

## 1.0 基础、标示符、常量、数据类型(enum 枚举,struct 结构体)、操作符、循环、数组

### 一、程序

- 现实生活中，程序是指完成某些事务的一种既定方法和过程，可以把程序看成是一系列动作执行过程的描述。
- 在计算机世界，程序是指令，即为了让计算机执行某些操作或解决某个问题而编写的一系列有序指令的集合
- **程序=数据结构**(指程序中的特定数据类型和数据组织形式)+**算法**(算法是指为达到某个目的所要执行的操作步骤)

### 二、标示符

- 标识符用来表示程序中的一个特定元素，如类名、方法名、变量名、项目名等等
- 标识符可以是字母、数字、下划线及“@”，但是必须以字母、下划线及@开头，不能以数字开头
- 标识符严格区分大小写
- 可以使用中文做为标识符，但不建议使用
- 不能使用关键字做标识符
- 对于类名和方法名的标识符，一般将每个单词的首字母大写，如 StudentInfo；字段及变量名首单词的首字母小写，其他单词首字母大写

### 三、常量

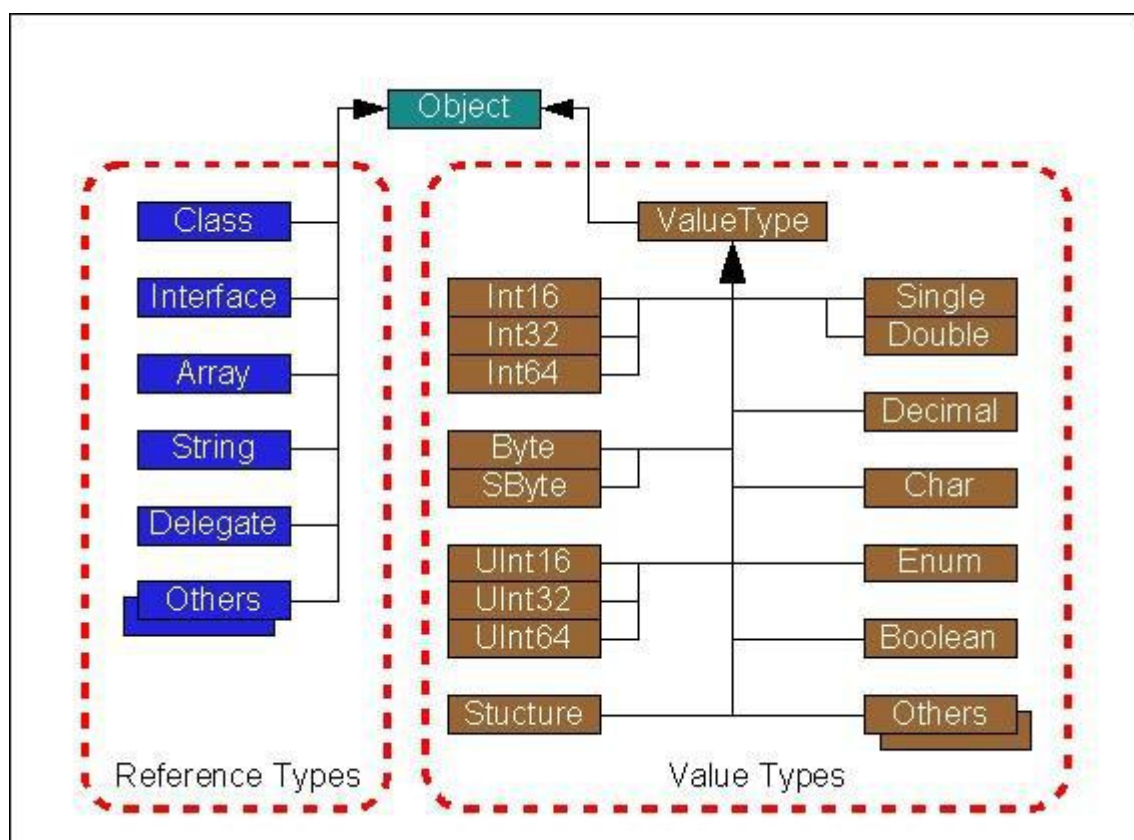
常量使用 **const** 关键字来定义,常量在声明时必须赋值,且以后不能再对其赋值 eg :const int a = 10;

## 四、数据类型

C#中的数据可以分为两大类：值类型（Value Type）和引用类型（Reference Type）。

值类型包括：结构体（数值类型，bool 型，用户定义的结构体），枚举，可空类型

引用类型包括：数组，接口，委托，类(用户自定义类、String 类、Object 类)



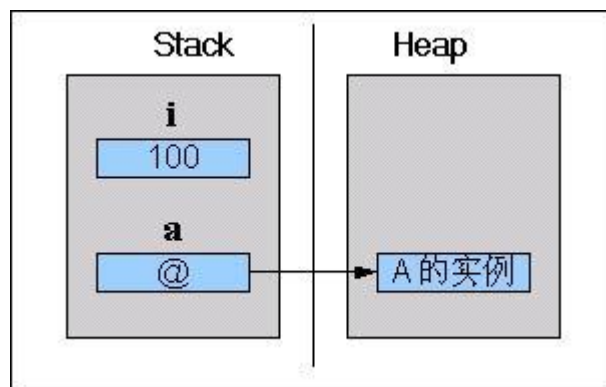
值类型的对象，继承自 `System.ValueType` 类（此类的对象却是引用类型的），内容直接储存在栈上（以及 MSDN 中提到的“或在结构中以内联方式分配的”——“or allocated inline in a structure”）；

引用类型的对象，其内容存储于堆上，栈上的相应变量名下只存储该处的堆地址，长度为4字节/32位（32位操作系统）或者8字节/64位（64位操作系统）。

值类型的对象不能为 null。不能从已有的值类型派生新的数据类型。

栈的特点:存放数据类型（值类型），其优点：比在堆上分配空间更快;用完之后立即自动释放;赋值方便（使用等号）

堆的特点:存放引用类型。



值类型的数据类型全部继承自 System.ValueType 类，它们又分为以下几类：

C#系统自定义了13种数值类型的数据类型：

整型 9种：byte, short, int(系统默认), long sbite, ushort, uint, ulong,char ----类型装换

Convert 类

| 名称   | 值范围                | 系统名称        | 属于 CLS | 占用内存长度 |
|------|--------------------|-------------|--------|--------|
| byte | 0至<br>$2^8-1(128)$ | System.Byte | Yes    | 1字节/8位 |

|        |  |               |     |         |
|--------|--|---------------|-----|---------|
| sbyte  | $-2^7$ 至 $2^7-1$                               | System.SByte  | No  | 1字节/8位  |
| ushort | 0至<br>$2^{16}-1$ (65535)                       | System.UInt16 | No  | 2字节/16位 |
| short  | $-2^{15}$ 至 $2^{15}-1$                         | System.Int16  | Yes | 2字节/16位 |
| char   | \0000至<br>\ffff                                | System.Char   | Yes | 2字节/16位 |
| uint   | 0至<br>$2^{32}-1$ (约<br>$4.29 \times 10^9$ )    | System.UInt32 | No  | 4字节/32位 |
| int    | $-2^{31}$ 至 $2^{31}-1$                         | System.Int32  | Yes | 4字节/32位 |
| ulong  | 0至<br>$2^{64}-1$ (约<br>$1.84 \times 10^{19}$ ) | System.UInt64 | No  | 8字节/64位 |
| long   | $-2^{63}$ 至 $2^{63}-1$                         | System.Int64  | Yes | 8字节/64位 |

浮点型3种：float, double(系统默认), decimal

| 名称           | 值范围  | 系统名称           | 属于<br>CLS | 精度         | 占用内存<br>长度    | 实例                |
|--------------|--|----------------|-----------|------------|---------------|-------------------|
| float(单精度)   | $\pm 1.5 \times 10^{-45}$ 至 $\pm 3.4 \times 10^{38}$   | System.Single  | Yes       | 7位         | 4字节<br>/32位   | float f = 3.14f   |
| double(双精度)  | $\pm 5.0 \times 10^{-324}$ 至 $\pm 1.7 \times 10^{308}$ | System.Double  | Yes       | 15至<br>16位 | 8字节<br>/64位   | double d = 2.14   |
| decimal(高精度) | $\pm 1.0 \times 10^{-28}$ 至 $\pm 7.9 \times 10^{28}$   | System.Decimal | Yes       | 28至<br>29位 | 16字节<br>/128位 | decimal d = 2.13M |

布尔型(逻辑)1种：bool

可能的取值为 true 和 false，占用内存长度1字节/8位。C#中不再有类似于“零等于 false，非零 int 值等于 true”的变换。bool 型的系统名称为 System.Boolean，属于 CLS。

其他2种：

enum 枚举

枚举的数据类型的声明：[属性(attributes)] [访问标识(modifiers)] enum 名称(identifier) [:

基类型(base-type)] {枚举列表(enumerator-list)} [;]

枚举类型的本意有两点：一是将难以记忆的整型数常量(除 char 外的8种整型数都可以)标记为

更好理解的变量名。二是引入强类型，在需要采用整形数常量表示的变量之间建立屏障。虽然每个字符串代表了一个整形数常量，但是具体使用枚举类数据类型的整型数常量时仍需要强制类型转换。这个整数值在 C# 默认是 int 型的。

枚举列表中各个变量的赋值默认是从0开始，递增1的。

例如：enum days:int {Sat, Sun, Mon}; 取值(int)days.Sun 为1 或 enum days:int {Sat=1, Sun, Mon}; 取值(int)days.Sun 为2

## struct 结构体

用户自定义的值类型数据类型，和类相似也有不同(有篇博客具体介绍过);本身没有继承这一特点，因此与继承相关的一切（例如抽象方法）结构体均不具备，除了两点：所有的结构体继承自 System.Object 类、结构体可以继承(实现)接口。

|                 | 类                                  | 结构体  |
|-----------------|------------------------------------|--|
| <b>类型</b>       | 引用类型                               | 值类型  |
| <b>实例化</b>      | 必须通过 new 实例化                       | 可以不 new，直接声明,但声明后必须赋值                              |
| <b>构造方法(函数)</b> | 方法不受限制(有、无参数均可，默认是无参)              | 方法必须带有参  |
| <b>继承</b>       | 自身继承 System.Object，可继承(包含接口)，也可被继承 | 除自身继承 System.ValueType 外，不可继承（除接口，通常叫做实现接口），也不可被继承 |
| <b>成员</b>       | 字段声明时可赋值(初始化)                      | 字段声明时不可赋值，方法一般公有                                   |

|                  |                          |  |
|------------------|--------------------------|--|
| 作参数<br>传递给<br>方法 | 传递的是引用(值一但修改所有指向改类值都已改变) | 传递的是实际值(值修改不会影响其它指向值，除非被 ref 修饰;ref 修饰后传递引用) |
|------------------|--------------------------|--|

性能介绍:结构是值类型，所以会影响性能，但根据使用结构的方式，这种影响可能是正面的，也可能是负面的。正面的影响是为结构分配内存时，速度非常快，因为它们将内联或者保存在堆栈中。在结构超出了作用域被删除时，速度也很快。另一方面，只要把结构作为参数来传递或者把一个结构赋给另一个结构(例如  $A=B$ ，其中  $A$  和  $B$  是结构)，结构的所有内容就被复制，而对于类，则只复制引用。这样，就会有性能损失，根据结构的大小，性能损失也不同。注意，结构主要用于小的数据结构。但当把结构作为参数传递给方法时，就应把它作为 ref 参数传递，以避免性能损失——此时只传递了结构在内存中的地址，这样传递速度就与在类中的传递速度一样快了。另一方面，如果这样做，就必须注意被调用的方法可以改变结构的值。

## 五、操作符

分类:

|          |                                       |
|----------|---------------------------------------|
| 赋值操作符:   | =                                     |
| 复合赋值运算符: | += , -= , *= , /= , %=                |
| 算数运算符:   | +, -, *, /, %(β二元运算符), ++, --(β一元运算符) |
| 条件运算符:   | >, <, >=, <=, !=, ==                  |
| 逻辑运算符:   | &&(遇假为假),   (同真为真), !                 |

|       |   |
|-------|---|
| 位运算符: | &, ,^,~,<<,>>   |
| 备注:   | 几乎所有的操作符都只能操作基本数据类型,但是"==" ,"==" 和"!=" ,<br>这些操作符能操作所有的对象。除此以外 String 类支持" +" 和" +=". |

优先级：

|                         |                               |
|-------------------------|-------------------------------|
| postfix operators       | [] . (params) expr++ expr--   |
| unary operators         | ++expr --expr +expr -expr ~ ! |
| creation or cast        | new (type)expr                |
| multiplicative          | * / %                         |
| additive                | + -                           |
| shift                   | << >> >>>                     |
| relational              | < > <= >= instanceof          |
| equality                | == !=                         |
| bitwise AND             | &                             |
| bitwise exclusive<br>OR | ^                             |
| bitwise inclusive OR    |                               |



|             |  |
|-------------|--|
| logical AND | &&                                     |
| logical OR  |  |
| conditional | ?                                      |
| assignment  | = += -= *= /= %= &= ^=  = <<= >>= >>>= |
| 备注          | 上面的操作数优先级从上到下依次降低，同一个单元格内的优先级相同        |

## 六、循环 注：[]表示可有可无

while (先判断后执行,)

用法:while(循环条件){循环体}

其它用法:while(变量名){ case 变量值1:[执行语句] break;—default:[执行语句] break;}

do-while (先执行一次再判断)

用法:do{循环体}while(循环条件);

for (和 while 类似,先判断)

用法:for([初始化循环变量];[循环条件];[修改循环变量值]){循环体}

foreach (遍历)

用法:foreach(类型 变量名 in 类型数据列表){}

扩展:continue(跳出本次循环进行下次循环)、break(跳出整个循环)、return(跳出整个方法,可有可无返回值)

## 七、数组

一维数组:类型[] 变量名 = new 类型[长度];

多维数组(矩形数组):类型[,] 变量名 = new 类型[外长度,内长度];--注:[]里的,等于几维减1

交错数组(数组的数组):类型[][] 变量名 = 类型[长度][];

### 2.0 面向对象 类与实例(关键字)、封装、继承、多态(虚方法,抽象类,抽象方法,接口)

**一切事物皆为对象，对象是可以看到、感觉到、听到、触摸到、尝到、或闻到的东西。对象是一个自包含的实体，用一组可识别的特性和行为来标识。面向对象编程 (Object-Oriented Programming): 其实就是针对对象来进行编程。**

**面向对象的语言必须满足三大特性：封装、继承、多态**

#### 一、类与实例

定义:具有相同属性和功能的对象的抽象的集合，是对象的一种表现形式;实例化就是创建对象的过程(new)，实例就是一个真实的对象。

修饰符和关键字:

| 分<br>类<br>名<br>称      | 修饰符/关键字        | 概述   | 实例使用 |
|-----------------------|----------------|--|------|
| 属<br>性<br>修<br>饰<br>符 | [Serializable] | 按值将对象封送到远程服务器。在按值封送对象时，就会创建一个该对象的副本，并将其 <a href="#">序列化</a> 传送到服务器。任何对该对象的方法调用都是在服务器上进行的。          |      |
|                       | [STAThread]    | 是 Single-Threaded Apartment 单 <a href="#">线程</a> 套间的意思，是一种线程模型（线程模式用于处理组件在 <a href="#">多线程</a> 的环境 |      |

里并行与并互的方式), 套间线程 (STAThread) 模式中接口跨线程传递必须被调度 (Marshal), 不调度直传肯定会失败! 而 MTA 或 FreeThread 模式中的接口可以不经调度直接传递。这种调度在特定的环境中非常影响性能 (可有几百倍之差)。如 VB 里只支持 STAThread 模式。FreeThread 模式的组件会在里面表现成和跨进程一样慢! 线程模式是微软的 COM 基础中的极其重要的概

|               |                 |  |  |
|---------------|-----------------|--|--|
|               |                 | 念。一定要吃透！                                       |  |
|               | [MTAThread<br>] | 是 MultiThreaded Apartment 多线程套间的意思，同上也是一种线程模型。 |  |
| 访问<br>修饰<br>符 | private         | 只有包含该成员类可以使用,作用于类成员。                           |  |
|               | protected       | 只有包含该成员类以及派生类可以存取，作用于类成员。                      |  |
|               | internal        | 只有当前工程(程序集)可以存取，作用于程序集及以内。                     |  |
|               | public          | 存取不受限制，任何地方都可以访                                |  |


|                       |                 |   |  |
|-----------------------|-----------------|---|--|
|                       |                 | 问， <a href="#">作用于程序集</a> 及以内。                                  |  |
| 类<br>修<br>饰<br>符      | <b>abstract</b> | <a href="#">抽象类</a> 。指示一个类只能作为其它类的基类。                           |  |
|                       | <b>sealed</b>   | <a href="#">密封类</a> 。指示一个类不能被继承。理所当然，密封类不能同时又是抽象类，因为抽象总是希望被继承的。 |  |
| 成<br>员<br>修<br>饰<br>符 | <b>abstract</b> | 指示该方法或属性没有实现。   |  |
|                       | <b>sealed</b>   | 密封方法。可以防止在派生类中对该方法的 override（重载）。不是类的                           |  |

|  |                 |   |  |
|--|-----------------|---|--|
|  |                 | <p>每个成员方法都可以作为密封方法密封方法，必须对基类的虚方法进行重载，提供具体的实现方法。所以，在方法的声明中，sealed 修饰符总是和 override 修饰符同时使用。</p> |  |
|  | <b>delegate</b> | <p>委托。用来定义一个<a href="#">函数指针</a>。C#中的<a href="#">事件驱动</a>是基于 delegate + event 的。</p>          |  |
|  | <b>event</b>    | <p>声明一个事件。</p>  |  |
|  | <b>extern</b>   | <p>指示方法在外部实现。</p>   |  |
|  | <b>override</b> | <p>重写。对由基类继承成员的新实现。</p>   |  |
|  | <b>readonly</b> | <p>指示一个域只能在</p>   |  |

|  |                |  |  |
|--|----------------|--|--|
|  |                | 声明时以及相同类的内部被赋值。  |  |
|  | <b>static</b>  | 指示一个成员属于类型本身，而不是属于特定的对象。即在定义后可不经实例化，就可使用。                            |  |
|  | <b>virtual</b> | 指示一个方法或存取器的实现可以在继承类中被覆盖。   |  |
|  | <b>new</b>     | 在派生类中隐藏指定的基类成员，从而实现重写的功能。若要隐藏继承类的成员，请使用相同名称在派生类中声明该成员，并用 new 修饰符修饰它。 |  |
|  | <b>const</b>   | 指定该成员的值只读不允许修改。  |  |



|       |               |   |   |
|-------|---------------|---|---|
| 其他关键字 | <b>this</b>   | 关键字只能在方法内使用，包括构造方法，用来指代当前类本身的对象，关键字只能访问实例成员                 |   |
|       | <b>base</b>   | base 关键字的中文意思是超级的，使用 base 关键字可以在子类中引用父类部分的内容。               |   |
|       | <b>typeof</b> | 获取某一类型的 System.Type 对象，不能重载 typeof 运算符。<br>eg : typeof(int) |  <pre>using System;  namespace OperatorTest{      public class OperatorTestClass{          public static void Main(string[] args){              Console.WriteLine(typeof(int));              Console.WriteLine(typeof(Int32));              Console.WriteLine(typeof(string));              Console.WriteLine(typeof(double[]));</pre> |

|  |    |  |  |
|--|----|--|--|
|  |    |  | <pre>         }      }  } </pre>  |
|  | as | 通过引用转换或装箱转换将对象转换成引用指定类型，若失败则返回 NULL                  | <pre> Object ob1 = "My";  Object ob2 = 5; string a = ob1 as string; 成功 a 为"My"，失败 a 为 NULL </pre>                  |
|  | is | 动态的检查运行时对象类型是否和给定的类型兼容,它返回一个布尔值，表示能否通过引用转换、装箱转换或拆箱转换 | <pre> if(1 is int){} </pre>  |

类的声明：[修饰符] class 类名{

类的实例化：类名 实例名 = new 类名() ---实例化时调用默认无参构造方法，若编写了构造方法默认无参构造方法将失效

构造方法默认无参构造方法将失效

构造方法(构造函数)：**就是对类进行初始化,构造方法与类同名,无返回值(也不需要 void), 在 new 时候调用.**声明: [修饰符] 类名(参数列表){}

方法重载：创建同名的多个方法，但参数类型或个数(参数列表)不同，提供了函数的扩充能力。

字段和属性：属性是一个方法或一对方法(get、set),适合于**以私有字段的方式使用**方法调用的场合。

## 二、封装

定义:封装是一种把代码和代码所操作的数据捆绑在一起，使这两者不受外界干扰和误用的机制;可理解为一种用做保护的包装器，以防止代码和数据被包装器外部所定义的其他代码任意访问。

说明:1.良好的封装能够**减少耦合**2.类内部的实现**可以自由地修改**3.类具有**清晰的对外接口**.

委托和事件：---**委托**是对函数的封装,可以当作给方法的特征指定一个名称;**事件**是委托的一种特殊形式,当发生有意义的事情时,事件对象处理通知过程.(有篇博客深入研究)

封箱和拆箱:实际就是引用类型和 object 类型(引用类型)之间相互转换;封箱实例:object obj = "str" //隐式封箱;拆箱实例: String s = (String)obj //显示拆箱;

属性和索引器:

| 属性 | 索引器 |
|----|-----|
|----|-----|

|  |   |
|--|---|
| 访问修饰符 返回类型 属性名{get{//<br>取值代码 return 属性值}set{ //赋值<br>代码}} | 访问修饰符 返回类型 this[索引列表]{ get{return<br>属性值;} set{}} |
| 定义:有效地封装数据,方便地操作数据   | 定义:是一种特殊的类成员,它能够让对象以类似数组的方式来存取,使程序看起来更为直观,更容易编写   |
| 允许调用方法,如同公共数据成员  | 允许调用对象上的方法,如同对象是一个数组                              |
| 可通过简单的名称进行访问   | 可通过索引器进行访问  |
| 可以为静态成员或实例成员   | 必须为实例成员   |
| 其 get 访问器没有参数  | 其 get 访问器具有与索引器相同的形参表                             |
| 其 set 访问器包含隐式 value 参数                                     | 除了 value 参数外,其 set 访问器还具有与索引器相同的形参表               |

### 三、继承 -----被继承者:父类,超类,基类;继承者:子类,派生类

定义:继承是指一个对象从另一个对象中获得属性的过程,继承与封装可以互相作用,是一种 类与类之间强耦合的关系。

优点:1.使得所有子类公共的部分都放在了父类,使得代码得到了共享,避免了重复(子类拥有父类 非 private 的属性和功能)。2.使得 修改或扩展 继承而来的实现 都较为容易(方法重写等)。

缺点:1.父类变,则子类不得不变;2.会破坏包装,父类实现细节暴露给了子类。增大了两


个类之间的耦合性。

注意事项:1.子类构造方法的书写：在子类的构造方法内部必须调用父类的构造方法，为了方便程序员进行开发，如果在子类内部不书写调用父类构造方法的代码时，则子类构造方法将自动调用父类的默认构造方法。而如果父类不存在默认构造方法时，则必须在子类内部使用 `base` 关键字手动调用。子类构造方法的参数列表和父类构造方法的参数列表不必完全相同;2.子类的构造过程：在构造子类时由于需要父类的构造方法，所以实际构造子类的过程就显得比较复杂了。

#### 四、多态-----分为静态多态(方法重载)和动态多态(建立在继承和方法重写基础上)

定义: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。

`virtual`(虚)方法: 被 `virtual` 修饰的是一个可被覆盖的基类方法，在派生类中用 `override` 覆盖该方法。( 不能为 `private` 方法 )



```
public class Animal
{
    ...

    public virtual void EatFood()
    {
```

```
        Console.WriteLine( "animal 在吃食物" );
    }
    ...
}

public class Cow : Animal
{
    ...

    public override void EatFood()
    {
        Console.WriteLine( "Cow 在吃食物" );
    }
    ...
}

class Program{...

    static void Main(string[] args)
    {...

        Animal animal1;

        animal1 = new Cow();

        animal1.EatFood();

    ...}

}

-----输出: Cow 在吃食物
```



抽象类:是特殊的类(对象)，只是不能被实例化，但可以包括抽象方法，这是普通类所不能的。

抽象类可以派生自抽象类(只可继承一个)，可以覆盖基类的抽象方法也可以不覆盖，如果不覆盖，则其派生类必须覆盖它们,必须用 **abstract** 修饰，不能用密封(sealed)来修饰。

抽象方法:是一个没有方法体的方法(行为),但不能实现，只能声明于抽象类中，派生类必须覆盖(实现)它们,必须用 **abstract** 修饰,本身是虚方法，不能在修饰。



```
class MyClass1
{
    public void Method1()
    {
        Console.WriteLine("wo shi yi ban de lei");
    }
}

abstract class MyAbs : MyClass1
{
    public abstract void AbMethod1();
}

class MyClass : MyAbs
```

```
{

    public override void AbMethod1()

    {

        Console.WriteLine("wo shi bei ji cheng hou chong xie de lei");

    }

}

class MyClient

{

    public static void Main()

    {

        MyClass mc = new MyClass();

        mc.Method1();

        mc.AbMethod1();

    }

}
```



接口:是单个或一组行为(属性、方法、事件、索引器),只是不能被实例化,不能加访问修饰符,所有成员都是抽象(虚)的,可以继承多个接口





```
interface IInterface1

{

    void Method1();

}

interface IInterface:IInterface1

{

}

abstract class MyAbs : IInterface

{

    public void Method1()

    {

        Console.WriteLine("wo shi xian le bei jie kou ji cheng de jie kou");

    }

}

class MyClass : MyAbs

{

}

class MyClient

{
```

```
public static void Main()

{

    MyClass mc = new MyClass();

    mc.Method1();

}

}
```



抽象类继承接口时必须实现接口，可处理为 virtual 方法和抽象方法



```
public abstract class AbsTest

{

    public virtual void Test()

    {

        Debug.WriteLine("Test");

    }

    public abstract void NewTest();

}

public interface ITest
```

```
{  
  
    void Test();  
  
    void NewTest();  
  
}
```



### 3.0 面向对象 委托和事件 异常和错误

#### 一、委托和事件

委托和事件这两个概念是完全配合的。委托仅仅是函数指针，那就是说，它能够引用函数，通过传递地址的机制完成。委托是一个类，当你对它实例化时，要提供一个引用函数，将其作为它构造函数的参数。事件则是委托的一种表现形式。

委托的声明:[修饰符] `delegate` 返回类型 委托名(参数列表);

简单的委托



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

```
using System.Text;

namespace TestApp
{
    /// <summary>

    /// 委托

    /// </summary>

    /// <param name="s1"></param>

    /// <param name="s2"></param>

    /// <returns></returns>

    public delegate string ProcessDelegate(string s1, string s2);

    class Program
    {
        static void Main(string[] args)
        {
            /* 调用方法 */          ProcessDelegate pd = new ProcessDelegate(new
Test().Process);

            Console.WriteLine(pd("Text1", "Text2"));
        }
    }
}
```

```
    }

}

public class Test

{

    /// <summary>

    /// 方法

    /// </summary>

    /// <param name="s1"> </param>

    /// <param name="s2"> </param>

    /// <returns> </returns>

    public string Process(string s1, string s2)

    {

        return s1 + s2;

    }

}
```



## 泛型委托



```
using System;

using System.Collections.Generic;

using System.Text;

namespace TestApp
{
    /// <summary>

    /// 委托

    /// </summary>

    /// <param name="s1"></param>

    /// <param name="s2"></param>

    /// <returns></returns>

    public delegate string ProcessDelegate<T,S>(T s1, S s2);

    class Program
    {
```

```
static void Main(string[] args)

{

    /* 调用方法 */           ProcessDelegate<string,int> pd = new

ProcessDelegate<string,int>(new Test().Process);

    Console.WriteLine(pd("Text1", 100));

}

}

public class Test

{

    /// <summary>

    /// 方法

    /// </summary>

    /// <param name="s1"> </param>

    /// <param name="s2"> </param>

    /// <returns> </returns>

    public string Process(string s1,int s2)

    {

        return s1 + s2;

    }

}
```

```
    }  
  
    }  
  
}
```



委托的回调方法



```
using System;  
  
using System.Collections.Generic;  
  
using System.Text;  
  
namespace TestApp  
{  
  
    /// <summary>  
  
    /// 委托  
  
    /// </summary>  
  
    /// <param name="s1"></param>  
  
    /// <param name="s2"></param>
```



```
/// <returns> </returns>
```

```
public delegate string ProcessDelegate(string s1, string s2);
```

```
class Program
```

```
{
```

```
    static void Main(string[] args)
```

```
    {
```

```
        /* 调用方法 */        Test t = new Test();
```

```
        string r1 = t.Process("Text1", "Text2", new ProcessDelegate(t.Process1));
```

```
        string r2 = t.Process("Text1", "Text2", new ProcessDelegate(t.Process2));
```

```
        string r3 = t.Process("Text1", "Text2", new ProcessDelegate(t.Process3));
```

```
        Console.WriteLine(r1);
```

```
        Console.WriteLine(r2);
```

```
        Console.WriteLine(r3);
```

```
    }
```

```
}
```

```
public class Test
```

```
{
```

```
    public string Process(string s1, string s2, ProcessDelegate process)
```

```
{  
  
    return process(s1, s2);  
  
}  
  
public string Process1(string s1, string s2)  
  
{  
  
    return s1 + s2;  
  
}  
  
public string Process2(string s1, string s2)  
  
{  
  
    return s1 + Environment.NewLine + s2;  
  
}  
  
public string Process3(string s1, string s2)  
  
{  
  
    return s2 + s1;  
  
}  
  
}  
  
}
```



事件的声明:[修饰符] **event** 委托名 事件名;

**事件应该由事件发布者触发，而不应该由客户端（客户程序）来触发。**注意这里术语的变化，当我们单独谈论事件，我们说发布者(publisher)、订阅者(subscriber)、客户端(client)。当我们讨论 Observer 模式，我们说主题(subject)和观察者(observer)。客户端通常是包含 Main()方法的 Program 类。

1.为什么要使用事件而不是委托变量？（摘自 <http://kb.cnblogs.com/page/45756/>）



```
using System;

using System.Collections.Generic;

using System.Text;

// 实例：为什么使用事件而不是委托变量

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Publishser pub = new Publishser();
```

```
Subscriber sub = new Subscriber();

pub.NumberChanged += new
NumberChangedEventHandler(sub.OnNumberChanged);

pub.DoSomething();           // 应该这样触发事件

pub.NumberChanged("使用委托");    // 但是被这样调用了,对委托变量的恰
当使用

    }
}

// 定义委托

public delegate void NumberChangedEventHandler(string Name);

// 定义事件发布者

public class Publishser
{
    private string Name;

    public NumberChangedEventHandler NumberChanged;    // 声明委
托变量

    //public event NumberChangedEventHandler NumberChanged;    // 声明一个事
```

件

```
public void DoSomething()
{
    // 在这里完成一些工作 ...

    if (NumberChanged != null)
    {    // 触发事件

        NumberChanged(Name);
    }
}

// 定义事件订阅者

public class Subscriber
{
    public void OnNumberChanged(string Name)
```

```
{  
  
    Console.WriteLine("{0}已响应", Name);  
  
}  
  
}
```



## 2.如何让事件只允许一个客户订阅？(事件访问器)



```
using System;  
  
using System.Collections.Generic;  
  
using System.Text;  
  
// 实例：让事件只允许一个客户订阅  
  
namespace ConsoleApp  
{  
  
    class Program3  
    {  
  
        static void Main(string[] args)
```

```
{

    Publishser pub = new Publishser();

    Subscriber1 sub1 = new Subscriber1();

    Subscriber2 sub2 = new Subscriber2();


    pub.NumberChanged -= sub1.OnNumberChanged;    // 不会有任何反应

    pub.NumberChanged += sub2.OnNumberChanged;    // 注册了 sub2

    pub.NumberChanged += sub1.OnNumberChanged;    // sub1将 sub2的覆盖
掉了

    pub.DoSomething();        // 触发事件

}

}

// 定义委托

public delegate string GeneralEventHandler();

// 定义事件发布者

public class Publishser
```

```
{

    // 声明一个委托变量或事件都无所谓

    private GeneralEventHandler numberChanged;

    // 事件访问器的定义

    public event GeneralEventHandler NumberChanged
    {
        add
        {
            numberChanged = value; //只允许注册一个事件，重复注册则替换前者
        }

        remove
        {
            numberChanged -= value;
        }
    }

    public void DoSomething()
    {
```



```
// 做某些其他的事情

if (numberChanged != null)

{    // 触发事件

    string rtn = numberChanged();

    Console.WriteLine("Return: {0}", rtn);    // 打印返回的字符串

}

}

}

// 定义事件订阅者

public class Subscriber1

{

    public string OnNumberChanged()

    {

        Console.WriteLine("Subscriber1 Invoked!");

        return "Subscriber1";

    }

}

public class Subscriber2
```

```
{  
  
    public string OnNumberChanged()  
  
    {  
  
        Console.WriteLine("Subscriber2 Invoked!");  
  
        return "Subscriber2";  
  
    }  
  
}
```



### 3.处理异常和订阅者方法超时的处理



```
using System;  
  
using System.Collections.Generic;  
  
using System.Text;  
  
using System.Threading;  
  
using System.Runtime.Remoting.Messaging;  
  
using System.IO;
```

```
// 处理异常

namespace ConsoleApp {

    class Program6 {

        static void Main(string[] args) {

            Publisher pub = new Publisher();

            Subscriber1 sub1 = new Subscriber1();

            Subscriber2 sub2 = new Subscriber2();

            Subscriber3 sub3 = new Subscriber3();

            pub.MyEvent += new EventHandler(sub1.OnEvent);

            pub.MyEvent += new EventHandler(sub2.OnEvent);

            pub.MyEvent += new EventHandler(sub3.OnEvent);

            pub.DoSomething();           // 触发事件

            Console.WriteLine("Control back to client!\n");    // 返回控制权

            Console.WriteLine("Press any thing to exit...\n");

            Console.ReadKey();           // 暂停客户程序，提供时间供订阅者完成方法
```

```
    }  
  
}  
  
public class Publisher {  
  
    public event EventHandler MyEvent;  
  
    public void DoSomething() {  
  
        // 做某些其他的事情  
  
        Console.WriteLine("DoSomething invoked!");  
  
        if (MyEvent != null) {  
  
            Delegate[] delArray = MyEvent.GetInvocationList();  
  
            foreach (Delegate del in delArray) {  
  
                EventHandler method = (EventHandler)del;  
  
                method.BeginInvoke(null, EventArgs.Empty, null, null);  
  
            }  
  
        }  
  
    }  
  
}  
  
public class Subscriber1 {
```

```
public void OnEvent(object sender, EventArgs e) {

    Thread.Sleep(TimeSpan.FromSeconds(3));    // 模拟耗时三秒才能完成方法

    Console.WriteLine("Waited for 3 seconds, subscriber1 invoked!");

}

}

public class Subscriber2 {

    public void OnEvent(object sender, EventArgs e) {

        throw new Exception("Subscriber2 Failed");    // 即使抛出异常也不会影响到客

        //Console.WriteLine("Subscriber2 immediately Invoked!");

    }

}

public class Subscriber3 {

    public void OnEvent(object sender, EventArgs e) {

        Thread.Sleep(TimeSpan.FromSeconds(2));    // 模拟耗时两秒才能完成方法

        Console.WriteLine("Waited for 2 seconds, subscriber3 invoked!");

    }

}
```

客户端

```
}
```



#### 4.委托和方法的异步调用



```
using System;

using System.Collections.Generic;

using System.Text;

using System.Threading;

using System.Runtime.Remoting.Messaging;

namespace ConsoleApp
{

    public delegate int AddDelegate(int x, int y);

    class Program9
    {

        static void Main(string[] args)
        {
```

```
Console.WriteLine("-----客户端执行开始，即将调用异步");

Thread.CurrentThread.Name = "客户端线程 Name";

Calculator cal = new Calculator();

AddDelegate del = new AddDelegate(cal.Add);

string data = "客户端数据."; //异步完成后挂载的方法返回的数据

AsyncCallback callBack = new AsyncCallback(OnAddComplete); //异步完成后挂
载方法

del.BeginInvoke(2, 5, callBack, data);           // 异步调用方法

// 做某些其它的事情，模拟需要执行3秒钟

for (int i = 1; i <= 3; i++)
{
    Thread.Sleep(TimeSpan.FromSeconds(i));

    Console.WriteLine("{0}: 模拟执行其他事情 {1} 秒钟.",
        Thread.CurrentThread.Name, i);
}

Console.WriteLine("\n-----客户端执行完毕...");
```

```
        Console.ReadKey();

    }

    static void OnAddComplete(IAsyncResult asyncResult)
    {
        AsyncResult result = (AsyncResult)asyncResult;
        AddDelegate del = (AddDelegate)result.AsyncDelegate;
        string data = (string)asyncResult.AsyncState;

        int rtn = del.EndInvoke(asyncResult);

        Console.WriteLine("{0}: 异步返回值, {1}; Data: {2}\n",
            Thread.CurrentThread.Name, rtn, data);
    }

}

public class Calculator
{
    public int Add(int x, int y)
    {
        if (Thread.CurrentThread.IsThreadPoolThread)
        {
```



```
        Thread.CurrentThread.Name = "异步线程 Name";

    }

    Console.WriteLine("-----异步开始!");

    // 执行某些事情，模拟需要执行2秒钟

    for (int i = 1; i <= 2; i++)

    {

        Thread.Sleep(TimeSpan.FromSeconds(i));

        Console.WriteLine("{0}: 模拟执行事情 {1} 秒钟.",

            Thread.CurrentThread.Name, i);

    }

    Console.WriteLine("-----异步结束!");

    return x + y;

}

}
```

## 二、异常和错误 (try-catch-finally)

基类: System.Exception

try 语句提供了一种机制来捕捉块执行过程中发生的异常。以下是它的三种可能的形式(s 可多个 catch) :

- try-catch(s)
- try-finally
- try-catch(s)-finally

try{申请资源，如数据库连接，网络连接，打开文件等可能出现异常的代码}

catch(异常类型 e){处理异常一类型的异常}

finally{释放资源}

手动跑出异常:throw new System.Exception();