بعض ميزات Framework 3.5: Visual Studio 2008 Orcas, وصف بعض المزايا الجديدة

مرحبا،،
دعنا مباشرة نبدأ الحديث عن بعض هذه التقنيات بدون أية مقدمات

1) Automatic Properities:

نعلم انه إذا أردنا إنشاء صف Point يحتوي على خاصيتين هما X و Y فإننا سوف نكتب الكود التالي:

```
1.    public class Point
2.    {
3.        private int _x;
4.        private int _y;
5.
6.        public int X
7.        {
8.            get { return _x; }
9.            set { _x = value; }
10.        }
11.
12.        public int Y
13.        {
14.      get { return _y; }
15.      set { _y = value; }
16.        }
17.     }
```

إذا فقد احتجنا لإنشاء هاتين الخاصيتن انشاء متحولات لهما من اجل تخزين القيم (خمن مالذي حدث!!!؟)، أصبح الآن بمقدورك كتابة الشيفرة السابقة بالشكل التالي:

```
1.    public class Point
2.    {
3.        public int X{ get; set; }
4.        public int Y { get; set; }
5.    }
```

(أهاكذا يكفي؟ نعم) هو سيتول عنك إنشاء متحولات لهذه الخصائص، للنتقل الآن إلى الميزة التالية.

2) Local Variable Type Inference (استنباط أنوع المتحولات المحلية):

ماذا يعني استنباط أنواع المتحولات المحلية، دعني أوضح لك ذلك من خلال المثال التالي، إذا أردت إنشاء متحول من نوع عدد صحيح، وآخر من أجل عدد حقيقي وآخر من أجل تخزين سلسلة نصية ماذا تكتب؟ "طلب سخيف"!!!

```
int n = 0;
float f = 10.1F;
string s = "Visual Studio Orcas 2008";
```

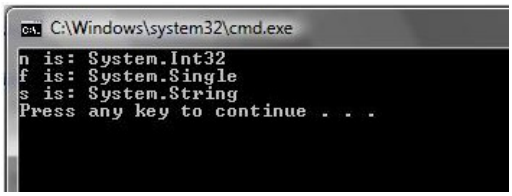
اما الان فقد اصبح لديك إمكانية كتابة مايلي لتحقيق نفس الشيفرة السابقة كما يلي:

```
var n = 0;
var f = 10.1F;
var s = "Visual Studio Orcas 2008";
```

يمكن اعتباره تصريح عن متحولات بشكل ديناميكي، أي أن المترجم هو الذي سيتولى تحديد أنواع هذه المتحولات وذلك من خلال القيم المسندة إلى هذه الى المتحولات، ربما يقول قائل أن المترجم قد اعتبر هذه المتحولات من نوع object لأنه يمكنك عندها استناد أي شيء، هذا الكلم غير صحيح على الاطلاق فالمترجم هنا في الشيفرة السابقة سوف يعرف المحتول n على أنه من نوعint ، والمتحول f من نوعfloat ، والكائن s من نوعstring ، (ربما يقول أحدنا أثبت ما تقول) يمكن أن تثبت صحة الكلام السابق بكتابة الشيفرة التالية:

```
Console.WriteLine("n is: {0}", num.GetType());
Console.WriteLine("f is: {0}", f.GetType());
Console.WriteLine("s is: {0}", s.GetType());
```

سوف ينتج لديك



هل اقنعنا الان؟
("طيب شو هالميزة يلي ما لا طعمة كان بإمكاني أن أصرح عن هذه المتحولات بشكل الاعتيادي ويكفي") سوف تكتشف فائدة الميزة هذه في الفقرات التالية.

3) التصريح عن الكائنات والمجموعات Object Initializers & Collection Initializers:

إذا أخبرتك يا عزيزي المبرمج أتذكر الصف Point السابق؟ أريد منك أن تنشئ كائنا منه وتضبط قيم خصائصة بقيمة الصفر. -شو هاشغلة ميت طلب مثل هالطلب- فستكتب الشكل التالي إلا إذا كنت قد سبقتني إلى التعرف ما سوف أخبرك عنه.

```
Point p = new Point(); // var p = new Point();
p.X = 0;
p.Y = 0;
```

ولكن لو أخبرتك أنه هل هناك طريقة تبسط علي إعطاء القيم الابتدائية اهذه الخصائص؟ فسيكون جوابك: بكل تأكيد كل ما عليك هو إنشاء باني لهذا الصف وتمرر من خلاله قيم الخصائص. كلامك سليم ولكن دعني الان أوريك ماذا يمكن عمله الان بدون بناء هذا الباني، كلما عليك هو تعديل شيفرة التصريح عن هذا الكائن ليصبح بالشكل التالي:

```
Point p = new Point() { X = 0, Y = 0 };
```

ما رؤيك؟؟ ولا ننسى القول أن هذه الميزة يمكن تطبيقها على المجمعات، على الشكل التالي:

```
List<Point> points = new List<Point>() {
    new Point() { X = 1, Y = 1 },
    new Point() { X = 2, Y = 3 }
};
```

4) الأنواع المجهولة Anonymous Types: (هل بقي أكثر من هذا جهل)

تمكنك اللغة الآن من التصريح عن أنواع جديدة -ليكون تمزح- بدون أن تقوم بإنشاء صف جديد -يعني كيف- لو أردنا إنشاء كائن يمثل نقطة بدون أن تقوم بإنشاء الصف السابق Point فستقول لي مستحيل، ولكنني سأقول لك لا شيء مستحيل أبدا، لاحظ الشيفرة التالية التي ستقوم بتأدية نفس الغرض:

```
var p = new { X = 1, Y = 2 };

Console.WriteLine("p.X = {0}", p.X);
Console.WriteLine("p.Y = {0}", p.Y);
```

أنوه مرة أخرى بدون كتابة الشيفرة السابقة للصفPoint .

5) Lambda Expressions (تعابير لامدا):

في C# 2.0 يوجد الاجراءات المجهولة (العامة(Generic ، فمثلا في C# 2.0 يمكن كتابة الشيفرة التالية:

```
class Program
{
    delegate R MyDelegate<A, R>(A arg);

    MyDelegate<int, bool> IsPositive = delegate(int num)
    {
        return num > 0;
    };

    static void Main(string[] args)
    {
        Program p = new Program();
        Console.WriteLine(p.IsPositive(2));
        Console.WriteLine(p.IsPositive(-2));
    }
}
```

طيب هذا الكلام أين الجديد فيه- الجديد فيه هو أنه يمكنك إعادة كتابة التابع IsPositive في سطر واحد بالشكل التالي:

```
class Program
{
    delegate R MyDelegate<A, R>(A arg);

    MyDelegate<int, bool> IsPositive1 = num => num > 0;

    static void Main(string[] args)
    {
        Program p = new Program();
        Console.WriteLine(p.IsPositive(2));
        Console.WriteLine(p.IsPositive(-2));
    }
}
```

6) Extension Methods (الإجرائيات المضافة):

ماذا يعني هذا الكلام وياعيني على هالكلام- لو طلبت منك أن تنشىء لي تابع يقوم بالتحقق من عنوان ايميل هل هو صحيح أم لا، - مافي أسهل من هذا الطلب باستخدام التعابير النظامية- تكتب التابع التالي:

```
public static class StringExtenstion
{
    public static bool IsValidEmailAddress(string s)
    {
        Regex regex =
            new Regex(@"\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*");
        return regex.IsMatch(s);
    }
}

class Program
{
    static void Main(string[] args)
    {

        string email = "Abdulkarim.Kanaan@gmail.com";
        bool isValidate = StringExtenstion.IsValidEmailAddress(email);

        Console.WriteLine(isValidate);
    }
}
```

ولكن لو أخبرتك بأني أود أن أن يكون تابعا عاما لكل سلسلة نصية، مثل توابع السلاسل النصية العامة ()ToString و ()ToTrim و ()ToCharArrayو ...، فربما ستقول لي هذا غير ممكن ولكن عندها سأقول لك أن هذا قد أصبح ممكنا وذلك فقط بإضافة كلمة واحدة قبل البرامتر s في التابع IsValidEmailAddress تأمل الشيفرة التالية:

```
public static class StringExtenstion
{
    public static bool IsValidEmailAddress(this string s)
    {
        Regex regex =
            new Regex(@"\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*");
        return regex.IsMatch(s);
    }
}

class Program
{
    static void Main(string[] args)
    {

        string email = "Abdulkarim.Kanaan@gmail.com";
        bool isValidate = StringExtenstion.IsValidEmailAddress(email);

        Console.WriteLine(isValidate);

        Console.WriteLine(email.IsValidEmailAddress());
    }
}
```

ياللهول؟!

7) Query Syntax (صيغ الاستعلام):

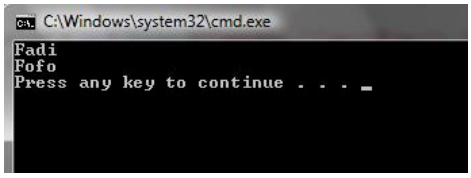
ربما عندما تقرأ هذا العنوان سيخيل لك أنني سأتحدث عنSQL ، ولكن دعني أقل لك أنك قد أخطأت ما سأتحدث عنه بشكل مختصر هو ما يعرف LINQ إختصار لـ–Langauge Integrated Query ممكن أن يكون هناك مقالة أخرى توضح هذه الميزة بشكل أوسع- ، لأبسط لك الأمور بأبسط شكل ممكن، لو أخبرتك بأن لدي مصفوفة من سلاسل نصية تمثل أسماء أصدقائي و أود أن أستخرج جميع الأسماء التي تبدأ بحرف'F ، فربما تقول بسيطة نعمل حلقة ونفحص إذا كانت بداية الاسم تبدأ بهذا الحرف وانتهت القصة، ولكن لو علمت ما في وراء هذا العنوان لعلمت أن ما تعمله هو قصة بحد ذاته، ولكن مع استخدام تقنية LINQ سيكون عمل المطلوب من أبسط مل يكون، تأمل الشيفرة التالية يارعاك الله:

```
static void Main(string[] args)
{
    string[] frindNames = new string[] {
        "Fadi",
        "Ahmad",
        "Rami",
        "Fofo",
        "Toto"};

    var names = from f in frindNames
                where f.StartsWith("F")
                select f;

    foreach (string n in names)
    {
        Console.WriteLine(n);
    }
}
```

سيكون الناتج :



وشكرا