("我是在校本科大学生"); }
                                    //子类的默认构造函数
    }
    class Program
    {
        static void Main(string[] args)
        {   Student student = new Student();  //通过子类无参构造函数创建子类实例
            Console.ReadKey();
        }
    }
}
```

程序运行结果如下：

```
我是学生
我是在校本科大学生
```

**【程序解析】** 从程序运行结果可见，通过调用子类无参构造函数创建子类实例，会默认调用父类无参构造函数。如果把父类的无参构造函数去掉，结果会报“Person 不包含 0 个参数的构造函数”之错。

（2）通过子类有参构造函数创建子类实例，同时为子类和父类添加有参构造函数。

```
namespace ConsoleApp031402
{
    public class Person             //定义基类 Person
    {   public Person() { Console.WriteLine("我是学生"); } //基类默认构造函数
    }
    public class Student : Person  //定义派生类 Student,基类是 Person
    {   //子类的有参构造函数
        public Student(string name){ Console.WriteLine("我的名字叫{0}", name); }
    }
```

```
    class Program
    {
        static void Main(string[] args)
         {  //通过子类的有参数构造函数创建子类实例,调用基类的构造函数
            Student student = new Student("李佳");
            Console.ReadKey();
        }
    }
}
```

程序运行结果如下：

```
我是学生
我的名字叫李佳
```

从程序运行结果可出，通过调用子类有参构造函数，同样默认会调用父类无参构造函数。

（3）在子类中明确指出调用父类相应的构造函数。

以上实例，默认调用了父类的无参构造函数。在子类中，通过使用 base 可以指定调用父类的某个有参构造函数。

```
namespace ConsoleApp031403
{
    public class Person
    {
        private string name;        //定义基类的字段
        public string Name          //设置基类的属性
        {   get { return name; }
            set { name = value; }
        }
        public Person(){ Console.WriteLine("我是学生."); }  //基类默认构造函数
        public Person(string name) {                         //基类构造函数重载
            this.Name = name;       //通过参数 name 的值修改了基类的属性 Name 的值
            Console.WriteLine("我的名字叫：{0}",name); }//输出参数变量 name 的值
    }
    public class Student : Person //定义基类 Person 的派生类 Student
    {
        //在子类中,通过 base(name)访问基类的字段 name 值,并将其中字母转化为大写
        //调用子类的方法 ConvertToUpper(name)将 name 变量值中的字母转化为大写字母
        public Student(string name) : base(ConvertToUpper(name))
                                    //重载子类构造函数
        {   //输出子类 Student 实例中的参数 name 的值
            Console.WriteLine("我是在校大学生,名字叫：{0}", name);
        }
        private static string ConvertToUpper(string name)  //定义子类的方法
        {  return name.ToUpper();  }
```

```
    }
    class Program
    {
        static void Main(string[] args)
        {   //通过子类的有参数构造函数创建子类实例
            Student student = new Student("Michael");
            Console.WriteLine("子类获取父类的Name属性值为：{0}", student.Name);
            Console.ReadKey();   }
    }
}
```

程序运行结果如下：

```
我的名字叫：MICHAEL                    //调用基类的有参数构造函数,输出参数变量name的值
我是在校大学生,名字叫：Michael     //调用子类的重载构造函数,输出name变量的值
子类获取父类的Name属性值为：MICHAEL  //在子类中访问了基类Person的属性Name的值
```

**【程序解析】**

① 通过子类设置父类的公共属性。在子类的构造函数中加入“：base（参数变量）”，通过参数变量设置父类的属性的值。在子类中，当父类通过 base 拿到子类的参数时，还可以对该参数做一些处理，例如代表父类的 base 把从子类拿到的参数（变量中的字母）转换成大写。

② 代码执行过程。调用子类有参构造函数，并把该参数传值给父类有参构造函数→调用父类有参构造函数，并给父类公共属性 Name 赋值（this.Name = name）→子类实例调用父类的公共属性（如本实例中的 Name）。

## 3.5.2 类的多态性

多态性是指两个或多个属于不同类的对象，对于同一个消息（如方法调用）作出不同的相应。多态的实现有如下两种方法。

**1．使用虚方法实现多态**

在父类中定义虚方法，在子类中重写该虚方法，以实现多态。定义虚方法的语法格式如下：

```
[访问修饰符] virtual 返回类型 方法名()
{
   方法体
}
```

在子类中，使用 override 重写虚方法。语法格式如下：

```
[访问修饰符] override 返回类型 方法名()
{
   方法体
}
```

【例 3.15】 类的多态性的实现应用实例解析。

源代码如下：

```
namespace ConsoleApp0315
{   class classA
    {   static int x=1;
        //基类默认的构造函数
        public classA() { Console.WriteLine("类 classA 的构造函数！"); }
        virtual public void m(){ Console.Write(x); }   //定义基类的虚函数
    }
    class classB : classA
    {
        new static int x=2;          //声明与基类同类型同变量名称的变量
        override public void m(){    //重载基类的虚函数
            base.m();                //通过“base.”调用了基类的方法 m()
            Console.Write(x);    }   //输出子类的变量的 x 的值
    }
```

程序运行结果如下：

```
类 classA 的构造函数！     //调用基类的构造函数输出的结果
1                          //调用基类的方法 m()输出的结果
2                          //调用子类的方法 m()输出的结果
```

### 2．使用抽象类和抽象方法实现多态

（1）抽象类。定义类时，用关键字 abstract 修饰的类就是抽象类。其中可以直接定义抽象方法（同样用关键字 abstract 声明之）。抽象类中，也可以不是抽象方法（即有具体的实现）。但含有抽象方法的类一定是抽象类。但是抽象类不能被实例化。由于无法创建该类的实例，所以只能通过该类的派生（子）类来实现它的方法，除非它的子类也是一个抽象类。

（2）抽象方法。定义方法时，通过关键字 abstract 声明抽象方法。抽象方法是一个没有实现的方法。其语法格式为：

```
访问修饰符 abstract 返回类型 方法名称();
```

【说明】抽象方法是没有实现的方法，因此，没有方法体，也就没有配对的大括号（{}），之后直接跟一个分号（;）结尾。

（3）抽象方法的实现与多态的实现。抽象方法必须在其子类中实现（除非它的子类也是一个抽象类）。在子类中通过关键字 override 重写抽象方法。其语法格式如下：

```
[访问修饰符] override 返回类型 方法名()
{
  方法体
}
```

【例 3.16】 抽象类及其抽象方法的定义与实现实例解析。

源代码如下：

```
namespace ConsoleApp0316
{   abstract class Vehicle              //定义抽象类（交通工具）
    {
        private string name;            //定义基类的字段
        public string Name
        {   get { return name; }
            set { name = value; }
        }
        public Vehicle(string name) { this.Name = name; }
        public abstract void run();     //定义类的抽象方法
    }
    class Train:Vehicle                 //定义抽象类的子类（火车类）
    {
        private string name;
        //子类中通过:base(name)传递数据 name 到基类 Vehicle 的 Name 属性
        public Train(string name):base(name)
        {   this.Name = name;       //通过子类的 name 字段值修改基类的属性 Name 值
            Console.Write("{0},是交通工具.", name);
        }
        public override void run()      //实现基类 Vehicle 的抽象方法
        {   //子类 Train 中引用基类 Vehicle 的属性 Name 的值,并输出
            Console.WriteLine("{0},运行在铁路上。",this.Name);
        }
    }
    class Cars : Vehicle                //定义抽象类的子类（小车类）
    {   //子类中通过:base(name)传递数据到基类 Vehicle 的 Name 属性
        public Cars(string name):base(name)
        {   //通过子类的 name 字段值修改基类 Vehicle 的属性 Name 值
            this.Name = name;
            Console.Write("{0},是交通工具.", name);    }
        public override void run()      //实现基类 Vehicle 的抽象方法
        {   //子类 Cars 中引用基类 Vehicle 的属性 Name 的值,并输出
            Console.WriteLine("{0},运行在公路上。", this.Name);    }
    }
    class Program
    {   static void Main(string[] args)
        {   Train t = new Train("火车"); //定义子类 Train 的对象并实例化
            t.run();                    //调用子类 Train 的抽象方法 run()
            Cars c = new Cars("小车");  //定义子类 Cars 的对象并实例化
            c.run();                    //调用子类 Cars 的抽象方法 run()
            Console.ReadKey();   }      //暂停程序运行
    } // class Program
}
```

# 3.6 接口

在 C#语言中，接口是一种引用类数据类型。接口把定义和实现进行分离，使得程序依赖于抽象而非具体，不代表实际的操作。例如，USB 接口只是一个定义，但实现 USB 接口的设备很多。

接口只表达了一个规范，规定了一系列标准。接口成员可以是方法、属性、事件、索引器，但不包括字段、构造函数。所有接口成员必须是公有（public）类型。

理论上，接口是继承者的父类，因此，它可以作为实现类的引用，以封装类的复杂实现。接口只定义了实现该接口的类所需的成员，接口本身不提供它所定义的成员的实现。

### 1．接口的声明

声明接口须使用关键字 interface，其语法格式如下：

```
[访问修饰符] interface 接口名称 [：继承的接口列表]
{
   接口成员
}
```

访问修饰符包括 public、private、protected 等，默认为 public。接口名称与类名称的命名规则相同。为了便于区别，接口名称一般使用大写字母 I 开头。类、结构、接口也可以继承其他多个接口。

### 2．接口的实现

接口的成员通过类继承加以实现，其实现方式分显示实现和隐式实现。一个类只能继承一个基类，但可以继承多个接口，因此，继承接口的类中必须实现接口中所有属性、方法、所引器、事件。

**【例 3.17】** 接口与类应用实例解析。本实例演示了接口的定义、继承，接口方法的实现。

源代码如下：

```
namespace ConsoleApp0317
{
    public interface IStudent                    //定义接口,名称为 IStudent
    {   //private string sno、sname;接口中作此定义将产生语法性错误,接口中不能定义字段
        string Sno { get; set; }
        string Sname { get; set; }
        void show();                             //接口中,声明一个方法,无实现部分
    }
    class Program : IStudent                     //定义类,继承于接口 IStudent
    {
        string strNo="", strName="";             //定义类的成员
        public string Sno                        //实现类的属性
        {   get { return strNo; }
```

```
            set { strNo = value; }
        }
        public string Sname              //实现类的属性
        {   get { return strName; }
            set { strName = value; }
        }
        public void show(){              //接口的方法 show()在类中的实现
            Console.WriteLine("学号：{0};姓名：{1}", Sno, Sname);   }
        static void Main(string[] args)
        {
            Program t = new Program();    //定义类的对象
            IStudent s = t;               //使用派生类的对象实例化接口对象
            s.Sno = "201606024205";   s.Sname = "李俊";    s.show();
            Console.ReadKey();
        }
    }
}
```

程序运行结果如下：

```
学号：201606024205;  姓名：李俊
```

### 3．接口的继承

接口可以从一个接口继承，也可以从多个接口继承。接口从多个接口继承时，基接口名称之间用逗号隔开。

**【例 3.18】** 接口与类应用实例解析。本实例演示了接口的定义、继承，接口方法的实现。

源代码如下：

```
namespace ConsoleApp0318
{   interface IPeople                        //定义接口,名称为 IPeople
    {   string Name { get; set; }            //声明基接口的属性
        string Sex { get; set; }             //声明基接口的属性
        int Age { get; set; }                //声明基接口的属性
    }
    interface ITeacher : IPeople             //继承公共接口 IPeople
    {  void teacher();  }                    //声明继承的接口的方法
    interface IStudent : IPeople             //继承公共接口
    {  void student();  }                    //声明继承的接口的方法
    class Program : IPeople, ITeacher,IStudent   //继承多个接口
    {
        string name, sex; int age;           //定义类的字段
        public string Name                   //实现基接口 IPeople 的属性 Name
        {   get { return name; }
            set { name = value; }
        }
```

```
        public string Sex                    //实现基接口 IPeople 的属性 Sex
        {   get { return name; }
            set { sex = value; }
        }
        public int Age                       //实现基接口 IPeople 的属性 Age
        {  get { return age; }
            set { age = value; }
        }
        //在派生类中,实现基类接口中声明的方法
        public void teacher() {
            Console.WriteLine("教师：{0},性别：{1},年龄：{2}",Name,Sex,Age); }
        public void student() {
            Console.WriteLine("学生:{0},性别:{1},年龄:{2}", Name, Sex, Age); }
        static void Main(string[] args)
        {   Program p = new Program();//实例化类的对象
            ITeacher t = p;              //使用派生类的对象实例化接口对象 ITeacher
            t.Name = "吴俊"; t.Sex = "男"; t.Age = 45;
            t.teacher();
            IStudent s = p;              //使用派生类的对象实例化接口对象 IStudent
            s.Name = "李佳"; s.Sex = "女"; s.Age = 23;
            s.student();                 //调用接口的方法
            Console.ReadKey();
        }
    }
}
```

程序运行结果如下：

```
教师：吴俊,性别：男,年龄：45
学生：李佳,性别：女,年龄：23
```