C# 6.0那些事

这两天期中考试没时间去看Connect();直播，挺可惜的，考完后补看了Connect(); 把C#6.0的新东西总结一下。

自动属性初始化 (Initializers for auto-properties)

以前我们是这么写的

```
0 references
public class User
{
    private bool _isEnabled = true;

    0 references
    public bool IsEndbled
    {
        get { return _isEnabled; }
        set { _isEnabled = value; }
    }
}
```

为一个默认值加一个后台字段是不是很不爽，现在我们可以这样写

```
0 references
public class User
{
    0 references
    public bool IsEndbled { get; set; } = true;
}
```

只读属性的初始化(Getter-only auto-properties)

像用户ID这种只读属性，我们以前是这样写的

```
1 reference
public class User
{
    0 references
    public User(int id)
    {
        _id = id;
    }

    private readonly int _id;

    0 references
    public int Id
    {
        get { return _id; }
    }
}
```

现在我们可以这样写

```
1 reference
public class User
{
    0 references
    public User(int id)
    {
        Id = id;
    }
    1 reference
    public int Id { get; }
}
```

只读属性可以和标了**readonly**的字段一样在构造函数里面赋值。

用Lambda作为函数体 (Expression bodies on method-like members)

```
0 references
public class User
{
    1 reference
    public int Id { get; }

    1 reference
    public string Name { get; set; }

    0 references
    public override string ToString()
    {
        return string.Format("Id:{0} Name:{1}", Id, Name);
    }
}
```

平时总是有一些短小精悍的代码，但我们不得不把他们放到两个括号中，现在我们可以这么写

```
0 references
public class User
{
    1 reference
    public int Id { get; }

    1 reference
    public string Name { get; set; }

    0 references
    public override string ToString() => string.Format("Id:{0} Name:{1}", Id, Name);
}
```

Lambda表达式用作属性 (Expression bodies on property-like function members)

```
0 references
public class User
{
    1 reference
    public string FirstName { get; set; }
    1 reference
    public string LastName { get; set; }

    0 references
    public string FullName
    {
        get
        {
            return string.Format("{0} {1}", FirstName, LastName);
        }
    }
}
```

这种用法同样可以用于属性

```
0 references
public class User
{
    1 reference
    public string FirstName { get; set; }
    1 reference
    public string LastName { get; set; }

    0 references
    public string FullName => string.Format("{0} {1}", FirstName, LastName);
}
```

字符串嵌入值 (String interpolation)

这个叫法有点怪，看个例子就明白了，上面那个**string.Format**其实可以这样写，不仅写起来方便，而且可读性也非常好。

```
0 references
public class User
{
    1 reference
    public string FirstName { get; set; }
    1 reference
    public string LastName { get; set; }

    0 references
    public string FullName => "\{FirstName} \{LastName}";
}
```

如果要用到格式控制，和以前一样加上就可以了。

Using静态类 (Using static)

如果一个静态类里面是一堆方法，比如**Math** 可以不用写类名，直接调用他的静态方法

```
using System.Math;

namespace HelloCSharp6
{
    0 references
    public class Point
    {
        2 references
        public double X { get; set; }
        2 references
        public double Y { get; set; }

        0 references
        public double Dist() => Sqrt(X * X + Y * Y);
    }
```

有人说这有破坏面向对象的嫌疑，我倒觉得这让C#在函数式编程上更进一步，至于到底是怎样，time will tell.

值得一提的是，这种**using** 也会引入扩展方法，之前**using System.Linq** 会把这个命名空间下所有的扩展方法引入，如果只需要一部分（比如**Enumerable**），这种用法会很方便。

空值判断 (Null-conditional operators)

```
if (xxx != null)
{
    xxx.DoSomething();
}
```

这种写法相信有非常多的朋友用过，经常为了一个是否为空的问题搞得代码非常难看，比如视频里的那个

Null-conditional operators

```
public static Point FromJson(JObject json)
{
    if (json != null &&
        json["x"] != null &&
        json["x"].Type == JTokenType.Integer &&
        json["y"] != null &&
        json["y"].Type == JTokenType.Integer)
    {
        return new Point((int)json["x"], (int)json["y"]);
    }

    return null;
}
```

再举个例子，我们要获取一个列表的长度

```
0 references
public int? GetElementCount(List<int> list)
{
    if (list!=null)
    {
        return list.Count();
    }
    return null;
}
```

这种写法真是太恶心了，在C#6.0中，我们可以这样写

```
0 references
public int? GetElementCount(List<int> list)
{
    return list?.Count();
}
```

从这里也可以看出这种操作符的一个规则：如果对象为空，则整个表达式的值为空。

后面的成员访问不限于方法，还可以是属性，索引器等。

给个实际应用的例子，在触发事件时，经常见到这样的写法，一个委托在调用前总是要判断是否为空

```
0 references
public class ViewModelBase : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    0 references
    protected void RaisePropertyChanged([CallerMemberName]string propertyName = "")
    {
        if (PropertyChanged!=null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

现在我们可以这样

```
0 references
public class ViewModelBase : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    0 references
    protected void RaisePropertyChanged([CallerMemberName]string propertyName = "")
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

如果**PropertyChanged**为**null**，那这句就什么也不做。

nameof表达式 (nameof expressions)

在方法参数检查时，经常会见到这样的代码

```
0 references
public class User
{
    0 references
    public void AddToRole(Role role)
    {
        if (role == null)
            throw new ArgumentNullException("role");
        //...
    }
}
```

里面有那个role是我们手写的字符串，在给role改名时，很容易把下面的那个字符串忘掉，C#6.0解决了这个问题，看看新写法

```
0 references
public void AddToRole(Role role)
{
    if (role == null)
        throw new ArgumentNullException(nameof(role));
    //...
}
```

带索引的对象初始化器 (Index initializers)

对象初始化器在C#3.0就已经有了，C#6.0的对象初始化器加入了对索引器的支持，使得字典一类的东西也可以轻松初始化

这是一个Json.NET使用的例子

```
var obj = new JObject
{
    ["firstName"] = "Henry",
    ["lastName"] = "Charles"
};
```

异常筛选器 (Exception filters)

这个在VB和F#中早就有的功能也加进来了，看看用法

```
try
{
    //...
}
catch (ArgumentNullException ex) if (ex.ParamName == "firstName")
{
    Console.WriteLine("FirstName can not be null.");
}
```

在微软的文档中还给出了另一种用法，这个异常会在日志记录失败时抛给上一层调用者

```
1 reference
private bool Log(Exception ex)
{
    //succeed
    return true;

    //failed
    return false;
}

0 references
public void DoSth()
{
    try
    {
        //...
    }
    catch (Exception ex) if (!Log(ex))
    {
    }
}
```

catch和finally 中的 await (Await in catch and finally blocks)

这是另一个和异常相关的特性，使得我们可以在catch 和finally中等待异步方法，看微软的示例

```
try
{
    res = await Resource.OpenAsync("res location"); // You could do this.
}
catch (ResourceException e)
{
    await Resource.LogAsync(res, e); // Now you can do this …
}
finally
{
    if (res != null) await res.CloseAsync(); // … and this.
}
```

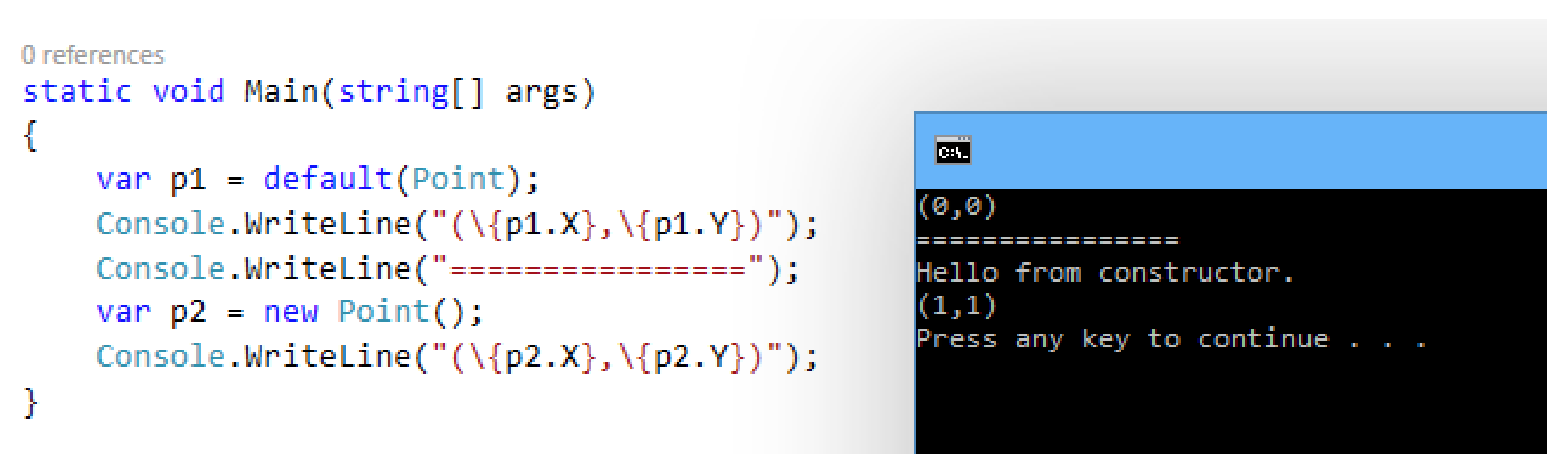
无参数的结构体构造函数 (Parameterless constructors in structs)

在之前版本的C#中是不允许结构体拥有无参构造函数的，在C#6.0中是允许的，但需要注意一点，通过new得到的结构体会被调用构造函数，而通过default得到的不会调用

看看这个例子

```
3 references
public struct Point
{
    1 reference
    public Point()
    {
        Console.WriteLine("Hello from constructor.");
        X = 1;
        Y = 1;
    }
    3 references
    public int X { get; set; }
    3 references
    public int Y { get; set; }
}
```

首先是一个结构体，通过两种不同的方式创建，然后输出



```
0 references
static void Main(string[] args)
{
    var p1 = default(Point);
    Console.WriteLine("(\{p1.X},\{p1.Y})");
    Console.WriteLine("================");
    var p2 = new Point();
    Console.WriteLine("(\{p2.X},\{p2.Y})");
}
```

```
(0,0)
================
Hello from constructor.
(1,1)
Press any key to continue . . .
```