变量与实例：

1.孩子与气球;类变量相当于孩子，实例相当于气球

2.气球不一定有孩子牵着；

3.可以多个孩子牵同一个气球，也就是多个变量可以共用一个实例；也可以都通过同一根绳子牵着气球（ref操作符）。

例如：

Form myForm1;

Form myForm2;

myForm1 = new Form();

myForm2 = myForm1;

myForm1.Text = "My Form";

myForm2.Text = "I changed it";

myForm1.ShowDialog();

//结果是显示 I changed it

类的三大成员（在vs中把光标移到类上按F1可定位到msdn文档）

属性

方法

事件

某些特殊类或对象在成员方面的侧重点不同

侧重属性 如Entity Framework

侧重方法 如Math

侧重事件 如Timer

静态成员与实例成员

静态 static 语义上表示是 类的成员

实例 非静态成员

Console.WriteLine("hello");//静态

Form form = new Form();//实例化

form.Text = "hello";//实例属性

form.ShowDialog();//实例方法

绑定指的是编译器把一个成员与类或对象关联起来

C#与语言的基本元素

标记：

1.关键字（keyword)

2.操作符（Operator) + - & << || ...

3.标识符（Identifier) 合法标识符，大小写规范（变量：驼峰法，myVarible；其他：帕斯卡尔法，首字母大写,MyClass），命名规范

4.标点符号

5.文本（字面值）

整数 int x = 2(默认int32);(32bit)/ long y = 3L;(64bit)

浮点 float x = 3.0F;(32bit)/ double y = 4.0(默认D);(64bit)

字符 char c = 'a';

字符串 string str = ""/"a"/"aasasa";

布尔 bool b = true;

空值 string str = null;

注释与空白：

单行注释 //注释内容

块注释 /\*内容\*/

格式化代码 Ctrl+KD

初识类型、变量和方法

1.类型（type），也称为数据类型

var i = 3L;

Console.WriteLine(i.GetType().Name);

性质相同的数据的集合，有着一套各自的运算操作。比如取整和除法的结果不同

C#为强类型语言 保证数据的正确性

C#强于C强于JavaScript

C#引入dynamic定义变量，类似于JavaScript中的变量定义。

dynamic myVar = 100;

Console.WriteLine(myVar);

myVar = "Mr.OK";

Console.WriteLine(myVar);

数据类型的6个非常重要的特性：

1.1 数据在内存中占用空间；

1.2 表示最大、最小 值的范围

1.3 此类型所包含的成员（如方法、属性、事件等）

Type myType = typeof(Form);

PropertyInfo[] pInfos = myType.GetProperties();

//MethodInfo[] mInfos = myType.GetMethods();

foreach(var p in pInfos)

Console.WriteLine(p.Name);

1.4 此类型由何基类派生而来

Type myType = typeof(Form);

Console.WriteLine(myType.BaseType.BaseType.FullName);

1.5 程序运行的时候，此类型的变量在分配在内存的什么位置

静态时，编译器分析代码，在硬盘中

动态时，在内存中

内存中有堆heap和栈stack

heap：存储对象，实例；可以有几个G

stack：函数调用；不超过2M

StackOverFlow:栈内存超过2M

C#有指针，但不推荐使用，使用时必须在指针代码函数用unsafe标识

两个都溢出的例子：

class Program

{

static void Main(string[] args)

{

//1

unsafe

{

int\* p = stackalloc int[9999999];

}

//2

BadGuy bg = new BadGuy();

bg.BadMethod();

}

}

class BadGuy

{

public void BadMethod()

{

int x = 100;

this.BadMethod();

}

}

1.6 此类型所允许的操作（运算）

double d = 3.0/4.0和3/4是不同的

2.变量是存放数据的地方，简称“数据”

声明和引用

3.方法（旧称函数）是处理数据的逻辑，又称“算法”

声明和调用

4.程序=数据+算法

有了变量和方法就可以写有意义的程序了

C#语言的类型系统

1. C#的五大数据类型(均由Object类派生)

1.1 类(class)(引用类型):如Window,Form,Console,String

1.2 结构体(structures)(值类型):如Int32,Int64,Single,Double

1.3 枚举(Enumerations)(值类型):如HorizontalAlignment,Visibility,MessageBoxResult

1.4 接口类型(interfaces)(引用类型)

1.5 委托类型(Delegate)(引用类型)

数据类型的派生谱系

object,string被C#吸收为关键字,常用数据类型,基本数据类型(构成其他类型),同时为引用数据类型

class,interface,delegate不是数据类型,而是去定义数据类型的

bool,byte,char,decimal,double,float,int,long,sbyte,short,uint,ulong,ushort被C#吸收关键字,基本数据类型,常用数据类型,值数据类型

struct,enum不是数据类型,而是去定义数据类型.

true,false为bool类型的值

void函数返回值为空,null一个引用变量值为空

var,dynamic声明变量

变量,对象和内存

1.变量->存储数据;变量表示存储位置;变量表示类型;

1.1 7种变量:静态变量,实例变量,数组变量,值参数,引用参数,输出形参,->局部变量<-,狭义的变量就是局部变量

2.引用类型变量定义时分配四个字节的内存,实例数据的地址(堆内存上)存储在前面分配的四个字节的地址里面.

3.局部变量在stack上分配内存

4.实例在heap上分配内存,类成员变量(字段)也在heap上分配内存

5.类成员变量有默认值(由构造器初始化),本地变量一定要赋值.

6.常量,值不能变//const

7.装箱和拆箱[boxing&unboxing],会损失性能.

装箱:int x = 100; object obj = x;//int值类型,直接在stack上给x分配4个字节空间存储值100||object为引用类型,在stack上给obj分配4个用于存储地址的4个字节空间后,把x变量的值复制到heap上,然后obj得到heap上的地址.隐式转换

拆箱:int y = (int)obj;//y变量是值类型,在stack分配4个字节的空间,通过obj找到在heap上的地址,然后把地址里面的值复制到y对应的stack内存里面.显式转换

方法的定义与应用

1.方法的由来

1.1方法的前身是C/C++语言的函数

方法是面向对象的范畴,在非面向对象的语言中仍然称为函数.

函数在类中定义时就称为方法,私下里可称为成员函数

1.2永远都是类(或结构体)的成员

C#语言中函数不可能独立于类(或结构体)之外

只有作为类(结构体)的成员时才被称为方法

1.3是类(或结构体)最基本的成员之一

最基本的成员只有两个---字段和方法(成员变量与成员函数),本质还是数据+算法

方法表示类(或结构体)"能做什么事"

1.4为什么需要方法和函数

目的一:隐藏复杂的逻辑

目的二:复用(reuse,重用)

2.方法的定义与调用

2.1声明方法的语法详解

声明和定义不分家

Parameter全称应该为formal parameter,形式上的参数,简称形参\\也是一种变量

2.2命名规范

大小写规范

需要以动词或者动词短语作为名字

2.3静态方法与实例方法要区分

2.4方法的调用

写入实际参数argument

可理解为调用方法时的真实条件

实参与形参的匹配(数量与类型)

3.构造器constructor(一种特殊的方法)

3.1构造器是类型的成员之一

无返回值类型

与类名完全一致

//ctor+tab\*2,快速定义构造器

3.2狭义的构造器是指"实例构造器"(instance constructor)

3.3构造器的内存原理

对heap的字段地址控件进行初始化或者赋值,其他与前面的引用类型变量的内存存储差不多.

如果构造函数中有string,object等引用类型的参数,则在heap中存储地址,该地址是string内容存储地址,并且也在heap的另一块可用的内存中.

4.方法的重载

名字一样,方法签名不一样

方法签名:组成->名称,类型参数<T>的个数和它的每一个形参(由左往右的顺序)的类型和种类(值,引用ref或输出out)组成.方法签名不包含返回类型.

构造器也可重载

5.如何对方法进行debug

5.1设置断点

5.2观察方法调用时的call stack

5.3step-into进入方法(一步一步最仔细的调用方法F11), step-over(跳过,大范围的定位,效率高F10),step-out(回到调用它的那一层上去,shift+F11).

5.4观察局部变量的值与变化,locals变量的变化

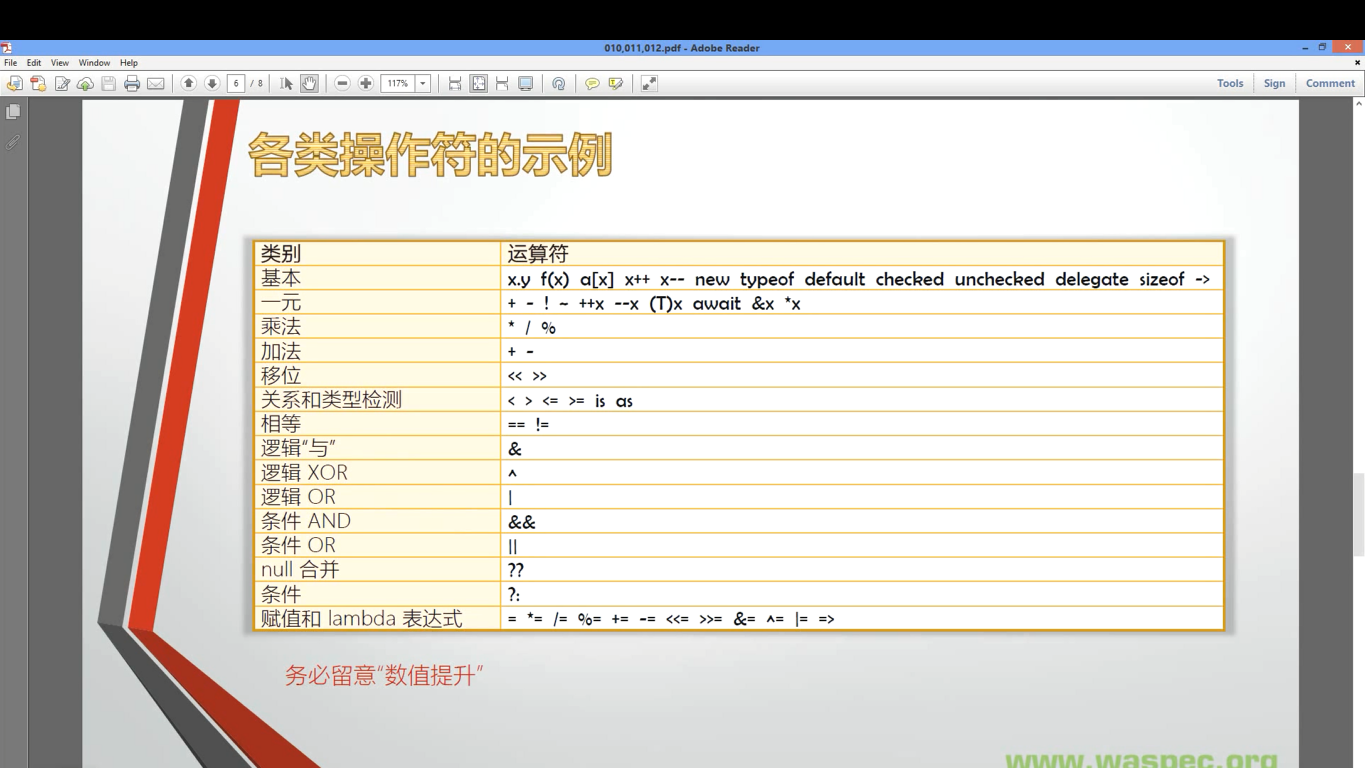
1. 方法的调用与栈

一层层调用,一层层返回

函数的返回值存在CPU的寄存器上

操作符(operator)内容

优先级:



1.操作符的本质

1.1操作符的本质是函数(即算法)的"简记法"

如果没有发明"+"和"\*",只有Add函数和Mul函数,那么3+4\*5将只能写成Add(3,Mul(4,5)),注意优先级.

1.2操作符不能脱离与它关联的数据类型

可以说操作符就是与固定数据类型相关联的一套基本算法的简记法.

2.操作符的优先级与运算顺序

2.1可以用圆括号提高被括起来表达式的优先级

圆括号可以嵌套

"[]","{}"在C#中有专门的用途:

int[] myIntArry = new int[5]{1,2,3,4,5};//int[] myIntArry = new int[5];//int[] myIntArry = new int[]{1,2,3,4,5};

//index

//Dictionary<TKey, TValue> Class.

class Program

{

static void Main(string[] args)

{

//Dictionary<TKey, TValue> Class.

Dictionary<string,Student> stuDic = new Dictionary<string, Student>();

for (int i = 0; i < 100; i++)

{

Student stu = new Student();

stu.Name = "s\_" + i.ToString();

stu.Score = 100 + i;

stuDic.Add(stu.Name,stu);

}

//The "[]" operator is not just int type inside, it maybe string type.

Student number6 = stuDic["s\_6"];

Console.WriteLine(number6.Score);

}

}

class Student

{

public string Name;

public int Score;

}

2.2同优先级操作符的运算顺序

除了带有赋值功能的操作符,同优先级操作符都是由左向右进行运算;3+4+5...

带有赋值功能的操作符的运算顺序是由右向左;int x=100;int y=200;x+=y;"+="有赋值功能;//int z=300;x+=y+=z;//c.w->z=300,y=500,x=600;

与数学不同,计算机语言的同优先级运算没有"结合律",3+4+5为Add(Add(3,4),5)不能是Add(3,Add(4,5)).

int y = x--;先赋值再运算

注意typeof():

Type t = typeof (int);

Console.WriteLine(t.Namespace);

Console.WriteLine(t.FullName);

Console.WriteLine(t.Name);

int c = t.GetMethods().Length;

foreach (var mi in t.GetMethods())

{

Console.WriteLine(mi.Name);

}

Console.WriteLine(c);

注意default();//引用类型null;int等值类型为0;

特别注意enum类型,没赋值第1个默认|赋值有0的,0那个字段默认|赋值没0的,0默认|

new操作符:

功能1:在内存中构建一个实例new Form,并且调用实例构造器new Form();还可以指向一个变量Form myForm = new Form();可以通过变量来操作实例.

功能2,实例的初始化器:可以给实例的属性赋值Form myForm = new Form(){Text = "Hello",FormBroderStyle = FormBroderStyle.SizableToolWindow};

new Form(){Text = "hello"}.ShowDialog();//也是可以显示一个窗口的,new和.的优先级相同,并且无赋值操作符,所以从左往右依次执行//但是这样也没有意义

不一定实例都要用new创建,如string name = "呵呵呵"|int[] myArray = {1,2,3,4,5};

new创建匿名类型:

// anonymous type use

var person = new {Name = "Mr.Ok", Age = 34};

Console.WriteLine(person.Age);

Console.WriteLine(person.Name);

Console.WriteLine(person.GetType().Name);

var关键字:声明隐式类型变量(int x;//显式);var = 100;//int32|var = 100L;//long|...多使用var定义

使用new操作符的时候也要十分小心:当类里面的主方法运行时,主方法里面new一个类的实例,那么这两个类就相互关联,依赖,紧耦合,若主方法里面的实例出现异常,必然会影响到外层类.

new可以做修饰符,子类对父类方法的隐藏,声明方法前new

比如你父类里有一个方法叫 Method(),然后你子类里也有一个方法叫 Method()  
原本，因为子类的继承关系，他自己就会有一个Method()了，然后你又新定义了一个，于是程序编译的时候，不知道这个Method()到底用哪一个，所以你在子类里使用new谓语以后，就会把子类中原有的（继承父类的）Method()改由你后写的那个替换掉

checked运算符和unchecked运算符,检查变量是否溢出:

对语句的检查:

// var overflow

uint x = uint.MaxValue;

Console.WriteLine(x);

string binStr = Convert.ToString(x, 2);

Console.WriteLine(binStr);

//uint y = x + 1;//y == 0 is true, default is unchecked mode

try

{

uint y = checked(x + 1);// overflow exception

//uint y = unchecked (x + 1);//default is unchecked mode

Console.WriteLine(y);

}

catch (OverflowException ex)

{

Console.WriteLine("There's overflow");

}

对语句块的检查:

//var overflow

uint x = uint.MaxValue;

Console.WriteLine(x);

string binStr = Convert.ToString(x, 2);

Console.WriteLine(binStr);

//uint y = x + 1;//y == 0 is true, default is unchecked mode

checked

{

try

{

//uint y = checked(x + 1);// overflow exception

uint y = x + 1;

//uint y = unchecked (x + 1);//default is unchecked mode

Console.WriteLine(y);

}

catch (OverflowException ex)

{

Console.WriteLine("There's overflow");

}

}

delegate操作符(过时的技术):

使用delegate操作符来声明匿名方法(不会被reuse,没有名字)

this.MyButton.Click += delegate(object sender, RouteEventArgs e)

{

this.MyTextbox.Text = "hello";

};

现在使用lambda表达式

this.MyButton.Click += (sender, e) =>

{

this.MyTextbox.Text = "hello";

};

sizeof操作符,获取对象在内存中字节数的尺寸 int x = sizeof(int);

注意1:sizeof只能获取基本类型实例(结构体数据类型),string和object除外;

注意2:非默认情况下(在unsafe作用域内),可以使用sizeof获取自定义结构体在内存中的字节大小;

->操作符

C#存在指针,->操作符就是结构体指针变量指向其结构体成员,指针不能使用在引用类型变量中

struct CsStudent

{

public int ID;

public int Score;

}

//main中

//"->" operator use

unsafe

{

CsStudent stu;

stu.ID = 1;

stu.Score = 99;

CsStudent\* pStu = &stu;

(\*pStu).Score = 1000;

pStu->Score = 100;

Console.WriteLine(stu.Score);

}

一元操作符,一个变量也可以操作

+,-,!,~,++x,--x,(T)x,await,&x,\*x;

&x和\*x,不安全操作符

CsStudent\* pStu = &stu;

(\*pStu).Score = 1000;//优先级..

pStu->Score = 100;

+,- (正,负)操作符,与数学类似

int x = 100;

int y = +x;//y==100

int y = -x;//y==-100,运算为按位取反再加1

int y = -(-x);//y==100

注意:

int x = int.MinValue;

int y = checked(-x);//overflow,若y=-x,则y还是等于int.MinValue

~按位取反

!取非操作符,操作bool类型值,常用在if判断中

++x,--x,先自我运算,如果有赋值运算符就后赋值(与x++等相反)

(T)x,强制类型转换

//(T)x, convert use

string str1 = Console.ReadLine();

string str2 = Console.ReadLine();

int x = Convert.ToInt32(str1);

int y = Convert.ToInt32(str2);

Console.WriteLine(x+y);

1.隐式类型转换

1.1不丢失精度的转换

隐式数值转换为：

•从 sbyte 到 short、int、long、float、double 或 decimal。

•从 byte 到 short、ushort、int、uint、long、ulong、float、double 或 decimal。

•从 short 到 int、long、float、double 或 decimal。

•从 ushort 到 int、uint、long、ulong、float、double 或 decimal。

•从 int 到 long、float、double 或 decimal。

•从 uint 到 long、ulong、float、double 或 decimal。

•从 long 到 float、double 或 decimal。

•从 ulong 到 float、double 或 decimal。

•从 char 到 ushort、int、uint、long、ulong、float、double 或 decimal。

•从 float 到 double。

从 int、uint、long 或 ulong 到 float 以及从 long 或 ulong 到 double 的转换可能导致精度损失，但决不会影响到它的数量级。其他的隐式数值转换决不会丢失任何信息。

不存在向 char 类型的隐式转换，因此其他整型的值不会自动转换为 char 类型。

隐式类型转换原理:

class Program

{

static void Main(string[] args)

{

Stone stone = new Stone();

stone.Age = 50000;

Monkey WukongSun = stone;//转换

Console.WriteLine(WukongSun.Age);

}

}

class Stone

{

public int Age;

//相当于一个monkey类型的一个构造器,但是写在被转换类型的类里面的.

public static implicit operator Monkey(Stone stone)

{

Monkey m = new Monkey();

m.Age = stone.Age/500;

return m;

}

}

class Monkey

{

public int Age;

}

1.2子类向父类的转换,隐式转换

Techer t = new Techer();

Human h = t;

Animal a = h;

a.Eat();//a只能eat

class Animal

{

public void Eat()

{

Console.WriteLine("Eating...");

}

}

class Human : Animal

{

public void Think()

{

Console.WriteLine("Who I am?");

}

}

class Techer : Human

{

public void Tech()

{

Console.WriteLine("I can tech");

}

}

1.3装箱操作,前面有

2.显式类型转换

丢失精度

Console.WriteLine(ushort.MaxValue);//65535

uint x = 65536;

ushort y = (ushort)x;

Console.WriteLine(y);//0

Covert 类转换

Parse方法转换,只能转换有效的格式

tryParse,返回bool型,带有输出类型

拆箱操作

显式类型转换操作符的原理

class Program

{

static void Main(string[] args)

{

Stone stone = new Stone();

stone.Age = 50000;

Monkey WukongSun = (Monkey) stone;

Console.WriteLine(WukongSun.Age);

}

}

class Stone

{

public int Age;

//相当于一个monkey类型的一个构造器,但是写在被转换类型的类里面的.

public static explicit operator Monkey(Stone stone)

{

Monkey m = new Monkey();

m.Age = stone.Age/500;

return m;

}

}

class Monkey

{

public int Age;

}

1.算术运算符

1.1 乘法类型运算符 \* / %

注意数值提升 var x = 3.0 \* 4;x的type为double

1.2 加减法运算符 + -

注意数值提升 var x = 3.0 + 4;x的type为double

1.位移操作符 shift-expression:

right and left

int x = 7;

int y = x << 1;

string strX = Convert.ToString(x, 2).PadLeft(32, '0');

string strY = Convert.ToString(y, 2).PadLeft(32, '0');

Console.WriteLine(strX);

Console.WriteLine(strY);

//00000000000000000000000000000111

//00000000000000000000000000001110

可以通过checked来检查是否溢出

关系操作符

> < >= <= is as 的优先级大于 == !=

返回类型为bool类型

int x = 5;

double y = 4.0;

var result = x > y;//char类型也是值类型,也可以比大小

Console.WriteLine(result.GetType().FullName);

Console.WriteLine(result);

//System.Boolean

//true

string类型不能比大小,只能比相等

is as类型检验操作符

is 检验一个对象是不是某个类型的对象

Techer t = new Techer();

var result = t is Techer;

Console.WriteLine(result.GetType().FullName);

Console.WriteLine(result);

//System.Boolean

//True

//null

Techer t = null;

var result = t is Techer;

Console.WriteLine(result.GetType().FullName);

Console.WriteLine(result);

//System.Boolean

//False

//inherit

Techer t = new Techer();

var result = t is Animal;

Console.WriteLine(result.GetType().FullName);

Console.WriteLine(result);

//System.Boolean

//True

//object everything is object

// human is teacher is false

as 操作符 "像..一样"

object o = new Teacher();

Techer t = o as Teacher;//o 是否和 Teacher一样,如果一样,把o赋值给t,如果不一样把Null赋值给t

if(t != null)

{

t.Teacher();

}

逻辑 |与 &|异或 ^|或 || 这三个都是位操作符

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

条件 |与 &&|或 ||| 条件运算符

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

Null 值合并操作符 ??

int? x = null;

int y = x ?? 1;//x为null吗,如果是,把1赋值给y.x的值不变

Console.WriteLine(y);

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

条件操作符 ?:

其实就是ifeles操作的简写

int x = 80;

string str = string.Empty;

str = (x >= 60)?"pass":"failed";

Console.WriteLine(str);

赋值有关的操作符和lambda表达式

= \*= %= += -= <<= >>= &= ^= |= =>

int x = 7;

x <<=2;//x=x<<2;

表达式和语句

什么是表达式:一种专门用来求值的语法实体.什么值?--->值,对象,方法(委托),命名空间等等

表达式是构成算法逻辑的最基本(最小)单元,表达一定的算法意图

因为操作符有优先级,所以表达式也就有了优先级.

C#表达式的分类

一个值.有关联的类型

块语句:可在其中写任何语句,当作一条语句来看待,变量的作用域:

block:

{statement-list-opt}

标记语句:

hello:Console.WriteLine("Hell, world");

goto hello;//死循环

技巧:在花括号行按Ctrl+}可以标记该花括号的作用域

try语句,异常处理:

namespace Try

{

class Program

{

static void Main(string[] args)

{

Calculator c = new Calculator();

int r = c.Add("0", "200");

Console.WriteLine(r);

}

}

internal class Calculator

{

public int Add(string arg1, string arg2)

{

var a = 0;

var b = 0;

bool hasError = false;

try

{

a = int.Parse(arg1);

b = int.Parse(arg2);

}

catch (ArgumentNullException ane)

{

Console.WriteLine(ane.Message);

hasError = true;

//Console.WriteLine("Your argument(s) are null!");

}

catch (FormatException fe)

{

Console.WriteLine(fe.Message);

//Console.WriteLine("your arguments have error!");

hasError = true;

}

catch (OverflowException ofe)

{

Console.WriteLine(ofe.Message);

//Console.WriteLine("out of range!");

hasError = true;

throw;//交给主方法处理;

}

finally

{

Console.WriteLine(hasError ? "Execution has error!" : "Done");

}

int result = a + b;

return result;

}

}

}

迭代语句

while//条件循环

do{}while

for//计数循环

foreach//循环遍历

跳转语句

continue;//结束此次循环,执行下一次

表达数据的: 常量,字段,

字段(field)是一种表示与对象或类型(类或结构体)关联的变量

字段是类型的成员,旧称:成员变量

静态字段与实例字段

静态字段表示的是类型当前的状态

namespace Field

{

class Program

{

static void Main(string[] args)

{

//static field and instance field.

List<Student> stuList = new List<Student>();

for (int i = 0; i < 100; i++)

{

Student stu = new Student();

stu.Age = 23;

stu.Score = i + 1;

stuList.Add(stu);

}

int totalAge = 0;

int totalScore = 0;

foreach (var stu in stuList)

{

totalAge += stu.Age;

totalScore += stu.Score;

}

Student.AveAge = totalAge/Student.Amount;

Student.AveScore = totalScore/Student.Amount;

Student.ReportAmount();

Student.ReportAveAge();

Student.ReportAveScore();

}

}

class Student

{

public int Age;

public int Score;

public static int AveAge;

public static int AveScore;

public static int Amount;

public Student()

{

Amount++;

}

public static void ReportAmount()

{

Console.WriteLine(Student.Amount);

}

public static void ReportAveAge()

{

Console.WriteLine(Student.AveAge);

}

public static void ReportAveScore()

{

Console.WriteLine(Student.AveScore);

}

}

}

字段的声明:名词(属性)

new public protected internal private static readonly volatile 的组合

readonly字段只能在构造时赋值

static字段只能静态构造函数里面赋值,以后不能再赋值,用来保存初始化后值不再改变的静态字段

属性:

?:属性(property)是一种用于访问对象或类型的特征的成员,特征反映了状态

属性是字段的自然扩展

1.从命名上看,field更偏向于实例对象在内存中的布局,property更偏向于反映现实世界的对象的特征

2.暴露数据,数据可以是存储在字段里面的,也可以是动态计算出来的

3.对内:保护字段不被非法值"污染"

namespace Property

{

class Program

{

static void Main(string[] args)

{

try

{

Student stu1 = new Student();

//stu1.SetAge(20);

stu1.Age = 20;//simple and safty;

Student stu2 = new Student();

//stu2.SetAge(30);

stu2.Age = 30;

Student stu3 = new Student();

//stu3.SetAge(24);

stu3.Age = 241;

//int aveAge = (stu1.GetAge() + stu2.GetAge() + stu3.GetAge())/3;

int aveAge = (stu1.Age + stu2.Age + stu3.Age)/3;

Console.WriteLine(aveAge);

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

}

}

class Student

{

private int age;//private field

public int Age//public property

{

get { return age; }

set

{

if (value >= 0 && value <= 120)

{

age = value;

}

else

{

throw new Exception("Age value has error!");

}

}

}

//原理,上面代码为改进

//public int GetAge()

//{

// return age;

//}

//public void SetAge(int value)

//{

// if (value <= 0 && value > 120)

// {

// this.age = value;

// }

// else

// {

// throw new Exception("Age value has error!");

// }

//}

}

}

属性的完整声明(propfull+tab\*2):

可加静态关键字

private static int amount;

public static int Amount

{

get { return amount; }

set { amount = value;//可加限定 }

}

属性的简略声明(prop+tab\*2):

public int Age { get; set; }//无保护措施

属性的只读方法:

删除set{}

还有一种特殊的:

public int Score

{

get { return score; }

private set { score = value; }//只能在类内部才能访问得到

}

private int score;

只读实例:

public bool CanWork

{

get

{

if (this.age >= 16)

{

return true;

}

else

{

return false;

}

}

}

Console.WriteLine(stu.CanWork);

索引器(indexer):一般检索集合的

namespace Indexer

{

class Program

{

static void Main(string[] args)

{

Student stu = new Student();

stu["Math"] = 90;

var mathScore = stu["Math"];

Console.WriteLine(mathScore);

}

}

class Student

{

private Dictionary<string,int> scoreDictionary = new Dictionary<string, int>();

public int? this[string subject]

{

get

{

if (this.scoreDictionary.ContainsKey(subject))

{

return this.scoreDictionary[subject];

}

else

{

return null;

}

}

set

{

if (value.HasValue == false)

{

throw new Exception("Score cannot be null.");

}

if (this.scoreDictionary.ContainsKey(subject))

{

this.scoreDictionary[subject] = value.Value;

}

else

{

this.scoreDictionary.Add(subject,value.Value);

}

}

}

}

}

常量(constant):

const修饰符修饰

使用常量可以提高程序的运行效率

Math.PI就是一个成员常量为3.14159

int.MaxValue...

局部常量:const int x = 100;

常量隶属于类型而不是对象,即没有"实例常量",而这个角色由只读实例字段来担当

不能用自定义的类类型和结构体类型来作为常量的类型,但可以这样:

public static readonly Building Location = new Building("Some Address");

class Building{...}

各种参数,对方法的进一步学习:

1.传值参数,值参数:声明时不带修饰符的形参.在传值后为实参创建一个副本,操作的是副本,不改变实际参数;

引用的类型的参数就不一样了,处理的是同一个引用变量,只要不对其进行新的实例化;

class Program

{

//引用类型的传值参数

static void Main(string[] args)

{

Student stu = new Student() { Name = "Tim" };

SomeMethod(stu);

Console.WriteLine("{0},{1}", stu.GetHashCode(), stu.Name);

}

static void SomeMethod(Student stu)

{

stu = new Student() { Name = "Tom" };

Console.WriteLine("{0},{1}", stu.GetHashCode(), stu.Name);

}

}

class Student

{

public string Name { get; set; }

}

2.引用参数,用ref修饰,引用参数与值形参不同,它不创建新的存储位置.

相反,引用形参表示的存储位置恰是方法调用中作为实参给出的那个变量所表示的存储位置.

class Program

{

static void Main()

{

int y = 1;

IWantSideEffect(ref y);//传递的是地址

Console.WriteLine(y);//101

}

static void IWantSideEffect(ref int x)

{

x = x + 100;

}

}

引用类型的引用参数:

class Program

{

static void Main()

{

Student outerStu = new Student() {Name = "Tim"};

Console.WriteLine("HashCode={0},Name={1}",outerStu.GetHashCode(),outerStu.Name);

Console.WriteLine("======================================");

IWantSideEffect(ref outerStu);

Console.WriteLine("HashCode={0},Name={1}", outerStu.GetHashCode(), outerStu.Name);

}

static void IWantSideEffect(ref Student stu)

{

stu = new Student() {Name = "Tom"};

//stu.Name = "Tom";

Console.WriteLine("HashCode={0},Name={1}",stu.GetHashCode(),stu.Name);

}

}

class Student

{

public string Name { get; set; }

}

Result:

HashCode=21083178,Name=Tim

======================================

HashCode=55530882,Name=Tom

HashCode=55530882,Name=Tom

当去掉ref变为值参数的引用类型时:

对比:

class Program

{

static void Main()

{

Student outerStu = new Student() {Name = "Tim"};

Console.WriteLine("HashCode={0},Name={1}",outerStu.GetHashCode(),outerStu.Name);

Console.WriteLine("======================================");

IWantSideEffect(ref outerStu);

Console.WriteLine("HashCode={0},Name={1}", outerStu.GetHashCode(), outerStu.Name);

}

static void IWantSideEffect(ref Student stu)

{

stu.Name = "Tom";

Console.WriteLine("HashCode={0},Name={1}",stu.GetHashCode(),stu.Name);

}

}

class Student

{

public string Name { get; set; }

}

和:

class Program

{

static void Main()

{

Student outerStu = new Student() {Name = "Tim"};

Console.WriteLine("HashCode={0},Name={1}",outerStu.GetHashCode(),outerStu.Name);

Console.WriteLine("======================================");

IWantSideEffect(outerStu);

Console.WriteLine("HashCode={0},Name={1}", outerStu.GetHashCode(), outerStu.Name);

}

static void IWantSideEffect(Student stu)

{

stu.Name = "Tom";

Console.WriteLine("HashCode={0},Name={1}",stu.GetHashCode(),stu.Name);

}

}

class Student

{

public string Name { get; set; }

}

其效果都是:

HashCode=21083178,Name=Tim

======================================

HashCode=21083178,Name=Tom

HashCode=21083178,Name=Tom

但是机制是完全不同的:前面一个,是引用类型的引用参数,在方法里面和外面操作的一个对象.后面一个,在方法体中新建了一个存储stu对象地址的变量,方法体里面和外面变量存储的地址都是同一个实例.

输出参数,用out修饰符声明的形参,一个方法产出多个返回值:

类似于引用形参,输出参数不创建新的存储位置.

static void Main(string[] args)

{

Console.WriteLine("Please input first number:");

string arg1 = Console.ReadLine();

double x = 1;

var b1 = double.TryParse(arg1, out x);

if (b1 == false)

{

Console.WriteLine("Input error!");

return;

}

Console.WriteLine("Input right");

}

轮子:

class Program

{

static void Main(string[] args)

{

double x = 0;

bool b = DoubleParser.TryParse("27d8", out x);

if (b == true)

{

Console.WriteLine(x+1);

}

else

{

Console.WriteLine(x);

}

}

}

class DoubleParser

{

public static bool TryParse(string intput, out double result)

{

try

{

result = double.Parse(intput);

return true;

}

catch

{

result = 0;

return false;

}

}

}

输入参数是引用类型:

namespace OutParameters

{

class Program

{

static void Main(string[] args)

{

Student stu = null;

bool b = StudentFactory.Create("Tim", 23, out stu);

if (b == true)

{

Console.WriteLine("Student {0}, age is {1}.",stu.Name,stu.Age);

}

}

}

class Student

{

public int Age { get; set; }

public string Name { get; set; }

}

class StudentFactory

{

public static bool Create(string StuName, int StuAge, out Student result)

{

result = null;

if (string.IsNullOrEmpty(StuName))

{

return false;

}

if (StuAge<20 || StuAge>80)

{

return false;

}

result = new Student() {Age = StuAge, Name = StuName};

return true;

}

}

}

数组参数:

class Program

{

static void Main(string[] args)

{

int[] myIntArray = new int[] {1, 2, 3, 4};

int result = CalculateSum(myIntArray);

Console.WriteLine(result);

}

static int CalculateSum(int[] intArray)

{

int sum = 0;

foreach (var i in intArray)

{

sum += i;

}

return sum;

}

}

改进,加入params标识:

//数组参数中只能有一个参数是数组参数,有其他参数时数组参数要在最后

class Program

{

static void Main(string[] args)

{

//int[] myIntArray = new int[] {1, 2, 3, 4};

int result = CalculateSum(1,2,3,4,5);

Console.WriteLine(result);

}

static int CalculateSum(params int[] intArray)

{

int sum = 0;

foreach (var i in intArray)

{

sum += i;

}

return sum;

}

}

其实在writeline中也有调用这种方法的重载

static void Main(string[] args)

{

string str = "H,U.G;U[I/";

string[] result = str.Split(',', '.', ';', '[', '/');//例子方法

foreach (var s in result)

{

Console.WriteLine(s);

}

}

具名参数(不是参数的种类而是参数的使用方法):

class Program

{

static void Main(string[] args)

{

PrintInfo("HuGui",34);//不具名调用

PrintInfo(name:"dru",age:11);//具名调用:1.提高代码的可读性,2.参数位置不受参数列表的顺序约束

}

static void PrintInfo(string name, int age)

{

Console.WriteLine("Hello {0}, you are {1}",name,age);

}

}

可选参数(在声明的时候具有默认值):

class Program

{

static void Main(string[] args)

{

PrintInfo();

}

static void PrintInfo(int age = 22, string name = "Hu")

{

Console.WriteLine("Hello {0}, you are {1}", name, age);

}

}

扩展方法(this参数):

为目标数据类型来追加方法.

扩展方法必须是公有的,静态的,即被public static修饰;

必须是形参列表中的第一个,由this修饰;

必须由一个静态类(一般类名为SomeTypeExtension)来统一收纳对SomeType类型的扩展;

using System;

namespace ExtensionMethod

{

public static class DoubleExtension

{

public static double Round(this double input, int digits)

{

double result = Math.Round(input, digits);

return result;

}

}

}

namespace ExtensionMethod

{

class Program

{

static void Main(string[] args)

{

double x = 3.14159;

double y = x.Round(3);

Console.WriteLine(y);

}

}

}

LINQ方法举例:

//LINQ

List<int> myList = new List<int>() {12,11,2,13,12,68};

bool result = myList.All(i => i > 10);//lameda表达式和Linq表达式.

Console.WriteLine(result);

总结:

1.传值参数:参数的默认传递方式.

2.输出参数:用于除返回值外还需要输出的场景.

3.引用参数:用于需要修改实际参数值的场景.

4.数组参数:用于简化方法的调用.

5.具名参数:提高可读性.

6.可选参数:参数拥有默认值.

7.扩展方法(this参数):为目标数据类型来追加方法

委托(delegate)

什么是委托:是函数指针的升级版,间接调用

一切皆地址:

1.变量,是以某个地址为起点的一段内存中所存储的值;

2.函数,是以某个地址为起点的一段内存中所存储的一组机器语言的指令

直接调用与间接调用

1.直接调用:通过函数名来调用函数,CPU通过函数名直接获得函数所在地址并开始执行->返回

2.间接调用:通过函数的指针来调用函数,CPU通过读取函数指针存储的值获得函数所在地址并开始执行->返回

执行的效果是一样的.

Java中没有与委托相对应的功能实体.

类库委托中的Action和function的使用

class Program

{

static void Main(string[] args)

{

Calculator calculator = new Calculator();

Action action = new Action(calculator.Report);//只能空类型的委托

calculator.Report();//直接调用

action.Invoke();//间接调用

action();//间接调用简写

Func<int,int,int> func1 = new Func<int, int, int>(calculator.Add);//泛型委托,int为返回类型

var func2 = new Func<int, int, int>(calculator.Sub);

//Func<int,int,int> func2 = calculator.Sub;//上面一样

int x = 100;

int y = 200;

int z = 0;

//z = func1.Invoke(x, y);

z = func1(x, y);//间接调用,简写

Console.WriteLine(z);

z = func2.Invoke(x, y);

Console.WriteLine(z);

}

}

class Calculator

{

public void Report()

{

Console.WriteLine("I have 3 methods");

}

public int Add(int a, int b)

{

return a + b;

}

public int Sub(int a, int b)

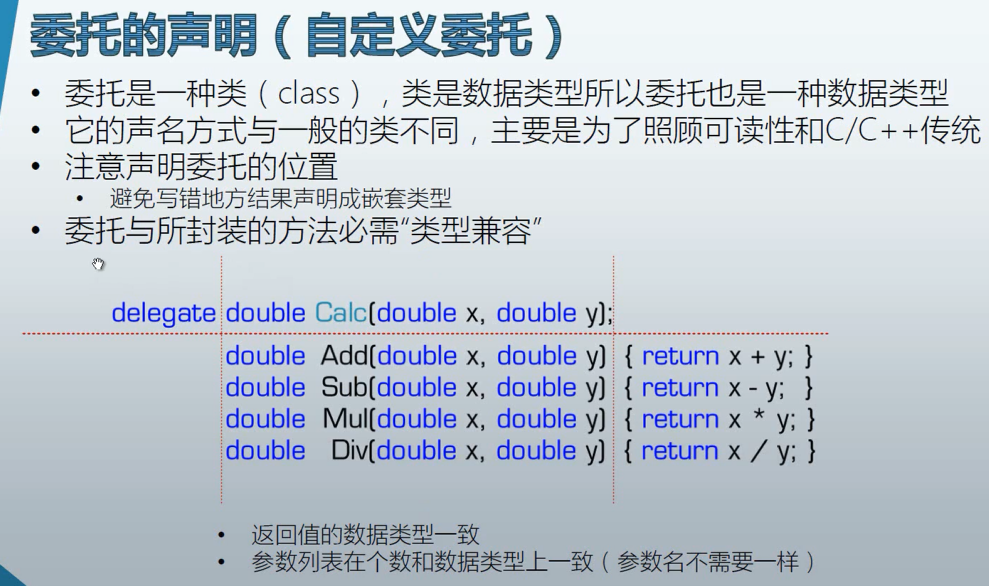
{

return a - b;

}

}

委托的声明(自定义委托)



1.委托是一种类,类是数据类型所以委托也是一种数据类//typeof(Action).isClass == true;

2.它的声明方式与一般类不同,主要是为了照顾可读性和C/C++传统.

namespace SelfDelegate

{

public delegate double Calc(double x, double y);//与类是同一级别

class Program

{

static void Main(string[] args)

{

Calculator calculator = new Calculator();

Calc calc1 = new Calc(calculator.Add);

Calc calc2 = new Calc(calculator.Sub);

Calc calc3 = new Calc(calculator.Mul);

Calc calc4 = new Calc(calculator.Div);

double a = 300;

double b = 200;

double c = 0;

c = calc1.Invoke(a, b);

Console.WriteLine(c);

c = calc2.Invoke(a, b);

Console.WriteLine(c);

c = calc3.Invoke(a, b);

Console.WriteLine(c);

c = calc4.Invoke(a, b);

Console.WriteLine(c);

}

}

class Calculator

{

public double Add(double x, double y)

{

return x + y;

}

public double Sub(double x, double y)

{

return x - y;

}

public double Mul(double x, double y)

{

return x \* y;

}

public double Div(double x, double y)

{

return x / y;

}

}

}

委托与所封装的方法必须"类型兼容",返回值的数据类型一致/参数列表在个数和数据类型上一致(参数名不需要一样)

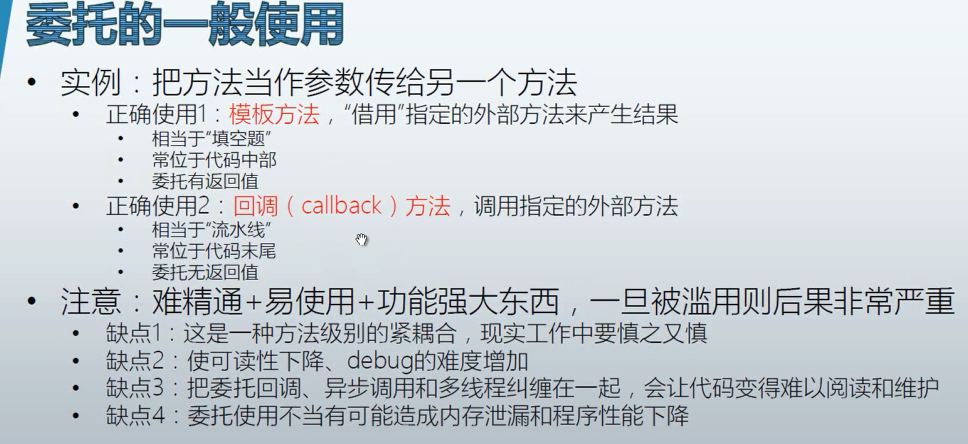
double Calc(double x, double y)

double Add(double x, double y)

double Sub(double x, double y)

...

注意委托声明的位置,与类的定义属于同一级别.



namespace TemplateMethodDelegate

{

class Program

{

static void Main(string[] args)

{

ProductFactory productFactory = new ProductFactory();

WrapFactory wrapFactory = new WrapFactory();

Func<Product> func1 = new Func<Product>(productFactory.MakePizza);

Func<Product> func2 = new Func<Product>(productFactory.MakeToyCar);

Box box1 = wrapFactory.WarpProduct(func1);

Box box2 = wrapFactory.WarpProduct(func2);

Console.WriteLine(box1.Product.Name);

Console.WriteLine(box2.Product.Name);

}

}

class Product

{

public string Name { get; set; }

}

class Box

{

public Product Product { get; set; }

}

class WrapFactory

{

public Box WarpProduct(Func<Product> getProduct)//模版方法,参数(得到产品),返回值(装箱)

{

Box box = new Box();

Product product = getProduct.Invoke();

box.Product = product;

return box;

}

}

class ProductFactory

{

//使用委托的好处就是,只需要扩展产品,就能够得到产品,不用修改其他的地方,增加代码的复用,

//提高效率,减少bug引入.

public Product MakePizza()

{

Product product = new Product();

product.Name = "Pizza";

return product;

}

public Product MakeToyCar()

{

Product product = new Product();

product.Name = "Toy Car";

return product;

}

}

}

回调方法,调用指定的外部方法

相当于"流水线"

常位于代码末尾

委托无返回值

namespace TemplateMethodDelegate

{

class Program

{

static void Main(string[] args)

{

ProductFactory productFactory = new ProductFactory();

WrapFactory wrapFactory = new WrapFactory();

Func<Product> func1 = new Func<Product>(productFactory.MakePizza);

Func<Product> func2 = new Func<Product>(productFactory.MakeToyCar);

Logger logger = new Logger();

Action<Product> log = new Action<Product>(logger.Log);

Box box1 = wrapFactory.WarpProduct(func1,log);

Box box2 = wrapFactory.WarpProduct(func2,log);

Console.WriteLine(box1.Product.Name);

Console.WriteLine(box2.Product.Name);

}

}

class Logger//记录程序的运行状态

{

public void Log(Product product)

{

Console.WriteLine("Product {0} create at {1}, Price is {2}.",product.Name,DateTime.UtcNow,product.Price);

}

}

class Product

{

public string Name { get; set; }

public double Price { get; set; }

}

class Box

{

public Product Product { get; set; }

}

class WrapFactory

{

public Box WarpProduct(Func<Product> getProduct,Action<Product> logCallback)//前面的Product是返回值,后面的Product是回调方法参数

{

Box box = new Box();

Product product = getProduct.Invoke();

if (product.Price >= 50)

{

logCallback(product);//不一定调用

}

box.Product = product;

return box;

}

}

class ProductFactory

{

public Product MakePizza()

{

Product product = new Product();

product.Name = "Pizza";

product.Price = 12;

return product;

}

public Product MakeToyCar()

{

Product product = new Product();

product.Name = "Toy Car";

product.Price = 100;

return product;

}

}

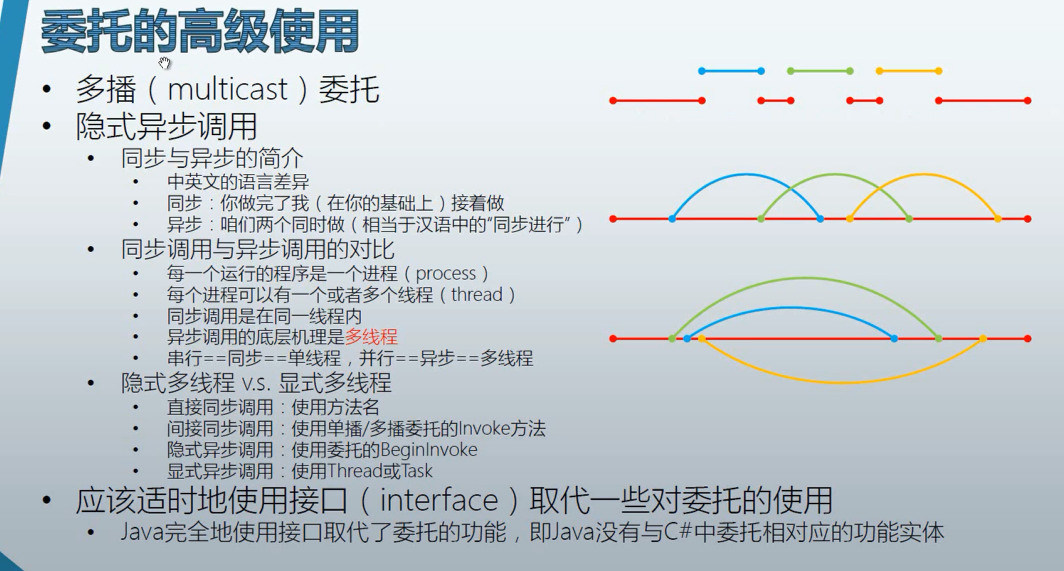
}

output:

Product Toy Car create at 2016/6/8 13:22:09, Price is 100.

Pizza

Toy Car



1.多播委托,同步调用的,接着上一个做

namespace MultiDelegate

{

class Program

{

static void Main(string[] args)

{

Student stu1 = new Student() { ID = 1, PenColor = ConsoleColor.Blue};

Student stu2 = new Student() { ID = 2, PenColor = ConsoleColor.Green};

Student stu3 = new Student() { ID = 3, PenColor = ConsoleColor.Red};

Action action1 = new Action(stu1.DoHomework);

Action action2 = new Action(stu2.DoHomework);

Action action3 = new Action(stu3.DoHomework);

action1 += action2;

action1 += action3;//用一个委托封装多个方法的使用方式叫多播委托,有顺序,执行按顺序

action1.Invoke();

}

}

class Student

{

public int ID { get; set; }

public ConsoleColor PenColor { get; set; }

public void DoHomework()

{

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

{

Console.ForegroundColor = this.PenColor;

Console.WriteLine("Student {0} doing homework {1} hour(s)", this.ID,i);

Thread.Sleep(1000);

}

}

}

}

输出: a line per second

Student 1 doing homework 0 hour(s)

Student 1 doing homework 1 hour(s)

Student 1 doing homework 2 hour(s)

Student 1 doing homework 3 hour(s)

Student 1 doing homework 4 hour(s)//blue

Student 2 doing homework 0 hour(s)

Student 2 doing homework 1 hour(s)

Student 2 doing homework 2 hour(s)

Student 2 doing homework 3 hour(s)

Student 2 doing homework 4 hour(s)//green

Student 3 doing homework 0 hour(s)

Student 3 doing homework 1 hour(s)

Student 3 doing homework 2 hour(s)

Student 3 doing homework 3 hour(s)

Student 3 doing homework 4 hour(s)//red

异步调用,各做各的

同步调用,接上一个做

直接同步调用,直接调用方法

间接同步调用,使用委托

多播委托也是同步调用,是间接同步调用

隐式异步调用,对委托使用GeginInvoke方法,它会自动生成一个分支线程,在分支线程里面调用方法.

class Program

{

static void Main(string[] args)

{

Student stu1 = new Student() { ID = 1, PenColor = ConsoleColor.Blue };

Student stu2 = new Student() { ID = 2, PenColor = ConsoleColor.Green };

Student stu3 = new Student() { ID = 3, PenColor = ConsoleColor.Red };

Action action1 = new Action(stu1.DoHomework);

Action action2 = new Action(stu2.DoHomework);

Action action3 = new Action(stu3.DoHomework);

action1.BeginInvoke(null, null);

action2.BeginInvoke(null, null);

action3.BeginInvoke(null, null);

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

{

Console.ForegroundColor = ConsoleColor.Cyan;

Console.WriteLine("Main thread {0}.", i);

Thread.Sleep(1000);

}

}

}

class Student

{

public int ID { get; set; }

public ConsoleColor PenColor { get; set; }

public void DoHomework()

{

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

{

Console.ForegroundColor = this.PenColor;

Console.WriteLine("Student {0} doing homework {1} hour(s)", this.ID, i);

Thread.Sleep(1000);

}

}

}

主线程和其他三个线程并行的执行,异步调用. 资源有冲突,都在访问控制台前色.

Output:

Main thread 0.

Student 3 doing homework 0 hour(s)

Student 2 doing homework 0 hour(s)

Student 1 doing homework 0 hour(s)

Main thread 1.

Student 3 doing homework 1 hour(s)

Student 1 doing homework 1 hour(s)

Student 2 doing homework 1 hour(s)

Main thread 2.

Student 3 doing homework 2 hour(s)

Student 1 doing homework 2 hour(s)

Student 2 doing homework 2 hour(s)

Main thread 3.

Student 3 doing homework 3 hour(s)

Student 1 doing homework 3 hour(s)

Student 2 doing homework 3 hour(s)

Main thread 4.

Student 3 doing homework 4 hour(s)

Student 1 doing homework 4 hour(s)

Student 2 doing homework 4 hour(s)

Main thread 5.

Main thread 6.

Main thread 7.

Main thread 8.

Main thread 9.

显式异步调用,直接定义线程和task的方法

class Program

{

static void Main(string[] args)

{

Student stu1 = new Student() { ID = 1, PenColor = ConsoleColor.Blue };

Student stu2 = new Student() { ID = 2, PenColor = ConsoleColor.Green };

Student stu3 = new Student() { ID = 3, PenColor = ConsoleColor.Red };

Thread thread1 = new Thread(new ThreadStart(stu1.DoHomework));

Thread thread2 = new Thread(new ThreadStart(stu2.DoHomework));

Thread thread3 = new Thread(new ThreadStart(stu3.DoHomework));

thread1.Start();

thread2.Start();

thread3.Start();

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

{

Console.ForegroundColor = ConsoleColor.Cyan;

Console.WriteLine("Main thread {0}.", i);

Thread.Sleep(1000);

}

}

}

class Student

{

public int ID { get; set; }

public ConsoleColor PenColor { get; set; }

public void DoHomework()

{

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

{

Console.ForegroundColor = this.PenColor;

Console.WriteLine("Student {0} doing homework {1} hour(s)", this.ID, i);

Thread.Sleep(1000);

}

}

}

结果也是异步调用,同上是一类

使用Task方法:替换为

Task task1 = new Task(new Action(stu1.DoHomework));

Task task2 = new Task(new Action(stu2.DoHomework));

Task task3 = new Task(new Action(stu3.DoHomework));

task1.Start();

task2.Start();

task3.Start();

结果同上:

使用接口(Interface)替代一些委托的功能:

namespace TemplateMethodDelegate

{

class Program

{

static void Main(string[] args)

{

IProductFactory pizzaFactory = new PizzaFactory();

IProductFactory toyCarFactory = new ToyCarFactory();

WrapFactory wrapFactory = new WrapFactory();

Box box1 = wrapFactory.WarpProduct(pizzaFactory);

Box box2 = wrapFactory.WarpProduct(toyCarFactory);

Console.WriteLine(box1.Product.Name);

Console.WriteLine(box2.Product.Name);

}

}

interface IProductFactory

{

Product Make();

}

class PizzaFactory : IProductFactory//Ctrl+.

{

public Product Make()

{

Product product = new Product();

product.Name = "Pizza";

product.Price = 12;

return product;

}

}

class ToyCarFactory : IProductFactory

{

public Product Make()

{

Product product = new Product();

product.Name = "Toy Car";

product.Price = 100;

return product;

}

}

class Product

{

public string Name { get; set; }

public double Price { get; set; }

}

class Box

{

public Product Product { get; set; }

}

class WrapFactory

{

public Box WarpProduct(IProductFactory productFactory)

{

Box box = new Box();

Product product = productFactory.Make();

box.Product = product;

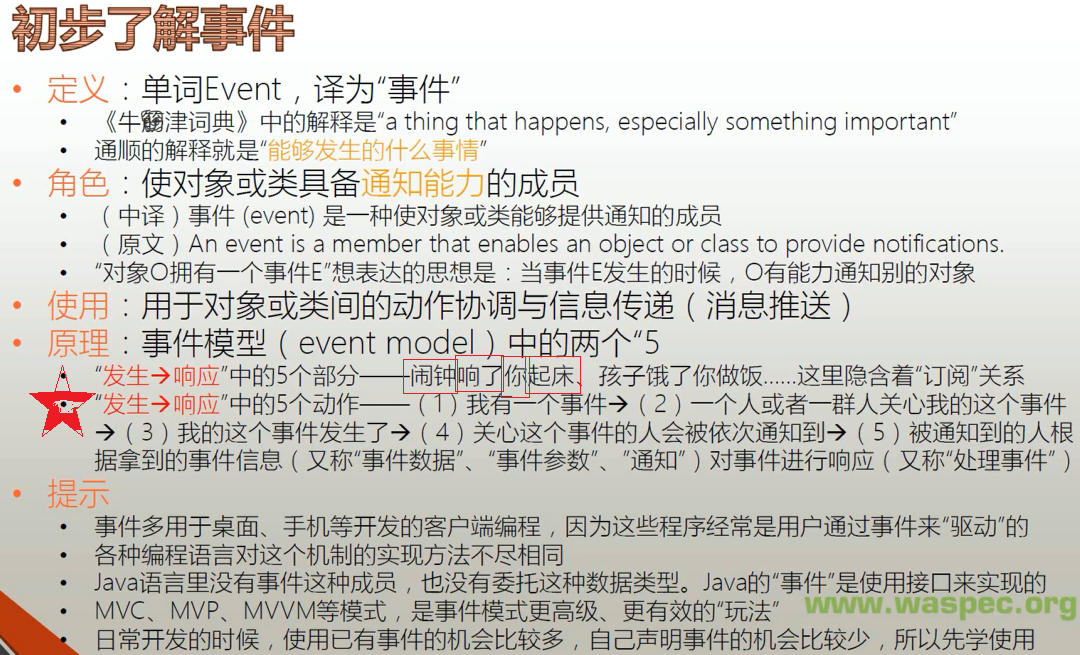
return box;

}

}

}

事件:



C#中,是一种类型的成员

事件模型的5个组成部分:

1.事件的拥有者(event source,对象或类)

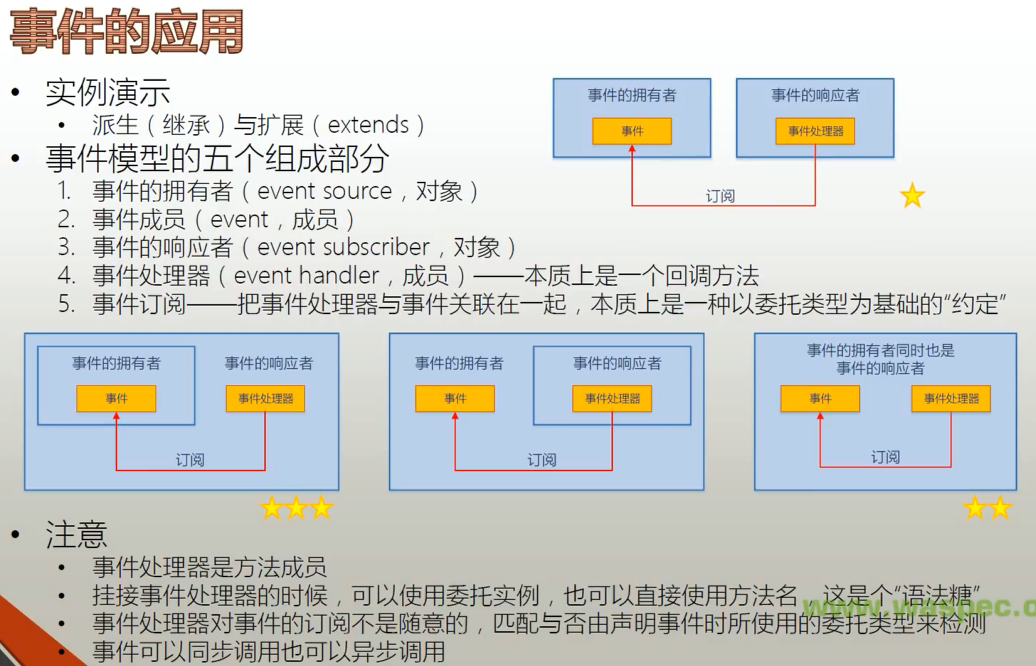
2.事件成员(event,事件本身,成员)

3.事件的响应者(event subscriber,对象)

4.事件处理器(event handler,成员)--本质上是一个回调方法

5.事件订阅---把事件处理器与事件关联在一起,本质上是一种以委托类型为基础的"约定"

事件的使用:



事件是事件拥有者内部逻辑触发的.

初识事件:

这里事件的拥有者和事件的响应者是完全不同的两个对象

class Program

{

static void Main(string[] args)

{

Timer timer = new Timer();

timer.Interval = 1000;

Boy boy = new Boy();

Girl girl = new Girl();

timer.Elapsed += boy.Action;//事件订阅

timer.Elapsed += girl.Action;//一个事件有两个事件处理器的时候

timer.Start();

Console.ReadLine();

}

}

class Boy

{

public void Action(object sender, ElapsedEventArgs e)

{

Console.WriteLine("Jump");

}

}

class Girl

{

public void Action(object sender, ElapsedEventArgs e)

{

Console.WriteLine("Sing");

}

}

事件拥有者form是事件响应者Controlller是两个完全不同的对象(\*):

class Program

{

static void Main(string[] args)

{

Form form = new Form();//事件拥有者form

Controller controller = new Controller(form);//Controlller事件响应者

form.ShowDialog();

}

}

class Controller

{

private Form form;

public Controller(Form form)

{

if (form != null)

{

this.form = form;

this.form.Click += this.FormClicked;//事件click;事件订阅

}

}

private void FormClicked(object sender, EventArgs e)//事件处理器

{

this.form.Text = DateTime.Now.ToString();

}

}

事件的拥有者同时也是事件的响应者(\*\*)

class Program

{

static void Main(string[] args)

{

MyForm myform = new MyForm();//myform事件拥有者和响应者;

myform.Click += myform.FormClick;//事件和事件订阅;

myform.ShowDialog();

}

}

class MyForm : Form

{

public void FormClick(object sender, EventArgs e)//事件处理器

{

this.Text = DateTime.Now.ToString(CultureInfo.CurrentCulture);

}

}

事件的拥有者是事件响应者的一个字段成员(事件响应者用自己的方法订阅自己成员的某个事件,如按钮是窗体的成员)(\*\*\*):

class Program

{

static void Main(string[] args)

{

MyForm form = new MyForm();//事件响应者

form.ShowDialog();

}

}

class MyForm : Form

{

private TextBox textBox;

private Button button;//事件拥有者,也是form的字段成员

public MyForm()

{

this.textBox = new TextBox();

this.button = new Button();

this.Controls.Add(this.button);

this.Controls.Add(this.textBox);

this.button.Click += this.ButtonClicked;//click事件;事件订阅

this.button.Text = "say hello";

this.button.Top = 100;

}

private void ButtonClicked(object sender, EventArgs e)//事件处理器,sender是事件拥有者对象;

{

this.textBox.Text = "hello,world!";

}

}

事件处理器可以重用,只要约束一致

namespace EventReuse

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

//this.Btn3.Click += Btn1\_Click;

this.Btn3.Click += new EventHandler(this.Btn1\_Click);//两种挂接方式

}

//one event handler can be reuse, button1 and button2

//need same constraint

private void Btn1\_Click(object sender, EventArgs e)

{

if (sender == this.Btn1)

{

TxtBox.Text = "Hello";

}

else if(sender == this.Btn2)

{

TxtBox.Text = "World";

}

else if(sender == this.Btn3)

{

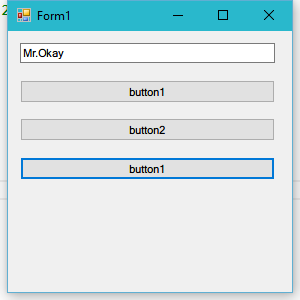
TxtBox.Text = "Mr.Okay";

}

}

}

}



事件处理器也可以是匿名方法(已经过时):

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

//this.Btn3.Click += Btn1\_Click;

//this.Btn3.Click += new EventHandler(this.Btn1\_Click);

this.Btn3.Click += delegate(object sender, EventArgs e)

{

this.TxtBox.Text = "Haha!";

};

}

//one event handler can be reuse, button1 and button2

//need same constraint

private void Btn1\_Click(object sender, EventArgs e)

{

if (sender == this.Btn1)

{

TxtBox.Text = "Hello";

}

else if(sender == this.Btn2)

{

TxtBox.Text = "World";

}

else if(sender == this.Btn3)

{

TxtBox.Text = "Mr.Okay";

}

}

}

现在采样一种Lambda表达式的方法:

this.Btn3.Click += (Object sender, EventArgs args) => {

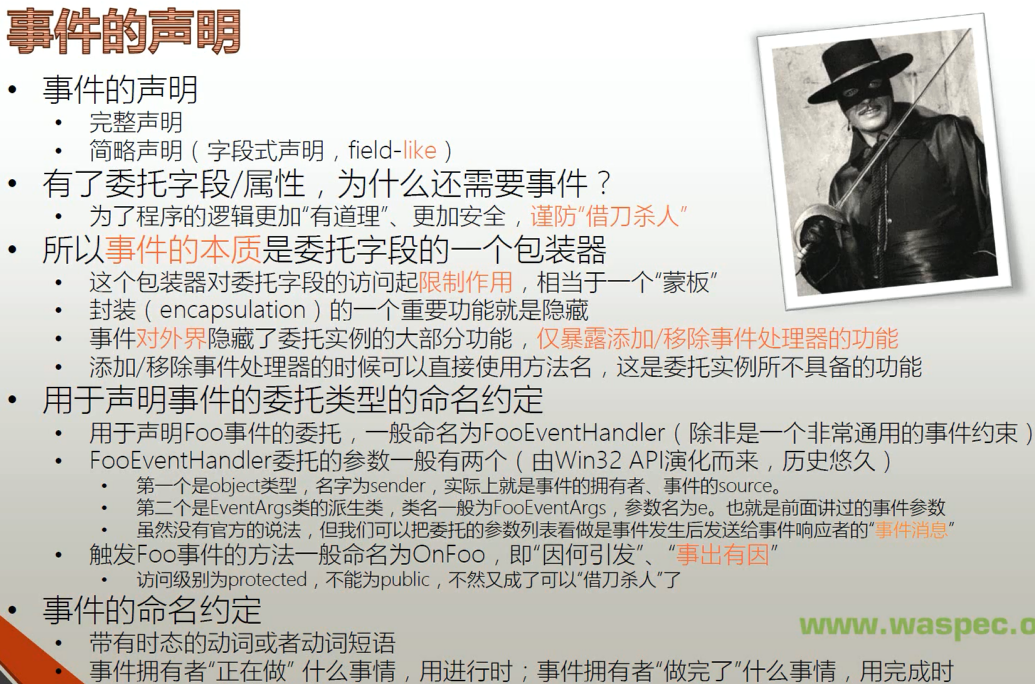
this.TxtBox.Text = "hihi";

};

甚至可以去掉Object 和EventArgs

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

事件详解:



完整声明:

using System;

using System.Threading;

namespace EventDetail

{

class Program

{

static void Main(string[] args)

{

Customer customer = new Customer();//拥有者

Waiter waiter = new Waiter();//订阅者

customer.Order += waiter.Action;//约束关系,只是开始订阅

customer.Action();//触发在Action方法里面. 触发事件是由事件订阅者的内部逻辑决定的

customer.PayTheBill();

}

}

public class OrderEventArgs : EventArgs//传递的事件参数

{

public string DishName { get; set; }

public string Size { get; set; }

}

public delegate void OrderEventHandler(Customer customer, OrderEventArgs e);//声明委托类型,事件依赖于委托.委托约束规范事件

public class Customer//事件拥有者

{

private OrderEventHandler orderEventHandler;//在事件拥有者定义的委托字段

public event OrderEventHandler Order//声明事件

{

add { this.orderEventHandler += value; }

remove { this.orderEventHandler -= value; }

}

public double Bill { get; set; }

public void PayTheBill()

{

Console.WriteLine("I will pay ${0}", this.Bill);

}

public void WalkIn()

{

Console.WriteLine("Walk into the restaurant.");

}

public void SitDown()

{

Console.WriteLine("Sit down.");

}

public void Think()

{

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

{

Console.WriteLine("Let me think...");

Thread.Sleep(1000);

}

if (this.orderEventHandler != null)//关键时刻--触发事件,如果事件有人订阅!

{

OrderEventArgs e = new OrderEventArgs();

e.DishName = "Kongpao Chicken";

e.Size = "large";

this.orderEventHandler.Invoke(this, e);//触发...

}

}

public void Action()//流程

{

Console.ReadLine();

this.WalkIn();

this.SitDown();

this.Think();

}

}

public class Waiter

{

public void Action(Customer customer, OrderEventArgs e)

{

Console.WriteLine("I will serve you the dish - {0}", e.DishName);

double price = 10;

switch (e.Size)

{

case "small":

price = price \* 0.5;

break;

case "large":

price = price \* 1.5;

break;

default:

break;

}

customer.Bill += price;

}

}

}

Output:

Walk into the restaurant.

Sit down.

Let me think...

Let me think...

Let me think...

Let me think...

Let me think...//触发

I will serve you the dish - Kongpao Chicken

I will pay $15

事件声明的简略格式:像字段声明一样

using System;

using System.Threading;

namespace EventSimple

{

class Program

{

static void Main(string[] args)

{

Customer customer = new Customer();//拥有者

Waiter waiter = new Waiter();//订阅者

customer.Order += waiter.Action;//约束关系,只是开始订阅

customer.Action();//触发在Action方法里面. 触发事件是由事件订阅者的内部逻辑决定的

customer.PayTheBill();

}

}

public class OrderEventArgs : EventArgs//传递的事件参数

{

public string DishName { get; set; }

public string Size { get; set; }

}

public delegate void OrderEventHandler(Customer customer, OrderEventArgs e);//声明委托类型,事件依赖于委托.委托约束规范事件

public class Customer//事件拥有者

{

/\*

private OrderEventHandler orderEventHandler;//在事件拥有者定义的委托字段

public event OrderEventHandler Order//声明事件

{

add { this.orderEventHandler += value; }

remove { this.orderEventHandler -= value; }

}

\*/

public event OrderEventHandler Order;//很像委托类型的字段.事件的简略声明

public double Bill { get; set; }

public void PayTheBill()

{

Console.WriteLine("I will pay ${0}", this.Bill);

}

public void WalkIn()

{

Console.WriteLine("Walk into the restaurant.");

}

public void SitDown()

{

Console.WriteLine("Sit down.");

}

public void Think()

{

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

{

Console.WriteLine("Let me think...");

Thread.Sleep(1000);

}

/\*

if (this.orderEventHandler != null)//关键时刻--触发事件,如果事件有人订阅!

{

OrderEventArgs e = new OrderEventArgs();

e.DishName = "Kongpao Chicken";

e.Size = "large";

this.orderEventHandler.Invoke(this, e);//触发...

}

\*/

if (this.Order != null)//迫不得已只能用事件的名字来判断,和调用

{

OrderEventArgs e = new OrderEventArgs();

e.DishName = "Kongpao Chicken";

e.Size = "large";

this.Order.Invoke(this, e);//触发...

}

}

public void Action()//流程

{

Console.ReadLine();

this.WalkIn();

this.SitDown();

this.Think();

}

}

public class Waiter

{

public void Action(Customer customer, OrderEventArgs e)

{

Console.WriteLine("I will serve you the dish - {0}", e.DishName);

double price = 10;

switch (e.Size)

{

case "small":

price = price \* 0.5;

break;

case "large":

price = price \* 1.5;

break;

default:

break;

}

customer.Bill += price;

}

}

}

效果是一样的.

EventHandler委托是常见的委托类型,C#已经定义好了, 传入的参数是Object类型和EventArgs类型,使用这个委托时也要进行类型转换.

public class Waiter

{

public void Action(Object sender, EventArgs e)

{

Customer customerInfo = sender as Customer;

OrderEventArgs orderEventHandlerInfo = e as OrderEventArgs;

Console.WriteLine("I will serve you the dish - {0}", orderEventHandlerInfo.DishName);

double price = 10;

switch (orderEventHandlerInfo.Size)

{

case "small":

price = price \* 0.5;

break;

case "large":

price = price \* 1.5;

break;

default:

break;

}

customerInfo.Bill += price;

}

}

前面的OrderEventHandler不要了,直接定义EventHandler的事件字段.

