**代码构造（程序结构）：**

引用头文件：

using System:

引入命名空间（本质上就是在命名空间自身内对一组类型（例如类）进行**分组**，便于成组管理类对象）：

namespace Code{}

类（既然命名空间是用来成组管理类的，那么其内部就应该是一个一个的类）即面向对象的对象：

class Class1{}

函数与代码块：

static void Main(string[] args)

{

Console.WriteLine("Hello world!");

Console.ReadKey();

}

因此，一个简单的C#程序最基本的情况就应该至少包含以上部分：

using System;

namespace Code

{

class Program

{

static void Main(string[] args)

{

//注释一

/\*注释二

。。。\*/

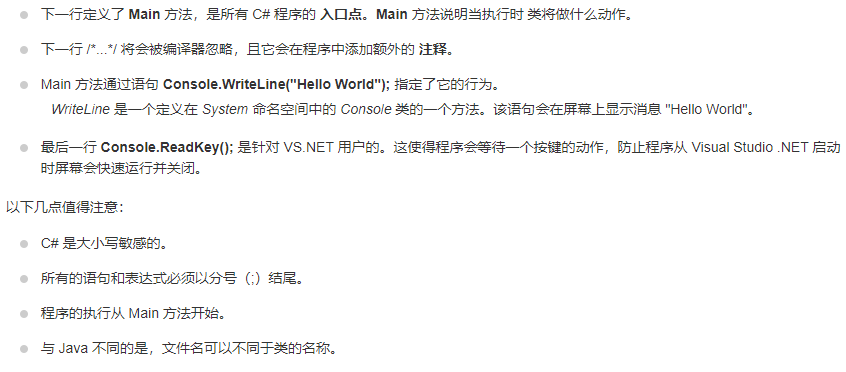
Console.WriteLine("Hello world!");

Console.ReadKey();

}

}



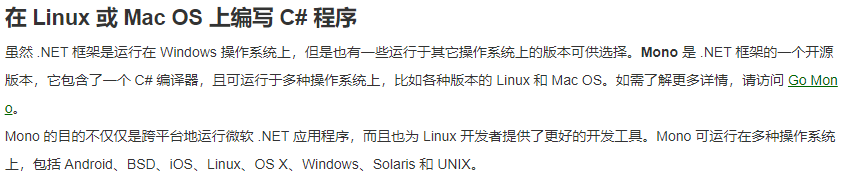


**C#环境：**

C#是.Net框架的一部分，且可用于编写.Net应用程序。

**.Net框架（.Net Framework）简介：**





**C#基本语法：**

C#是一种面向对象的编程语言。在面向对象的程序设计方法中，程序由各种相互交互的对象组成。相同种类的对象通常具有相同的类型，或者说，是在相同的class中。

例如，以Rectangle（矩形）对象为例。它具有length和width属性。根据设计，他可能需要接受这些属性值、计算面积和显示细节。下面以一个Rectangle类为例讨论C#的基本语法：

using System;

namespace Basic\_Syntax

{

class Program

{

static void Main(string[] args)

{

RectangleApplication.Rectangle r = new RectangleApplication.Rectangle();

r.Acceptdetails();

r.Display();

}

}

}

namespace RectangleApplication

{

class Rectangle

{

//成员变量

double length;

double width;

public void Acceptdetails()

{

length = 4.5;

width = 3.5;

}

public double GetArea()

{

return length \* width ;

}

public void Display()

{

Console.WriteLine("Length:{0}",length);

Console.WriteLine("Width:{0}",width);

Console.WriteLine("Area:{0}",GetArea());

}

}

}

**using关键字：**

在任何的C#应用程序中的第一条语句都是Using System; using关键字用于在程序中包含所需的命名空间。一个程序可以包含多个using语句。

**class关键字：**

用于声明一个类。

**C#中的注释：**

多行注释： 单行注释：

/\*这是 //这是一条注释

多行

注释\*/

**成员变量：**

变量是类的属性或数据成员，用于存储与类直接相关的数据，例如上面Rectangle类中有两个成员变量，分别为Length与Width。

**成员函数：**

函数是执行一系列指定任务的语句。类的成员函数是在类内声明的。例如上面Rectangle类中包含了三个成员函数：Acceptdetails、GetArea和Display。

**实例化一个类：**

在上面的程序中，类Program是一个包含Main方法和实例化Rectangle类的类。

**标识符：**

标识符是用来识别类、变量、函数或任何其他用户定义的项目。在C#中，类的命名必须遵循如下基本规则：

标识符必须以字母、下划线或@开头，后面可以跟一系列的字母、数字、下划线、@等。

标识符中的第一个字符不能是数字。

标识符必须不包含任何嵌入的空格或符号，比如? - + ! #等等。

标识符不能是C#的关键字，除非他们有一个@前缀，例如**@if**是有效的标识符，**if**就不是有效标识符。

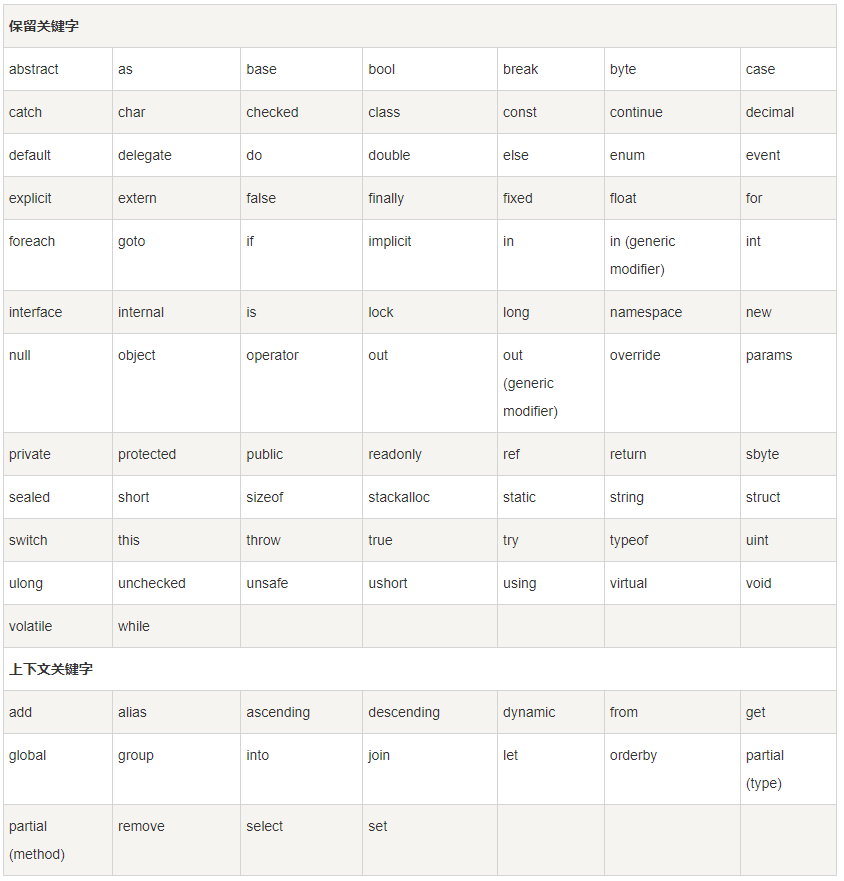
标识符必须区分大小写，大小学被认为是不同的字母。

标识符不能与C#的类库名称相同。

**C#关键字：**

关键字是C#编译器预定义的保留字，这些关键字不能被用作标识符，但是你若想使用这些关键字作为标识符，可以在关键字前添加@字符作为前缀。

在C#中，有些关键字在代码的上下文中有特殊意义，如get和set，这些被称为上下文关键字，见下表：



**C#数据类型：**

在C#中，变量分为以下几种类型：

值类型

引用类型

指针类型

**值类型：**

值类型变量可以直接分配给一个值，他们是从类System.ValueType中派生的。

值类型直接包含数据，比如int、char、float，他们分别存储数字、字符、浮点数。当声明一个int类型时，系统分配内存来存储值。下表中包含可用的值类型：



获取指定机器上一个类型或变量在特定平台上的准确尺寸，可以使用sizeof方法。如下，在我win64位操作系统下的int的size大小是4个**字节。**（8位为一字节，int的默认是int32，所以是4字节）（在32位操作系统下int的size大小同样是32位，所以同样是4字节）

Console.WriteLine("Size of int:{0}",sizeof(int));

out：Size of int：4

**引用类型：**

引用类型不包含存储在变量中的实际数据，但包含对变量的引用。换句话说，引用类型指的是一个**内存位置**。**一个数组标识符就可以被看做一个引用类型（可以理解为一个程序的启动快捷方式）。**使用多个变量时，引用类型可以指向一个内存位置。如果内存位置的数据是由一个变量改变的，其他变量会自动反映这种值的变化，一个很好的例子：



**堆空间：**

程序运行的开始，需要记录他的运行流程，和临时性数据。这个指挥程序怎么运行的地方必须很快，能一眼就找到需要的东西，就像桌面一样，不能放很多东西。

但是数组，代替你声明了变量，而就算你一次声明几百万个int类型，都是一句话的事情。但是这个几百万格数据不可能全部放在桌面上。于是规定不管数组创建了多少数据，一律放在堆空间中。堆空间相当于你的硬盘，随便存放所少数量的数据都无所谓。对于内存来说，这是专门放置数据的地方，不存指令。

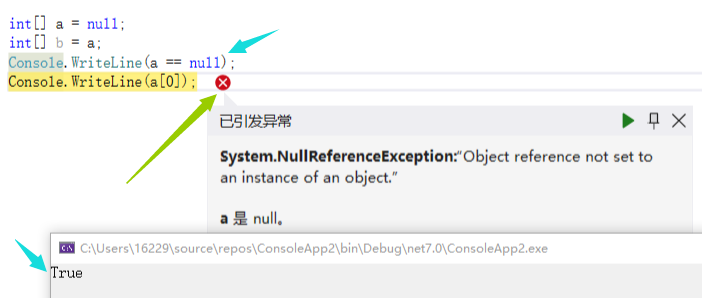
**引用类型：**

数组的建立如果是在堆中建立一堆的数据，那我们执行的赋值操作是在做什么呢？赋值时的的确确进行了一次赋值操作，而且这个值只有一个，并且这个值可以找到那几百万个数据。即一个内存地址。

首先创建好这个数组，并塞进数据，然后创建一个这个数组的快捷方式返回给你。这个快捷方式很小，就和一个普通数字一样大，可以直接放在桌面上。由于这种类型只是一个快捷方式，只存储地址，他的内容都是对别的东西的引用，所以被称为引用类型。

**引用类型的特殊值：**

对于所有的引用类型，都可以使用关键字null（空地址）进行赋值。此时你是可以访问他本身的，进行判断，进行赋值。但如果你访问他的内容，因为他没有引用东西，没有内容，此时的任何访问都是报错：

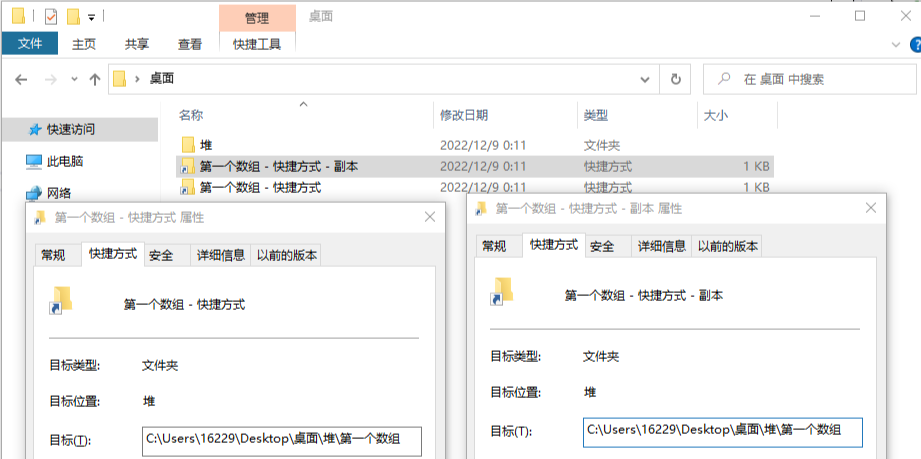


**引用类型的特点：**

**复制：**

最常见的复制就是赋值操作，这个过程会把值复制一份。

而这个引用类型就是个快捷方式，如果复制就会这样：



他们指向同一个文件夹，也就是说，使用其中任何一个快捷方式访问内容并修改后，使用另一个快捷方式访问到的内容也会被修改，因为他们访问的是同一个内容，这也就是前面数组发生窜号的情况的原因。

**只读：**

当你将一个引用类型设置为只读时（只读的作用是变量一经创建和赋值初始值后就不能更改），对引用类型的不许修改，是指不能再指向别的文件夹，可是文件夹没有只读，其中的文件也没有只读，因此一个只读的数组，其中的内容仍然可以被改变。

**比较：**

如果你要访问快捷方式的内容，相当于双击打开快捷方式，但是这个过程仅限于访问他的内容，而不比较他内部的内容，比较的就是他本身的值。而他本身只有一个地址，这意味着比较他们之比较他们是不是指向同一个文件夹，如果是两个文件夹，即使内部内容一模一样，二者也不相同：



**内置引用类型：**

object、dynamic和string。

**对象（object）类型：**

对象类型是C#通用类型系统中所有数据类型的终极基类。object是System.Object类的别名。所以对象（object）类型可以被分配任何其他类型（值类型、引用类型、预定义类型或用户自定义类型）的值。但是，在分配值之前，需要先进行类型转换。

当一个值类型转换为对象类型时，被称为**装箱**；另一方面，当一个对象类型转换为值类型时，被称为**拆箱。**

int i = 100;

object obj = i； //这是装箱

i = (int)obj; //这是拆箱

**动态（dynamic）类型：**

你可以存储任何类型的值在动态数据类型变量中，这些变量的检查是在运行时发生的。例如：

dynamic d = 20;

动态类型与对象类型相似，但是对象类型变量的类型检查是在编译时发生的，而动态类型变量的类型检查是在运行时发生的。

动态类型与var关键字也类似，区别也是在类型检查的时间不同，动态类型变量的类型检查时间是在运行时进行，而var是在赋值后就绑定了。

**字符串（string）类型：**

字符串（String）类型允许您给变量分配任何字符串值。字符串（String）类型是 System.String 类的别名。它是从对象（Object）类型派生的。字符串（String）类型的值可以通过两种形式进行分配：引号和 @引号。例如：

string str = "nba.com";

string str2 = @"nba.com";

C#中string字符串前可以加@将转义字符（\）当做普通字符对待，比如：

string str = @"C:\user";

string str2 = "C:\\user";

//二者是等价的

@字符串可以任意换行，换行符以及缩进空格等都计算在字符串长度之内：

string str = @"<script type=""text/javascript"">

<!--

-->

</script>";

Console.WriteLine(str.Lenth);

out:101(空格什么的都算在里边了)

**指针类型：**

指针类型变量存储另一种类型的内存地址。C#中的指针与C或C++指针具有相同的功能。声明指针类型的语法：

char\* cptr;

int\* iptr;

在C#中，不建议使用指针，原因在于指针不是安全类型，用完后需要手动释放分配的内存，若不注意容易出现内存泄露问题。C#与Java类似，都提倡使用封装好的轮子。

在后面章节“不安全的代码”中会讨论指针类型。

**C#类型转换：**

在C#中，类型转换是将一个数据类型转换为另一个数据类型的过程。在C#中，类型转换可以分为两种：隐式类型转换和显式类型转换（强制类型转换）。

**隐式类型转换：**

隐式类型转换是指将一个较小范围的数据类型转换为较大范围的数据类型时，编译器会自动完成类型转换，这些转换是C#默认的以安全方式进行的转换，不会导致数据丢失。

例如，从小的整数类型转换为大的数据类型，从派生类转换为基类。将一个byte类型的变量赋值给int类型的变量，编译器会自动完成转换（int-->long）而无需显式转换。

//隐式转换，不会丢失数据，合法

int a = 100;

long b = a;

**显式转换（强制类型转换）：**

显式转换是指将一个较大范围的数据类型转换为较小范围的数据类型时，或者将一个对象类型转换为另一个对象类型时，需要使用强制类型转换符号进行显式转换，强制类型转换可能会造成数据丢失。例如，将一个int类型变量赋值给byte类型的变量时，需要显示转换。实例：

//显式转换

//此时并没有发生数据丢失，但当i=256时，b就等于0了

int i = 10;

byte b = (byte)i;

//损失了小数部分

double doubleValue = 3.14;

int intValue = (int)doubleValue;

//输出为256989870，强制从int到float，数据可能损失精度

int intValue = 256989879;

float floatValue = (float)intValue;

Console.WriteLine(floatValue);

int intValue = 123;

string stringValue = intValue.ToString(); // 将 int 转换为字符串

//当转换为string时，有一些特殊情况：

float f = 53.005f;

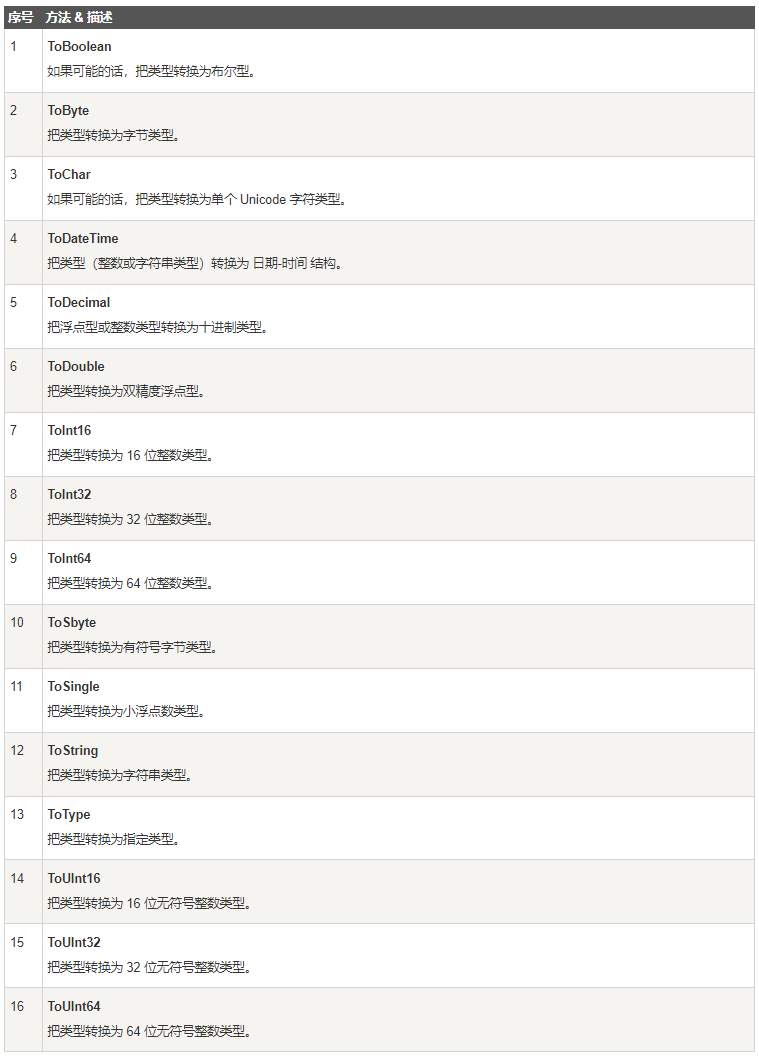
bool b = true;

out:

53.005(f当然被吞了)

True（不知道为啥但确实首字母变成大写了）

**C#类型内置转换方法（需要前面加Convert.）：**



例如：

int a = 100;

byte b = Convert.ToByte(a);

**注意：**

隐式转换只能将较小范围的数据类型转换为较大范围的数据类型，不能将较大范围的数据类型转换为较小范围的数据类型；

显式转换可能会导致数据丢失或精度降低，需要进行数据类型的兼容性检查；

对于对象类型的转换，需要进行类型转换的兼容性检查和类型转换的安全性检查。

**C#变量：**

一个变量只不过是一个供程序操作的存储区的名字。在C#中，每个变量都有一个特定的类型，类型决定了变量的内存大小和布局。当然，C#允许定义其他值类型的变量，比如enum，也允许定义引用类型变量，比如class。在此处只研究基本变量类型。

**C#中的变量定义方式：**

int i,j,k;

int i = 100; //定义时并初始化

int a,b,c = 10;

string a,b,c = "CSharp";

string a = "a",b="b",c="c";

//不合法的定义方式：

string a,b,c = "a","b","c";

**接收来自用户的值：**

System命令空间中的Console类提供了类函数ReadLine()，用于接收来自用户的输入，并把它存储到变量中：

int num;

num = Convert.ToInt32(Console.ReadLine());

函数Convert.ToInt32()把用户输入数据转换为int数据类型，原因在Console.ReadLine()只接受字符串格式的数据。

**C#中的左值与右值：**

Lvalue：左值，可以出现在赋值语句的左边或者右边

int g = 10; //变量g为左值

int a = g; //g在右边

g=10; //g在左边

Rvalue：右值，只能出现在赋值语句的右边

int a = 10； //常量10为右值，只能出现在赋值语句的右边

10 = 20； //这样就不行了

**C#常量：**

常量是固定值，程序执行期间不会改变。常量可以是任何基本数据类型，不如整数常数、浮点型常数、字符常数，以及枚举常量。

常量可以被当做常规的变量，只是他们的值在定义后不能被修改

**整数常量：**

整数常量可以是十进制、二进制或十六进制的常量。前缀表示基数：0x或0X表示十六进制，0b或0B表示二进制，**八进制的前缀0表示法在C#中已经被删除了**，没有前缀表示十进制。整数常量也可以有后缀，可以是U和L的组合。其中U代表unsigned，L代表long。后缀可以是大写或者小写，多个后缀可以以任意顺序进行组合。例子：

212 //合法

215u //合法

0xFeeL //合法，0x前缀表示十六进制，Fee是数据，L表示长整型

0b12 //非法，0b前缀表示二进制，但2不是八进制数字

032UU //非法，相同后缀不能重复

以下是各种类型的整数常量的实例：

85 //十进制

0x4b //十六进制

30 //int

30u //无符号int

30l //long

30ul //无符号long

**浮点常量：**

一个浮点常量是由整数部分、小数点、小数部分和指数部分组成。可以使用小数形式或者指数形式来表示浮点常量，实例：

.28 //合法，浮点形式是可以不要整数部分，默认会取0

3.14159 //合法

314159E-5L //合法

510E //非法，不完全指数（E后面没有指数的数值）

210f //不是浮点常量，没有小数或指数

.e55 //非法，缺少整数或小数

使用浮点形式表示时，必须包含小数点、指数或者同时包含二者。使用指数形式表示时，必须包含整数部分、小数部分或同时包含两者。有符号的指数使用e或E表示的。

**字符常量：**

字符常量是括在单引号中的如'x'，且可以存储在一个简单的字符类型变量中。一个字符常量可以是一个普通字符（'x'）、一个转义序列（'\t'）或者一个通用字符（'\u02C0'）。

在C#中，有一些特定的字符，当他们前面有反斜杠时有特殊意义，称为转义字符，如：



实例：

Console.WriteLine("hello\tWorld\n\n");

out:hello World

**字符串常量：**

字符串常量是括在双引号" "里或者括在@" "。字符串常量包含的字符与字符常量相似，可以是普通字符、转义字符和通用字符。使用字符串常量时，可以把一个很长的行拆分成多行，可以使用空格分隔各个部分，实例：

string a = "hello,world"; //hello,world

string b = @"hello, world"; //hello, world

string c = "hello\tworld"; //hello world

string d = @"hello\tworld"; //hello\tworld

string e = "Joe said \"hello\" to me"; //Joe said "hello" to me

string f = @"Joe said ""hello"" to me"; //Joe said "hello" to me //需要注意的是当我在@中使用双引号时，需要使用两次""""才行，不然他会以为是字符串的结束标志

string g = "\\\\server\\share\\file.txt"; //\\server\share\file.txt

string h = @"\\server\share\file.txt"; //\\server\share\file.txt

string i = "one\r\ntwo\r\nthree"; //等价于下面这个 //需要注意的是，虽然\r表示回车，但与其他语言不同，单独的\r在C#中并不会导致换行，他必须与\n一起使用

//才能创建一个新行，单独使用一个\r的话输出有点莫名其妙，所以为什么不直接使用\n算了？？

string j = @"one

two

three";

**定义常量：**

常量是用**const**关键字定义的，定义常量的语法：

const <data\_type> <constant\_name> = value;

实例：

const int c1 = 5;

const int c2 = c1 + 2; //定义完就确定了他的值为7了

int res = c2 \* 5;

Console.WriteLine(res);

out:35

//并不会像宏定义那样res=c1+2\*5=5+2\*5=15 //而是=35

//宏定义是直接替换

**C#运算符：**

运算符是一种告诉编译器执行特定的数字或逻辑操作的符号。在C#中有如下内置运算符：

算数运算符

关系运算符

逻辑运算符

位运算符

赋值运算符

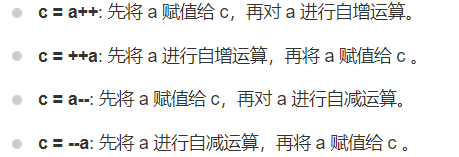
其他运算符

**算数运算符：**

下表中显示了C#支持的所有算数运算符。假设变量A的值为10，变量B的值为20，则有：



关于自增自减，请注意：



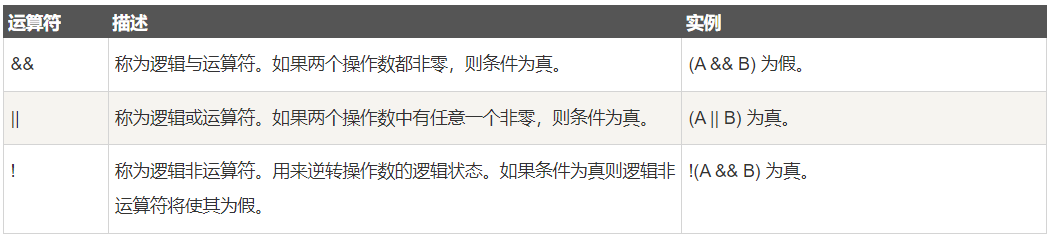
**关系运算符：**

下表显示了C#所有的关系运算符，假设变量A值为10，变量B值为20，则：



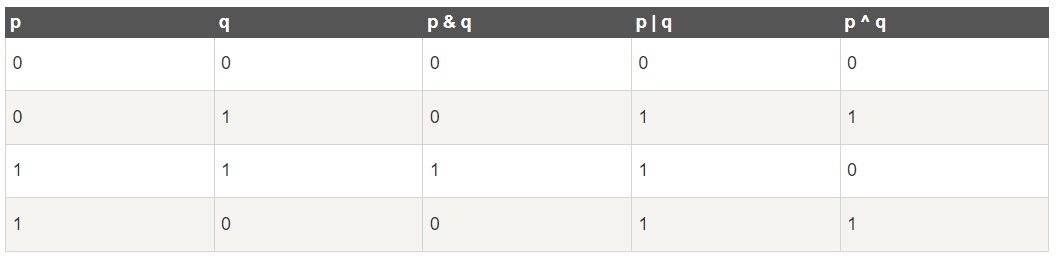
**逻辑运算符：**

下表显示了C#中所有的逻辑运算符。假设变量A为布尔值True，变量B为布尔值False，则：



**位运算符：**

位运算符作用于位，并逐位执行操作。&、|和^（即异或，相同取0，不同取1）的真值表如下所示：



假设A=60，B=13，现在以二进制表示，则：

A=0011 1100B=0000 1101

----------------------------------------

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001//异或运算符

~A = 1100 0011//按位取反运算符

下表列出了C#支持的位运算符。假设变量A的值为60，变量B的值为13，则：



**赋值运算符：**

需要注意一点，下面这种表达只是给b赋值了10，a还是没有被赋值：

int a,b=10;

下表列出了C#所有支持的赋值运算符：



**其他运算符：**

下表列出了C#支持的其他一些重要运算符：



注：typeof(class\_name)会返回namespace\_name.class\_name，几个例子：

using System;

namespace Code

{

class CodeClass1

{

static void Main(string[] args)

{

Console.WriteLine(typeof(CodeClass1)); //输出Code.CodeClass1

Type type = type(string);

Console.WriteLine(type); //输出System.string

Console.ReadKey();

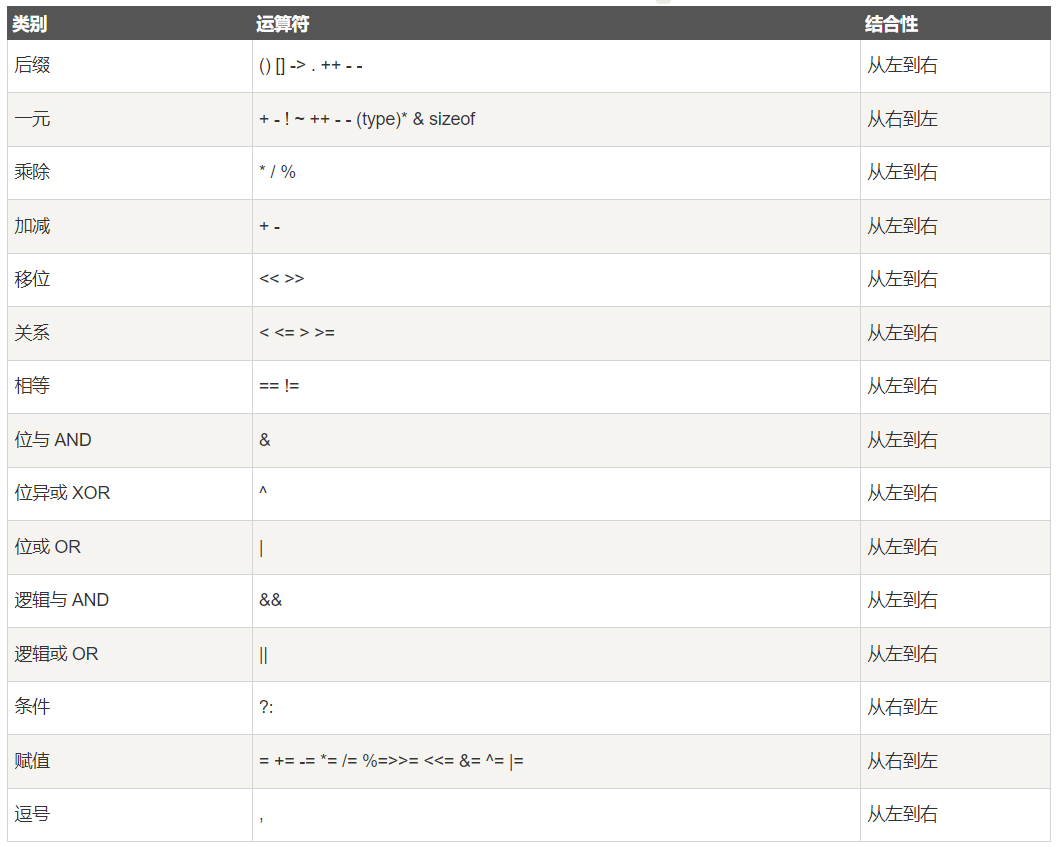
}

}

}

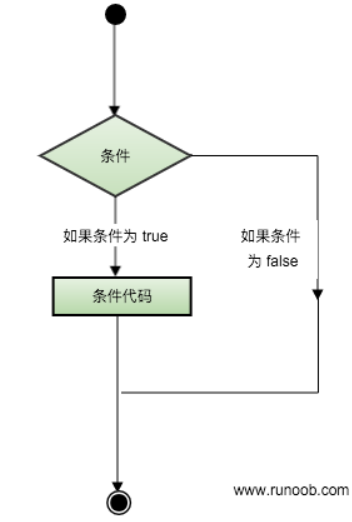
**C#中的运算符优先级：**

运算符的优先级确定表达式中项的组合。这会影响到一个表达式如何计算。某些运算符比其他运算符有更高的优先级，例如，乘除运算符具有比加减运算符更高的优先级。例如x=7+3\*2，此处x被赋值为13而非20，因为乘除的优先级更高，下表将按运算符优先级从高到低进行排列。



**C#判断：**

判断结构要求程序员指定一个或多个要评估或测试的条件，以及条件为真时要执行的语句（必需的）和条件为假时要执行的语句（可选的），大多数编程语言中典型的判断结构一般形式：



**判断语句：**

C#提供了以下类型的判断语句：



**Switch语句：**

一个Switch语句允许测试一个变量等于多个值的情况。每个值称为一个case，且被测试的变量会对每个switch case进行测试。

**语法：**

switch(expression){

case constant-expression :

statement(s);

break;

case constant-expression :

statement(s);

break;

//可以有任意数量的case语句

default：

statement(s);

break;

}

**实例：**

class Program

{

static void Main(string[] args)

{

char grade = Convert.ToChar(Console.ReadLine());

switch (grade)

{

case 'A': //此处意思就是A与B的情况共用case B的语句（若在case A中写了任何语句都必须加break）

case 'B': //So:要么全写（break也写），要么全不写（直接共用下一case中的语句）

Console.WriteLine("很棒");

break;

case 'C':

Console.WriteLine("不错");

break;

case 'D' :

Console.WriteLine("及格啦");

break;

default:

Console.WriteLine("错误输入");

break;

}

Console.WriteLine("您的成绩是{0}",grade);

}

}

int day = 2;

switch(day)

{

case 1:

Console.WriteLine("Monday");

break;

case 2:

Console.WriteLine("Tuesday");

break;

......

default:

Console.WriteLine("False data");

break;

}

**嵌套Switch语句：**

可以把switch作为一个外部switch的语句序列的一部分，即可以在一个switch语句中使用另一个switch语句。即便内部和外部switch的case常量包含共同的值，也没有矛盾。

switch(ch1) //外部判断ch1

{

case 'A':

Console.WriteLine("这个A是外部switch中的A");

switch(ch2) //内部判断ch2

{

case 'A':

Console.WriteLine("这个A是内部switch中的A");

break;

case 'B': //内部case B

}

break; //一条包含了语句的case必须在最后加上break，此处对应的是外部case A

case 'B' : //外部case B

default:

}

**？：判断符：**

可以用来替代简单的if...else语句，例如：

int a, b = 10;

a = 20;

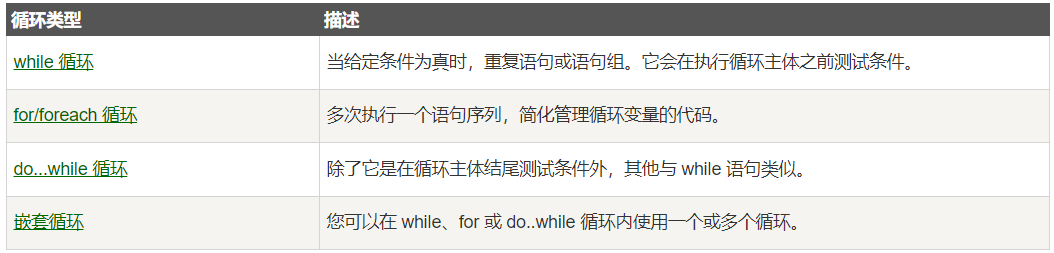
a = a == b ? 30 : 60;

Console.WriteLine(a);

out:60

**C#循环：**

有时需要多次执行同一代码，一般情况下，语句是顺序执行的。C#提供了允许更为复杂的执行路径的多种控制结构。循环语句允许我们多次执行一个语句或语句组。C#支持如下几种循环类型：



**循环控制语句：**

循环控制语句更改执行的正常序列。当执行离开一个范围时，所有在该范围内创建的对象都会被销毁。

**break：**

跳出当前循环，执行循环后的语句（整个循环都跳过了）

**continue：**

跳出本次循环，执行下一次循环（只是跳过了本次循环）

**无限循环：**

for(; ;)while(True)当条件表达式不存在时，默认为真

**注意事项：**

foreach常被用来遍历一个序列内的所有元素

foreach不能直接修改正在遍历序列中的元素，原因在于foreach循环在遍历过程中使用的是序列的副本而不是原始数据

**C#封装（public、internal等）：**

封装被定义为"把一个或多个项目封闭在一个物理的或者逻辑的包中"。在面向对象程序设计方法论中，封装是为了防止对实现细节的访问（我不想让你看的你别看）。

抽象和封装是面向对象程序设计的相关特性。抽象允许相关信息可视化，封装则使开发者实现所需级别的抽象。

C#中封装根据具体的需要，设置使用者的访问权限，通过**访问修饰符**来实现。访问修饰符定义了一个类成员的范围和可见性。C#支持的访问修饰符如下所示：

public：任何公有成员可以被外部的类访问。

private：只有同一个类中的函数可以访问它的私有成员。

protected：该类内部和继承类中可以访问。

internal：同一个程序集的对象可以访问。

protected internal：protected 和 internal 的并集，符合任意一条都可以访问。

一个例子：

A为父类，B为子类，C为妻子，D为私生子（D不在家中）。若我们给A的事件添加上修饰符，则有

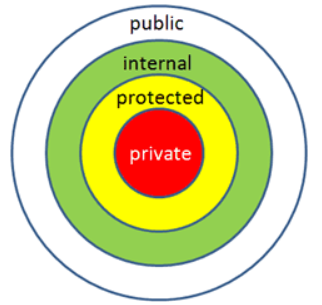
public事件，全地球都知道A的事

protected事件，A的事情只有ABD知道（A和他所有的儿子知道）

private事件，只有A自己知道（心事？秘密？）

internal事件，ABC都知道（A的家里人都知道，私生子不在家（该程序）中就不知道了）

protected internal事件，ABCD知道，其他人不知道



**Public访问修饰符：**

public修饰符允许一个类将其成员变量和成员函数暴露给其他的函数和对象。任何public成员都可以被外部的类访问。实例：

using System;

namespace fengZhuang

{

class Rectangle

{

public double length;

public double width;

public double GetArea()

{

return length \* width;

}

public void Display()

{

Console.WriteLine("长度： {0}",length);

Console.WriteLine("宽度： {0}",width);

Console.WriteLine("面积： {0}",GetArea());

}

}

class ExecuteRectangle

{

static void Main()

{

Rectangle r = new Rectangle();

r.length = 4.5; //在外部可以直接访问Rectangle class

中的length与width两个public变量与Display方法

r.width = 3.5;

r.Display();

}

}

}

在上例中，成员变量length与width被定义为public，所以他可以被其他类的Main()方法使用Rectangle类的实例r访问，成员函数Display和GetArea也能直接访问这些变量。同样的，成员函数DIsplay也被声明为public，所以在其他类中被实例r访问。

**private访问修饰符：**

private访问修饰符允许一个类将成员变量和成员函数对其他的函数和对象进行隐藏。只有在同一个类中的函数可以访问该类的私有对象。即使是类的实例也不能访问他的私有成员。实例：

namespace RectangleApplication

{

class Rectangle

{

private double length;

private double width;

public void AcceptDetails()

{

Console.WriteLine("请输入长度： ");

length = Convert.ToDouble(Console.ReadLine());

Console.WriteLine("请输入宽度： ");

width = Convert.ToDouble(Console.ReadLine());

}

public double GetArea()

{

return width \* length;

}

public void Display()

{

Console.WriteLine("长度： {0}",length);

Console.WriteLine("宽度： {0}",width);

Console.WriteLine("面积： {0}",GetArea());

}

}

class ExecuteRectangle

{

static void Main()

{

Rectangle r = new Rectangle();

r.AcceptDetails();

r.Display();

}

}

}

在上例中，成员变量width和length被定义为private，他们不能被其他类的任何函数访问。成员函数AcceptDetails与Display与两者同属一个类，因此可以访问。

由于成员函数AcceptDetails与DIsplay被设置为public，所以能在其它类中使用Rectangle类的实例r访问。

**protected访问修饰符：**

protected访问修饰符允许子类访问他的基类的成员变量和函数。这样有助于继承。在后续的继承章节详细讲解。

**internal访问修饰符：**

internal访问修饰符允许一个类将其成员变量和成员函数暴露给当前程序中的其他函数和对象。换句话说，带有internal访问修饰符的任何成员可以被定义在该成员所定义的应用程序内的任何类和方法访问。实例：

namespace RectangleApplication

{

class Rectangle

{

internal double length;

internal double width;

double GetArea()

{

return length\*width;

}

public void Display()

{

Console.WriteLine("长度： {0}",length);

Console.WriteLine("宽度： {0}",width);

Console.WriteLine("面积： {0}",GetArea());

}

}

class ExecuteRectangle

{

static void Main()

{

Rectangle r = new Rectangle();

r.length = 4.5;

r.width = 3.5;

r.Display();

}

}

}

在此类中的GetArea函数未定义任何访问修饰符，这种情况会默认使用private作为修饰符。而internal在大多数情况下与public是类似的，只有在多个程序交互的时候才会有区别。多个程序交互时，internal变量和函数在其他程序中是不能被访问的。

**protected internal访问修饰符：**

protected internal访问修饰符允许在本类、派生类或者包含该类的程序集中访问。这也被用作继承，在继承章节详细讲。

**C#方法：**

一个方法是把一些相关的语句组织在一起，用来执行一个任务的语句块。每个C#程序至少有一个带有Main（可能不是这个名字）方法的类。要使用一个方法，就得**定义方法**、**调用方法**。

**定义与调用方法：**

定义一个方法时，从根本上说就是在声明他的结构的元素，在C#中，定义方法的语法如下：

<Access Specifier> <Return Type> <Method Name>(Parameter List)

{

Method Body

}

Access Specifier：访问修饰符，这个决定了变量或方法对于另一个类的可见性。

Return type：返回类型，一个方法可以返回一个值。返回类型是方法返回的值的数据类型。如果方法不返回任何值，则返回类型为 void。

Method name：方法名称，是一个唯一的标识符，且是大小写敏感的。它不能与类中声明的其他标识符相同。

Parameter list：参数列表，使用圆括号括起来，该参数是用来传递和接收方法的数据。参数列表是指方法的参数类型、顺序和数量。参数是可选的，也就是说，一个方法可能不包含参数。

Method body：方法主体，包含了完成任务所需的指令集。

在下面的实例中，函数FindMax接收两个整数值，并返回两个中的较大值。修饰符为public，所以他可以使用类的实例在类的外部进行访问：

class NumberManipulator

{

public int FindMax(int a,int b)

{

return a>b?a:b;

}

}

调用一个方法时，直接使用方法名调用方法（如果有参数要求添加指定参数即可），实例调用刚写的FindMax方法：

int a = 10;

int b = 20;

int res = FindMax(a,b);

Console.WriteLine(res);

out:20

同样的，由于FindMax方法是public的，所以我们也可以在其他类中使用类的实例去调用方法。例如FindMax方法属于NumberManipulator类，我们可以在另一个类中实例化调用他：

class OtherClass

{

static void Main(string[] args)

{

NumberManipulator m = new NumberManipulator();

int a = m.FindMax(10,8);

}

}

**递归方法调用：**

一个方法自己调用自己即称为递归，下例中使用递归方法计算一个数的阶乘：

internal int factorial(int num)

{

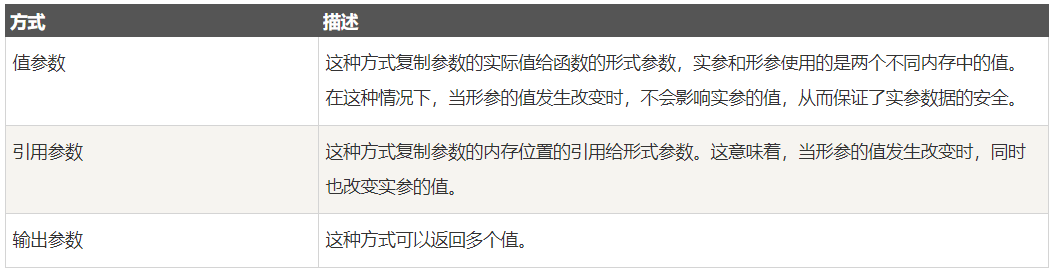
if(num==1) return 1;

else return num \* factorial(num-1);

}

**参数传递：**

当调用带有参数的方法时，需要向方法中传递参数。在C#中，有三种传递参数的方式：



**值参数传递：**

这是参数传递的默认情形。在此种方式下，当调用一个方法时，会为每一个参数创建一个**新的存储位置**。实际参数的值会复制给形参（函数名后的括号里的参数），实参和形参使用的是两个不同的内存中的值，所以当形参发生改变时，不影响实参的值，从而保证了实参的数据安全。实例：

using System;

namespace fangFa

{

class NumberManipulator

{

public void swap(int x,int y)

{

int temp;

temp = x;

x = y;

y = temp;

}

static void Main()

{

NumberManipulator n = new NumberManipulator();

int a = 100;

int b = 200;

Console.WriteLine("交换前a的值： {0}",a);

Console.WriteLine("交换前b的值： {0}",b);

n.swap(a,b);

Console.WriteLine("交换后a的值： {0}", a);

Console.WriteLine("交换后b的值： {0}", b);

}

}

}

out：

交换前a的值： 100

交换前b的值： 200

交换后a的值： 100

交换后b的值： 200

结果表明，通过值参数传递的方法，即使在函数内改变了实参a，b的值，实际上a，b也没有发生任何变化。

**引用参数传递：**

引用参数是一个对变量的**内存位置的引用**。当按引用传递参数时，与值参数不同的是，他不会为这些参数新建一个存储地址。引用参数表示与提供给方法的实际参数具有相同的内存位置。

在C#中，使用**ref**（reference）关键字声明引用参数，实例：

using System;

namespace fangFa

{

class NumberManipulator

{

public void swap(ref int x,ref int y)

{

int temp;

temp = x;

x = y;

y = temp;

}

static void Main()

{

NumberManipulator n = new NumberManipulator();

int a = 100;

int b = 200;

Console.WriteLine("交换前a的值： {0}",a);

Console.WriteLine("交换前b的值： {0}",b);

n.swap(ref a,ref b);

Console.WriteLine("交换后a的值： {0}", a);

Console.WriteLine("交换后b的值： {0}", b);

}

}

}

out：

交换前a的值： 100

交换前b的值： 200

交换后a的值： 200

交换后b的值： 100

从结果中不难发现，swap函数内的值改变了，并且同时在实参中的其他位置体现出来了，真真实实的改变了传递进去的参数的值。

**输出参数传递：**

return语句可用于只从函数中一个值。但是，可以使用输出参数(**out关键字**)从函数中返回两个或多个值。输出参数会把方法输出的数据赋值给自身，**其他方面与引用参数相似**，实例：

class NumberManipulator

{

public void getValue(out int x)

{

int temp = 5;

x = temp;

}

static void Main()

{

NumberManipulator n = new NumberManipulator();

int a = 100;

Console.WriteLine("在方法调用前a的值： {0}",a);

n.getValue(out a);

Console.WriteLine("在方法调用后a的值： {0}",a);

}

}

out:

在方法调用前a的值： 100

在方法调用后a的值： 5

提供给输出参数的变量不需要赋值。当需要从一个参数没有指定初始值的方法中返回值时，输出参数特别有用，下面的实例中很好的说明了这一点：

class NumberManipulator

{

public void getValue(out int x,out int y)

{

Console.WriteLine("请输入第一个值： ");

x = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("请输入第二个值： ");

y = Convert.ToInt32(Console.ReadLine());

}

static void Main()

{

NumberManipulator n = new NumberManipulator();

int a,b;

n.getValue(out a,out b);

Console.WriteLine("方法调用后a的值： {0}",a);

Console.WriteLine("方法调用后b的值： {0}",b);

}

}

注意：return与out是可以同时使用的（比如可以使用return一个bool值表示我们的操作是否成功）

输出参数传递具有**只出不进**的特点，即你无论在外部有没有初始化out参数的值，在传递输出参数的方法中你都必须对out参数进行赋值，并且方法完全不接受之前的初始化out参数的内容。

**C#可空类型：**

**单问号？与双问号？？**

单问号用于对int、double、bool等无法直接赋值为null的数据类型进行null的赋值，意思是这个数据类型是nullable类型的，例：

int? i = 3;

等价于

Nullable<int> i = new Nullable<int>(3);

**可空类型：**

C#提供了一个特殊的数据类型，Nullable类型（可空类型），可空类型可以表示在其基础数据类型的基础之上，再加上一个null值。例如Nullable<Int32>表示可空的Int32，可以被赋值为 -2,147,483,648 到 2,147,483,647 之间的任意值，也可以被赋值为 null 值。类似的，Nullable< bool > 变量可以被赋值为 true 或 false 或 null。

在处理数据库和其他包含可能未赋值的元素的数据类型时，将null赋值给数值类型或者bool型的功能特别有用，实例：

class NullablesAtShow

{

static void Main()

{

int? num1 = null;

int? num2 = 45;

double? num3 = new double?();

double? num4 = 3.14159;

bool? boolVal = new bool?();

Console.WriteLine("显示可空的数据类型的值:{0},{1},{2},{3}",num1,num2,num3,num4);

Console.WriteLine("显示可空的布尔值：{0}",boolVal);

}

}

out:

显示可空的数据类型的值: ,45, ,3.14159

显示可空的布尔值：

**Null合并运算符（？？）：**

Null合并运算符，如果第一个操作数的值为null，则运算符返回第二个操作数的值，否则返回第一个操作数的值，实例：

double? num1 = null;

double? num2 = 3.14159

double num3;

num3 = num1??5.55;

Console.WriteLine(num3);

out:5.55

//num1为空，所以取到后面的值

//可以理解为三元运算符?:的简化形式num3 = (num1==null)?5.55:num1;

**C#数组：**

数组是一个存储相同类型元素的固定大小的顺序集合。

**声明、初始化、赋值给数组：**

//声明数组

datatype[] arrayName;

int[] intArr;

//初始化数组

datatype[] arrayName;

int[] intArr = new int[10]; //初始化时必须指定数组大小

//赋值给数组:

//方式一，索引号单独赋值给指定元素

intArr[0] = 10;

//方式二，声明数组的同时给数组赋值

int[] intArr = {1,6,9};

//方式三，创建并初始化一个数组

int[] intArr = new int[3] {3,6,9};

//方式四，省略数组大小但给出值

int[] intArr = new int[] {3,6,9};

//方式五，赋值一个数组变量到另一个数组变量，但需要注意这种情况下两个数组指向同一个内存地址**，一个跟随**另一个改变而改变

int[] intArr1 = new int[] {1,3,5,7,9};

int[] intArr2 = intArr1;

//注意在创建数组时，系统会分配默认值，int默认为0，bool默认为false

//访问数组元素,数组名加[索引]

intArr[5] = 100;

**二维数组：**

C#中的二维数组更像是一个矩阵，其声明与初始化方式如下：

//声明

int[,] intArr;

//初始化

int[,] intArr = new int[3, 3] {

{ 1,2,3},

{ 1,2,3},

{ 1,2,3}

};

//二维数组中获取到相关参数

int[,] x = new int[2, 3] {

{ 1,2,3},

{ 3,4,6},

};

int temp1 = x.Rank;

int temp3 = x.GetLength(1);

int temp2 = x.Length;

out：

temp1 = 2 //获取到数组的行

temp2 = 3 //获取到数组的列

temp3 = 6 //获取到数组的总长度

//三维数组的定义，二维数组是n个一维数组，三维数组不就是n个二维数组了吗

int[,,] a = new int[3, 3, 3] {

{ { 1,1,1},{ 2,2,2},{ 3,3,3} },

{ { 1,1,1},{ 2,2,2},{ 3,3,3} },

{ { 1,1,1},{ 2,2,2},{ 3,3,3} }

};

注意注意，int[][]这种格式的并不是二维数组，而是交错数组

**交错数组：**

声明并初始化交错数组：

int[][] scores = new int[3][]{

new int[]{1,2,3},

new int[]{4,5,6,7},

new int[]{8,9,10,11}

};

交错数组与二维数组的区别在于，交错数组的每一行的长度可以是不一样的，如果说二维数组是唐诗，那交错数组就是宋词，每一句的长短不限制。本质上交错数组是一维数组，使用foreach遍历交错数组：

foreach (int[] temp in scores)

{

foreach (int m in temp)

{

Console.WriteLine(m);

}

}

使用for循环遍历交错数组（本质是一维数组）：

for (int i = 0; i < scores.Length; i++)

{

for (int j = 0; j < scores[i].Length; j++)

{

Console.WriteLine(scores[i][j]);

}

}

**参数数组：**

当你不确定需要传递多少个参数给函数时，就可以使用参数数组作为形参：

**params关键字：**

在使用数组作为形参时，C#提供了params关键字，使调用数组为形参的方法时，既可以传递数组实参，也可以传递一组数组元素，使用实例：

public int addArr(params int[] arr)

{

int res = 0;

foreach(int a in arr)

{

res += a;

}

return res;

}

int c = addArr(1,6,8,9,7); //可以直接这样用

如此，我们甚至可以使用object[](对象数组)实现一些骚操作：

public class MyClass

{

public static void UseParams(params int[] list)

{

for(int i = 0; i < list.Length; i++)

{

Console.Write( list[i] + " ");

}

Console.WriteLine();

}

public static void UseParams2( params object[] list)

{

for(int i = 0; i < list.Length; i++)

{

Console.Write(list[i] + " ");

}

Console.WriteLine();

}

static void Main()

{

UseParams(1, 2, 3, 4);

UseParams2(1, 'a', "Apple");

UseParams2(); //不填入参数，执行还是会执行的

int[] myIntArray = {5, 6, 7, 8, 9, };

UseParams(myIntArray);

object[] myObjArray = { 1, 'b', "boom", "app" };

UseParams2(myObjArray);

UseParams2(myIntArray); //输出为"System.Int32[]"

}

}

out：

1 2 3 4

1 a Apple

5 6 7 8 9

1 b boom app

System.Int32[]

关于params的注意事项：

params关键字的参数类型必须是一维数组

不能和ref，out同时使用

带 params 关键字的参数必须是最后一个参数，并且在方法声明中只允许一个 params 关键字

**Array类：**

Array类中提供了各种用于数组的属性和方法。

常用属性：



常用方法：



**C#字符串：**

string[] sarray = {"hello","from","Point"};

string message = String.Join(" ",sarray); //将字符或字符串数组中返回字符串的方法，" "表示分隔符

Console.WriteLine(message);

out:

hello from Point

DateTime dt = new DateTime(2023,10,24,14,34,10);

string time = String.Format("Message sent at {0:t} on {0:D}",dt);//将字符串格式化输出

Console.WriteLine(time);

out:

Message sent at 14:34 on 2023年10月24日 //中文系统所以输出中文文字

**String类的方法：**



**注：有的方法已经过时**

**C#结构体：**

在C#中，结构体是值类型的数据结构，它使得一个单一变量可以存储各种数据类型的相关数据。**struct**关键字以创建结构体，结构体是用来表达一个记录。假设你想跟踪图书馆中书的动态。你可能想跟踪书的以下属性

Title

Author

Subject

BookID

**定义结构体：**

struct关键字定义了一个包含多个成员的新的数据类型，例如声明一个书的结构体：

struct Books

{

public string Title;

public string Author;

public string Subject;

public int BookID;

}; //这个分号是可加可不加的

using System;

using System.Text;

struct Books

{

   public string title;

   public string author;

   public string subject;

   public int book\_id;

};

public class testStructure

{

   public static void Main(string[] args)

   {

      Books Book1;        /\* 声明 Book1，类型为 Books \*/

      Books Book2;        /\* 声明 Book2，类型为 Books \*/

      /\* book 1 详述 \*/

      Book1.title = "C Programming";

      Book1.author = "Nuha Ali";

      Book1.subject = "C Programming Tutorial";

      Book1.book\_id = 6495407;

      /\* book 2 详述 \*/

      Book2.title = "Telecom Billing";

      Book2.author = "Zara Ali";

      Book2.subject =  "Telecom Billing Tutorial";

      Book2.book\_id = 6495700;

      /\* 打印 Book1 信息 \*/

      Console.WriteLine( "Book 1 title : {0}", Book1.title);

      Console.WriteLine("Book 1 author : {0}", Book1.author);

      Console.WriteLine("Book 1 subject : {0}", Book1.subject);

      Console.WriteLine("Book 1 book\_id :{0}", Book1.book\_id);

      /\* 打印 Book2 信息 \*/

      Console.WriteLine("Book 2 title : {0}", Book2.title);

      Console.WriteLine("Book 2 author : {0}", Book2.author);

      Console.WriteLine("Book 2 subject : {0}", Book2.subject);

      Console.WriteLine("Book 2 book\_id : {0}", Book2.book\_id);

      Console.ReadKey();

   }

}

结构体需要定义在namespace之外。

**C#结构体的特点：**

如上述中的例子。可见C#中的结构体与传统的C与C++中的结构不同，C#中的结构体有如下特点：

结构体可以带有方法、字段、属性、索引、运算符方法和事件

结构体可以定义构造函数，但不能定义析构函数。但是，不能为结构体定义无参构造函数。无参构造函数是默认自动定义的，且不能被改变。

与类不同，结构体不能继承其他的结构或类。

结构不能作为其他结构体或类的基础结构。

结构体可以实现一个或多个接口。

结构体不能被定义为abstract、virtual或protected。

当使用New操作符创建一个结构体对象时，会调用适当的构造函数来创建结构体。与类不同，结构体可以不使用New操作符就能被实例化。

当不使用New操作符，只有在所有的字段都被初始化之后，字段才被赋值，对象才被使用。

**类VS结构体：**

类和结构体有几个基本的不同点：

类是引用类型，结构体是值类型

结构体不支持继承

结构体不能声明默认的构造函数

结构体中的声明的字段无法被赋初始值，类可以

结构体的构造函数中必须为所有的字段赋值，类不用

结构体被存放在栈空间、类被存放在堆空间，因此，当我们描述一个轻量级对象的时候，结构可提高效率，成本更低。当然，这也得从需求出发，假如我们在传值的时候希望传递的是对象的引用地址而不是对象的拷贝，就应该使用类了。

using System;

struct Books

{

private string title;

private string author;

private string subject;

private int book\_id;

public void setValues(string t, string a, string s, int id)

{

title = t;

author = a;

subject = s;

book\_id = id;

}

public void display()

{

Console.WriteLine(title);

Console.WriteLine(author);

Console.WriteLine(subject);

Console.WriteLine(book\_id);

}

};

namespace Struct

{

class Program

{

static void Main(string[] args)

{

Books Book1 = new Books(); /\* 声明 Book1，类型为 Books \*/

Books Book2 = new Books(); /\* 声明 Book2，类型为 Books \*/

/\* book 1 详述 \*/

Book1.setValues("C Programming",

"Nuha Ali", "C Programming Tutorial", 6495407);

/\* book 2 详述 \*/

Book2.setValues("Telecom Billing",

"Zara Ali", "Telecom Billing Tutorial", 6495700);

/\* 打印 Book1 信息 \*/

Book1.display();

/\* 打印 Book2 信息 \*/

Book2.display();

Console.ReadKey();

}

}

}

**C#枚举：**

枚举是一种命名整形常量。枚举类型是使用enum关键字声明的。换句话说，C#中的枚举类型是值类型，并且包含自己的值，且不能被继承或传递继承。

声明枚举的一般语法：

enum <enum\_name>

{

enumration list;

};

//枚举列表中的每个符号代表一个整数值，一个比他前面的符号大的整数值。默认情况下第一个枚举符号的值是0，如：

enum Days {Sun,Mon,Tue,Wed,Thu,Fri,Sat};

实例：

using System；

public class EnumTest

{

enum Days {Sun,Mon,Tue,Wed,Thu,Fri,Sat}；

static void Main()

{

int x = (int)Days.Sun;

int y = (int)Days.Fri;

Console.WriteLine("Sun = {0}",x);

Console.WriteLine("Fri = {0}",y);

}

}

out:

Sun = 0

Fri = 5

//注意，当你在枚举中为某一个枚举常量添加一个标号后，他后面一个的标号就变成前一标号加1了

enum Test {One,Two = 6,Three}; //那么此处的Three的标号就是7了

**C#类（class）:**

当定义一个类时，就相当于定义了一个数据类型的蓝图。对象是类的实例。确定了类的对象由什么组成及在这个对象上执行什么操作。构成类的方法和变量称为类的成员。

**类的定义：**

类的定义是从关键字**class**开始的，后跟类的名称，类主体则在类的名称后的一对花括号中：

<access specifier> class class\_name

{

<access specifier> <data type> variable1;

<access specifier> <data type> variable2;

<access specifier> <return type> method1(parameter\_list)

{

}

}

注意：

访问修饰符<access specifier>指定了类及其成员的访问规则。若没有指定则使用默认的访问修饰，类的默认修饰符是internal（程序集内），成员的默认修饰符是private（当前类内）

数据类型data type指定了变量的类型，return type指定了函数的返回类型

若访问类中的成员则使用点运算符（.）

点运算符连接了对象的名称和成员的名称

实例：

using System;

namespace Class

{

internal class Box

{

public double length;

public double width;

public double height;

}

class Program

{

static void Main(string[] args)

{

Box Box1 = new Box();

Box Box2 = new Box();

double volume = 0.0;

//分别给Box1和Box2赋值

Box1.height = 5.0;

Box1.length = 6.0;

Box1.width = 7.0;

Box2.height = 10.0;

Box2.length = 12.0;

Box2.width = 13.0;

volume = Box1.height \* Box1.width \* Box1.length;

Console.WriteLine("Box1's volume is {0}",volume);

volume = Box2.height \* Box2.width \* Box2.length;

Console.WriteLine("Box2's volume is {0}", volume);

}

}

}

**成员函数与封装：**

类的成员函数是一个在类定义中有它的定义或原型的函数，就像其他变量一样可以直接作为类的一个成员使用点运算符进行调用，并且能够在类的任何对象上操作，且能访问类的任何成员。

成员函数同样能够进行封装：

using System;

namespace Class

{

internal class Box

{

public double length;

public double width;

public double height;

public void setLength(double len)

{

length = len;

}

public void setWidth(double wid)

{

width = wid;

}

public void setHeight(double hei)

{

height = hei;

}

public double getVolume()

{

return height \* width \* length;

}

private void showHeight()

{

Console.WriteLine(height);

}

}

class Program

{

static void Main(string[] args)

{

Box Box1 = new Box();

Box Box2 = new Box();

double volume = 0.0;

//设定Box对象的值

Box1.setHeight(5.0);

Box1.setWidth(6.0);

Box1.setLength(7.0);

Box2.setHeight(10.0);

Box2.setWidth(12.0);

Box2.setLength(13.0);

volume = Box1.getVolume();

Console.WriteLine(volume);

volume = Box2.getVolume();

Console.WriteLine(volume);

Box1.showHeight(); //报错Box.showHeight无法访问，它具有一定的保护级别，被封装为private了

}

}

}

**构造函数：**

类的构造函数是一个特殊的成员函数，当创建类的新对象时执行。构造函数名称与类名相同并且没有返回类型，实例：

public class Box{

public double length;

public double width;

public double height;

public Box(double len,double hei,double wid)

{

Console.WriteLine("对象已被创建，长宽高分别为：{0}，{1}，{2}",len,wid,hei);

length = len;

height = hei;

width = wid;

}

}

Main()

Box box1 = new Box(1.0,2.0,3.0); //有构造函数的类就必须要按照构造函数所要求的格式进行类实例化的创建了

**析构函数：**

类的析构函数是类的一个特殊的成员函数，析构函数是在**垃圾回收**或**资源释放**时使用的。析构函数的名称是在类的名称前添加一个波浪号~作为前缀，它**没有返回值**，也**没有任何的参数**。析构函数用于在结束程序时（比如关闭文件、释放内存等）之前释放资源。析构函数不能继承或重载。实例：

class Line

{

private double length;

public Line() //构造函数

{

Console.WriteLine("对象已创建！");

}

~Line() //析构函数

{

Console.WriteLine("对象已删除！");

}

public void setLength(double len)

{

length = len;

}

public double getLength()

{

return length;

}

}

注意：

析构函数只有类可以有，结构体不能有

一个类只能有一个析构函数

无法主动调用析构函数，它是被自动调用的

程序员无法控制何时调用析构函数，因为这是由垃圾回收器决定的。垃圾回收器检查是否存在应用程序不再使用的对象。如果垃圾回收器认为某个对象符合析构，则调用析构函数（如果有）并回收用来存储此对象的内存。程序退出时也会调用析构函数。

**C#中类的静态成员：**

**静态即只有一个。**

我们可以使用**static**关键字将类成员定义为静态的，当我们声明一个类成员为静态时，意味着无论有多少个类的对象被创建，都只会有一个该静态成员的副本。即**所有对象中只能有一个该成员的实例**（main方法作为入口函数字自然只能有一个所以必须是static）。静态变量用于定义常量，因为它们的值**可以通过直接调用类而不需要创建类的实例来获取**。静态变量可以在成员函数或者类的定义外部进行初始化，也可以在类的定义内部初始化静态变量。实例（我虽然创建了两个类，但因为其中的num是static，他只会有一个，所以每个类中的改变都会映射到唯一的一个num上）：

using System;

namespace Class

{

class StaticVar

{

public static int num;

public void Add()

{

num++;

}

public int getNum()

{

return num;

}

}

class code

{

static void Main()

{

StaticVar var1 = new StaticVar();

StaticVar var2 = new StaticVar();

var1.Add();

var1.Add();

var1.Add();

var2.Add();

var2.Add();

var2.Add();

Console.WriteLine("var1中的静态变量num的值为：{0}",var1.getNum());

Console.WriteLine("var2中的静态变量num的值为：{0}",var2.getNum());

Console.WriteLine("静态变量num的值为：{0}",StaticVar.num); //可以直接使用类名调用静态成员，而不用实例化，静态方法同样是这样

}

}

}

out：

var1中的静态变量num的值为：6

var2中的静态变量num的值为：6

静态变量num的值为：6

可以把类中的一个成员函数定义为static，这样的函数只能访问静态变量。静态函数在对象被创建之前就已经存在，实例：

using System;

namespace StaticVarApp

{

class StaticVar

{

public static int num;

public void Add()

{

num++;

}

public static int getNum()

{

return num;

}

}

class program

{

static void Main()

{

StaticVar s1 = new StaticVar();

s.Add();

s.Add();

s.Add();

Console.WriteLine("变量num的值：{0}",StaticVar.getNum()); //若使用s1.getNum会报错，得知static方法前缀只能是类名而不能是实例化对象

}

}

}

out：

变量num的值：3

**C#继承：**

继承是面向对象程序设计中最重要的概念之一。继承允许我们根据一个类来定义另一个类，这使得创建和维护应用程序变得更容易，同时也更有利于重写代码和节省开发时间。

当创建一个类时，程序员可以不用完全重新编写新的数据成员和成员函数，只需要设计一个新的类，继承已有的类的成员即可。这个已有的类被称为**基类**，新的类被称为派生类。

继承的思想实现了**属于（IS-A）**关系。例如，哺乳动物**属于**动物，狗**属于**哺乳动物，因此狗**属于**动物。

**基类与派生类：**

一个类可以派生自多个类或接口，这意味着他可以从多个基类或接口中继承数据和函数。假设，有一个基类Shape，他的派生类是Rectangle：

using System;

namespace jicheng

{

//基类

class Shape

{

public void setWidth(int w)

{

width = w;

}

public void setHeight(int h)

{

height = h;

}

protected int width;

protected int height;

}

//派生类

class Rectangle : Shape

{

public int getArea()

{

return width \* height;

}

}

class program

{

static void Main()

{

Rectangle r = new Rectangle();

r.setWidth(5);

r.setHeight(7);

Console.WriteLine("总面积为：{0}",r.getArea());

}

}

}

**基类的初始化：**

派生类继承了基类的成员变量与成员方法。因此父类对象应当在子类对象创建之前被创建。可以在成员初始化列表中进行父类的初始化，实例：

using System;

namespace jicheng

{

class Rectangle

{

protected double length;

protected double width;

public Rectangle(double l, double w)

{

length = l;

width = w;

}

public double getArea()

{

return (width\*length);

}

public void Display()

{

Console.WriteLine("长度为： {0}\n宽度为： {1}\n面积为： {2}",length,width,getArea());

}

}

class Tabletop : Rectangle

{

private double cost;

public Tabletop(double l, double w) : base(l, w) { } //这样的格式是没问题的，子类的构造函数直接继承父类的构造函数

public double GetCost()

{

double cost;//作用域不同，优先选择局部作用域，两个同样的cost在C#中是合法的

cost = getArea() \* 70;

return cost;

}

public void Display()

{

base.Display();

Console.WriteLine("成本为：{0}",GetCost());

}

}

class program

{

static void Main()

{

Tabletop t = new Tabletop(4.5,7.5);

t.Display();

}

}

}

**C#多重继承：**

多重继承是指一个类可以同时从多于一个父类中继承行为与特征的功能。与单一继承相对，单一继承指一个类只能继承一个父类。C#中不允许多重继承（与C++不同），但是，可以通过接口的方式实现多重继承，实例：

using System;

namespace jicheng

{

//基类Shape

class Shape

{

public void setWidth(int w)

{

width = w;

}

public void setHeight(int h)

{

height = h;

}

protected int width;

protected int height;

}

//基类PaintCost

public interface PaintCost

{

int getCost(int area);

}

//派生类

class Rectangle : Shape, PaintCost

{

public int getArea()

{

return (width \* height);

}

public int getCost(int area)

{

return area \* 70;

}

}

class program

{

static void Main()

{

Rectangle Rect = new Rectangle();

int area;

Rect.setWidth(5);

Rect.setHeight(7);

area = Rect.getArea();

Console.WriteLine("总面积为：{0}",Rect.getArea());

Console.WriteLine("油漆总成本为：${0}",Rect.getCost(area));

}

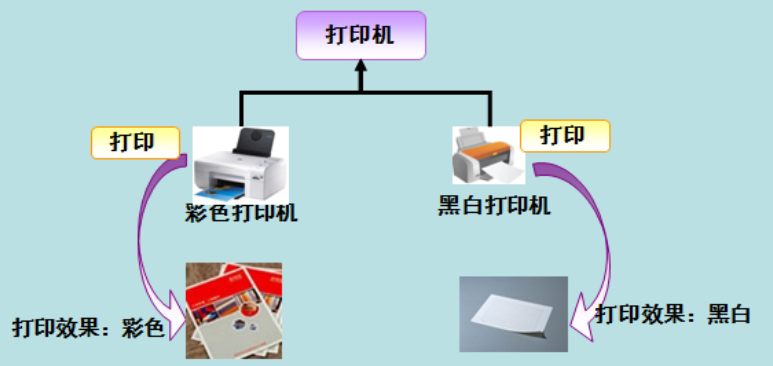
}

}

展示了继承和接口的用法。通过继承，派生类可以获得基类的属性和方法，并可以进行自定义的拓展。接口则提供了一种规范，要求实现了该接口的类必须提供指定的方法。通过继承和接口的组合使用，可以实现更加灵活和可扩展的代码结构。关于接口的内容在后面的章节讲。

**C#多态性：**

多态性意味着有多种形式。在面向对象编程范式中，多态性往往表现为“一个接口，多个功能”。多态性可以是静态或者动态的。在**静态多态性**中，函数的响应是在编译时发生的。在**动态多态性**中，函数的响应是在运行时发生的。在C#中，每个类型都是多态的，因为包括用户定义类型在内的所有类型都继承自**Object**，多态就是同一个接口（在此不是指interface），使用不同的实例而执行不同的操作，如图：



又如，当按下F1键时：

如果在Flash界面下弹出的就是AS3的帮助文档

如果在Word界面下弹出的就是Word帮助

在Windows下弹出的就是Windows帮助和支持

同一个事件发生在不同的对象上会产生不同得到结果。

**静态多态性：**

在**编译**时，函数和对象的连接机制被称为早期绑定或静态绑定，C#中提供了两种技术来实现静态多态性。分别为：

函数重载

运算符重载（下一章节讨论）

**函数重载：**

你可以在同一个范围内对相同的函数名有多个定义。函数的定义必须彼此不同，可以是参数列表中的参数类型不同，也可以是参数个数不同，也可以是返回值类型不同。但不能重载只有返回类型不同的函数声明。实例中演示了几个相同的函数Add()，用于对不同个数参数进行相加处理：

using System;

namespace PolymorphismApplication

{

    public class TestData

    {

        public int Add(int a, int b, int c)

        {

            return a + b + c;

        }

        public int Add(int a, int b) //名字一样是可以的，但是得参数类型或参数个数不同才行，仅仅是返回值类型不同是不可以的

        {

            return a + b;

        }

public double Add(int a, int b) //这个函数就会报错

{

return Convert.ToDouble(a+b);

}

    }

    class Program

    {

        static void Main(string[] args)

        {

            TestData dataClass = new TestData();

            int add1 = dataClass.Add(1, 2);

            int add2 = dataClass.Add(1, 2, 3);

            Console.WriteLine("add1 :" + add1);

            Console.WriteLine("add2 :" + add2);

        }

    }

}

**动态多态性：**

**抽象类：**

C#允许使用关键字**abstract**创建抽象类，用于提供接口的部分类的实现。当一个派生类继承自该抽象类时，实现即完成。**抽象类**包含抽象方法，抽象方法可以被派生类实现。派生类具有更专业的功能。以下有几点抽象类的注意：

不能创建一个抽象类的实例

不能在一个抽象类的外面声明一个抽象方法

通过在类定义前放置关键字**sealed**，可以将类声明为**密封类**。当一个类被声明为**sealed**时，他就**不能被继承**了。抽象类是不能被声明为sealed的。

实例：

using System;

namespace duotaixing

{

abstract class Shape

{

abstract public int area();

}

class Rectangle : Shape

{

private int length;

private int width;

public Rectangle(int a = 0, int b = 0)

{

length = a;

width = b;

}

public override int area()

{

Console.Write("Rectangle类的面积为:");

return (width \* length);

}

}

class Program

{

static void Main(string[] args)

{

Rectangle r = new Rectangle(10,7);

double a = r.area();

Console.WriteLine(a);

}

}

}

out：

Rectangle类的面积为:70 //先write再writeLine还是会显示在一行中的

**虚方法：**

当有一个定义在类中的函数需要在继承类中实现的时候，可以使用**虚方法**。虚方法通过**virtual**关键字声明。虚方法可以在不同的继承类下有不同的实现。对虚方法的调用是在**运行时**发生的。**动态多样性就是通过抽象类和虚方法来实现的**。下面的实例中，创建Shape基类，派生Circle、Rectangle等类，Shape类提供一个名为Draw的虚方法，在每个派生类中重写该方法从而绘制不同的形状。

using System;

using System.Collections.Generic;

namespace duotaixing

{

public class Shape

{

public int X { get; private set; }

public int Y { get; private set; }

public int Height { get; set; }

public int Width { get; set; }

public virtual void Draw()

{

Console.WriteLine("执行基类的画图任务");

}

}

class Circle : Shape

{

public override void Draw()

{

Console.WriteLine("画一个圆形");

base.Draw();

}

}

class Rectangle1 : Shape

{

public override void Draw()

{

Console.WriteLine("画一个矩形");

base.Draw();

}

}

class Triangle : Shape

{

public override void Draw()

{

Console.WriteLine("画一个三角形");

base.Draw();

}

}

class program

{

static void Main()

{

var shapes = new List<Shape>()

{

new Rectangle1(),

new Triangle(),

new Circle()

};

foreach (var s in shapes)

{

s.Draw();

}

Console.WriteLine("按下任意键以退出");

Console.ReadKey();

}

}

}

实例二，通过虚方法area计算不同形状图像的面积：

using System;

namespace Code

{

class Shape

{

protected int width, height;

public Shape(int a = 0,int b = 0)

{

width = a;

height = b;

}

virtual public int area()

{

Console.WriteLine("父类的面积：");

return 0;

}

}

class Rectangle : Shape

{

public Rectangle(int a = 0, int b = 0) : base(a, b) { }

public override int area()

{

Console.WriteLine("Rectangle类的面积：");

return width \* height;

}

}

class Trigger : Shape

{

public Trigger(int a = 0, int b = 0) : base(a, b)

{

}

public override int area()

{

Console.WriteLine("Trigger类的面积为：");

return width\*height/2;

}

}

class Caller

{

public void CallArea(Shape sh) //形参为Shape类，其子类当然也可以

{

int a = sh.area(); //已经用虚方法重写了

Console.WriteLine("面积：{0}",a);

}

}

class Program

{

static void Main()

{

Caller c = new Caller();

Rectangle r = new Rectangle(10,7);

Trigger t = new Trigger(10, 5);

c.CallArea(r);

c.CallArea(t);

}

}

}

out：

Rectangle类的面积：

面积：70

Trigger类的面积为：

面积：25

**隐藏方法：**

隐藏方法是指在派生类中重新定义与基类中同名的方法。

隐藏方法使用 **new** 关键字进行修饰，显示告诉编译器这是一个隐藏方法。

**隐藏方法不会隐藏基类中的同名方法，即在基类类型的引用下（即Animal animalCat = new Cat();这种形式；即虽然实际上是Cat类）但调用该方法时，实际执行的仍然是基类中的方法。**

隐藏方法不涉及运行时多态性，只是在编译时根据引用的类型确定调用哪个方法。

隐藏方法可以用于扩展或修改基类中的方法实现。

实例：

using System;

class Animal

{

public void Eat()

{

Console.WriteLine("Animal is eating.");

}

// 隐藏方法

public new void Sleep()

{

Console.WriteLine("Animal is sleeping.");

}

}

class Cat : Animal

{

public new void Sleep()

{

Console.WriteLine("Cat is sleeping.");

}

}

class Program

{

static void Main(string[] args)

{

Animal animal = new Animal();

animal.Eat();

animal.Sleep(); // 输出 "Animal is sleeping."

Cat cat = new Cat();

cat.Eat();

cat.Sleep(); // 输出 "Cat is sleeping."

Animal animalCat = new Cat();

animalCat.Eat();

animalCat.Sleep(); // 输出 "Animal is sleeping."

}

}

**virtual与abstract的区别：**

virtual与abstract都是用来修饰父类的，通过覆盖父类的定义，让子类重新定义。

**virtual**修饰的方法**必须要实现**（哪怕就只是一对空的大括号）；而**abstrac**t修饰的方法**必须不实现**，不能有任何的方法实体。

virtual可以被子类重写也可以不重写，abstract必须被子类重写

如果类中的成员被abstract修饰，那么类也必须是abstract，只有抽象类才可以有抽象方法

无法创建abstract类的实例，abstract类只能被继承无法实例化

简单来说，就是抽象方法需要子类去实现，而虚方法已经实现了，可以被子类覆盖。

**重载和重写的区别：**

重载（overload）是提供一种机制，相同函数名不同参数或不同参数类型。

重写（override）是用于重写基类的虚方法，这样在派生类中能提供一个新的方法。

**运算符重载 ：**

运算符重载是静态多态性中的一种实现方式，在编译时即调用。可以重定义或重载C#中内置的运算符。因此，程序员可以使用用户自定义类型的运算符。重载运算符是具有特殊名称的函数，通过关键字**operator**后跟运算符的符号进行定义的。与其他函数一样，重载运算符有返回类型与参数列表。实例：

public static Box operator+(Box b,Box c)

{

Box box = new Box();

box.length = b.length + c.length;

box.breadth = b.breadth + c.breadth;

box.height = b.height + c.height;

return box;

}

上面的函数为用户自定义类Box实现了加法运算符。他把两个Box对象的属性相加，并返回相加后的Box对象。

完整实现：

using System;

namespace yunsuanfuchongzai

{

class Box

{

private double length;

private double breadth;

private double height;

public double getVolume()

{

return length \* breadth \* height;

}

public void setLength(double len)

{

length = len;

}

public void setBreadth(double bre)

{

breadth = bre;

}

public void setHeight(double hei)

{

height = hei;

}

//重载+运算符来把两个Box对象相加

public static Box operator +(Box b, Box c)

{

Box box = new Box();

box.length = b.length + c.length;

box.breadth = b.breadth + c.breadth;

box.height = b.height + c.height;

return box;

}

}

class Program

{

static void Main(string[] args)

{

Box box1 = new Box();

Box box2 = new Box();

Box box3 = new Box();

double volume = 0.0;

box1.setLength(6.0);

box1.setBreadth(7.0);

box1.setHeight(5.0);

volume = box1.getVolume();

Console.WriteLine("box1的体积为：{0}",volume);

box2.setLength(12.0);

box2.setBreadth(13.0);

box2.setHeight(10.0);

volume = box2.getVolume();

Console.WriteLine("box2的体积为：{0}", volume);

box3 = box1 + box2; //运算符重载完之后，直接使用就可以了

volume = box3.getVolume();

Console.WriteLine("box3的体积为：{0}", volume);

}

}

}

所以，其实重载运算符看起来跟一般的函数形式是类似的，同样是需要返回值、形参、函数体。唯一有点不同的就是重载运算符必须要使用static。且与函数名不同，重载运算符需要使用**operator**关键字后面加上运算符作为重载名。

**可重载与不可重载运算符：**



**C#命名空间：**

命名空间的设计目的是提供一种让一组名称与其他名称分隔开的方式。在一个命名空间中声明的类的名称与另一个命名空间中类的名称可以是一样的。可以理解为在计算机系统中，两个文件夹下可以有重名的文件名。

**定义命名空间：**

namespace namespace\_name

{

//代码主体

}

为了调用支持命名空间版本的函数或者变量，会把命名空间的名称放置于前面，如下：

namespace\_name.item\_name;

实例：

using System;

namespace first\_space

{

class namespace\_c1

{

public void func()

{

Console.WriteLine("Inside first\_space");

}

}

}

namespace second\_space

{

class namespece\_c1

{

public void func()

{

Console.WriteLine("Inside second\_space");

}

}

}

class program

{

static void Main()

{

first\_space.namespace\_c1 fc = new first\_space.namespace\_c1();

second\_space.namespece\_c1 sc = new second\_space.namespece\_c1();

fc.func();

sc.func();

}

}

out：

Inside first\_space

Inside second\_space

**using关键字：**

using关键字表明程序使用的是给定命名空间中的名称。例如我们常用的System命空间，其中定义了Console类，我们可以直接写Console.WriteLine();而不用写全为System.Console.WriteLine();

以此为引导，我们也可以使用using命名空间指令，这样在使用的时候就不用在前面加上命名空间名称。该指令直接告诉了编译器我们要使用到哪个命名空间中的应用。如此一来，我们重写上面的例子代码就能变得简洁的多：

using System;

using first\_space;

using second\_space;

namespace first\_space

{

class abc

{

public void func()

{

Console.WriteLine("Inside first\_space");

}

}

}

namespace second\_space

{

class def

{

public void func()

{

Console.WriteLine("Inside second\_space");

}

}

}

class program

{

static void Main()

{

abc a = new abc();

def d = new def();

a.func();

d.func();

}

}

当然，这种情况下就不再适合两个同名的类了，编译器就分不清到底应该调用哪个命名空间中的那个同名类了。

**嵌套命名空间：**

命名空间可以嵌套，可以使用点（.）运算符访问嵌套的命名空间中的成员（例如using UnityEngine.UI;外层命名空间为UnityEngine，内层命名空间为UI），实例：

using System;

using SomeNameSpace;

using SomeNameSpace.Nested;

namespace SomeNameSpace

{

public class MyClass

{

static void Main()

{

Console.WriteLine("In SomeNameSpace");

Nested.NestNameSpaceClass.SayHello();//因为开始using添加了Nested命名空间，

//所以其实是可以直接使用NestNameSpaceClass的

}

}

//内嵌命名空间

namespace Nested

{

public class NestNameSpaceClass

{

public static void SayHello()

{

Console.WriteLine("Hello world!");

}

}

}

}

**using关键字的所有用法：**

引入命名空间

using System;

using NameSpace1.InsideNameSpace;

using static指令：指定无需指定类型名称即可直接访问其静态成员的类型。如下面的实例中，引用了System.Math命名空间，该命名空间中有静态成员PI，那么我就可以直接在代码中使用double a = PI;

using static System.Math;

...

double a = PI;

起别名：

using Project = PC.MyCompany.Project;

**C#接口(interface)：**

接口定义了所有类继承接口时应遵循的语法合同。接口定义了语法合同**“是什么”**部分，派生类定义了语法合同“**怎么做**”部分。

接口定义了**属性**、**方法**和**事件**，这些都是接口的成员。接口只包含了成员的声明。成员的定义是派生类的责任。接口定义了派生类应遵循的标准结构。

接口使得实现接口的类或结构在形式上保持一致。抽象类在某种程度上与接口类似，但是，它们大多只是用在只有少数方法由基类声明由派生类实现的情况。

接口本身不实现任何功能，他只是和声明实现该接口的对象订立一个**必须实现**哪些行为的契约。抽象类不能直接实例化，但允许派生出具体的，具有实际功能的类。

**定义接口：**

接口使用**interface**关键字声明，他与类的声明类似。接口声明默认是public的。实例：

interface IMyInterface

{

void MethodToImplement（）;

}

以上代码定义了接口IMyInterface。通常接口命令由**I**字母开头，这个接口只有一个方法MethodToImplement()，没有参数和返回值，当然我们也可以按照需求设置参数和返回值。值得注意的是，该方法并没有具体的实现。

接下来我们来实现以上的接口：

using System;

interface IMyInterface

{

//接口成员

void MethodToImplement();

}

class InterfaceImplementer : IMyInterface

{

static void Main()

{

InterfaceImplementer iImp = new InterfaceImplementer();

iImp.MethodToImplement();

}

public void MethodToImplement()

{

Console.WriteLine("接口成员函数实现.");

}

}

上述例子中，InterfaceImplementer类实现了IMyInterface接口，接口的实现与类的继承语法格式类似。

继承接口后，我们必须实现接口的方法MethodToImplement()，且方法名必须与接口定义的方法一致。

**接口继承：**

以下实例定义了两个接口IMyInterface和IParentInterface。如果一个接口继承其他接口，那么实现类或接口就必须实现所有接口的成员（包括直接继承的接口的成员和继承接口的父接口的成员）

以下实例IMyInterface接口继承了IParentInterface接口，因此接口的实现类必须实现MethodToImplement()和ParentInterfaceMethod()方法：

using System;

interface IParentInterface

{

void ParentInterfaceMethod();

}

interface IMyInterface : IParentInterface

{

//接口成员

void MethodToImplement();

}

class InterfaceImplementer : IMyInterface

{

static void Main()

{

InterfaceImplementer iImp = new InterfaceImplementer();

iImp.MethodToImplement();

iImp.ParentInterfaceMethod();

}

public void MethodToImplement()

{

Console.WriteLine("直接接口成员函数实现。");

}

public void ParentInterfaceMethod()

{

Console.WriteLine("父接口成员函数实现。");

}

}

**完整的接口：**

实例中包含了接口能包含的所有成员（属性、方法、事件），其中需要注意的是接口中属性的实现方式

using System;

// 定义接口

public interface IExampleInterface

{

// 属性

int MyProperty { get; set; }

// 方法

void MyMethod();

// 事件

event EventHandler MyEvent;

}

// 实现接口的类

public class ExampleClass : IExampleInterface

{

private int myProperty;

// 实现属性

public int MyProperty

{

get { return myProperty; }

set { myProperty = value; }

}

// 实现方法

public void MyMethod()

{

Console.WriteLine("MyMethod called");

}

// 实现事件

public event EventHandler MyEvent;

public ExampleClass()

{

// 在构造函数中订阅事件

MyEvent += ExampleEventHandler;

}

private void ExampleEventHandler(object sender, EventArgs e)

{

Console.WriteLine("MyEvent triggered");

}

}

// 使用示例

public class Program

{

public static void Main()

{

ExampleClass example = new ExampleClass();

example.MyProperty = 42;

Console.WriteLine(example.MyProperty);

example.MyMethod();

// 触发事件

example.MyEvent += null;

}

}

**C#预处理指令：**

预处理指令是在编译器在实际编译开始前对信息进行预处理。所有的预处理指令都是以#开始。且在一行上，只能有一个预处理指令，并且因为是指令而不是语句所以结尾不需要使用分号；结束。下图为C#中可用的预处理指令：



实例：

#define PI

using System;

namespace App1

{

class Program

{

static void Main()

{

#if(PI)

Console.WriteLine("PI is defined");

#else

Console.WriteLine("PI is not defined");

#endif

Console.ReadKey();

}

}

}

out：

PI is defined

上例中，使用#if判断PI是否已经被定义了，#else进行#if条件不成立时的操作，#endif是必须需要的，用以终结#if预处理指令。

**条件指令：**

可以使用#if指令来创建一个条件指令。条件指令用于测试符号是否为真。若为真，编译器会执行#if和下一个指令之间的代码。其中，是可以使用true和false，或者在符号前放置否定运算符或者使用关系运算符或逻辑运算符等。条件指令用于在调试版本或编译指定配置时编译代码。一个以#if开始的指令必须以一个#endif作为终止。实例：

#define ZZZ

using System;

class Program

{

static void Main()

{

#if ZZZ

Console.WriteLine("OK");

#else

Console.WriteLine("No);

#endif

}

}

实例二：

#define BUG

#define VC\_V10

using System;

public class TestClass

{

   public static void Main()

   {

#if BUG && !VC\_V10

Console.WriteLine("BUG is defined");

#elif !BUG && VC\_V10

Console.WriteLine("VC\_V10 is defined");

#elif BUG && VC\_V10

Console.WriteLine("BUG and VC\_V10 are defined");

#else

        Console.WriteLine("BUG and VC\_V10 are not defined");

#endif

Console.ReadKey();

   }

}

**预处理指令的作用：**

**#if家族：**在程序调试和运行上有重要的作用。比如预处理指令可以禁止一部分代码编译，达到很好的控制代码的作用，总体上来说与直接的if else类似，但是#if等是面向代码块的。

**#warning和#error：**当编译器遇到这俩时会分别产生警告和错误，会给用户在下方错误和警告列表中显示error或warning后的内容，并且，当存在的是error时还会无法编译：

#if DEBUG && RELEASE

#error "You've defined DEBUG and RELEASE simultaneously!"

#endif

#warning "Don't forget to remove this line before the boss tests the code!"

Console.WriteLine("\*I hate this job.\*");

**#region和#endregion：**这俩会将一段代码标记为一个有名称的块，并能够实现折叠该块（只是折叠，并不是不使用）如下所示：

using System;

class Program

{

static void Main()

{

#region hello world

Console.WriteLine("hello world!");

#endregion

}

}

**C#正则表达式：**

正则表达式是一种匹配输入文本的模式。.Net框架提供了允许这种匹配的正则表达式引擎。模式由一个或多个字符、运算符和结构组成。

**定义正则表达式：**

下面列出了用于定义正则表达式的各种类别的字符、运算符和结构：

字符转义

字符类

定位点

分组构造

限定符

反向引用构造

备用构造

替换

杂项构造

具体内容参见网址：[C#正则表达式](https://www.runoob.com/csharp/csharp-regular-expressions.html)

一个使用正则表达式的实例：

using System;

using System.Text.RegularExpressions;

public class Program

{

public static void Main()

{

string input = "Hello, World! This is a test.";

// 使用正则表达式匹配所有小写字母

Regex regex = new Regex("[a-z]");

MatchCollection matches = regex.Matches(input);

// 判断是否存在小写字母

if (matches.Count > 0)

{

Console.WriteLine("输入中存在小写字母。");

// 输出匹配到的结果

Console.WriteLine("匹配到的结果：");

foreach (Match match in matches)

{

Console.Write(match.Value+" ");

}

}

else

{

Console.WriteLine("输入中不存在小写字母。");

}

}

}

**C#异常处理：**

异常是在程序执行期间出现的问题。C#中的异常是对程序运行时出现的特殊情况的一种响应，比如尝试除以零。

异常提供了一种把程序控制权从某个部分转移到另一个部分的方式。C#异常处理时建立在四个关键词之上的：**try、catch、finally、throw**。

**try**：一个try块标识了一个被激活的特定的异常的代码块。后面跟一个或多个catch块

**catch**：程序通过异常处理程序捕获异常，catch关键字表示异常的捕获。

**finally**：finally块用于执行给定的语句，不管异常是否被抛出都会执行。例如，当你打开一个文件，不管是否出现异常文件都要被关闭。

**throw**：当问题出现时，程序抛出一个异常。使用throw关键字来完成。

**语法：**

假设一个块将出现异常，一个方法使用try和catch关键字捕获异常。try/catch块内的代码为受保护的代码，语法如下：

try

{

//引起异常的语句

}

catch(ExceptionName e1)

{

//错误处理代码

}

catch(ExceptionName e2)

{

//错误处理代码

}

catch(ExceptionName e3)

{

//错误处理代码

}

finally

{

//要执行的语句

}

可以列出多个catch语句来捕获不同类型的异常，以防try块在不同的情况下生成多个异常。

**C#中的异常类：**

C#异常是使用类来表示的。C#中的异常类最主要是直接或者派生于System.Exception类。System.ApplicationException和System.SystemException类是派生于System.Exception类的异常类。System.ApplicationException类支持由应用程序产生的异常。所以程序员定义的异常都应该派生自该类。System.SystemException类是所有预定义的系统异常的基类。下表列出了一些派生自System.SystemException类的预定义的异常类：



**异常处理：**

C#以try和catch块的形式提供了一种结构化的异常处理方案。使用这些块，把核心程序语句与错误处理语句分离开。这些错误处理块是使用try、catch和finally关键字实现的，实例：

using System;

namespace yichangchuli

{

class DivNumbers

{

int result;

DivNumbers()

{

result = 0;

}

public void division(int num1, int num2)

{

try

{

result = num1 / num2;

}

catch (DivideByZeroException e)

{

Console.WriteLine("Exception caught:{0}", e);

}

finally

{

Console.WriteLine("Result:{0}",result);

}

}

static void Main(string[] args)

{

DivNumbers d = new DivNumbers();

d.division(25,0);

}

}

}

out：

Exception caught:System.DivideByZeroException: Attempted to divide by zero.

at yichangchuli.DivNumbers.division(Int32 num1, Int32 num2) in D:\\...Program.cs:line 16

Result:0

异常会在程序运行后在控制台抛出。请注意finally中的语句是一定会执行的。并且，注意到上面的DivideByZeroException这个异常是系统自带的异常，我们可以直接使用。同样的，我们也能定义自己特有的异常：

**创建用户自定义异常：**

用户自定义的异常派生自ApplicationException类。实例：

using System;

namespace Code

{

class TestTemperature

{

static void Main()

{

Temperature temp = new Temperature();

try

{

temp.ShowTemp();

}

catch (TempIsZeroException e)

{

Console.WriteLine("TempIsZeroException:{0}",e.Message);

}

}

}

//自定义异常类，所有自定义异常类都继承自ApplicationException类

public class TempIsZeroException : ApplicationException

{

//这就是一个构造函数啊

public TempIsZeroException(string message) : base(message)

{

}

}

public class Temperature

{

int temprature = 0;

public void ShowTemp()

{

if (temprature == 0)

{

//throw关键字用以抛出一个异常对象

throw (new TempIsZeroException("Zero Temperature found"));

}

else

{

Console.WriteLine("Temperature:{0}",temprature);

}

}

}

}

out:

TempIsZeroException:Zero Temperature found

**C#的文件输入输出：**

一个文件是一个存储在磁盘中带有指定名称和目录路径的数据集合。当打开文件进行读写时，它变成了一个**流**。

从根本上说，流是通过通信路径传递的字节序列。有两个主要的流：**输入流**和**输出流**。输入流用于从文件读取数据(读操作)，输出流用于向文件写入数据(写操作)。

**C#的I/O类：**

System.IO命名空间中有各种不同的类，用于执行各种文件操作，如创建文件和删除文件、读取和写入文件，关闭文件等。下表列出了一些System.IO命名空间中常用的非抽象类：

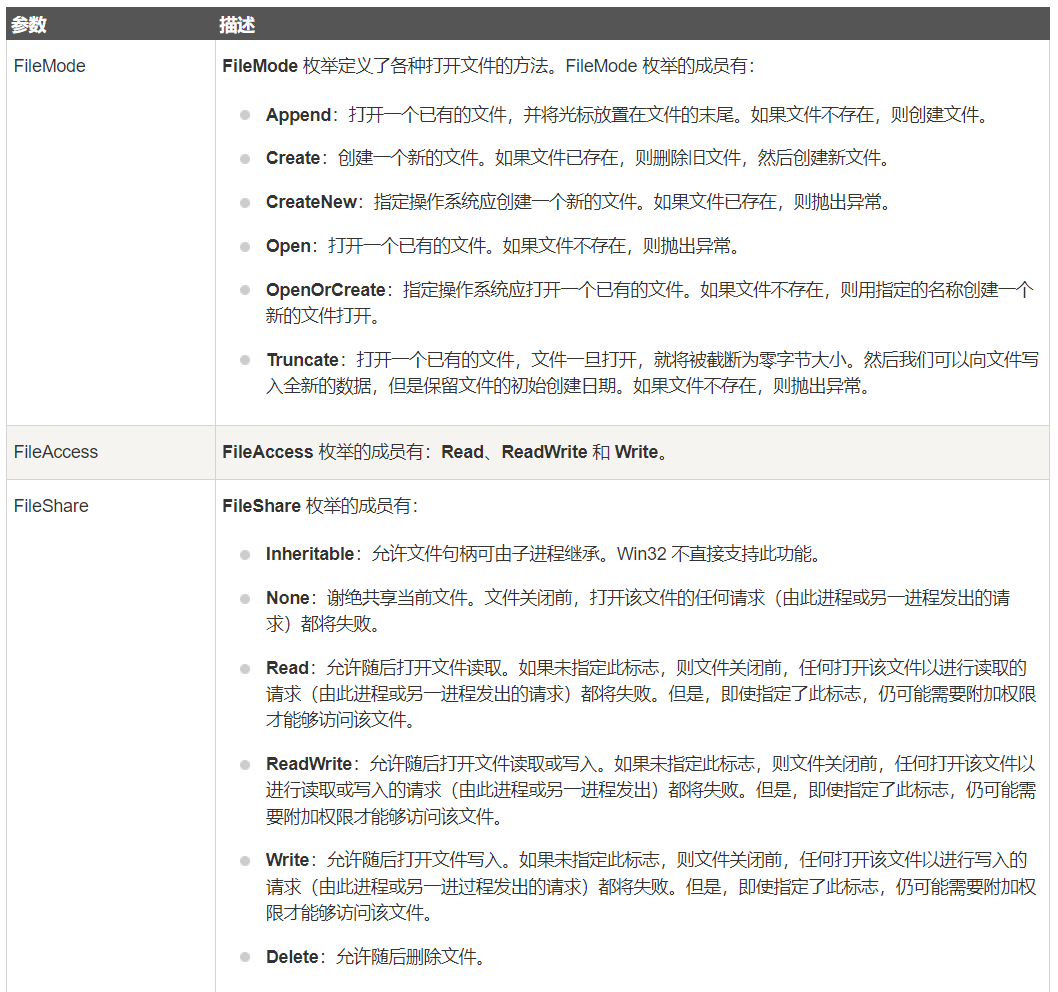


**FileStream类：**

System.IO命名空间中的FileStream类有助于文件的读写和关闭。该类派生自抽象类Stream。需要创建一个FileStream对象来创建一个新的文件或者打开一个已有的文件。实例：

FileStream F = new FileStream("sample.txt",FileMode.Open,FileAccess.Read,FileShare.Read);

在上例中，创建了一个FileStream对象F来读取名为sampl.txt的文件。



实例：

using System;

using System.IO;

namespace FileHandle

{

class Program

{

static void Main(string[] args)

{

//创建了一个test.dat的文件（因为本来没有）

FileStream F = new FileStream("test.dat",FileMode.OpenOrCreate

,FileAccess.ReadWrite);

for (int i = 1; i <= 20; i++)

{

F.WriteByte((byte)i);

}

F.Position = 0;

for (int i = 0; i <= 20; i++)

{

Console.Write(F.ReadByte()+" ");

}

F.Close();

}

}

}

out：

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 -1

**C#特性（Attribute）：**

特性（Attribute）是用于在运行时传递程序中各种元素（比如类、方法、结构、枚举、组件等）的行为信息的声明性标签。可以通过使用特性向程序添加声明性信息。一个声明性标签是通过放置在他所应用的元素前面的方括号（[ ]）来描述的。

特性（Attribute）用于添加元数据，如编译器指令和注释、描述、方法、类等其他信息。.Net框架提供了两种类型的特性：**预定义特性**和**自定义特性**。

特性的名称和值是在方括号内规定的，放置在它所应用的元素之前。

**概念：**

特性（Attribute）是用户可以自定义的修饰符，这种修饰符可以被放在被修饰目标上（比如类、方法、结构、枚举、组件等），换句话说，Attribute可以在被修饰目标上添加额外的扩展信息和行为，而对被修饰目标没有任何影响。表面上看，单独的追加信息，与注释是不是有点“类似”呢。但是特性与注释的区别在于：

注释是对开发者或用户用的，是对源码的一种说明，在编译过程中是不参与的，不影响编译，只是一种解释，方便“我们”理解代码。

特性是源码的一部分，在编译过程中，会放到元数据中，程序运行时会动态得到特性，方便“编译器”理解代码，编译器提取到这个特性的信息后，会进行相关的逻辑处理。

VS的编译原理：C#代码-->编译成exe/dll文件-->CLR(.Net的运行时环境，用于执行和管理托管代码)-->IL（中间语言）-->编译成机器码（1010）。

特性的位置在：CLR-->IL+元数据。

由上述的内容，我们可以推断出特性的作用主要有：

增加标识，让编译器知道这个代码是什么内容

注入信息，让编译器了解关于这个代码更多的信息来进行更多的操作

**预定义特性实例：**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Attribute

{

public class ReviewAttribute

{

}

public class MyClass

{

//[Obsolete]这个特性为当前的属性或者方法添加一个提醒

//使用场景为比如当我们更换代码时，不能直接删除某一个属性，那会报错很多，但我们可以提供一个提醒

//让开发者注意这个属性已经过时了

[Obsolete("该属性已过时，请使用新属性：TextContent")]

public string Content { get; set; }

public string TextContent { get; set; }

public void Test()

{

MyClass mc = new MyClass();

mc.Content = "No";

mc.TextContent = "Yes";

}

}

//[Serializable]这个特性（Attribute）用来标识当前实体类是可以被序列化的

[Serializable]

public class Student

{

public int StudentID { get; set; }

}

}

**特性的应用一：**

特性的基本使用1：增加标识

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace AttributeStudy

{

public class AttributeUsage1

{

}

//AttributeUsage被叫做特性的特性，用来修饰特性的，主要用来说明特性在哪里使用。

//AttributeTarget表示该特性能被哪些对象所添加，如下例中可以被类和属性添加

//AllowMultiple表示这个特性能不能在一个对象上被追加多个相同特性（可能只是构造函不同）

[AttributeUsage(AttributeTargets.Class|AttributeTargets.Property,AllowMultiple = true,Inherited = true)]

public class MyCustomAttribute : Attribute

{

public int CourseId { get; set; } = 1000;

public string Comment { get; set; }//备注类的信息

//特性可以有多个构造方法，这点与普通类是一样的

public MyCustomAttribute() { }

public MyCustomAttribute(int courseId)

{

this.CourseID = courseId;

}

public MyCustomAttribute(int courseId, string comment)

{

this.CourseID = courseId;

this.Comment = comment;

}

//特性也可以添加方法

public string GetMyCustomInfo()

{

return $"课程编号:{CourseID}{Comment}";

}

//$符号是插值字符串，表示后面的{}内的内容会被替换为值

}

//特性是可以被继承的，当然，继承的对象肯定也是特性，不能是类

public class YourCustomAttribute : MyCustomAttribute

{

public string YourName { get; set; }

}

}

//Conditional特性的作用

[Conditional("Hello")]

public static void outNum1()

{

Console.WriteLine(1);

}

[Conditional("world")]

public static void outNum2()

{

Console.WriteLine(2);

}

//只有在头文件中定义了Hello才会调用outNum1方法（#define Hello）

//只有在头文件中定义了world才会调用outNum2方法(#define world)

特性的基本使用2：注入信息

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace AttributeStudy

{

[MyCustum]

[MyCustum(CourseId = 3000)] //一般不会这样做，因为一般只添加一个同样的特性就够用了

[MyCustum(CourseId = 20000,Comment = "常老师")]

//这个特性就起到了为Teacher类起别名MyTeachers的作用

[TableName("MyTeachers")]

class Teacher

{

public int TeacherID { get; set; }

public string TeacherName { get; set; }

public int Salary { get; set; }

}

//这个特性是一个非常标准的用来增加标识的特性

//这种特性我们一般都会让其有一个bool类型的属性

public class PrimaryKeyAttribute : Attribute

{

public bool isPKey { get; set; } = true;

}

//特性作用二：注入信息，如此处的添加别名

//这个特性我的目的是用来为其他对象改别名且不会报错

//下面的AttributeTargets用来限定这个特性只用在类和属性上面

[AttributeUsage(AttributeTargets.Class|AttributeTargets.Property)]

public class TableNameAttribute : Attribute

{

public TableNameAttribute(string tableName)

{

this.TableName = tableName;

}

//为了安全性限定一下这个属性只读（只在构造函数中可写）

public string TableName { get; }

}

}

获取到特性的信息：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace AttributeStudy

{

public class AttributeUsage1

{

public void Test()

{

Teacher teacher = new Teacher();

DisplayClassAttribute(teacher);

string a = GetDBTableName(teacher);

Console.WriteLine(a);

}

//泛型方法，参数T为一个属于class的泛型

//作用为获取类的特性信息，此处获取的是MyCustomAttribute(自建)特性中的属性

public void DisplayClassAttribute<T>(T model)where T : class

{

Type type = typeof(T);

if (type.IsDefined(typeof(MyCustomAttribute),true))

{

var myCustomAttribute = type.GetCustomAttributes(typeof(MyCustomAttribute),true);

foreach (var item in myCustomAttribute)

{

MyCustomAttribute attribute = item as MyCustomAttribute;

if (attribute != null)

{

Console.WriteLine($"课程编号={attribute.CourseId} {attribute.Comment}");

}

}

}

}

//获取前面那个表名称特性的信息

public string GetDBTableName<T>(T model) where T : class

{

Type type = typeof(T);

if (type.IsDefined(typeof(TableNameAttribute), true))

{

var myCustomAttribute = type.GetCustomAttributes(typeof(TableNameAttribute),true);

return ((TableNameAttribute)myCustomAttribute[0]).TableName;

}

else

{

return type.Name;

}

}

}

}

**特性的应用二：**

将特性应用到枚举当中：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Reflection;

using System.Text;

using System.Threading.Tasks;

namespace AttributeStudy

{

//开发中，我们常用到做一些状态的存储

//比如在电商的订单中，订单状态可能为已加购，未付款，已付款，未发货等这样的状态

class EnumAndAttribute

{

}

//订单状态枚举

public enum OrderStatus

{

[Description("未付款")]

Unpaid = 0,

[Description("已付款")]

AlreadyPaid = 1,

[Description("已发货")]

Delivery = 2,

[Description("已确认")]

Confirm = 3

}

//订单状态描述特性

//使用特性来进行状态描述的话就相当于不用写死当前的订单状态，我们只需要改变特性的值就能起到改变订单状态的效果

[AttributeUsage(AttributeTargets.Enum|AttributeTargets.Field)]

public class DescriptionAttribute : Attribute

{

public DescriptionAttribute(string desc)

{

Description = desc;

}

public string Description { get; }

}

public static class MyEnumExtend

{

//获取当前传递的枚举数据

public static string GetOrderStatus(this Enum e)

{

FieldInfo field = e.GetType().GetField(e.ToString());

if (field.IsDefined(typeof(DescriptionAttribute), true))

{

DescriptionAttribute attribute = (DescriptionAttribute)field.GetCustomAttribute(typeof(DescriptionAttribute), true);

return attribute.Description;

}

else

{

return e.ToString();

}

}

}

public class Order

{

public int OrderID { get; set; }

public DateTime OrderTime { get; set; }

public OrderStatus Status { get; set; }

}

}

class AttributeUsage2

{

public void Test()

{

Order order = new Order { OrderID = 100000, Status = OrderStatus.AlreadyPaid };

Console.WriteLine(order.Status.GetOrderStatus());

}

}

**特性的应用三：**

使用特性验证数据是否合规

//第一个类：

//员工信息的实体类，在此添加特性验证相关的员工信息是否是符合我们所需的规范

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace AttributeStudy

{

//员工信息实体类

class Employee

{

[DisplayName("员工编号")]

public int EmployeeID { get; set; }

[DisplayName("员工姓名")]

[Required]

public string EmployeeName { get; set; }

[Range(20,35)]

[Required]

[DisplayName("员工年龄")]

public int Age { get; set; }

public DateTime DateOfBirth { get; set; }

public string Email { get; set; }

}

}

//第二个类：

//在该类中设定一个方法用以测试是否是合规数据

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace AttributeStudy

{

public class AttributeUsage3

{

public void Test()

{

Employee e1 = new Employee

{

EmployeeID = 1000,

Age = 28,

DateOfBirth = Convert.ToDateTime("1990 - 12 - 02"),

Email = "193663@163.com",

EmployeeName = "Michael"

};

//调用模型验证类中成员是否合法

bool res1 = ModelValidate.AutoValidate(e1);

Console.WriteLine(res1);

}

}

}

//第三个类：

//数据验证类，数据验证方法

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Reflection;

namespace AttributeStudy

{

public class AttributeValidate

{

}

//模型验证类

public class ModelValidate

{

public static bool AutoValidate<T>(T model)

{

//获取所有的属性和特性

PropertyInfo[] properties = model.GetType().GetProperties();

//遍历每一个属性并读取特性

foreach (var property in properties)

{

if (property.IsDefined(typeof(CommonValidateAttribute), true))

{

var attributes = property.GetCustomAttributes(typeof(CommonValidateAttribute), true);

foreach (var item in attributes)

{

CommonValidateAttribute attr = item as CommonValidateAttribute;

//验证属性值是否符合要求（多态验证）

if (!attr.BeginValidate(property.GetValue(model)))

{

return false;

}

}

}

//看看各属性有无添加DisplayName的特性，有的话就输出信息

if (property.IsDefined(typeof(DisplayNameAttribute), true))

{

var attribute = property.GetCustomAttribute<DisplayNameAttribute>();

Console.WriteLine(attribute.DisplayName);

}

}

return true;

}

}

#region 设计需要验证的特性

//属性显示的名称

public class DisplayNameAttribute:Attribute

{

public string DisplayName { get; set; }

public DisplayNameAttribute(string name)

{

this.DisplayName = name;

}

}

//抽象一个验证类(特性)的父类：每个验证都是不一样的，但是都是需要调用验证方法的

public abstract class CommonValidateAttribute : Attribute

{

public abstract bool BeginValidate(object dataValue);

}

//几个验证数据格式是否满足要求的特性

public class RequiredAttribute : CommonValidateAttribute

{

public override bool BeginValidate(object dataValue)

{

return dataValue != null && dataValue.ToString().Length != 0;

}

}

[AttributeUsage(AttributeTargets.Field|AttributeTargets.Property)]

public class RangeAttribute : CommonValidateAttribute

{

private int min = 0;

private int max = 0;

public RangeAttribute(int min, int max)

{

this.min = min;

this.max = max;

}

public override bool BeginValidate(object dataValue)

{

return this.min < (int)dataValue && (int)dataValue < this.max;

}

}

#endregion

}

//入口函数进行调用：

public static void Main()

{

AttributeUsage3 a3 = new AttributeUsage3();

a3.Test();

}

out:

True

**特性总结：**

特性可以应用于类、方法、属性、字段和其他代码元素

特性其实就是一个类，直接或间接的继承于Attribute类

自定义类时约定俗成以Attribute为后缀，调用时就可以省略掉了

使用反射机制可以在运行时访问应用于代码元素的特性

**反射（Reflection）：**

反射是指程序可以访问、检测和修改它本身状态或行为的一种能力。

可以使用反射动态地创建类型的实例，将类型绑定到现有对象，或从现有对象中获取到类型。然后调用类型的方法或访问其字段和属性。

**反射作用一：查看元数据**

在前一章学到了，特性本质上就是元数据，而使用反射整好能够查看元数据，因此可以使用反射（Reflection）查看特性（Attribute）信息。

using System;

using System.Reflection;

[AttributeUsage(AttributeTargets.All)]

public class HelpAttribute : Attribute

{

public readonly string Url;

public string str = "66666";

//Topic是一个命名参数

public string Topic

{

get { return topic; }

set { topic = value; }

}

public HelpAttribute(string url)

{

this.Url = url;

}

private string topic;

}

[Help("Information on the class MyClass")]

[Serializable]

class MyClass

{

}

namespace ReflectionStudy

{

class Program

{

static void Main(string[] args)

{

MemberInfo info = typeof(MyClass);

object[] attributes = info.GetCustomAttributes(true);

for (int i = 0; i < attributes.Length; i++)

{

Console.WriteLine(attributes[i]);

}

//用来提取出特性中的属性

HelpAttribute helpAttribute = (HelpAttribute)info.GetCustomAttribute(typeof(HelpAttribute));

Console.WriteLine(helpAttribute.str);

}

}

}

out：

System.SerializableAttribute

HelpAttribute

66666

**实例二：**

在本实例中，我们通过创建DebugInfo特性，并使用反射来读取Rectangle类中的元数据：

using System;

using System.Reflection;

namespace ReflectionStudy2

{

[AttributeUsage(AttributeTargets.Class|AttributeTargets.Constructor|

AttributeTargets.Field| AttributeTargets.Method|

AttributeTargets.Property,AllowMultiple =true)]

public class DebugInfo : Attribute

{

private int bugNo;

private string developer;

private string lastReview;

private string message;

public DebugInfo(int bg, string dev, string d)

{

this.bugNo = bg;

this.developer = dev;

this.lastReview = d;

}

public int BugNo

{

get { return bugNo; }

}

public string Developer

{

get { return developer; }

}

public string LastReview

{

get { return lastReview; }

}

public string Message

{

get { return message; }

set { message = value; }

}

}

[DebugInfo(45,"Zara Ali","12/8/2012",

Message = "return type mismatch")]

[DebugInfo(49,"Nuha Ali","10/10/2012",

Message = "Unused variable")]

class Rectangle

{

protected double length;

protected double width;

public Rectangle(double l,double w)

{

length = l;

width = w;

}

[DebugInfo(55, "Zara Ali", "19/10/2012",

Message = "Return type mismatch")]

public double GetArea()

{

return length \* width;

}

[DebugInfo(56, "Zara Ali", "19/10/2012")]

public void Display()

{

Console.WriteLine("Length:{0}",length);

Console.WriteLine("Width:{0}",width);

Console.WriteLine("Area:{0}",GetArea());

}

}

class ExecuteRectangle

{

static void Main()

{

Rectangle r = new Rectangle(4.5,7.5);

r.Display();

Type type = typeof(Rectangle);

//遍历Rectangle类的所有特性

foreach (Object attributes in type.GetCustomAttributes(false))

{

DebugInfo dbi = (DebugInfo)attributes;

if (null != dbi)

{

Console.WriteLine("Bug no:{0}",dbi.BugNo);

Console.WriteLine("Developer:{0}",dbi.Developer);

Console.WriteLine("Last Reviewed:{0}",dbi.LastReview);

Console.WriteLine("Remarks:{0}",dbi.Message);

}

}

//使用反射来得到type的所有方法，从而得到方法的特性

foreach (MethodInfo m in type.GetMethods())

{

foreach (Attribute a in m.GetCustomAttributes(true))

{

DebugInfo dbi = a as DebugInfo;

if (null != dbi)

{

Console.WriteLine("Bug no:{0},for Method:{1}", dbi.BugNo,m.Name);

Console.WriteLine("Developer:{0}", dbi.Developer);

Console.WriteLine("Last Reviewed:{0}", dbi.LastReview);

Console.WriteLine("Remarks:{0}", dbi.Message);

}

}

}

}

}

}

out：

Length:4.5

Width:7.5

Area:33.75

Bug no:45

Developer:Zara Ali

Last Reviewed:12/8/2012

Remarks:return type mismatch

Bug no:49

Developer:Nuha Ali

Last Reviewed:10/10/2012

Remarks:Unused variable

Bug no:55,for Method:GetArea

Developer:Zara Ali

Last Reviewed:19/10/2012

Remarks:Return type mismatch

Bug no:56,for Method:Display

Developer:Zara Ali

Last Reviewed:19/10/2012

Remarks:

如上例所见，我们可以可以向目标对象添加调试信息（比如在类和方法上添加Bug编号、开发者、最后一次审核日期、消息等信息），能够帮助我们在程序运行过程中进行调试和追踪。

**总结：**

优点：

提高了程序的灵活性和扩展性

降低耦合性，提高自适应能力

允许程序创建和控制任何类的对象

缺点：

性能降低

模糊程序内部逻辑

反射的用途：

允许在运行时查看特性的信息

**C#属性（Property）：**

属性（Property）是类、结构、接口的命名成员。类或结构中的成员变量或方法称为**域（Field）**。属性是域的扩展，且可使用相同的语法来进行访问。他们使用**访问器（accessors）**让私有域的值可被读写或操作。

属性不会确定存储位置。相反，他们具有可读写或计算他们值的**访问器**。

例如，有一个Student类，带有age、name和code的私有域。我们不能在类外直接访问这些域，但是我们可以拥有访问这些私有域的属性。

**访问器：**

属性的访问器有助于获取或设置属性的可执行语句。访问器声明可包含一个get访问器，一个set访问器或同时包含二者，例如：

public int Age

{

get{return age;}

set{age = value;}

}

//缩写形式，但请注意，缩写与全写并不完全等价

//缩写形式当不涉及其他操作，一个类中本来就有相关变量，对其进行属性缩写完全没问题

//但若像下面抽象属性中的例子那样，如果我还缩写，那么就会出现问题

//缩写形式下，当我不对Name属性赋值，那么在我输出s的信息时，Name的get得到的就是null

//反之，若我写全形式，那么当我不对Name属性赋值时输出s的信息，Name的get得到的就是Student类下的初值“N.A”了而不是null

public int Age { get; set; }

实例：

using System;

namespace PropertyStudy

{

class Student

{

private string code = "N.A";

private string name = "not known";

private int age = 0;

//声明code属性来获取和设置成员变量code的值

public string Code

{

get { return code; }

set { code = value; }

}

//声明Name属性来获取和设置成员变量name的值

public string Name { get; set; }

//声明Age属性来获取和设置成员变量age的值

public int Age { get; set; }

public override string ToString()

{

return "Code = " + Code + ",Name = " + Name + ",Age = " + Age;

}

}

class Program

{

static void Main(string[] args)

{

Student s = new Student();

s.Code = "001";

s.Name = "Michael";

s.Age = 19;

Console.WriteLine("Student Info:{0}",s);

s.Age += 1;

Console.WriteLine("Student Info:{0}",s);

}

}

}

out：

Student Info:Code = 001,Name = Michael,Age = 19

Student Info:Code = 001,Name = Michael,Age = 20

**抽象属性：**

抽象类可以有抽象属性，这些属性应该在派生类中被实现，实例：

using System;

namespace PropertyStudy

{

public abstract class Person

{

public abstract string Name { get; set; }

public abstract int Age { get; set; }

}

class Student : Person

{

private string code = "N.A";

private string name = "N.A";

private int age = 0;

public string Code { get; set; }

public override string Name { get { return name; } set { name = value; } }

public override int Age { get; set; }

public override string ToString()

{

return "Code = " + Code + ", Name = " + Name + ", Age = " + Age;

}

}

class Program

{

public static void Main()

{

Student s = new Student();

s.Code = "001";

//s.Name = "Michael";

s.Age = 19;

Console.WriteLine("Student Info:{0}",s);

s.Age++;

Console.WriteLine("Student Info:{0}", s);

}

}

}

out：

Student Info:Code = 001, Name = N.A, Age = 19

Student Info:Code = 001, Name = N.A, Age = 20

**C#索引器（Indexer）：**

**索引器**允许一个对象可以像数组一样使用下标来进行访问。

当你为类定义一个索引器时，该类的行为就会像一个虚拟数组一样，可以使用数组访问运算符[]进行访问该类的成员。

**索引器的用途：**

索引器的行为的声明在某种程度上类似于属性。就像属性，可以使用get和set访问器来定义索引器。但是，属性返回或设置一个特定的数据成员，而索引器返回或设置对象实例的一个特定值。换句话说，它把实例数据分为更小的部分，并索引每个部分，获取或设置每个部分。

定义一个属性包括提供属性名称。索引器的定义不带有名称而使用**this**关键字，它指向对象实例。实例：

using System;

namespace Indedxer

{

class IndexNames

{

static public int size = 10;

private string[] namelist = new string[size];

public IndexNames()

{

for (int i = 0; i < size; i++)

{

namelist[i] = "N.A";

}

}

public string this[int index]

{

get

{

string tmp;

if (index >= 0 && index <= size - 1)

{

tmp = namelist[index];

}

else

{

tmp = "";

}

return tmp;

}

set

{

if (index >= 0 && index <= size - 1)

{

namelist[index] = value;

}

}

}

static void Main(string[] args)

{

IndexNames names = new IndexNames();

names[0] = "Zara";

names[1] = "Riz";

names[2] = "Nuha";

names[3] = "Asif";

names[4] = "Sunil";

names[5] = "Michael";

for (int i = 0; i < IndexNames.size; i++)

{

Console.WriteLine(names[i]);

}

}

}

}

out：

Zara

Riz

Nuha

Asif

Sunil

Michael

N.A

N.A

N.A

N.A

**重载索引器：**

索引器可以被重载，索引器声明的时候也可以带有多个参数，且每个参数可以是不同的类型。没有必要让索引器必须是整形的。C#允许索引器可以是其他类型如字符串等。

实例：

using System;

namespace Indedxer

{

class IndexNames

{

private string[] namelist = new string[size];

public static int size = 10;

public IndexNames()

{

for (int i = 0; i < size; i++)

{

namelist[i] = "N.A";

}

}

//数字索引

public string this[int index]

{

get

{

string tmp;

if (index >= 0 && index < size)

{

tmp = namelist[index];

}

else

{

tmp = "";

}

return tmp;

}

set

{

if (index >= 0 && index < size)

{

namelist[index] = value;

}

}

}

//字符串索引

public int this[string name]

{

get

{

int index = 0;

while (index < size)

{

if (namelist[index] == name)

{

return index;

}

index++;

}

return index;

}

}

static void Main(string[] args)

{

IndexNames names = new IndexNames();

names[0] = "Zara";

names[1] = "Riz";

names[2] = "Nuha";

names[3] = "Asif";

names[4] = "Sunil";

names[5] = "Michael";

//使用带有int参数的第一个索引器

for (int i = 0; i < IndexNames.size; i++)

{

Console.WriteLine(names[i]);

}

Console.WriteLine();

//使用带有string参数的第二个索引器

Console.WriteLine(names["Michael"]);

}

}

}

out：

Zara

Riz

Nuha

Asif

Sunil

Michael

N.A

N.A

N.A

N.A

5

**C#委托（delegate）：**

C#中的委托类似于C或C++中**函数的指针**。委托是存有对某个方法的引用的一种**引用类型变量**。引用可以在运行时被改变。

委托特别用于实现事件和回调方法。所有的委托都派生自System.Delegate类。

**声明与实例化委托：**

委托声明决定了可由委托引用的方法。委托可以指向一个与其具有相同标签的方法。例如，声明一个委托：

public delegate int MyDelegate(string s);

上面的委托可用于引用任何一个带有单一string参数、返回值为int类型的方法。

一旦声明了委托类型后，当创建委托时，传递到new语句的参数就像方法调用一样书写，但是不带参数，例如：

public delegate void printString(string s);

//后面的方法后不用加括号哦

printString ps1 = new printString(WriteToScreen);

printString ps2 = new printString(WriteToFile);

下面的实例演示一个委托的声明、实例化和使用，该委托可用于引用带有一个整形参数的方法，并返回一个整型值：

using System;

//声明委托

delegate int NumberChanger(int n);

namespace DelegateStudy

{

class TestDelegate

{

static int num = 10;

public static int AddNum(int p)

{

num += p;

return num;

}

public static int MultNum(int q)

{

num \*= q;

return num;

}

public static int GetNum()

{

return num;

}

static void Main(string[] args)

{

//实例化委托

NumberChanger nc1 = new NumberChanger(AddNum);

NumberChanger nc2 = new NumberChanger(MultNum);

//与直接使用AddNum(25)效果是一样的

nc1(25);

Console.WriteLine("Value of num is {0}",GetNum());

//与直接使用MultNum(5)效果是一样的

nc2(5);

Console.WriteLine("Value of num is {0}", GetNum());

}

}

}

**委托的多播：**

委托对象可以使用“+”运算符进行合并。一个合并委托调用他所合并的两个委托。只有相同类型的委托才可以被合并。“-”运算符可用于从合并委托中移除组件委托。

使用委托这个有用的特点，可以创建一个委托被调用时要调用的方法的调用列表，这样我们就能使用一个委托来实现一系列方法的调用（比如我们有时要对一个对象先后执行多个方法，我们将那些方法都存在委托中，那么我们只需要调用一个委托就能实现顺序执行多个方法，很方便）。这被称为委托的**多播**（也叫组播）。实例：

using System;

//声明委托

delegate int NumberChanger(int n);

namespace DelegateStudy

{

class TestDelegate

{

static int num = 10;

public static int AddNum(int p)

{

num += p;

return num;

}

public static int MultNum(int q)

{

num \*= q;

return num;

}

public static int GetNum()

{

return num;

}

static void Main()

{

NumberChanger nc;

NumberChanger nc1 = new NumberChanger(AddNum);

NumberChanger nc2 = new NumberChanger(MultNum);

//必须先把委托等于一个后再去使用+，因为必须要有顺序性

nc = nc1;

nc += nc2;

//调用多播,下面这个就相当于分两次调用：

//先调用AddNum(5)再调用MultNum(5)

nc(5);

Console.WriteLine("Value of num is {0}",GetNum());

}

}

}

**委托的用途：**

除了上面的多播外，委托还可被用于将委托作为参数传递给方法，这样做的优点是，可以只写一个方法，但能实现多种功能，如下例中，我们只写了一个sendString方法，却能实现控制台打印和文件写入两种功能，实例：

using System;

using System.IO;

namespace DelegateStudy

{

class PrintString

{

static FileStream fs;

static StreamWriter sw;

public delegate void printString(string s);

//打印到控制台

public static void WriteToConsole(string str)

{

Console.WriteLine("The string is {0}",str);

}

//写入到文件

public static void WriteToFile(string s)

{

fs = new FileStream("d:\\message.txt",FileMode.Append,FileAccess.Write);

sw = new StreamWriter(fs);

sw.Write(s);

//将缓冲区数据写入并刷新缓冲区

sw.Flush();

sw.Close();

fs.Close();

}

//该方法将委托作为参数，并使用委托来调用方法

public static void sendString(printString ps)

{

ps("hello world!");

}

static void Main()

{

printString ps1 = new printString(WriteToConsole);

printString ps2 = new printString(WriteToFile);

sendString(ps1);

sendString(ps2);

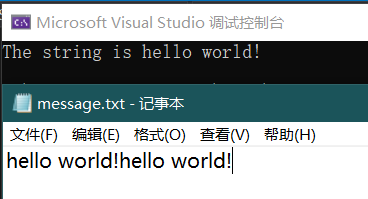
sendString(ps2);

}

}

}

结果输出：



**C#事件：**

**事件**基本上说是一个用户操作，如按键、点击、鼠标移动等等，或是一些提示信息，如系统生成的通知。应用程序需要在事件发生时响应事件，例如中断等。

C#中使用事件机制实现**线程间的通信。**

事件在类中声明且生成，且通过使用同一个类或其他类中的委托与事件处理程序关联。包含事件的类用于发布事件。被称为**发布器类**。其他接受该事件的类被称为**订阅器类**。事件使用发布-订阅模型。

**发布器**是一个包含事件和委托定义的对象。事件和委托之间的联系也定义在这个对象中。发布器类的对象调用这个事件，并通知给其他的对象。

**订阅器**是一个接收事件并提供事件处理程序的对象。在发布器类中的委托调用订阅器类中的方法（事件处理程序）。

**声明事件：**

在类的内部声明事件，首先必须声明该事件的委托类型。如：

public delegate void BoilerLogHandler(string status);

然后声明事件本身，使用**event**关键字：

//基于上面的委托定义事件

public event BoilerLogHandler BoilerEventLog;

上面的代码定义了一个名为BoilerLogHandler的委托和一个名为BoilerEventLog的事件。

**实例一：**

using System;

namespace EventStudy1

{

using System;

//publisher发布器类

public class EventTest

{

private int value;

//定义一个委托并将该委托作为事件的条件

public delegate void NumManipulationHandler();

public event NumManipulationHandler ChangeNum;

protected virtual void OnNumChanged()

{

if (ChangeNum != null) //表示事件已经被注册（即有订阅者了）

{

ChangeNum(); //事件被触发

}

else

{

Console.WriteLine("event not fire");

}

}

public EventTest()

{

int n = 5;

SetValue(n);

}

public void SetValue(int n)

{

if (value != n)

{

value = n;

OnNumChanged();

}

}

}

//subscriber订阅器类

public class subscriberEvent

{

public void printf()

{

Console.WriteLine("event fire!");

}

}

//触发

class Program

{

public static void Main()

{

EventTest e = new EventTest();

subscriberEvent v = new subscriberEvent();

e.ChangeNum += new EventTest.NumManipulationHandler(v.printf);//注册事件（给事件添加上委托）

e.SetValue(7);

e.SetValue(11);

}

}

}

out：

event not fire

event fire!

event fire!

**实例二：**

在本实例中，提供一个简单的用于热水锅炉系统故障排除的应用程序。当维修工程师检查锅炉时，锅炉的温度和压力会随着维修工程师的备注自动记录到日志文件中：

using System;

using System.IO;

namespace EventStudy1

{

//boiler类

class Boiler

{

private int temp;

private int pressure;

public Boiler(int t, int p)

{

temp = t;

pressure = p;

}

public int getTemp()

{

return temp;

}

public int getPressure()

{

return pressure;

}

}

//publisher事件发布类

class DelegateBoilerEvent

{

public delegate void BoilerLogHandler(string status);

//基于上面的委托定义事件

public event BoilerLogHandler BoilerEventLog;

public void LogProcess()

{

string remarks = "O.K";

Boiler b = new Boiler(100,12);

int t = b.getTemp();

int p = b.getPressure();

if (t > 150 || t < 80 || p < 12 || p > 15)

{

remarks = "Need Maintenance";

}

OnBoilerEventLog("Logging Info:");

OnBoilerEventLog("Temparature: " + t + "\nPressure: " + p);

OnBoilerEventLog("Message: "+ remarks+"\n");

}

public void OnBoilerEventLog(string message)

{

if (BoilerEventLog != null)

{

BoilerEventLog(message);

}

}

}

//该类保留写入日志文件的条款

class BoilerInfoLogger

{

FileStream fs;

StreamWriter sw;

public BoilerInfoLogger(string filename)

{

fs = new FileStream(filename, FileMode.Append, FileAccess.Write);

sw = new StreamWriter(fs);

}

public void Logger(string info)

{

sw.WriteLine(info);

}

public void Close()

{

sw.Close();

fs.Close();

}

}

//事件订阅器

public class RecordBoilerInfo

{

static void Logger(string info)

{

Console.WriteLine(info);

}

static void Main()

{

BoilerInfoLogger filelog = new BoilerInfoLogger("d:\\boiler.txt");

DelegateBoilerEvent boilerEvent = new DelegateBoilerEvent();

boilerEvent.BoilerEventLog += new DelegateBoilerEvent.BoilerLogHandler(Logger);

boilerEvent.BoilerEventLog += new DelegateBoilerEvent.BoilerLogHandler(filelog.Logger);

boilerEvent.LogProcess();

filelog.Close();

}

}

}

out：

Logging Info:

Temparature: 100

Pressure: 12

Message: O.K

在上面例子中可以非常清晰的了解到，当我们事件有了订阅者之后，只要当我们调用事件的时候，所有的订阅者都会收到消息，从而进行相关的操作。

**实例三：**

在本实例中，学习事件的触发顺序：

using System;

namespace EventStudy1

{

//事件发布器类

class DelegateTest

{

public delegate void delegate\_tz();

public event delegate\_tz delegate\_tz0;

public static void get()

{

Console.WriteLine("这是触发事件的第一个方法，此方法在事件发布类中。");

}

public void enter()

{

Console.WriteLine("开始调用事件方法：");

delegate\_tz0();

}

}

//事件订阅器类

class Test

{

public Test() { }

public void tryEvent()

{

Console.WriteLine("这是触发事件的第三个方法，在订阅器中，这才是正宗的订阅器类。");

}

}

class Program

{

static void Main()

{

DelegateTest delegateTest0 = new DelegateTest();

//在主体函数中依据需求组装事件，组装过程类似于委托多播

delegateTest0.delegate\_tz0 += new DelegateTest.delegate\_tz(DelegateTest.get);

delegateTest0.delegate\_tz0 += new DelegateTest.delegate\_tz(method1);

delegateTest0.delegate\_tz0 += new DelegateTest.delegate\_tz(new Test().tryEvent);

delegateTest0.enter();//执行操作，以触发事件，上面的三个+=只是订阅，但还没有调用啊

}

static public void method1()

{

Console.WriteLine("这是触发事件的第二个方法，在主体Main函数中");

}

}

}

out：

开始调用事件方法：

这是触发事件的第一个方法，此方法在事件发布类中。

这是触发事件的第二个方法，在主体Main函数中

这是触发事件的第三个方法，在订阅器中，这才是正宗的订阅器类。

**总结：**

首先，提出一个例子，方便下面的总结：

有一个Player类，当Player类中的player对象死亡后，会给UI处理类一个消息，从而UI进行一些UI变化表明对象狗带了。在这个例子中，Player类就是发布器类，UI类就是订阅器类。

事件的优点：

高内聚，低耦合（比如player对象死亡之后他就不管死亡之后的事情了，别的类中的对象需要干嘛是别的类的事情）

安全性（加了事件之后，其他类就只能订阅+=和取消订阅-=，而不能对事件的拥有者再去做其他事情了，比如Player类是狗带事件的拥有者，那么其他地方只能通过订阅狗带事件知道狗带了，然后做一些操作，但不能对Player类进行任何操作）

可扩展性，事件可以提高代码的可扩展性，因为可以轻松地添加新的事件处理程序而无需修改原有的代码。（除了例子中的UI类需要对player狗带做一些反应，可能其他类也需要做一些反应，那直接在事件中添加指定类需要做的方法就行了）

**C#集合（Collection）：**

集合类是专门用于数据存储和检索的类。这些类提供了对栈、队列、列表和哈希表的支持。

集合类服务于不同的目的，如为元素动态分配内存，基于索引访问列表项等。这些类创建Object类的对象的集合。在C#中，Object类是所有数据类的基类。

**各种集合类和他们的用法：**

下面是各种常用的**System.Collection**命名空间的类：



**动态数组（ArrayList）：**

其实就是一个特殊的数组，它的大小不固定，当你使用Add成员方法对其进行添加，若数组越界时他就会扩充数组的大小（0->4->8->16），实例：

using System;

using System.Collections;

namespace CollectionStudy

{

class Program

{

static void Main(string[] args)

{

//声明动态数组，需要添加System.Collection头文件

ArrayList al = new ArrayList();

Console.WriteLine("Add numbers");

al.Add(45);//Add方法是在动态数组末尾添加一个对象

al.Add(78);

al.Add(33);

al.Add(56);

al.Add(12);

al.Add(29);

al.Add(66);

Console.WriteLine("Capacity: {0}",al.Capacity);//"Capacity"属性获取或设置ArrayList可以包含的元素个数

Console.WriteLine("Count: {0}",al.Count);//"Count"属性表示实际包含的元素个数

Console.WriteLine();

foreach (int l in al)

{

Console.Write(l+" ");

}

al.Sort(); //排序动态数组

Console.WriteLine();

foreach (int l in al)

{

Console.Write(l + " ");

}

}

}

}

out：

Add numbers

Capacity: 8

Count: 7

45 78 33 56 12 29 66

12 29 33 45 56 66 78

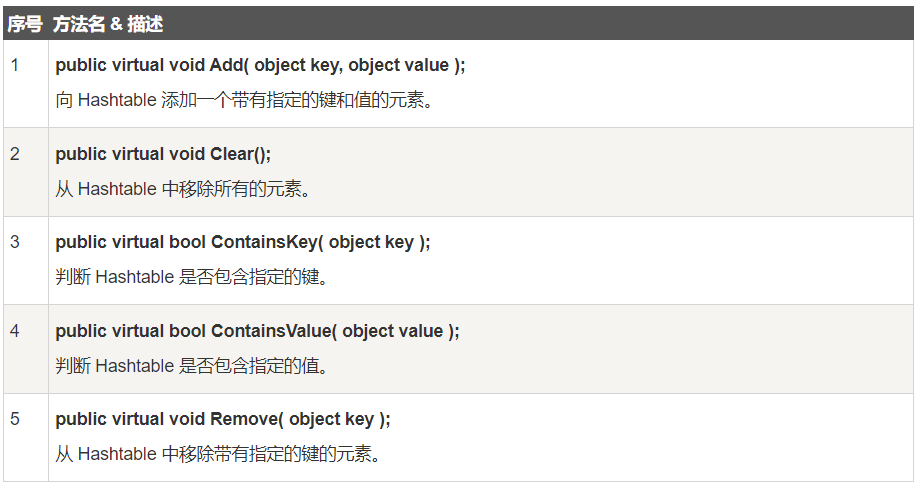
**哈希表：**

Hashtable类代表了一系列基于键的哈希代码组织起来的**键/值对**，他用**键**来访问集合中的元素。

当使用**键**访问元素时，则使用哈希表，哈希表中的每一项都一个**键/值对**。键用于访问集合中的项目。

常用属性和方法：





实例：

using System;

using System.Collections;

namespace CollectionStudy

{

class Program

{

//哈希表Hashtable

static void Main()

{

Hashtable ht = new Hashtable();

//前键后值

ht.Add("001","Zara");

ht.Add("002","Mike");

ht.Add("003","Sara");

ht.Add("004","Zim");

ht.Add("005","Tim");

ht.Add("006","Tom");

ht.Add("007","Aix");

if (ht.ContainsValue("Zara"))

{

Console.WriteLine("This student is already in the list");

}

else

{

ht.Add("008","Zara");

}

var key = ht.Keys;//获取所有键的集合

foreach (string k in key)

{

//直接用ht[k]的方式可以直接访问到值

Console.Write(k+"：" +ht[k]+" ");

//输出是无序的，并不是按照开始时的Add的顺序输出

}

}

}

}

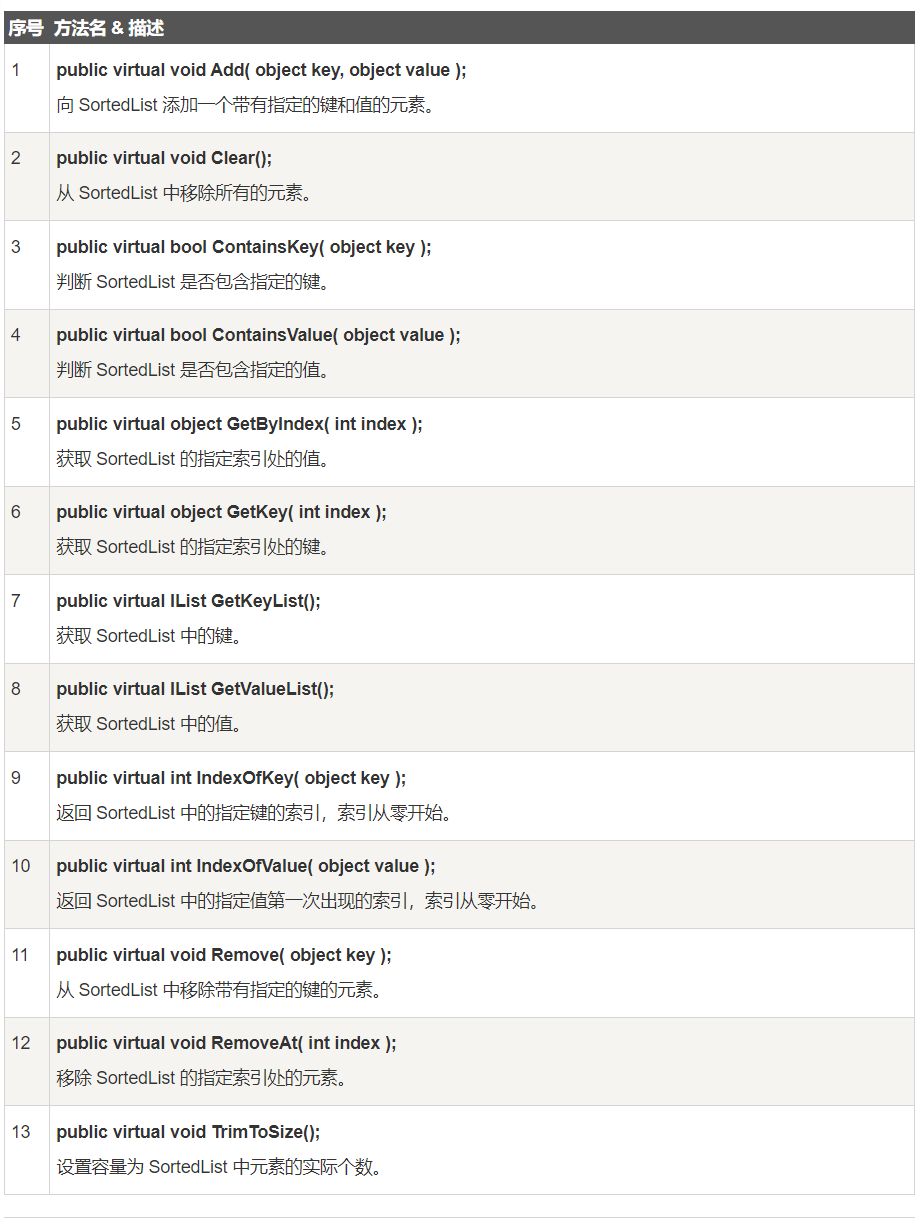
**排序列表（SortedList）：**

排序列表SortedList代表一系列按照键来排序的键/值对，这些键值对可以通过键和索引来访问。

排序列表是数组和哈希表的结合。它包含一个可使用键或索引访问各项的列表。如果使用索引访问各项，则他是一个动态数组（ArrayList），若使用键访问各项，则他就是一个哈希表（Hashtable）。集合中的各项总是按照键值排序的。与哈希表最大的区别也就是在此，一个有序一个无序。

常用的方法与属性也与哈希表类似





实例：

using System;

using System.Collections;

namespace CollectionStudy

{

class Program

{

//SortedList排序列表

static void Main()

{

SortedList sl = new SortedList();

sl.Add("001", "Zara");

sl.Add("002", "Abida");

sl.Add("004", "Mausam");

sl.Add("005", "Amlan");

sl.Add("006", "Zoe");

sl.Add("007", "Zed");

sl.Add("003", "Joe");//虽然是后加的，但仍然会放在第三位

if (sl.ContainsValue("Nuha"))

{

Console.WriteLine("already in list");

}

else

{

sl.Add("008","Nuha");

}

//获取键的集合

var key = sl.Keys;

foreach (var k in key)

{

Console.WriteLine(k+":"+sl[k]);

}

}

}

}

out：

001:Zara

002:Abida

003:Joe

004:Mausam

005:Amlan

006:Zoe

007:Zed

008:Nuha

**栈（Stack）：**

栈代表了一个**后进先出**的对象集合。当需要对各项进行后进先出的访问时可以使用栈。在列表中添加一项称为入栈（Push），反之称为出栈（Pop）。

常用的属性与方法：





实例：

using System;

using System.Collections;

namespace CollectionStudy

{

class Program

{

//栈Stack

static void Main()

{

Stack st = new Stack();

st.Push('A');

st.Push('M');

st.Push('G');

st.Push('W');

Console.WriteLine("Current stack is :");

foreach (var c in st)

{

Console.Write(c+" ");

}

Console.WriteLine();

Console.WriteLine("Remove some values");

st.Pop();

st.Pop();

st.Pop();

Console.WriteLine("Current stack:");

foreach (var c in st)

{

Console.Write(c + " ");

}

}

}

}

out:

Current stack is :

W G M A

Remove some values

Current stack:

A

**队列（Queue）：**

队列代表了一个**先进先出**的对象集合。当需要对各项进行先进先出的访问时，则使用队列。在列表中添加一项称为**入队，**反之称为**出队**。

方法与属性：





实例：

using System;

using System.Collections;

namespace CollectionStudy

{

class Program

{

//Queue队列

static void Main()

{

Queue q = new Queue();

q.Enqueue('A');

q.Enqueue('M');

q.Enqueue('G');

q.Enqueue('W');

Console.WriteLine("Current queue:");

foreach (char c in q)

{

Console.Write(c+" ");

}

Console.WriteLine();

q.Enqueue('V');

q.Enqueue('H');

Console.WriteLine("Current queue:");

foreach (char c in q)

{

Console.Write(c + " ");

}

Console.WriteLine();

Console.WriteLine("Remove some values");

char ch = (char)q.Dequeue();

Console.WriteLine("The remove value is {0}",ch);

Console.WriteLine("Current queue:");

foreach (char c in q)

{

Console.Write(c + " ");

}

}

}

}

out：

Current queue:

A M G W

Current queue:

A M G W V H

Remove some values

The remove value is A

Current queue:

M G W V H

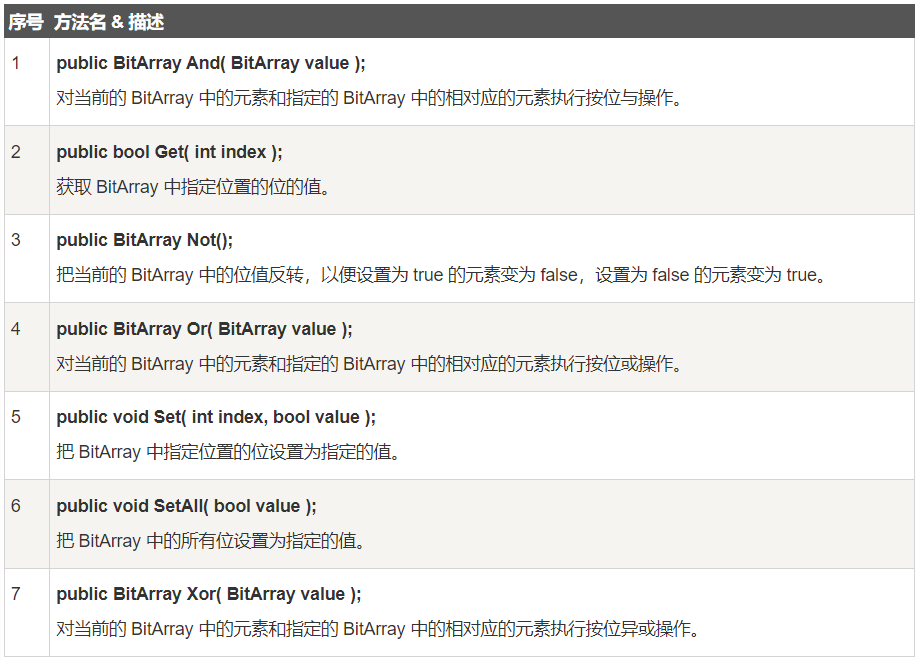
**点阵列（BitArray）：**

BitArray类管理一个紧凑型的位值数组，它使用布尔值表示，其中true表示位是开启的（1），false表示是关闭的（0）。

当需要存储位，但事先不知道位数时，可以使用点阵列。可以使用整型索引从点阵列集合中访问各项，索引从0开始。

常用的属性和方法：





实例：

using System;

using System.Collections;

namespace CollectionStudy

{

class Program

{

//BitArray点阵列

static void Main()

{

//必须要声明点阵列的大小

BitArray ba1 = new BitArray(8);

BitArray ba2 = new BitArray(8);

byte[] a = { 60 };

byte[] b = { 13 };

//把值60和13存储到点阵列中

//过程中会将其转换为二进制，因为bitarray是布尔值存储的位啊

ba1 = new BitArray(a);

ba2 = new BitArray(b);

/\*\*\*注意：存储之后我们的顺序是"反的"，还原为二进制的话得反过来\*\*\*/

//ba1的内容：

Console.WriteLine("Bit Array ba1:60");

for (int i = 0; i < ba1.Count; i++)

{

//-6表示左对齐，占位为6位（字符串结尾会有一个空格，所以False会占6位）

Console.Write("{0,-6}",ba1[i]);

}

Console.WriteLine();

//ba2的内容：

Console.WriteLine("Bit Array ba2:13");

for (int i = 0; i < ba2.Count; i++)

{

Console.Write("{0,-6}", ba2[i]);

}

Console.WriteLine();

BitArray ba3 = new BitArray(8);

ba3 = ba1.And(ba2);//进行与运算

//ba3的内容：

Console.WriteLine("Bit Array ba3 after and operation:12");

for (int i = 0; i < ba3.Count; i++)

{

Console.Write("{0,-6}", ba3[i]);

}

Console.WriteLine();

BitArray ba4 = new BitArray(8);

ba4 = ba1.Or(ba2);//进行与运算

//ba4的内容：

Console.WriteLine("Bit Array ba4 after or operation:61");

for (int i = 0; i < ba4.Count; i++)

{

Console.Write("{0,-6}", ba4[i]);

}

Console.WriteLine();

}

}

}

out：

Bit Array ba1:60

False False True True True True False False

Bit Array ba2:13

True False True True False False False False

Bit Array ba3 after and operation:12

False False True True False False False False

Bit Array ba4 after or operation:61

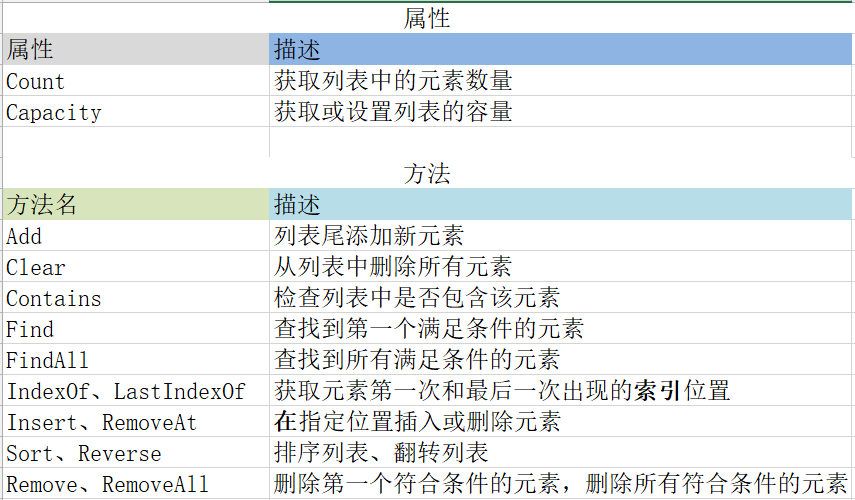
True False True True False False False False

**列表（List）：**

列表List<T>是一种动态泛型数组（也称为**可变大小数组，数据类型是泛型**），它提供了在列表中存储和操作元素的功能。

List通过索引分配，索引与数组一样从0开始，可通过索引[]读取值。

常用的属性和方法：



实例：

using System;

using System.Collections;

using System.Collections.Generic;

using static System.Console;

namespace CollectionStudy

{

class Program

{

//List列表

static void Main()

{

List<int> a = new List<int>();

a.Add(2);

a.Add(6);

a.Add(2);

a.Add(10);

Console.WriteLine($"第一个数为{a[0]}");

a.Remove(2);

a.Sort();

foreach (var temp in a)

{

Console.WriteLine(temp);

}

bool a3 = a.Contains(2);

WriteLine(a3);

}

}

}

out：

第一个数为2

2

6

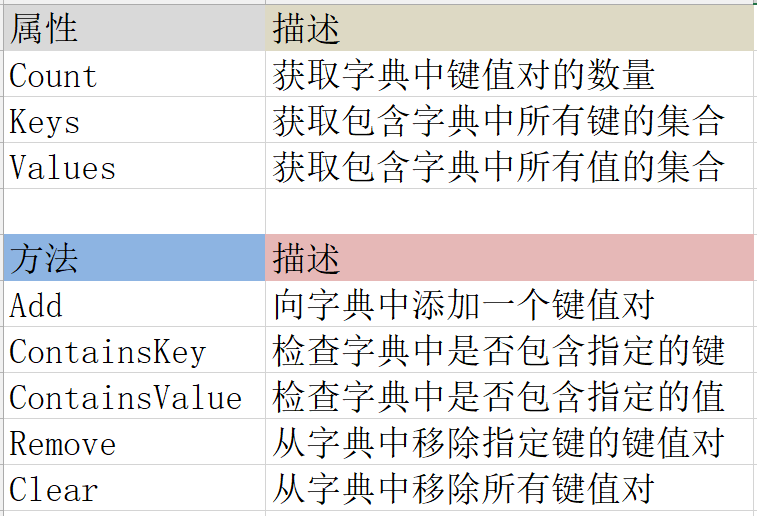
10

True

**字典（Dictionary）：**

字典Dictionary<TKey,TValue>是一个常用的泛型集合类，它提供了一种键值对的存储方式。**每个键都必须是唯一的，并且键不能是空引用。**可以使用键来快速查找和访问对应的值。同样也是使用中括号[]访问字典中的值，但是中括号中的值就不是序号了，而是键。另外，字典也不能排序。

字典的常用属性与方法：



实例：

using System;

using System.Collections;

using System.Collections.Generic;

using static System.Console;

namespace CollectionStudy

{

class Program

{

//Dictionary字典

static void Main()

{

var a = new Dictionary<int, int>();

a.Add(12, 14);

a.Add(0, 1);

a.Add(6, 9);

Console.WriteLine("删去前的Count为"+a.Count);

a.Remove(0);

Console.WriteLine(a[12]);

WriteLine("删去后的Count为" + a.Count);

Console.WriteLine(a.ContainsKey(12));

}

}

}

out：

删去前的Count为3

14

删去后的Count为2

True

**C#泛型：**

泛型允许延迟编写类或方法中的编程元素的数据类型，简言之，就是给出函数体或类体，而不指定数据类型（包括参数类型与返回值类型），直到程序实际使用它的时候才确定具体的数据类型。

泛型允许编写一个可以与任何数据类型一起工作的类或方法，调用时编译器会自动帮我们进行更换为具体类型，实例：

using System;

namespace GenericStudy\_fanxing

{

public class MyGenericArray<T>

{

private T[] array;

public MyGenericArray(int size)

{

array = new T[size + 1];

}

public T getItem(int index)

{

return array[index];

}

public void setItem(int index, T value)

{

array[index] = value;

}

}

class Program

{

static void Main(string[] args)

{

MyGenericArray<int> intArray = new MyGenericArray<int>(5);

//set values

for (int i = 0; i < 5; i++)

{

intArray.setItem(i,i\*5);

}

//get values

for (int i = 0; i < 5; i++)

{

Console.Write(intArray.getItem(i)+" ");

}

Console.WriteLine();

MyGenericArray<char> charArray = new MyGenericArray<char>(5);

//set values

for (int i = 0; i < 5; i++)

{

charArray.setItem(i, (char)(i+97));

}

//get values

for (int i = 0; i < 5; i++)

{

Console.Write(charArray.getItem(i) + " ");

}

Console.WriteLine();

}

}

}

out：

0 5 10 15 20

a b c d e

上面的例子中，使用一个类来创建一种泛型的数组，在上例中创建一个字符数组和整型数组看起来是没有意义，但是如果我们用来创建一个object数组，那么用处就多了。比如，我们使用这个类来存一些点：

...using System.Drawing;

MyGenericArray<object> objArray = new MyGenericArray<object>(5);

for (int i = 0; i < 5; i++)

{

objArray.setItem(i, new Point(0,i\*20));

}

**泛型的特性：**

使用泛型是一种增强程序功能的技术，具体体现在：

最大限度的重用代码、保护类型的安全和提高性能

可以创建泛型集合类，类似于上面我来创建的Point数组。

可以创建泛型接口、泛型类、泛型方法、泛型事件甚至是泛型委托

关于泛型数据类型中使用的类型信息可以在运行时通过反射获得

**泛型方法实例：**

using System;

using System.Drawing;

namespace GenericStudy\_fanxing

{

class Program

{

static void Swap<T>(ref T lhs, ref T rhs)

{

T temp;

temp = lhs;

lhs = rhs;

rhs = temp;

}

static void Main()

{

int a, b;

char c, d;

a = 10;

b = 20;

c = 'I';

d = 'V';

//调用swap

Swap(ref a,ref b);

Swap(ref c,ref d);

Console.WriteLine("changed:a is {0}",a);

Console.WriteLine("changed:b is {0}",b);

Console.WriteLine("changed:c is {0}",c);

Console.WriteLine("changed:d is {0}",d);

}

}

}

out：

changed:a is 20

changed:b is 10

changed:c is V

changed:d is I

泛型方法能够很好的重用代码，因为时常有一些参数类型不一致的东西可能需要做一样的事情。

**泛型委托：**

声明一个返回值类型为T，参数类型为T的委托的声明格式：

delegate T NumberChanger<T>(T n);

使用实例：

using System;

using System.Drawing;

//无论是泛型类、方法还是委托，都用的是<T>这个

delegate T NumberChanger<T>(T n);

namespace GenericStudy\_fanxing

{

class Program

{

static int num = 10;

public static int Addnum(int p)

{

num += p;

return num;

}

public static int Multnum(int p)

{

num \*= p;

return num;

}

public static int Getnum()

{

return num;

}

static void Main()

{

NumberChanger<int> nc1 = new NumberChanger<int>(Addnum);

NumberChanger<int> nc2 = new NumberChanger<int>(Multnum);

nc1(25);

Console.WriteLine("Value of num is {0}",Getnum());

nc2(5);

Console.WriteLine("Value of num is {0}", Getnum());

}

}

}

out：

Value of num is 35

Value of num is 175

使用泛型委托时，返回值与参数值的类型必须一致。

**总结：**

总结一下，首先，在格式上，无论是泛型类、方法、接口还是委托，在声明的时候我们都需要用到<T>这样的格式在我们的标识符的后面。在实例化的时候，都需要在<>中指定具体的格式。

其次，我们可以在声明泛型的时候，给泛型添加一定的约束条件。常用的泛型限定条件：

where T : class:T必须是引用类型

where T : struct:T必须是值类型

where T : new():T必须具有public的无参数构造函数

where T : SomeBaseClass：T必须是指定的基类或其派生类

where T : ISomeInterface：T必须实现指定的接口

where T : U:T必须是U或者其派生类

这些限定条件可以单独也可以组合使用，示例：

public class MyClass<T>where T:class,new()

{ //... }

在上例，MyClass类限定了T必须是引用类型，并且T必须具有公共的无参构造函数。

MyCLass<string> obj = new MyClass<string>();

string是引用类型并且具有无参构造函数，满足条件。

MyClass<int> obj = new MyClass<int>();

编译错误，int是值类型，不满足class约束。

**匿名方法与Lambda表达式：**

匿名方法，顾名思义就是没有方法名（标识符）的方法，他没有显式的名称，通常用于将代码块作为委托参数传递，或者在需要时定义临时的、短小的函数（看起来匿名函数与Lambda表达式是作用一样的）。

匿名方法实例：

//声明一个委托类型

delegate void MyDelegate(int x);

class Program

{

static void Main()

{

//创建一个匿名方法并赋值给委托对象

MyDelegate myDelegate = delegate(int x)

{

Console.WriteLine("匿名方法被调用，参数为："+x);

};

//调用委托对象会执行匿名方法

myDelegate(10);

}

}

我们可以直接将上例换成使用更为简洁和方便的Lambda表达式：

MyDelegate myDelegate = （int x） =>

{

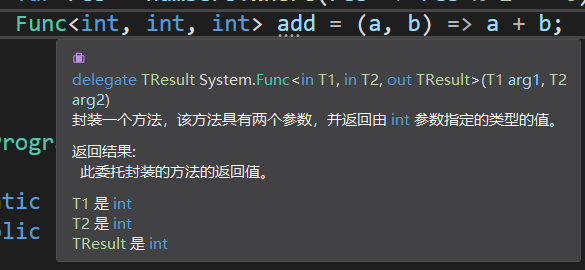
Console.WriteLine("Lambda表达式被调用，参数为："+ x);

};

Lambda表达式的语法（前面的两个int表示参数，后面的那个int表示返回值）：

Func<int,int,int>add = (a,b) => a+b;

int result = add(3,5);



在上例中，我们将Lambda表达式赋值给了一个委托变量add（Func是关键字），然后可以调用add（3,5）来执行Lambda表达式。

根据以上的匿名方法和Lambda表达式的实例，我们其实可以看出来，使用匿名方法和Lambda表达式的目的，主要是两点：

1.简洁性：我们可以不额外单独再写一个函数来引用，而是直接写在匿名方法或者Lambda表达式中，更为简洁。

2.闭包（安全性）：匿名方法和Lambda表达式连函数名都没有，外部不可能访问到，更安全。

**不安全代码（指针）：**

完整讲解：[C#不安全代码](https://www.runoob.com/csharp/csharp-unsafe-codes.html)

为什么C#不推荐使用指针：

1.不安全：使用指针需要在代码块上添加unsafe关键字，顾名思义，不安全。指针不是安全类型，使用完毕后需要程序员自行释放分配的内存，如果不注意很容易造成内存泄漏。

2.与C#设计初衷背离：C#与C语言最大的区别就是具有独立的内存管理机制、垃圾回收器。C#中声明的变量在内存中的存储受垃圾回收器管理，因此有可能一个变量（例如一个庞大的数组）在运行过程中会被移动到内存中的其他位置。指针就是地址，地址都换了，运用指针自然没有意义。

3.可替代：C#中有更方便的东西来替代指针，如泛型、委托，使用远比指针方便，若是取地址也可以用ref关键字替代。

**C#多线程：**

**线程**被定义为程序的执行路径。每个线程都定义了一个独特的控制流。若应用程序涉及到复杂的和耗时的操作，那么设置不同的线程执行路径往往是有益的，每个线程执行特定的工作。

一般使用线程是为了并行操作的实现，使用线程节省了CPU周期的浪费，同时提高了应用程序的效率。

为了同时执行多个任务，我们在编写程序的时候可以划分更小的线程。

**线程生命周期:**

线程的生命周期开始于System.Threading.Thread类的对象被创建时，结束于线程被终止或完成执行时。线程的生命周期中有四种状态：

**未启动状态：**当线程实例被创建但Start方法未被调用时的状况。

**就绪状态：**当线程准备好运行并等待CPU周期时的状况。

**不可运行状态：**下面情况下线程时不可运行的：

已经调用Sleep方法

已经调用Wait方法

通过I/O操作阻塞

**死亡状态：**当线程执行完毕或已中止时的状况。

**主线程：**

在C#中，System.Threading.Thread类用于线程的工作。它允许创建并访问多线程应用程序中的单个线程 。进程中第一个被执行的线程称为**主线程**。当C#程序开始执行时，主线程自动创建。使用Thread类创建的线程被主线程的子线程调用。可以使用Thread类的**CurrentThread**属性访问线程。

主线程的执行实例：

using System;

using System.Threading;

namespace Threading

{

class Program

{

static void Main(string[] args)

{

Thread th = Thread.CurrentThread;

th.Name = "MainThread";

Console.WriteLine("This is {0}",th.Name);

Console.WriteLine("This is {0}",th.IsAlive);

}

}

}

**Thread类常用属性和方法：**

属性：



方法：



**创建线程：**

线程是通过扩展Thread类创建的。扩展的Thread类调用**Start()**方法来开始子线程的执行。实例：

using System;

using System.Threading;

namespace Threading

{

class ThreadCreationProgram

{

public static void CallToChidThread()

{

Console.WriteLine("Child thread starts");

}

static void Main()

{

Console.WriteLine("In Main:Creating the child thread");

Thread childThread = new Thread(CallToChidThread);

childThread.Start();

}

}

}

out：

In Main:Creating the child thread

Child thread starts

**管理线程：**

Thread类提供了各种管理线程的方法。下面实例演示了**Sleep()**方法的使用，用于在一个特定的时间暂停线程。

using System;

using System.Threading;

namespace Threading

{

class ThreadCreationProgram2

{

public static void CallToChildThread()

{

Console.WriteLine("Child thread starts");

//线程暂停5000毫秒

int sleepfor = 5000;

Console.WriteLine("Child Thread Paused for {0} seconds.",

sleepfor/1000);

Thread.Sleep(sleepfor);

Console.WriteLine("Child thread resumes");

}

static void Main()

{

Console.WriteLine("In Main:Creating the Child thread");

Thread childThread = new Thread(CallToChidThread);

childThread.Start();

}

}

}

out：

In Main:Creating the Child thread

Child thread starts

Child Thread Paused for 5 seconds.//停顿了五秒

Child thread resumes

**销毁线程：**

尽量不要使用Thread.Abort方法来中止线程，因为这是一种粗暴的方式，容易引起资源泄漏或不稳定的行为。当线程中的方法执行完毕后，线程会自动终止并销毁。

**Abort()**方法用于销毁线程。通过抛出**threadabortexception**在运行时中止线程。

下面的实例说明了这一点：

using System;

using System.Threading;

namespace MultithreadingApplication

{

    class ThreadCreationProgram

    {

        public static void CallToChildThread()

        {

            try

            {

                Console.WriteLine("Child thread starts");

                // 计数到 10

                for (int counter = 0; counter <= 10; counter++)

                {

                    Thread.Sleep(500);

                    Console.WriteLine(counter);

                }

                Console.WriteLine("Child Thread Completed");

            }

            catch (ThreadAbortException e)

            {

                Console.WriteLine("Thread Abort Exception");

            }

            finally

            {

                Console.WriteLine("Couldn't catch the Thread Exception");

            }

        }

        static void Main(string[] args)

        {

            Console.WriteLine("In Main: Creating the Child thread");

            Thread childThread = new Thread(CallToChidThread);

            childThread.Start();

            // 停止主线程一段时间

            Thread.Sleep(2000);

            // 现在中止子线程

            Console.WriteLine("In Main: Aborting the Child thread");

            childThread.Abort();

            Console.ReadKey();

        }

    }

}

out：

In Main: Creating the Child thread

Child thread starts

0

1

2

In Main: Aborting the Child thread

Thread Abort Exception

Couldn't catch the Thread Exception

主线程的操作不会影响其他已启动的线程的操作（如上例中主线程休眠，但子线程仍在执行）。

**.Net4.0后的线程创建：**

利用Thread类自己创建独立的线程，优先级高。

ThreadPool由.Net自行管理，只需要将需要处理的方法写好，然后交给线程池，线程池自行执行、自行管理。

Task，类似于ThreadPool，但效率比线程池略高，Task对多核的支持更为明显，所以在多核处理器中，Task的优势更为明显

实例：

using System;

using System.Threading;

using System.Threading.Tasks;

namespace Threading

{

class ThreadCreationProgram5

{

static void Main()

{

//Thread类独立创建线程

Thread t = new Thread(ThreadProcess);

t.Start(new object());

//线程池

ThreadPool.QueueUserWorkItem(ThreadProcess,new object());

//Task方式

Task.Factory.StartNew(ThreadProcess,new object());

Task.Run(()=> { });

}

private static void ThreadProcess(object tag)

{

int i = 100;

while (i >= 0)

{

Console.WriteLine(string.Format("i:{0}",i));

Thread.Sleep(10);

i--;

}

}

}

}

**单例模式：**

单例模式即只能存在一个这个类的实例化对象（全局唯一性），与Unity中常用的instance=this;还是有一点不一样的，Unity中这种做法不算严格的单例模式
，因为Unity中的脚本需要被挂载在对象上调用而不是直接实例化所以可以这样用。

严格意义上的单例模式必须是具有私有构造函数（不写他会自动有一个构造函数的）
，然后给出一个静态方法get到我们需要的类的单个实例(即只有第一次才帮我们new一个)

实例：

using System;

using System;

namespace WinformStudy1

{

public class GlobalClass2

{

private static GlobalClass2 instance;

//私有构造函数

private GlobalClass2()

{

}

//提供一个静态的Instance属性供外部访问

public static GlobalClass2 Instance

{

get

{

if (instance == null)

{

instance = new GlobalClass2();

}

return instance;

}

}

//全局变量，任何地方都能通过GlobalClass2.Instance.accountG来访问了这个了

public int accountG;

}

}

**用处：**

用处其实很简单，使用一个全局类能更好的管理所有需要线程间、窗体程序之间通信的一些对象，比如我在线程1、2、6中都需要访问当前我们的用户账号，把他存在全局类中多好，因为全局唯一性的存在，在任何地方访问的都是我需要的那个用户账号。**许多地方都要用到这个对象的时候，就可以考虑使用单例模式，将那个对象存到全局类中，方便其他类进行访问**。

但是同样也需要注意，因为全局唯一性，我们在任何地方对全局类中的全局变量进行改变，都会反映到那唯一的一个实例上，所以我们尽量需要设置全局变量为只读（readonly）比较安全。

**链表：**

**链表的创建：**

public class Node

{

public int Value {get； set；}

public Node Next {get； set；}

public Node(int value)

{

Value = value;

Next = null;

}

}

public class LinkedList

{

private Node head;

public LinkedList()

{

head = null;

}

public void Add(int value)

{

Node newNode = new Node(value);

if (head == null)

{

head = newNode;

}

else

{

Node current = head;

while (current.Next != null)

{

current = current.Next;

}

current.Next = newNode;

}

}

public void Print()

{

Node current = head;

while (current != null)

{

Console.WriteLine(current.Value);

current = current.Next;

}

Console.WriteLine();

}

}

如此，使用以上代码来创建一个单链表并向其中添加节点的操作就为：

LinkedList linkedList = new LinkedList();

linkedList.Add(1);

linkedList.Add(2);

linkedList.Add(3);

linkedList.Print(); // 输出：1 2 3

**一些C#的小知识点：**

**Action：**

Action是一个泛型委托，但是**只能是无返回值的**。直接看示例：

using System;

using System.Text.RegularExpressions;

public class Program

{

public static void Main()

{

//不接受参数也无返回值的委托

Action act1 = () => {

Console.WriteLine("这是一个无参无返回值的泛型委托");

};

//有一个参数无返回值的委托

Action<int> act2 = (x) => {

Console.WriteLine("有参无返回值的泛型委托，参数为{0}",x);

};

//有两个参数无返回值的委托

Action<string,int> act3 = (name,age) => {

Console.WriteLine("有多参无返回值的泛型委托，参数为:姓名与年龄：{0},{1}", name,age);

};

//调用泛型委托

act1();

act2(15);

act3("张三",15);

}

}

out：

这是一个无参无返回值的泛型委托

有参无返回值的泛型委托，参数为15

有多参无返回值的泛型委托，参数为:姓名与年龄：张三,15

**Func：**

与Action类似，同样也是泛型委托，但是**必须要有返回值**，直接看实例：

//接收一个只有int类型返回值，无参数的结果

Func<int> func1 = () => { return 10; };

// 接受一个 int 参数并返回 int 类型的结果

Func<int, int> square = x => x \* x;

// 接受两个 string 参数并返回 string 类型的结果

Func<string, string, string> concat = (s1, s2) => s1 + s2;

// 接受三个 double 类型的参数并返回 double 类型的结果

Func<double, double, double, double> average = (x, y, z) => (x + y + z) / 3;

// 使用 Func 执行方法

int result1 = square(5); // 输出：25

string result2 = concat("Hello", "World"); // 输出："HelloWorld"

double result3 = average(1.0, 2.0, 3.0); // 输出：2