C#每日学习笔记

2019年7月9日 11:08

1.Nullable 类型

属性：HasValue ----判断值是否为null

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp12
{
    class Program
    {
        static void Main(string[] args)
        {
            int? oneValue = null;
            if (oneValue.HasValue)
            {
                Console.WriteLine(oneValue.Value);
            }
            else
            {
                Console.WriteLine("oneValue的值为null");
            }
        }
    }
}
```

输出结果：

```
oneValue的值为null
请按任意键继续. . .
```

2.dynamic 类型

略过编译期的类型检查，直接在运行时处理；

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp12
{
   class Program
   {
      static void Main(string[] args)
      {
         dynamic dyn = 1;//dynamic 类型 略过编译期类型检测，在运行时处理！只存在于编译期，运行时dynamic 不存在；
         object obj = 1;
         Console.WriteLine(dyn.GetType());
         Console.WriteLine(obj.GetType());
         //dyn+=1;//编译通过
         //obj+=1;//编译失败  错误提示："+="无法应用于"object"和"int"类型的操作数
      }
   }
}
```

输出结果为：

```
System.Int32
System.Int32
请按任意键继续. . .
```

3.溢出检查

checked 关键字 用于对整型算术运算和转换显式启用溢出检查

unchecked 关键字用于取消整型算术运算和转换的 溢出检查

```
//部分代码
byte  Mybyte =checked((byte)MyInt);
```

4. as 和 is 运算符

使用as 运算符进行类型转换;

先使用 is 运算符 判断 类型是否可以转换，再使用（）运算符进行显式类型转换;

5.可选参数

```
int   result = ms.add(20);
int  result = ms.Add(20,10);
public int Add (int a,int b=1)
{
        return  a+b;
}
```

6.使用命名参数

```
//部分代码
Public int Calculate (int length,int width)
{
        ……..
}

Int result = Calculate(width: w,length: l);
                              //无顺序要求
```

7.四种类型的参数

1.按值传递参数
2.按引用传递参数 ref
3.输出参数   out
4.参数数组   params

用法:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace ConsoleApp4
{
    class Program
    {
        static void Main(string[] args)
        {
            ShowNums(1, 2, 3, 4, 5, 6, 7, 8);
            int a = 10;
            int b = 80;
            string msg = "";
            Swap(ref a, ref b);
            Console.Write("a="+a+"\tb="+b);
            string zh = "admin";
            string mm = "123456";
            Login(zh, mm,out msg);
            Console.WriteLine(msg);
            Console.Read();
        }
        public static void ShowNums(params int [] arr)
        {
            foreach(int i in arr)
            {
                Console.WriteLine(i);
            }
        }
        public static void Swap(ref int a,ref int b)
        {
            int temp = a;
            a = b;
            b = temp;
        }
        public static void Login(string zh, string mm, out string  msg)
        {
            if (zh == "admin" && mm == "123456")
                msg = "ok";
            else
                msg = "error";
        }
    }
}
```

8.静态方法

1.它不属于特定对象的方法，它属于某一类的具体实例
2.它只可以访问静态成员变量，而不可以直接访问实例变量
3.它不用创建类的对象即可访问；
4.静态方法不能使用this 关键字

9.类的成员

类的成员包括：字段，属性，索引器，构造函数，方法

数据成员　　函数成员

10.索引器

```
class  Class
{
        private string student = new string[3];
        public string this [studentNo] //属性名统一为 this 不能为其他，专门用于定义索引器
        {                                    //索引器的参数列表包含在方括号内
                get
                {
                        Return students[studentNo];
                }
                set
                {
                        S8tudents[studentNo]=value;
                }
        }
}
```

11.分部类型和分部方法

修饰符：partial

分部类：

Files1.cs

```
namespace PatialTestSpace
{
        partial class A
        {
                void test1();
        }
}
```

分部类

Files.cs

```
namespace PatialTestSpace
{
        partial  class A
        {
                void test2();
        }
}
```

分部类

程序代码：

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

```
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp14
{
    class Program
    {
        static void Main(string[] args)
        {
            Car car = new Car();
            car.Dosomething1();
            car.Dosomething2();
        }
    }
    public partial class Car
    {
        public void Dosomething1()
        {
            Console.WriteLine("Car 类的部分类1 被使用！");
        }
    }
    public partial class Car
    {
        public void Dosomething2()
        {
            Console.WriteLine("Car 类的部分类2 被使用！");
        }
    }

}
```

分部方法：

分部方法的声明分为两部分：定义和实现；

注意：

1.声明必须以上下文关键字partial 开头

2.声明不可以出现访问修饰符，因此时隐式私有的

3.不能有返回值

4.可以有ref参数，不能有out参数

5.分部方法可以使用static 和 unsafe 修饰符

6.没有实现的分部方法的调用都会被编译器移除

程序代码：

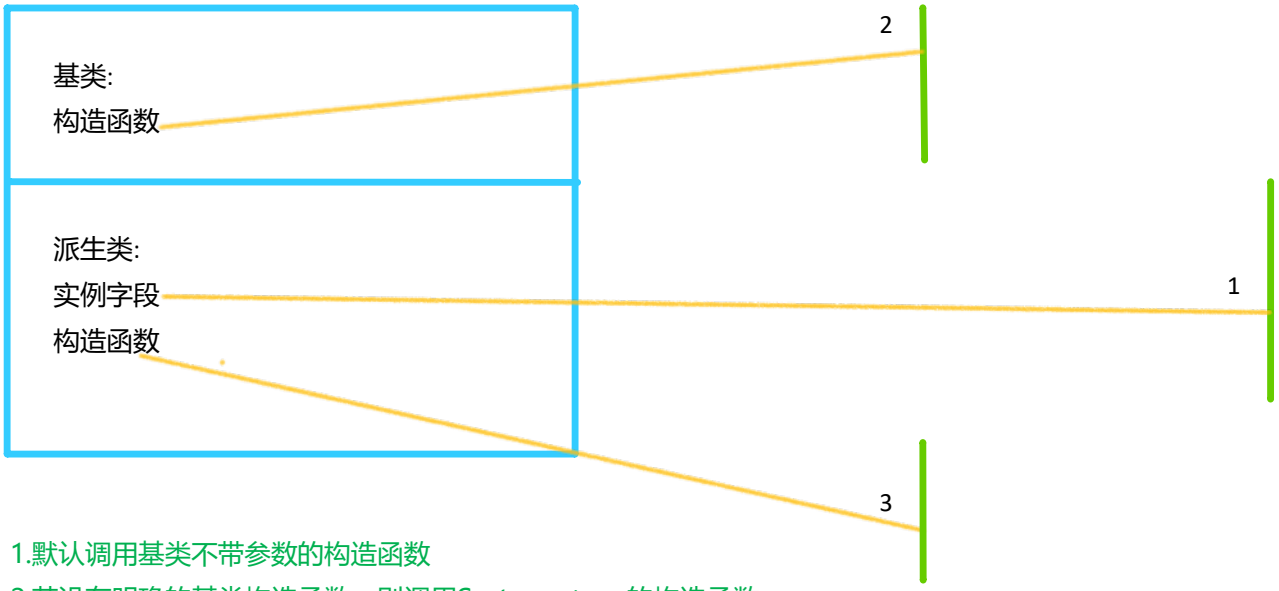
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp14
{
```

```
class Program
{
    static void Main(string[] args)
    {
        Car car = new Car();
       Console.WriteLine(car.Dosomething1());
       Console.WriteLine(car.Dosomething2());
    }
}
 partial class Car
{
    public string  Dosomething1()
    {
        return "DoSomething1()";
    }
    partial void Dosomething3(string sth1);//分部方法的定义
}
 partial class Car
{
    public string Dosomething2()
    {
        Dosomething3("sth");//分部方法被调用
        return "Dosomething2()";
    }
    partial void Dosomething3(string sth1)//分部方法的实现
    {
        Console.WriteLine("Dosomething3()");
        Console.WriteLine(sth1);
    }
}

}
```

运行结果：

```
DoSomething1()
Dosomething3()
sth
Dosomething2()
请按任意键继续. . .
```

12.类的初始化顺序



1.默认调用基类不带参数的构造函数

2.若没有明确的基类构造函数，则调用System.Object的构造函数

程序代码：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp9
{
   class ChildClass : BaseClass
   {
      public bool FieldA = true;
      public bool FieledB;
      public ChildClass()
      {
         Console.WriteLine("自己构造函数被调用！！");
      }
      public void print()
      {
         Console.WriteLine(FieldA);
         Console.WriteLine(FieledB);
      }
   }
}
```

运行结果：

```
基类的构造函数被调用!
自己构造函数被调用！！
True
False
请按任意键继续. . .
```

13.在派生类中指定基类的构造函数

1.使用base关键字，指定使用基类的某个构造函数
2.使用this关键字，指定使用当前类的某个构造函数

1.base：

```
class SomeClass ： BaseClass
{
        public SomeClass()
```

:base()	指定调用默认基类不带参数的构造函数

```
        {

        }
}
public SomeClass(string str)
```

:base(str)	指定后，调用基类的含参str的构造函数且不再调用基类默认不带参数的构造函数

```
{

}
```

2.this

//模式：

```
class SomeClass : BaseClass
{
        public SomeClass()
                :this("something")
        {

        }
        Public SomeClass(string str)
        {

        }
}
```

程序代码：

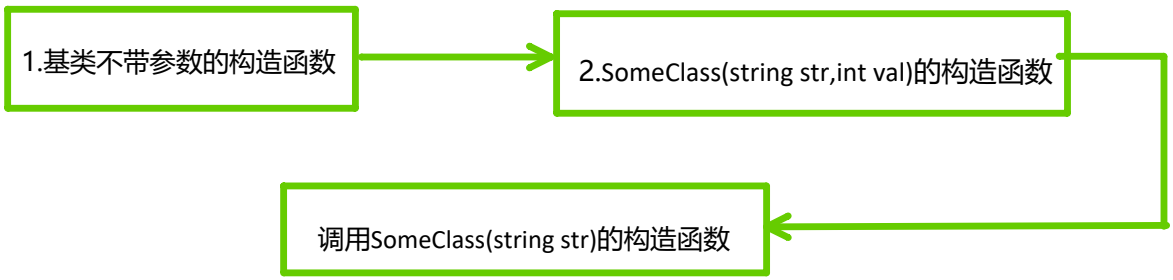
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp9
{
   class Program
   {
      static void Main(string[] args)
      {
         SomeClass cc = new SomeClass("something");//传入参数类型为str型

      }
   }

 class BaseClass
   {
```

```
    public BaseClass()
    {
      Console.WriteLine("基类的构造函数");
    }
    public BaseClass(string str)
    {
      Console.WriteLine("基类的构造函数，参数:{0}",str);
    }
  }
class SomeClass : BaseClass
  {
   //预先调用基类不带参数的构造函数
    public SomeClass()
      :this("something")//指定不带参数的本类构造函数 自动 调用本类SomeClass(string str)构造函数 并 为 str 赋默认值 something
    {
      Console.WriteLine("调用SomeClass()的构造函数！！");
    }
    public SomeClass(string str)
          :this(str,10)//指定参数为string类型的参数  自动 调用本类SomeClass(string str,int val)构造函数  并 为val赋值默认值10
    {
      Console.WriteLine("调用SomeClass(string str)的构造函数");
    }
    public SomeClass(string str,int val)
    {
      Console.WriteLine("调用SomeClass(string str,int val)的构造函数");
    }
  }
}
```

运行结果：

基类的构造函数
调用SomeClass(string str,int val)的构造函数
调用SomeClass(string str)的构造函数
请按任意键继续. . .

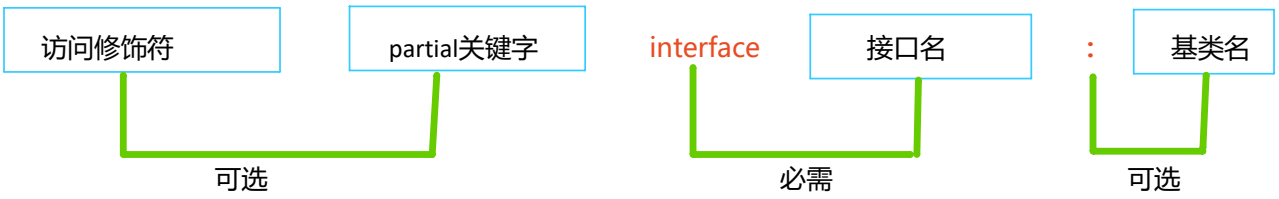
调用顺序：



基类默认不带参数的构造函数先被调用
再调用最终指定本类的构造函数
依次向前调用指定本类的构造函数

14.接口

语法：



接口成员示例：

```
interface Sample
{
        void SayHello();//方法签名
        int Age{get;set;}//属性签名
        eventHandler SizeChanged();//事件签名
        int this [int index ]{get;set;}//索引器
}
```

注意：

接口的成员只包含签名，不包含内容，且不能包含任何访问修饰符（默认为public 级别，但不能将public 写出，否则会出现异常),且必须以分号结尾；

接口类型变量可以指向任何shi

声明和实现接口：

程序代码示例：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp15
{
   class Program
   {
      static void Main(string[] args)
      {
         Interface1 inter1 = new Class1();
         Interface2 inter2 = new Class2();
        inter1.method1();
        inter2.method2();
         //将Interface1类型的引用向下转型到Class1类型的引用
        Class1 class1 = (Class1)inter1;
         class1.method2();
         class1.method1();//依然可调用接口1方法1；
```

```
    }

interface Interface1//接口Interface1
  {
    void method1();//接口Interface1的方法1
  }
interface Interface2//接口Interface2
  {
    void method2();//接口Interface2的方法
  }
 class Class1 : Interface1   {
    public void method1()
    {
      Console.WriteLine("from Class1.method1()");//实现接口Interface1的方法1
    }
    public void method2()
    {
      Console.WriteLine("from Class1.method2()");//类Class1的专属方法
    }
  }
class Class2:Interface2
  {
    public void method2()
    {
      Console.WriteLine("from Class2.method2()");
    }
  }
  }
}
```

运行结果：

```
from Class1.method1()
from Class2.method2()
from Class1.method2()
from Class1.method1()
请按任意键继续. . .
```

基类中的实现作为