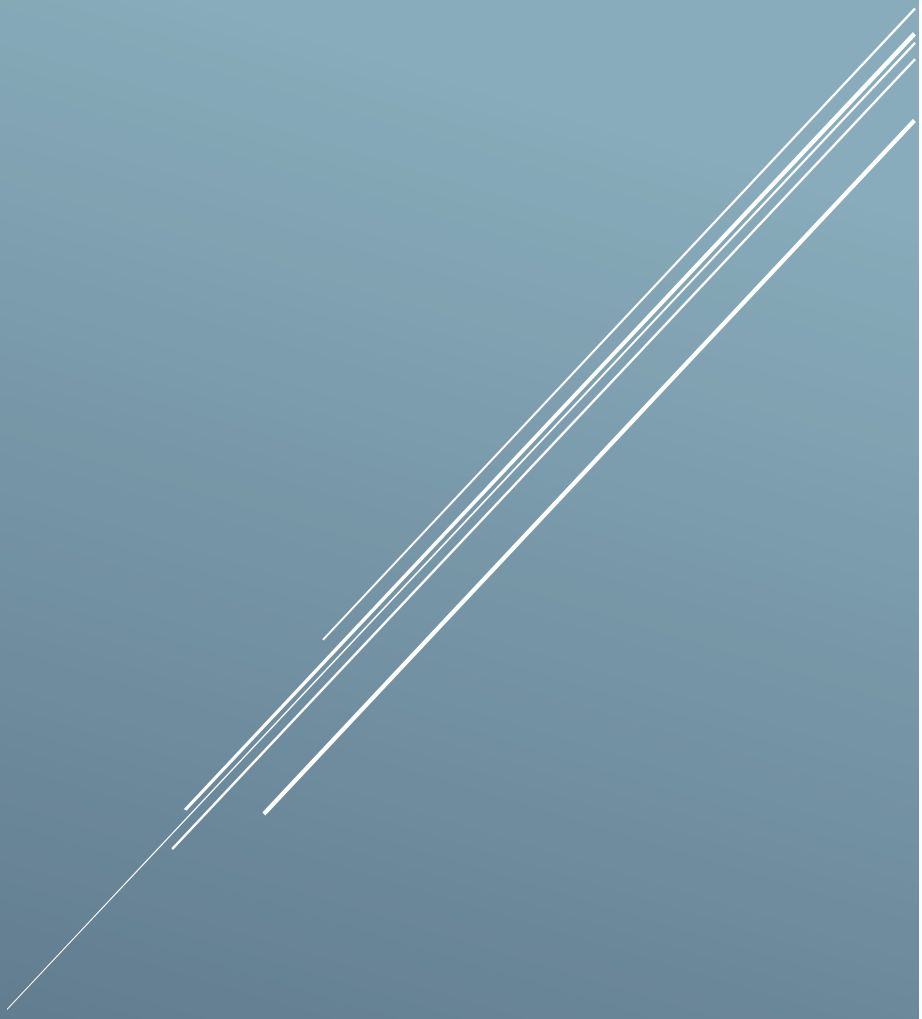


OBJECT ORIENTED PROGRAMMING IN C#

Mohammed Eydan

عنوان الكتاب : برمجة الكائنات الموجهة

المؤلف : محمد عيدان

المستوى: متوسط الى متقدم

عدد الصفحات : 67

نوع الكتاب : الكتروني

الى من كان السبب في كتابة صفحات هذا

الكتاب . . . أهديك هذا العمل .

عن المؤلف:

Mohammed Eydan

18 -Jan -1988 – Iraq – Maysan Governorate

BCs Computer Engineering -University Of Basra - IQ

.Net Framework Senior , System Administrator , Back-End Developer , Database Administrator & Developer .

1. **Microsoft Certified Professional - MCP**
2. **Microsoft Certified Solutions Associate – MCSA**
3. **ITIL**

مقدمة لابد منها :

ليش (بالعراقي) لازم أستخدام OOP ؟

في أحد الأيام سألني شخص : لو أستغنيت عن OOP , هل استطيع تطوير تطبيقات ؟ ثم لماذا كل هذا التعقيد بالبرمجة ؟ نعم تقدر وبكل سهولة , لكن ! الموضوع لا يقتصر فقط على التطوير ! أنت تصرف الكثير من الوقت (الوقت يعني مال) حتى تطور تطبيق بسيط , هذا الوقت لو استفدت منه بفهم مبادئ برمجة الكيانات الموجهة فهو أفضل بكثير لك . كذلك لنفترض أنت كتبت شفرات التطبيق وبعد أسبوع ظهر عندك خطأ (مشكلة) في أحد الشفرات ماذا ستفعل ؟ هل ستراجع جميع اسطر الشفرات ؟ ماذا لو كان في التطبيق 50 الف سطر ! ! !

هنا تبرز الحاجة الى OOP , وبعيداً عن المصطلحات المعقدة فأن الغاية منها هو تنظيم الشفرات وبالتالي سوف تساعدك على أكتشاف الأخطاء بسهولة وبوقت قصير (الصيانة) وتقليل وقت التطوير .

طيب, ماذا يعني تنظيم الشفرات ؟ يعني بدل ما يكون كل شفراتك بنفس الصندوق وهذا ما يسمى بالبرمجة التقليدية (Traditional Programming) , فأن تنظيمها يعني فرزها حسب الصنف (شويه صعبه) (Model) ! مثلاً أنت تريد تتعامل مع قاعدة البيانات فأن شفرات هذا التعامل نعزلها بصندوق خاص, وتريد أيضا تتعامل مع الملفات كذلك نعمل لها صندوق مفصول عن صندوق قاعدة البيانات , بالتالي أي خطأ يصير بشفرة الملفات وبدلا من البحث في كل الشفرات فأنت مباشرة تذهب لصندوق الملفات وتبحث فيه وهكذا مع البقية , الامر لا يقتصر على هذا فقط وانما ممكن بعض الصناديق التي استخدمتها في تطبيق اخر تجلبها وتجري عليها تعديلات بسيطة وتستخدمها في تطبيق جديد .

الكلام أعلاه هو نظري , وسوف يتوضح أكثر عندما نتكلم عن بصورة تقنية (Technical) , المهم تفهم الفكرة الأساسية وهي تسهيل صيانة الشفرات و سرعة تطويرها .

وحتى نفهم برمجة الكيانات الموجهة فالمفروض نكون فاهمين بعض اساسيات السي شارب (Fundamentals) مثلا كيف تعرف متغير وأنواع بيانات المتغيرات والدوارات (Loops) والعبارات الشرطية (Conditionals) , كذلك مفاهيم أخرى مثل الوظائف او الأساليب (Methods) والأصناف (Classes) الكثير يسمونها بالطراز (Model) وهذا ما سوف نشرحه قبل ما نبدأ ب OOP .

ملاحظة : الأفضل حفظ المفهوم باللغة الإنكليزية لان ترجمته تعطي معنى بعيد عن تفاصيله .

Chapter I

Methods

ماهي (Methods) ؟

بصورة نظرية وكما قلنا سابقآ اننا سوف ننظم الشفرات داخل صناديق والاصح نسميها (Model) وكل مودل يتألف من جزئين هما خصائصه وسلوكياته , أذا يمكننا القول ان الميثود عبارة عدة أسطر من الشفرات (برنامج فرعي بصورة أدق) تنفذ عمل معين دخل المودل , وكل مودل يجب بالاقل ان يحتوي على ميثود واحدة .

بالسي شارب الميثود تتكون من قسمين الأول نسميه الرأس (Header) والثاني هو الجسم (Body) , طريقة التعريف جدا بسيطة حيث لدينا صيغة ثابته كالتالي :

```
Access Modifier  Return_Type Name (Parameters) ;
```

الصيغة أعلاه عامة , ليس شرط تلتزم بكل ما فيها لكن فيها أشياء يجب ذكرها عند تعريف الميثود ومنها :

Return_Type – نوع بيانات القيم التي ترجعها الميثود بعد أن تكمل عملها . الميثود بالبرمجة ممكن ترجع قيم ونسميها Function وممكن ما ترجع أي قيمة فنسميها بـ Procedure .

لو أردناه ترجع قيمة فأننا يجب ان نحدد نوع البيانات لهذه القيمة كأن تكون رقم او نص او تاريخ (int , string , datetime) وهكذا , اما أذا رغبنا ان لا ترجع قيمة بهذه الحالة سوف نستخدم الكلمة المفتاحية (void) .

Name : اسم الميثود , تسمية المتغيرات أو الميثود أو الكلاس تعتمد بالأساس على مزاج المبرمج ☹ , لكن يفضل الابتعاد عن الكلمات المفتاحية المحجوزة , او بدأ الأسم برقم وهكذا , هناك طريقتين للتسمية الأولى تسمى بالباسكال وتكون بالشكل التالي لو كان لدينا ميثود نريد تسميتها helloworld :

HelloWorld

يعني اول الاحرف تكون كبيره , هذه الطريقة تنفع بتسمية الكلاسات .

الطريقة الثانية تسمى بسنام الجمل , بمعنى لو عندنا نفس الميثود أعلاه سيكون تسميتها بهذا الشكل :

helloWorld

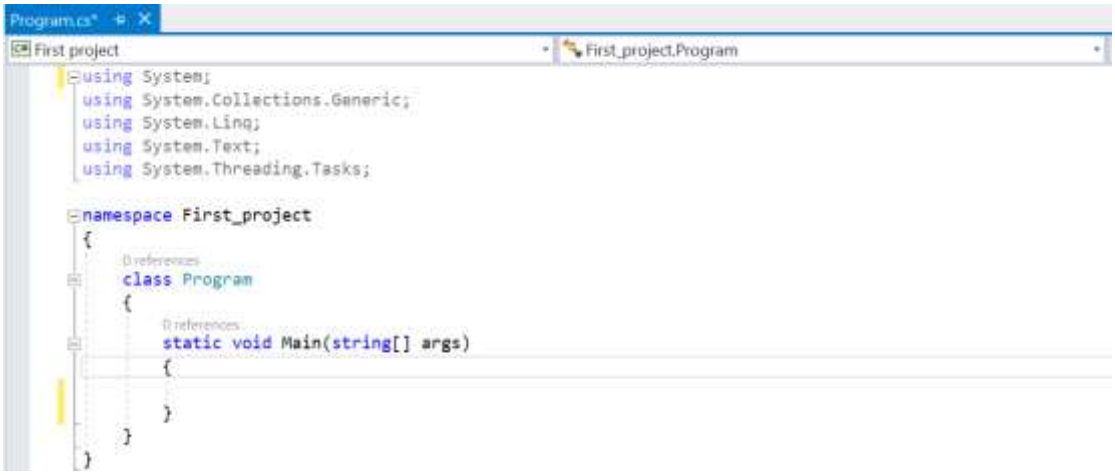
أي أول حرف من الكلمة الثانية يكون كبير , مثل هذه الطريقة يفضل استعمالها عند تسمية الميثود .

Parameters : المعاملات التي تمرر للميثود , طبعا شيء أختياري أعتمادا على طبيعة عمل الميثود ممكن تحتاج لمعاملات وممكن لا . وحتى نضع معاملات للميثود يجب ان نذكر نوع بيانات كل معامل واسمه (نعرفه) .

Access Modifier – بالنسبة للوصولية (يعني المستوى الذي نقدر نشوف فيه الميثود) مثل ما نعرف ان الميثود تكون داخل المودل (Class), كيف يكون الوصول لها ؟ عن طريق تحديد ذلك في بداية تعريف الميثود , كأن نريد نصل لها من كلاس اخر , أو مشروع آخر وهكذا .

بينما Modifier للميثود بغالبية لغات البرمجة يكون على نوعين أما Static أو Instance , الفرق بينهما ان Static لا نحتاج الى عمل أوبجكت من الكلاس الذي داخله الميثود , بينما الثاني يحتاج ذلك .

لنعمل مشروع جديد من نوع (Console Application) في الفيجوال ستوديو (أي أصدار) , بصورة أفتراضية سوف نجد الفيجوال ستوديو عمل لنا ميثود أسمها (Main)



هذه الميثود من نوع Static حتى تنفذ بالبداية قبل أنشاء أي (Instance) للكلاسات , ومن نوع Void وكما قلنا سابقآ هذا النوع من الميثود يسمى بالبروسيجر (Procedure) وهي تنفذ ولا تعيد أي قيمة , البراميترز لهذه الميثود من نوع مصفوفة نصية .

طيب لنعمل ميثود أخرى تقوم بأيجاد أكبر رقم من بين ثلاث ارقام يدخلها المستخدم .

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

```
namespace First_project
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Enter First Number : ");
            double f_Number = double.Parse(Console.ReadLine());
            Console.Write("Enter Second Number : ");
            double s_Number = double.Parse(Console.ReadLine());
            Console.Write("Enter Third Number : ");
            double t_Number = double.Parse(Console.ReadLine());
            double result = maxValue(f_Number, s_Number, t_Number);
            Console.WriteLine("the max value is : " + result);
            Console.ReadKey();
        }
        static double maxValue (double x ,double y ,double z)
        {
            double max = x;
            if (y > max)
                max = y;
            if (z > max)
                max = z;
            return max;

        }
    }
}
```

في الشفرة أعلاه عملنا ميثود باسم (maxValue) من نوع double وبالتالي فأنها يجب أن ترجع قيمة من نوع double, والموديفاير لها static حتى ما نحتاج نعرف متغير للكلاس الذي يحويها , هذه الميثود فيها ثلاث معاملات من نوع double أيضاً , ثم أجرينا المقارنات حتى نستخرج الأكبر .

استدعاء الميثود جدا بسيط , كما تلاحظ أننا استدعينا الميثود داخل main وكما نعرف كلاهما داخل نفس الكلاس وان maxValue من نوع static , لذلك فقط كتبنا أسمها , بينما لو كانت في كلاس آخر وحتى نستدعيها يجب ان تكون الوصولية لها أما public او protected أو Internal (سنتكلم عن هذا فيما بعد) . وتكون صيغة الوصول لها بهذا الشكل :

Classname.methodname();

لنعيد المثال السابق ونحذف static من الميثود maxValue , حيث أفتراضياً اذا لم نذكر كلمة Static عند تعريف الكلاس او الميثود فهو سوف يعتبره Instance , وبالتالي وحتى نصل للميثود يجب علينا عمل أنستانس للكلاس .

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

namespace First_project
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Enter First Number : ");
            double f_Number = double.Parse(Console.ReadLine());
            Console.Write("Enter Second Number : ");
            double s_Number = double.Parse(Console.ReadLine());
            Console.Write("Enter Third Number : ");
            double t_Number = double.Parse(Console.ReadLine());
            Program prg = new Program();
            double result =prg.maxValue(f_Number, s_Number, t_Number);
            Console.WriteLine("the max value is : " + result);
            Console.ReadKey();
        }
         double maxValue (double x ,double y ,double z)
        {
            double max = x;
            if (y > max)
                max = y;
            if (z > max)
                max = z;
            return max;

        }
    }
}
```

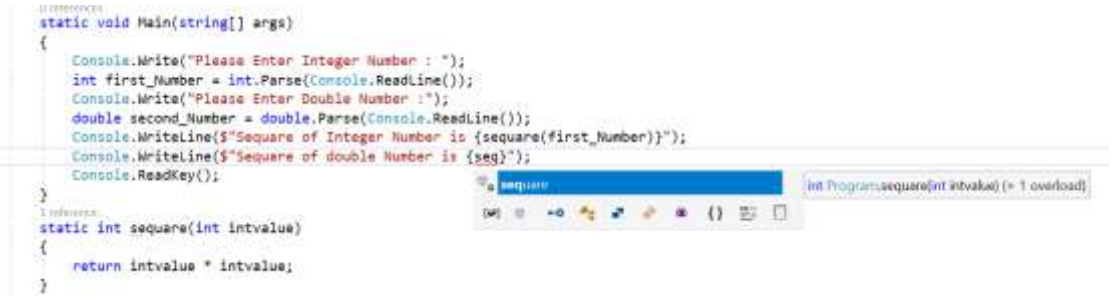
السطر (Program prg = new Program();) هذا لعمل انستانس للكلاس حتى نستطيع الوصول للميثود (maxValue) .

عندنا في الميثود مصطلح مهم هو (OverLoading) هذا يعني ان نعمل أكثر من ميثود بنفس الأسم , مثلا لو أردنا أيجاد مربع الأرقام التي من نوع int وكذلك مربع الأرقام التي من نوع double .

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

namespace First_project
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Please Enter Integer Number : ");
            int first_Number = int.Parse(Console.ReadLine());
            Console.Write("Please Enter Double Number :");
            double second_Number = double.Parse(Console.ReadLine());
                      Console.WriteLine($"Sequare of Integer Number is
          {sequare(first_Number)}");
                      Console.WriteLine($"Sequare of double Number is
          {sequare(second_Number)}");
```

```
            Console.ReadKey();
        }
        static int sequare(int intvalue)
        {
            return intvalue * intvalue;
        }
        static double sequare(double doubleValue)
        {
            return doubleValue * doubleValue;
        }

    }
}
```

بحيث عند كتابة فقط أسم الميثود بالفيجوال سيظهر لنا بجانب تفاصيل الميثود كلمة overloading 1+ كما في الصورة ادناه :



نرجع للمعاملات في الميثود , كما قلنا سابقا أن المعاملات تكون أختيارية وحسب الحاجة , المعاملات (parameters) تكون على عدة أنواع وكالتالي :

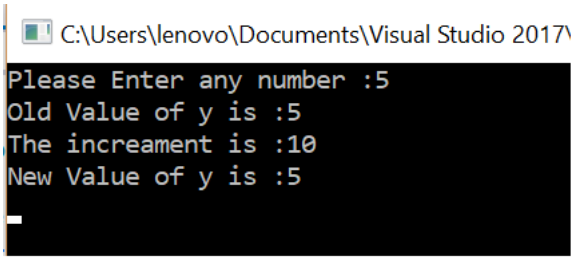
Value Parameters

عندما نمرر متغير الى الميثود فأن قيمة وعنوانها يبقى كما هو لا يتغير خارج الميثود , شلون ؟ ؟ ؟ ناخذ مثال بسيط آخر

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

namespace First_project
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Please Enter any number :");
            int y = int.Parse(Console.ReadLine());
            Console.WriteLine($"Old Value of y is :{y}");
            Console.WriteLine($"The increament is :{Inc( y)}");
            Console.WriteLine($"New Value of y is :{y}");
            Console.ReadKey();
        }
        static int Inc( int x)
        {
            x += 5;
```

```
            return x;
        }


    }
}
```

عندنا ميثود Inc مهمتها جمع المعامل مع خمسة وارجاع الناتج , اذا نفذت الشفرة أعلاه سوف يكون كالتالي :



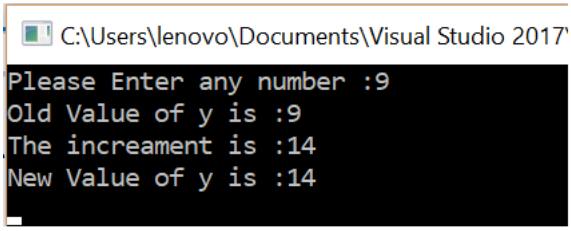
أي أن قيمة y لم تتغير رغم أنها جمعت مع 5 في الميثود Inc , هذا المقصود Value Parameter أي ان قيمة المتغير تبقى كما هي بدون أن تتأثر , هذا النوع يكون افتراضي بحيث أنك حتى تستخدمه لا يجب ان تكتب شيء عند تعريف المعاملات في الميثود (int x) .

Reference Parameters

بهذا النوع قيمة المتغير سوف تتأثر , لو أعدنا المثال السابق :

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

namespace First_project
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Please Enter any number :");
            int y = int.Parse(Console.ReadLine());
            Console.WriteLine($"Old Value of y is :{y}");
            Console.WriteLine($"The increament is :{Inc(ref y)}");
            Console.WriteLine($"New Value of y is :{y}");
            Console.ReadKey();
        }
        static int Inc(ref int x)
        {
            x += 5;
            return x;
        }
```

```
    }
}
```



اذا تلاحظ في تعريف الميثود ذكرنا (ref) وحتى عند الأستدعاء أيضا . وقيمة y الأولى كانت 9 بينما بعد تمريرها للميثود Inc أصبحت 14 .

: Out Parameters

هذا النوع مشابه لـ Reference Parameter , حيث نضع out بدل ref في التعريف والاستدعاء , الاختلاف فقط ان المعامل الذي يمرر سوف ستند له قيمة داخل الميثود ! (صعب ☹) لنأخذ مثال ميثود التربيع :

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

namespace First_project
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Enter y Value : ");
            int y = int.Parse(Console.ReadLine());
            int x;
            Console.WriteLine($"The original y is {y}");
            Console.WriteLine("The Original x is uninitialized ");
            seqRef(ref y);
            seqOut(out x);
            Console.WriteLine($"The square Ref y is {y}");
            Console.WriteLine($"The square Out x is {x}");
            seq( y);
            seq( x);
            Console.WriteLine($"Thesequare y is {y}");
            Console.WriteLine($"The square x is {x}");
            Console.ReadKey();
        }
        static void seqRef (ref int y)
        {
          y=  y * y;
        }
        static void seqOut(out int x)
```

```
        {
            x = 3;
            x = x * x;
        }
        static void seq ( int val)
        {
            val = val * val;
        }

    }
}
```

التنفيذ يكون بالشكل التالي :

كما تلاحظ أن قيمة x أسندت في الميثود seqout , وهكذا أصبحت قيمتها تساوي 9 بعد عملية الضرب . السؤال الذي يطرح نفسه الان : لماذا بعد استدعاء الميثود seq قيمة y و x لم يتغيرو ؟ لأن الميثود seq معاملاتها من نوع value Parameters لذلك خارج الميثود لن يتأثرو .

: Optional Parameters

نأخذ مثال حتى تتوضح فكرته بدل من الكلام النظري , لنفترض أردنا عمل ميثود نجري بداخلها عملية ضرب لثلاث أرقام :

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

namespace First_project
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(sum(3, 4, 5));
            Console.WriteLine(sum(3, 4));
            Console.WriteLine(sum(3));
            Console.ReadKey();
        }
        static int sum (int first , int second , int third)
        {
```

```
            return first + second + third;
        }
        static int sum (int first , int second)
        {
            return first + second;
        }
        static int sum (int first)
        {
            return first;
        }

    }
}
```

أذا تلاحظ الميثود sum عملنا لها اثنين overload , طيب في مثل هكذا حالات ما تحتاج تعمل overloading وتعقد الموضوع على نفسك , كيف ؟ غالبية لغات البرمجة تعطيك شي اسمه المعامل الاختياري (Optional Parameters) يعني تعطي قيمة افتراضية لبعض المعاملات , فعند استدعاء الميثود تكون غير مجبور على تمرير بيانات لها واذا تركتها فارغة فأنها سوف تأخذ القيمة الأفتراضية .

لو كتبنا البرنامج أعلاه بأستخدام optional parameters سيكون بالشكل التالي :

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

namespace First_project
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(sum(3, 4, 5));
            Console.WriteLine(sum(3, 4));
            Console.WriteLine(sum(3));
            Console.ReadKey();
        }
        static int sum (int first , int second =0, int third =0)
        {
            return first + second + third;
        }

    }
}
```

أعتقد وضحت الصورة أكثر .

طيب بقى لنا شي أخير بالميثود , مثل ما تعرف Net Framework. فيها مئات الكلاسات , وكل كلاس فيه عدة ميثود , هذه الميثود والكلاسات تأتينا جاهزة وتسمى (Built-in) أي أنها مبنيه مسبقآ , لكن ! ماذا بعد , لنأخذ واحدة من الميثود المبنية مسبقآ وهي (console.writeline()) لاحظت أن هذه الميثود تقبل عدد غير محدود من المعاملات , كيف ذلك ؟

في السي شارب وبقية اللغات عندنا مفهوم أسمه (**Parameter Array**) هذا شنو يعني ؟ يعني نمرر مصفوفة من المعاملات للميثود ونقدر نحدد عدد المعاملات في المصفوفة او نتركه غير محدود لنأخذ مثال :

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

namespace First_project
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(Summation(2, 3, 4, 5, 6, 7, 8, 9));
            Console.ReadKey();
        }
       static int Summation (params int[] numbers)
        {
            int sum = 0;
            foreach (var n in numbers )
            {
                sum += n;
            }
            return sum;
        }

    }
}
```

في تعريف المعاملات في رأس الميثود نذكر pararms ثم نوع المصفوفة (طبعا بالإمكان تمرين أي نوع من Generic) , اما الاستدعاء فيكون نمرر القيم مفصولة بفارزة , او بهذا الشكل :

```
int s = Summation(new[] { 1, 2, 3 });
```

: Extension Methods

كما أسلفنا ان الدوت نت فريموورك تتكون من مئات الكلاسات (Built-in) وهذه الكلاسات بدورها تحتوي على Methods جاهزة , طيب لو أفترضنا رغبت أن تضيف ميثود لأحد هذه الكلاسات تقوم بمهمة معينة ؟ الجواب يكون بعمل

Extension Method , بمعنى آخر أنه يمكنك أضافة ميثود لكلاس خاص موجود أو الى أحد كلاسات الدوت نت فريموورك أو كلاس Third Parity بل وحتى للـ Interface أو struct . . . الخ , بدون التعديل على الكلاس الأصلي أو تغييره .

طبعاً هذا النوع من الميثود تعتمد عليه تقنية LINQ وبكثرة .

لنأخذ مثال بسيط حتى تتوضح الفكرة , المثال التالي سوف نضيف ميثود تستخرج الأكبر لنوع البيانات int (الأتجر هو Struct) , بالبداية نعرف Extension Method داخل مشروعنا ويكون كالتالي :

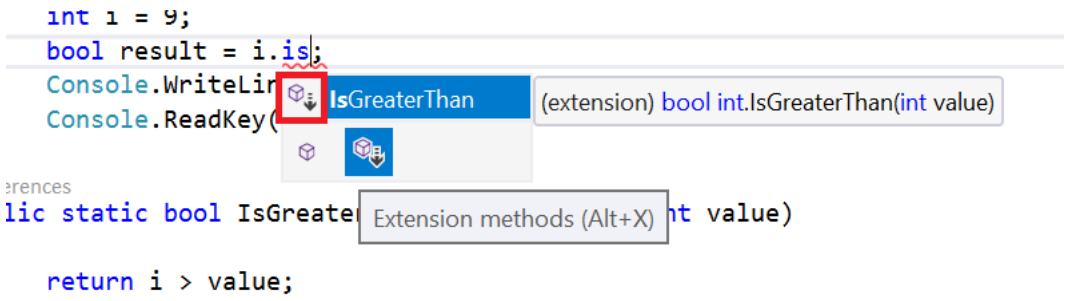
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace First_project
{
    static class Program
    {
        static void Main(string[] args)
        {
            int i = 9;
            bool result = i.IsGreaterThan(13);
            Console.WriteLine(result);
            Console.ReadKey();
        }
      public static bool IsGreaterThan (this int i , int value)
        {
            return i > value;
        }
    }
}
```

في الشفرة أعلاه عرفنا ميثود من نوع bool أسمها IsGreaterThan , وهذه الميثود تتقبل معاملين , لاحظ أستخدمنا كلمة this مع المعامل الأول وبالتالي ستكون هذه الميثود مرتبطة به (أي مرتبطة بالـ Integer) , والمعامل الثاني هو الرقم الاخر الذي سوف نقارن به .

الأستدعاء جداً بسيط :

```
bool result = i.IsGreaterThan(13);
```

لاحظ شكل Extension method كيف يكون :



أعتقد فكرتها وضحت !

لنأخذ مثال آخر عنها , كما تعرف ان نوع البيانات string ليس فيه ميثود تقلب النص (Reverse) لنعمل ميثود تنفذ ذلك :

```
public static string Reverse (this string s)
        {
            var chars = s.ToCharArray();
            Array.Reverse(chars);
            return new string(chars);
        }
```

ربطنا ال Extension Method بنوع البيانات string من خلال إضافة كلمة this للبارامتر الأول , ثم عرفنا أوبجكت من نوع مصفوفة حروف أسمه chars , ووضعنا فيه النص الذي نريد معكوسه , ثم عكسنا مصفوفة الحروف , وأرجعناه على شكل نص جديد (أي أننا أستفدنا من ميثود Reverse الموجوده في المصفوفات) .

أما الأستدعاء سيكون بالشكل التالي :

```
static void Main(string[] args)
        {
            Console.Write("Enter Your Name :");
            string name = Console.ReadLine();
            Console.WriteLine("the reverse of your name is :" +name.Reverse()
);
            Console.ReadKey();
        }
```

هل وضحت فكرتها ؟ طيب هناك ملاحظتين يجب الأنتباه لها عند أستخدام هكذا نوع من الميثود وهي :

1- المعامل الأول يجب أن نسبقه بكلمة this , ويجب أن يكون نوعه نفس النوع الذي نريد التطبيق عليه .

2- ممكن أن نستخدم Extension Methods في أي مكان بالمشروع , أو استدعاء ال Namespace الذي بداخله هي .

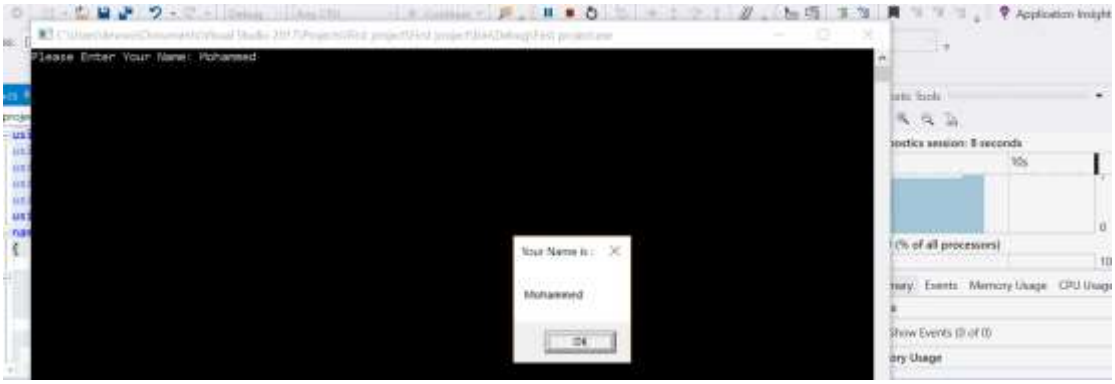
: Extern Methods

نوع آخر من الميثود , فكرته أنك ممكن تنشىء ميثود داخل الكلاس وتنفذ شيء خارجي ! الاستعمال الشائع لها يكون مكتبات dll حتى لو كانت هذه المكتبات مكتوبة بأية لغة أخرى غير السي شارب .

لنأخذ مثال بسيط , لنفترض في بيئة الكونسول أردت أن يقوم المستخدم بكتابة أسمه ويظهر Message Box فيه أسمه (هذا الشيء غير موجود في بيئة الكونسول) .

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Runtime.InteropServices;
namespace First_project
{
    static class Program
    {
        [DllImport("User32.dll", CharSet = CharSet.Unicode)]
        public static extern int MessageBox(IntPtr h, string m, string c, int
type);

        static int Main()
        {
            string myString;
            Console.Write("Please Enter Your Name: ");
            myString = Console.ReadLine();
            return MessageBox((IntPtr)0, myString, "Your Name is :", 0);
        }
    }
}
```

وعند التنفيذ :



كما تلاحظ طريقة تعريف هذه الميثود نذكر فيها كلمة extern , وليس فيها Body مثل باقي الميثود . والاستدعاء مثل باقي الميثود .

Chapter II

Classes, Objects, Constructors

ماهو الـ Class ؟

أي تطبيق سواء مكتبي أو ويب اوشبكي فهو عبارة عن مجموعة من الكلاسات وهي المسؤولة عن عمل التطبيق , والكلاس يشبه الى حد ما القالب , وعن طريق هذا القالب سوف نصنع كائنات جديدة تسمى ب (Object) .

الكلاس برمجيا يتكون من جزئين هما :

اولآ : البيانات (Data) : حيث نمثلها بالحقول (Fields)

ثانياً : السلوكيات (Behaviour) : وتمثل الميثودز .

على سبيل المثال لدينا كلاس خاص بالطلاب , سوف نمثله بلغة (Uml) بهذا الشكل :

persron
Id : int Name:string Birth :Date Photo :Byte
Eat() Walk() Sleep() Talk()

لغة uml عبارة عن لغة رسومية نمثل فيها الكلاسات والعلاقة بينهم , في هذه اللغة يكون الكلاس يتكون من ثلاث أجزاء يبدأ باسم الكلاس ثم بيانات الكلاس او الفيلدز , وبالاخير سلوكيات الكلاس او الميثودز , في مثالنا أعلاه اسم الكلاس person والفيلدز هم id ,name, birth ,photo اما سلوكيات البيرسون وهم المشي والأكل والنوم التكلم . . . الخ .

الأوبجكت يُشتق من الكلاس ويكون موجود في الذاكرة Memory على بسيل المثال الكلاس السابق نستطيع أن نعمل منه أوبجكت لشخص أسمه محمد و مصطفى وعلي . . . . الخ

بمعنى آخر الكلاس مثل النوع (Data Type) ونعرف أوبجكت من نوعه .

كيف نعرف الكلاس ؟

حسب الصيغة التالية :

```
<access_modifier> class <classname>
{

}
```

مثلا لو عرفنا الكلاس person :

```
public class person
{
    // Fields
    public string Name ;

    // Methods
    public void Introduce()
    {
        Console.Writeline("Hi , my name is : "+ Name);
    }
}
```

مثل ما تلاحظ الأكسس موديفاير جعلناه Public حتى يمكن الوصول له من أي مكان , اما الحقول فهو Name عبارة عن متغير عادي , بالنسبة للميثود Introduce أعتقد واضحة (سنتكلم بالتفصيل عن أجزاء الكلاس فيما بعد) .

ولأنشاء أوبجكت من الكلاس person فالأمر بسيط جداً :

أما بهذا الشكل :

```
Person ahmed =new Person();
```

أوبهذا الشكل :

```
var ali = new Person();
```

وبعد أن عرفنا الأوبجكت , وحتى نستخدمه أي بمعنى نستخدم الفيلدز أو الميثود بداخله يكون بالشكل التالي :

```
ali.Name = "Ali";
```

```
ali.Introduce();
```

أعضاء الكلاس (Members) ونقصد بهم (Fields) و (Methods) يكونون على نوعين (سبق وتكلمنا عن هذا الشيء) :

الأول : Instance : يعني الوصول لهم يكون عن طريق الأوبجكت وبهذا الشكل :

```
var p = new Person();
p.Introduce();
```

الثاني : Static – يكون الوصول لهم عن طريق الكلاس فقط وليس من الأوبجكت , هذا النوع نستخدمه لما يكون عندنا بيانات بالذاكرة واحدة مثلا الوقت الحالي بصورة منطقية ما ممكن يكون عندنا أكثر من وقت حالي في نفس اللحظة وبهذا الشكل :

```
Person.Introduce();
```

مثال آخر على ميثود نستخدمها كثيراً من نوع static هي WriteLine الموجوده في الكلاس Console :

```
Console.Writeline();
```

وهذا هو الفرق بين نعرف الميثود او الفيلد ستاتك أو أنستانس .

: Constructors

بعد ما ننشـــىء الكلاس الذي من نوع Instance (يعني نقدر نعمل منه أوبجكت) , ونشـتق منه أوبجكتز وكما نعرف أن الأوبجكت الجديد فيه فيلدز (متغيرات) ليس فيها بيانات , وحتى نسند لها قيم (يعني قيم أولية Initialize) سوف نحتاج الى تعريف ميثود خاصة داخل الكلاس الأصلي تحمل نفس أسم الكلاس وهذا ما يسمى (Constructor) ويستدعى بصورة تلقائية عند أنشاء أي أوبجكت .

في الأمثلة السابقة لم نصنع أي كونستركر لذلك الكومبايلر الخاص بالسي شارب يصنعه بصورة أفتراضية ويعطي قيمة صفر للفيلدز التي من نوع أرقام و Null لبقية الحقول التي من نوع Reference وقيمة False للذي من نوع Boolean .

طريقة تعريف الكونستركتر في الكلاس بسيطة :

```
<access_modifier> <name_of_class>()
{

}
```

على سبيل المثال لو أردنا أن نعمل كونستركتور لكلاس الزبائن الذي يحوي على حقل Name .

```
public class Customer
{
    //fields
    public string Name ;
    public int Age ;
    //Constructor
    public Customer (string n )
    {
        this.Name = n ;
        this.Age = 29 ;
    }
}
```

لدينا حقلين Name , Age , فقط عرفناهم ولم نسند لهم أي قيمة لأن هذه الحقول سوف يستخدمها أكثر من أوبجكت , وحتى ننشىء قيم أبتدائية لهذه الحقول عرفنا الكونستركتر ومررنا له معامل من نوع نص هو n (الذي سوف يحمل القيمة للحقل

Name) , وكذلك الحقل Age , هنا أستخدمنا كلمة this والتي نعني بها ان الاسناد سوف يكون من الأوبجكت , بالتالي لو أفترضنا عندنا أثنين أوبجكت من هذا الكلاس فأن كل أوبجكت سوف يكون فيه حقل الأسم مختلف عن الآخر بأستثناء حقل العمر الذي أسندنا له قيمة أفتراضية (29) .

طيب من نعرف أوبجكت كيف يكون السيناريو ؟ بهذا الشكل :

```
var customer1 = new Customer ("Zaid");
var customer2= new Customer ("Mohammed");
```

كما تلاحظ ان الأوبجكت الأول حقل الأسم فيه أسند له أسم زيد , بينما الأوبجكت الثاني محمد ,بينما حقل العمر بقي كما هو 29 .

وبما أن الكونستركتر هو ميثود لذلك ممكن نعرف اكثر من كونستركتر بنفس الأسم وبمعاملات مختلفة (Overloading) حسب الحاجة كما أدناه :

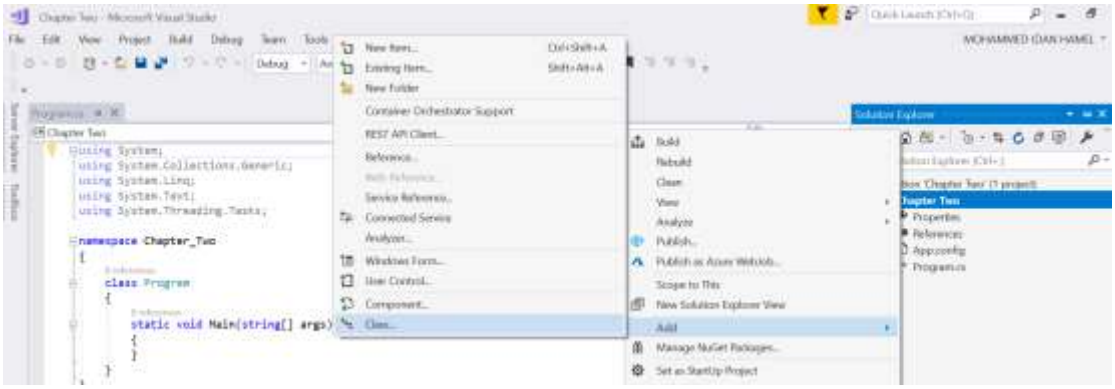
```
public Customer ()
    {

    }
    public Customer (string n )
    {
        this.Name = n ;

    }
        public Customer (string n , int a )
    {
        this.Name = n ;
        this.Age = a ;
    }
```

الكونستركتر لا يرجع أي قيمة ولا يمكن أن نستخدم معه void مثل بقية الميثودز .

في الأمثلة السابقة كانت الفيلدز من نوع (Value Type) أو كما تسمى في لغة الجافا بـ (Primitive Type) ماذا لو كان لدينا فيلد من نوع Reference وهذا شائع الاستخدام في كثير من التطبيقات ؟

على سبيل المثال لدينا فيلد من نوع Generic list هذه القائمة لنفترض تحمل مجموعة أوبجكت من نوع كلاس اخر ,

لنعمل مشروع جديد من نوع كونسول نسميه (Chapter Two) وداخله نضيف كلاس نسميه (Customer) وكلاس آخر من اسمه (Orders) :



داخل الكلاس Customer :

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

namespace Chapter_Two
{
   public class Customer
    {
        //Fields
        public int Id;
        public string Name;


        //Constructors
        public Customer()
        {

        }
        public Customer(int id)
        {
            this.Id = id;
        }
        public Customer(int id ,string name)
        {
            this.Id = id;
            this.Name = name;
        }

    }
}
```

أما كلاس Order فهو فارغ مجرد كلاس عادي .

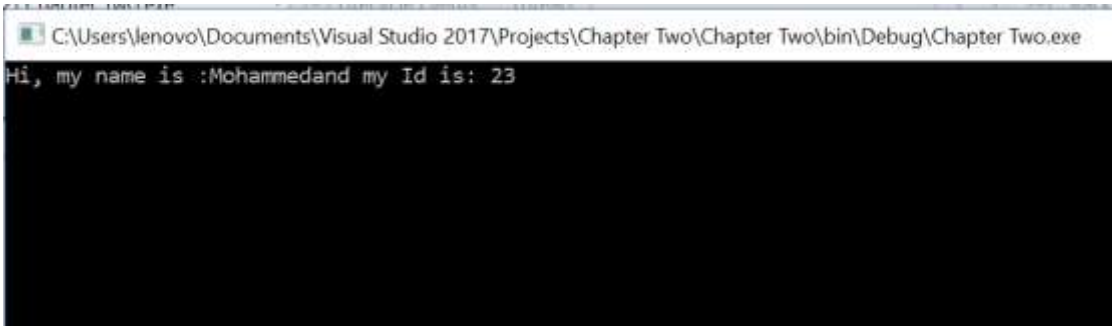
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Chapter_Two
{
  public class Orders
    {
    }
}
```

طيب, نرجع الى البرنامج الأصلي الذي اسمه (Program)

وفي الشفرة التالية :

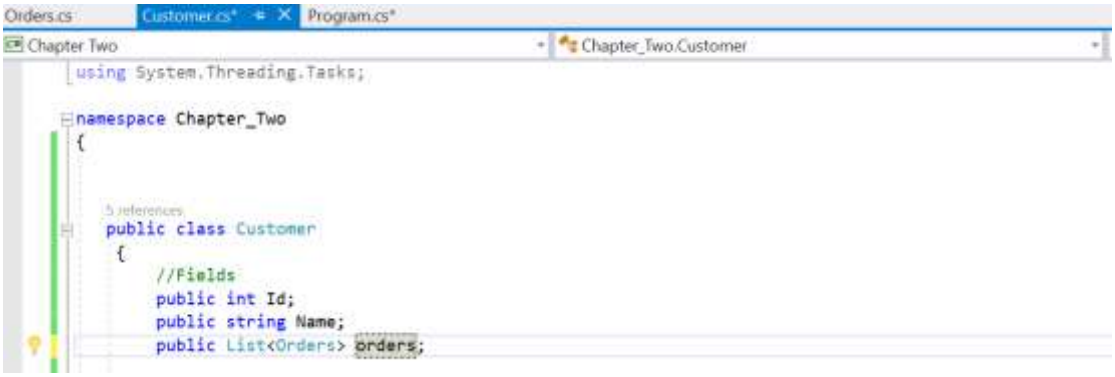
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Chapter_Two
{
    class Program
    {
        static void Main(string[] args)
        {
            Customer ctm = new Customer();
            ctm.Id = 23;
            ctm.Name = "Mohammed";
            Console.WriteLine("Hi, my name is :"+ctm.Name + "and my Id is: "+
ctm.Id);
            Console.ReadKey();
        }
    }
}
```

الان عند التنفيذ الأمور تمام وليس لدينا مشكلة لان الحقول التي تعاملنا معها من نوع Permitve type ,

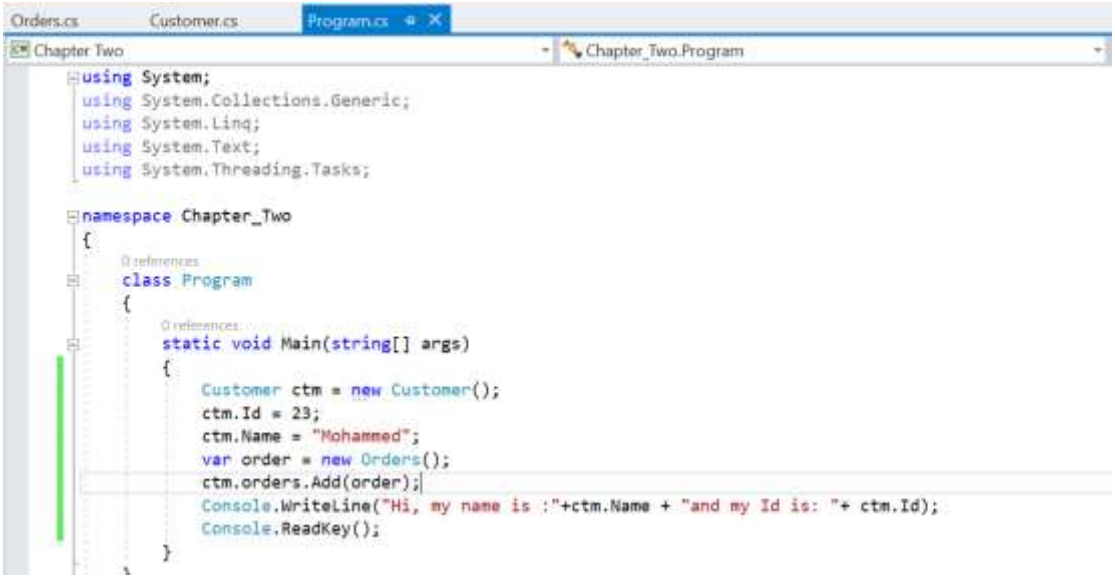


قلت سابقا ان الحقول التي من نوع Reference اذا لم نعمل لها قيم أبتدائية فأنها سوف تكون Null , والمشكلة التي ستواجهنا ان هذه الحقول مادام Null فأننا لن نستطيع ان نجري عليها أي شي .

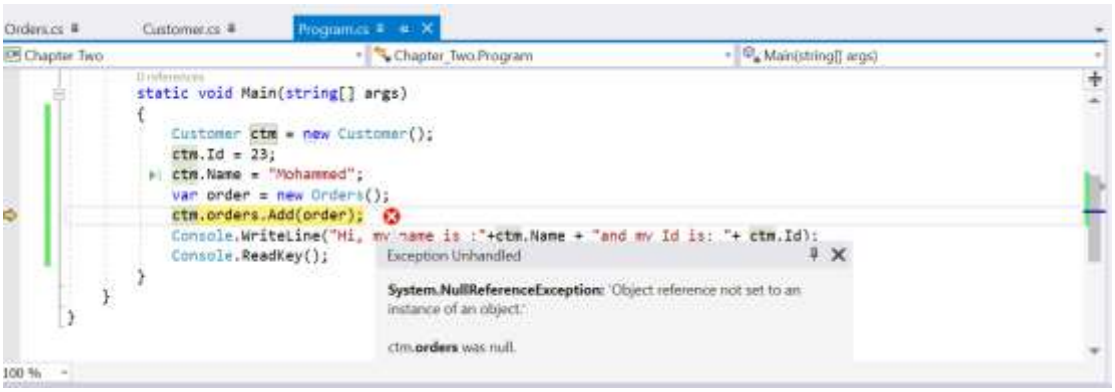
نعود الى مثالنا أعلاه ونضيف حقل جديد من نوع Generic list يحمل مجموعة من الأوبجكت التي من نوع Orders .



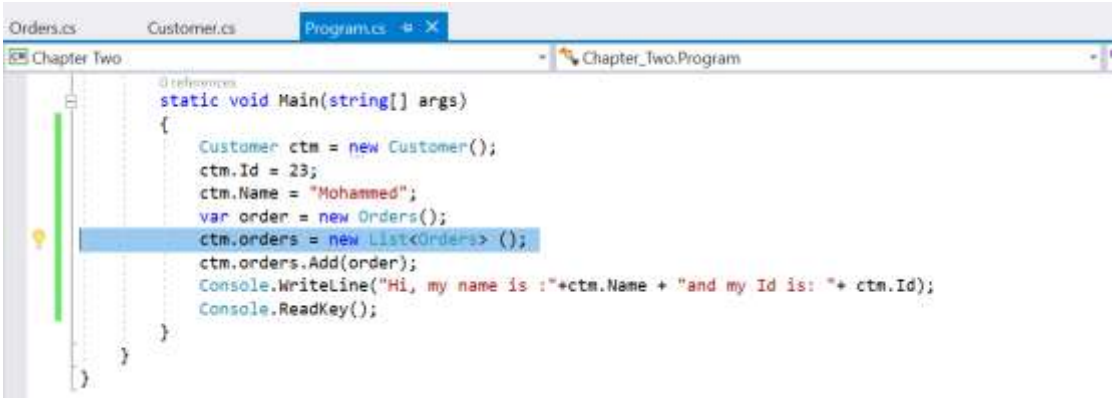
نرجع لـ Program , ونضيف Order جديد , كالتالي :



طيب نفذ البرنامج الآن :



هذا الخطأ يحصل لإن الأوبجكت Orders قيمة الأبتدائية Null لذلك عندما أستدعينا الميثود Add أظهر لنا هذا الخطأ , طيب وحتى نحل هذه الخطأ لدينا حل بسيط جدا وهو :



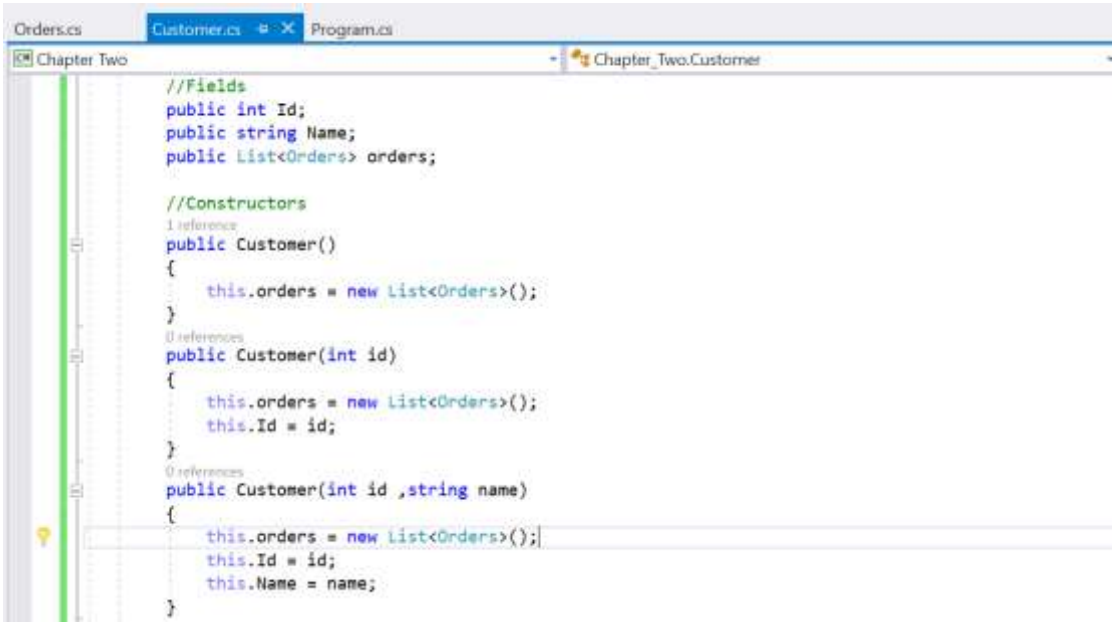
لكن هل هذا صحيح ؟ أي هل صحيح ان نعمل Instance لفيلد معين داخل كلاس Program ؟ بالطبع لا ! لأن واحده من مهام برمجة الكيانات الموجهة هي Encapsulation أي أخفاء هكذا تفاصيل عن المستخدم وعدم وضعها في الكلاسات التي يتعامل معها المستخدم مباشرة .

طيب ما الحل ؟ الأمر بسيط أيضاً , تذكر أننا عملنا كونستركتر للكلاس Customer , نقدر من هناك نعمل Instance للـ Orders وكالتالي :

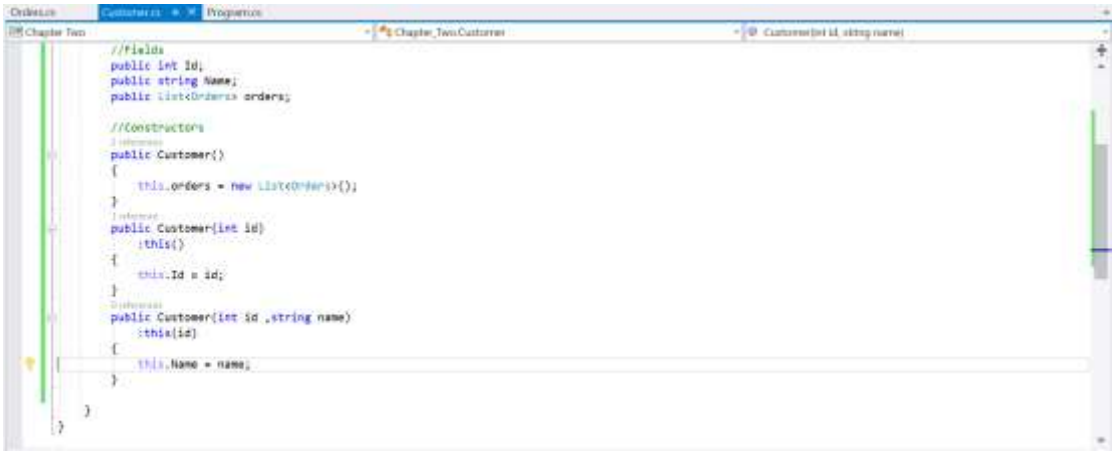
قبلها لا تنسى تحذف السطر المظلل في الصورة السابقة :

الان نفذ الأمر طبيعي !

اذا لاحظت في كلاس Program نحن أستدعينا الكونستركتر الأول الخاص بالكوستمر (الذي وضعنا فيه أنستانس الOrders) ماذا لو أستدعينا غير كونستركتر , ايضا سيظهر لنا خطأ , لذلك يجب ان نكتب سطر الانستانس الخاص بال Orders في جميع كونستركترز الكلاس كوستمر , وبهذا الشكل :



على سبيل الفرض لو كان لدينا خمسة حقول من نوع List , هل صحيح أن نكتب سطر الانستانس لهم في جميع الاسطر ؟ الا يعني هذا ان الأسطر قد زادت ؟ طيب لدينا حل بسيط جدا وهو نستدعي كونستركتر داخل كونستركتر وكالتالي :



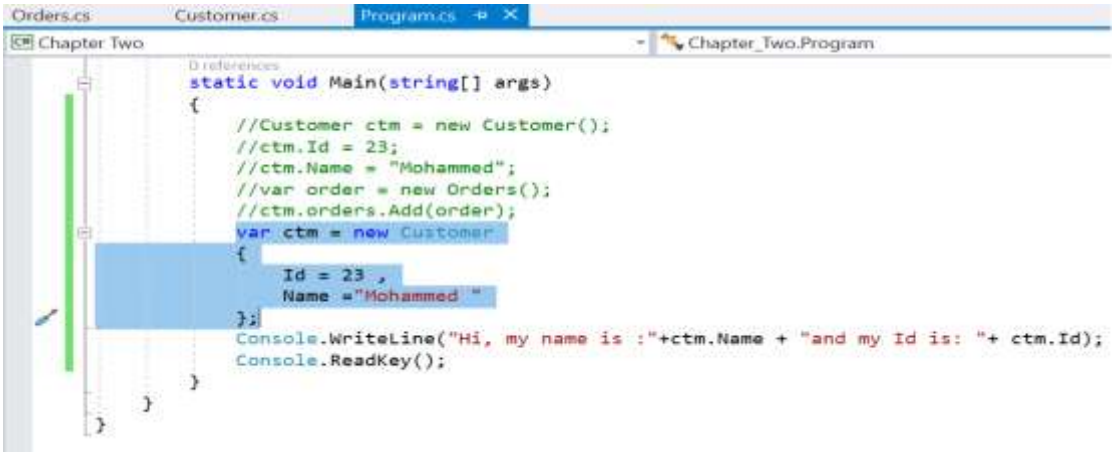
```
//Fields
public int Id;
public string Name;
public List<Orders> orders;

//Constructors
public Customer()
{
    this.orders = new List<Orders>();
}
public Customer(int id)
    :this()
{
    this.Id = id;
}
public Customer(int id ,string name)
    :this(id)
{
    this.Name = name;
}
}
}
```

شنوصار ؟

لدينا ثلاثة Constructors , الأول ليس فيه معاملات , الثاني فيه واحد فقط وهو id , والثالث فيه أثنين هما (id , name لاتنسى أن السي شارب لغة حساسة للحروف الكبيرة والصغيرة أي أن Id ليس مثل id) , كل ما فعلنا وحتى لا نعيد كتابة سطر عمل انستانس للـ Orders في بقية الكونستركترز , قمنا في الكونستركتر الثاني باستدعاء الأول من خلال (()this) , أما في الكونستركتر الثالث قمنا بأستدعاء الثاني وحذفنا سطر (this.Id=id) لأن هذا السطر هو موجود في الكونستركتر الثاني . واذا تلاحظ عندما استدعيناه في الثالث مررنا لها معامل (this(id)) .

أخيراً حاول قدر الأمكان لا تعمل الكثير من الكونستركتر , وانما حسب الحاجة لأن استدعاء كونستركتر أو تكرار الشفرات قد يسبب الكثير من المشاكل .

والأفضل عندما نعمل أوبجكت نعمل Initialize للفيلدز , يعني مثلاً كلاس Customer عملنا له اثنين كونستركتر حتى نسند قيم للحقلين Id , Name بينما بأمكاننا ان نعمل الاوبجكت من داخل الكلاس Program بهذا الشكل ونسند قيم للفيلدز .



```
Orders.cs    Customer.cs    Program.cs
Chapter Two                 Chapter_Two.Program
        static void Main(string[] args)
        {
            //Customer ctm = new Customer();
            //ctm.Id = 23;
            //ctm.Name = "Mohammed";
            //var order = new Orders();
            //ctm.orders.Add(order);
            var ctm = new Customer
            {
                Id = 23 ,
                Name ="Mohammed "
            };
            Console.WriteLine("Hi, my name is :"+ctm.Name + "and my Id is: "+ ctm.Id);
            Console.ReadKey();
        }
    }
}
```

Chapter III

Fields , Properties, Indexers

الحقول Fields :

عبارة عن متغيرات نعرفها داخل الكلاس لخزن البيانات والأكسس موديفاير لها اما Private أو Protected وذلك لاخفاء البيانات فيها عن المستخدم (هذا واحد من غايات برمجة الكائنات الموجهة) , طريقة تعريفها دائما تكون في بداية الكلاس ومثلها مثل أي متغير عادي , طيب مادام هي برايفت كيف راح نوصل لها من خارج الكلاس ونخزن أو نستدعي منها البيانات ؟ طبعاً هناك طرق كثيرة للوصول الى الحقول وحسب نوع الحقل هل هو Static أو Instance فيما بعد سوف نختار افضل طرق الوصول .

أن كان الحقل من نوع Instance فالوصول له قد يكون بأبسط طريقة وهي كالتالي :

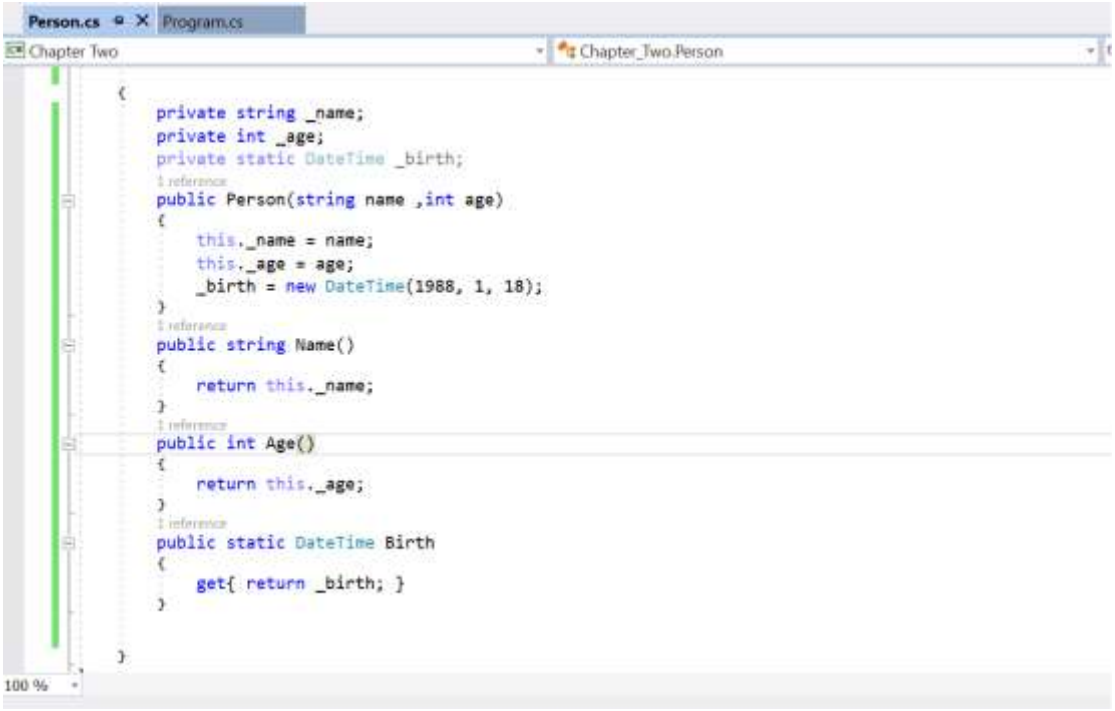
```
Person.cs  Program.cs
Chapter Two                                  Chapter_Two.Person
namespace Chapter_Two
{
    2 references
    public class Person
    {
        private string _name;
        private int _age;

        1 reference
        public Person(string name ,int age)
        {
            this._name = name;
            this._age = age;
        }
        1 reference
        public string GetInfo()
        {
            return this._name + this._age;
        }

    }
}
```

في المثال أعلاه لدينا كلاس فيه حقلين , وقد رنا نسند قيم للحقول من خلال الكونستركتر , اما استرجاع البيانات من هذه الحقول عملنا لها ميثود ترجع قيم الحقلين لكن ! كيف نرجع قيمة كل حقل منفصلة عن الأخرى , الامر بسيط نعمل ميثود الاسترجاع لكل واحد وكالتالي :

```
Person.cs  Program.cs
Chapter Two                                  Chapter_Two.Person
{
    2 references
    public class Person
    {
        private string _name;
        private int _age;

        1 reference
        public Person(string name ,int age)
        {
            this._name = name;
            this._age = age;
        }
        0 references
        public string Name()
        {
            return this._name;
        }
        0 references
        public int Age()
        {
            return this._age;
        }

    }
}
```

اما الاستدعاء كالتالي يكون :

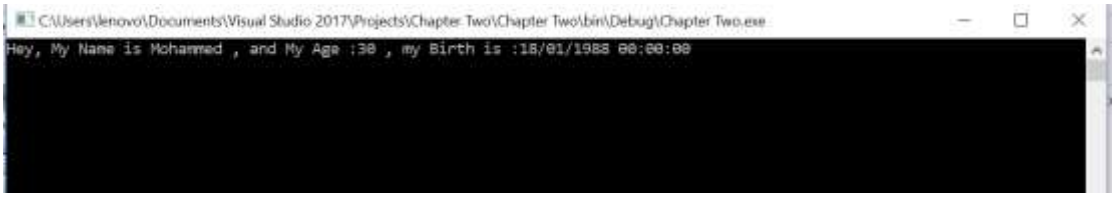
هذه طريقة من طرق كثيرة تقدر نستخدمها للتعامل مع الحقول , طيب ماذا عن الحقل الذي يكون Static ؟



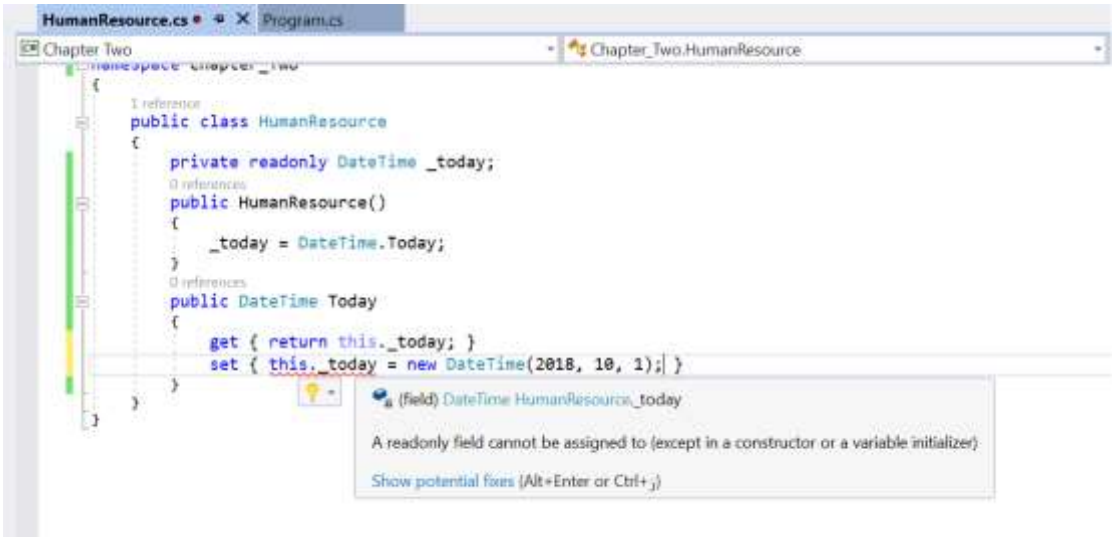
لاحظ أننا استعملنا ميثود خاصة لاسترجاع الحقل الذي من نوع Static وهذه الميثود تسمى ب Properties التي سوف نتكلم عنها لاحقآ بالتفصيل , وجعلنا نوعها Static . اما بالنسبة لأسناد القيمة فداخل الكونستركتر اسندنا القيمة .

طيب كيف نستدعي هذه الميثود في الكلاس الاخر :

ولأن الحقل من نوع Static لذلك الوصول له يكون من الكلاس وليس الأوبجكت (الكلاس اسمه Person بينما الأوبجكت أسمه person) .

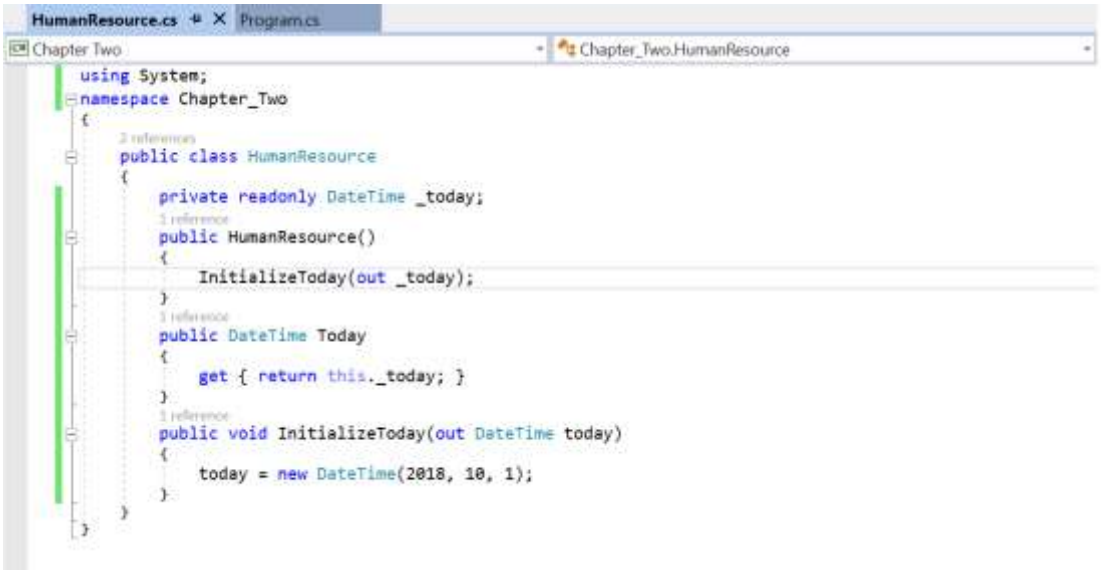


لنفترض وحسب الحاجة أردت أن يكون أحد الحقول للقراءة فقط بحيث عندما تنشيء أوبجكت من الكلاس لا تستطيع تغيير قيمة الحقل , هذا يكون عن طريق شيء أسمه Read only , حيث نعمل له Initialization من داخل الكونستركتر أو اثناء انشاءه , على سبيل المثال التالي :



في المثال أعلاه اسندنا له قيمة من داخل الكونستركتر , وحين أردنا اسناد له قيمة مرة ثانية من داخل ميثود ال Properties ظهر لنا خطأ يقول أن أي حقل لا يمكن اسناد له قيمة الا من خلال الكونستركتر او انشاء المتغير .

طيب لو أنك أردت التعديل عليه من داخل ميثود ثانية فقط لا غير بهذه الحالة ومن داخل الكونستركتر ترسل الحقل على شكل out Parameter (راجع شرح الميثودز في بداية الدروس)



ماهي الخصائص Properties :

واحد من الأشياء المهمة في برمجة الكائنات الموجهة هو أخفاء البيانات أو ما يسمى بـ(Encapsulation) , يعني أن الحقول (Fields) الخاصة بكل كلاس لازم تكون مخفية , بمعنى تقني آخر أن نجعل الوصول لها Private , في الأمثلة السابقة جميعها كنا نجعلها Public , وكما نعرف أن الوظيفة الرئيسية للحقول في الكلاس هي تمرير البيانات للميثودز في نفس الكلاس , فأذا جعلنا الوصول لهذه الحقول برايفت معناه ما نقدر نمرر فيها أي شيء بل أصلا لن نصل لها , ومن هنا جائت فكرة الخصائص , حيث نعمل ميثودز لكل حقل , واحدة لأسناد القيم والثانية لأسترجاع القيم من الحقول , والكثير من الناس يسمون هذه الميثودز بـ (Get/Set) .

لننظر لهذا المثال , حيث لدينا كلاس فيه حقل واحد , ولاحظ كيف مررنا له البيانات :

```
class Students
    {
        private DateTime _birth;

        public void SetBirth (DateTime birth)
        {
            this._birth = birth;
        }
        public DateTime getBirth()
        {
            return this._birth;
        }
    }
```

طيب الآن في الكود أعلاه عملنا وصولية للحقل , طيب ماذا لو كان لدينا أكثر من خمسة حقول , لاحظ كم سيكون لدينا من أسطر كثيره , ولتقليل الشفرات سوف نستخدم ميثودز (get/set) وكالتالي :

```
class Students
```

```
    {
        private DateTime _birth;
        public DateTime Birth
        {
            get { return _birth; }
            set { _birth = value; }
        }
    }
```

الجميل في الدوت نت هناك خاصية جميلة تسمى Auto-Implemented Properties , هذه بدل ما تكتب الشفرة أعلاه وتعرف الحقل private , فأن كومبايلر السي شارب هو من سيقوم بذلك بدل عنك , يعني الشفرة أعلاه سوف نكتبها بهذا الشكل :

```
class Students
    {
        public DateTime Birth { get; set; }
    }
```

طيب ماذا لو أردنا واحدة من الخصائص تكون للقراءة فقط أي ممكن نقرأ من الحقل بدون ما نقدر نسند له قيمة وذلك بأضافة private للset :

```
class Students
    {
        public DateTime Birth { get; private set; }
    }
```

ماذا لو أردنا اجراء بعض التعديلات على الخاصية قبل ان نستدعيها , مثلا في المثال السابق لو أردنا ان نعمل خاصية تعطينا العمر , يكون ذلك كالتالي :

```
class Students
 {
     public DateTime Birth { get;set; }
     public int Age
     {
         get
         {
             var age = DateTime.Today - Birth;
             var years = age.Days / 365;
             return years;
         }
     }
 }
```

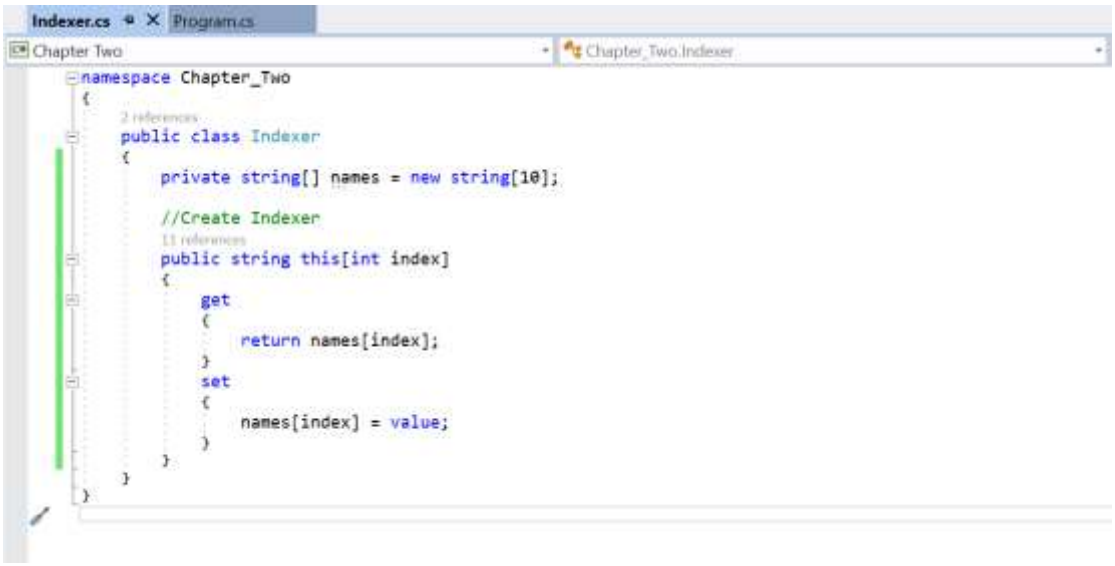
أما في الكلاس الرئيسي سوف نعمل اوبجكت من الكلاس student , ونستدعي العمر :

```
class Program
   {
       static void Main(string[] args)
       {
           var studnet = new Students();
           studnet.Birth =new DateTime(1988,1,18);
           Console.WriteLine(studnet.Age);
           Console.ReadKey();
       }
   }
```

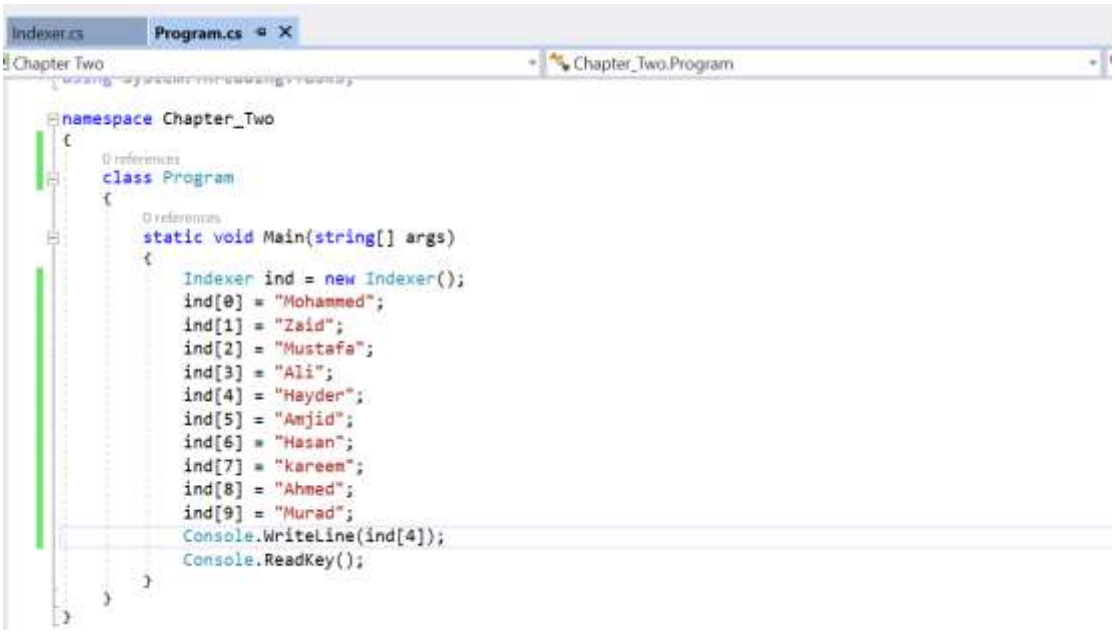
Indexers :

في السي شارب لو أفترضنا لدينا أوبجكت من نوع مصفوفة أو List أو Dictionary وحتى نصل لعناصر هذا الأوبجكت بسهولة وبسرعة نستخدم Indexer ,بشكل آخر الأندكسر عبارة عن أوبجكت من مصفوفة او قائمة لنأخذ مثال بسيط لتوضيح ذلك :

نعمل كلاس جديد باسم Indexer



وفي البرنامج:



في المثال السابق عرفنا الأندكسر من خلال الصيغة التالية:

```
<Access_Modifier> <type_of_Indexer> this [<typeIndex> <nameofIndex>]
{
……
}
```

ملاحظة جدا مهمة وهي الأندكسر يستخدم فقط عندما يكون لدينا حقل من الحقول من نوع Generic List او مصفوفة ... الخ.

Chapter IV

Inheritance , Composition

برمجة الكائنات الموجهة :

قلنا سابقاً أن الغاية الرئيسية من البرمجة الكائنية الموجهة هو تقليل وقت التطوير وسهولة صيانة التطبيق في حالة حدوث أي خطأ أو Bug (ال Error يعني خطأ في كتابة الشفرة بينما Bug تعني في خطأ منطقي أي ممكن التطبيق ينجز مهمة بصورة خاطئة) .

الأعمدة الرئيسية التي ترتكز عليها OOP ثلاثة سوف نتناولها بالتفصيل والأمثلة وهي :

1- Encapsulation
2- Relationships
3- Polymorphism

1- Encapsulation :

ترجمته للعربية التمحفظ ! أرجع وأقول الأفضل حفظ المفهوم على لغته الإنكليزية بدل مأيترجم , الأنكابسوليشن شو تعني ؟ قلت ولازلت أقول ممكن نعرف OOP على أنها طريقة لتنظيم الشفرات وهذه الطريقة تساعدنا على اكتشاف الأخطاء وتطوير التطبيق بصورة اسرع لأنها تقلل من تكرار الشفرات . التنظيم يعني أن نجمع الأشياء المتجانسة في مودل على سبيل المثال عندنا مجموعة من الشفرات خاصة بالتعامل مع قاعدة البينات وشفرات أخرى تتعامل مع الملفات وأخرى مع الطلبيات وأخرى مع الزبائن . . . الخ, فبدل ما نجمع تلك الشفرات في مكان واحد فأننا سوف نعزلها عن بعض بمودل لكل واحدة (يعني بكلاس لكل واحدة) وهذا الكلاس سوف يخزن في Storage (يعني في الهارد ديسك) ثم نعمل أوبجكت واحد أو أكثر لكل الكلاس حسب الحاجة ونتعامل معه في الأماكن الأخرى (الأوبجكت يخزن في الذاكرة RAM او الكاش) .

أذا الإنكابسوليشن نظيم الشفرات بكلاسات ولكل كلاس خصائص و سلوكيات , ولها وظيفة أخرى هي أخفاء البيانات (يعني المتغيرات التي تحمل البيانات) , وبالتالي تسمى ب Abstraction .

أعتقد مفهوم الإنكابسوليشن واضح وتناولناه كثيرا في الدروس السابقة .

2- Relationships :

أي تطبيق هو عبارة عن مجموعة من الكلاسات , والعلاقة بين هذه الكلاسات مهمة جداً (يعني الأرتباط وطبيعته بينهم) فمثلا قد يكون لدينا كلاس تعتمد عليه كلاسات أخرى في التطبيق وأي تغيير في هذا الكلاس سوف يؤثر على بقية الكلاسات مثل هذا النوع من العلاقات يسمى Tightly Coupling ولا ينصح به , ولدينا نوع ثاني من العلاقات يسمى بـ Loosely Coupling في هذا التغيير الذي يحصل بالكلاس لا يؤثر على البقية , أو قد يؤثر لكن بمقدار قليل .

أذا هذا المفهوم خاص بكيفية الأرتباط بين الكلاسات , ويكون على نوعين وهما :

أولاً : الوراثة Inheritance

ثانياً :التنضيد Composition

أولاً : الوراثة Inheritance :

نوع من العلاقات بين الكلاسات و تعني أن كلاس يرث من كلاس آخر الخصائص والميثودز الفائدة الرئيسية منها تقليل الشفرات من خلال إعادة أستخدام الشفرات مرة أخرى بالوراثة من الكلاس الذي يحويها , على سبيل المثال عندنا كلاس فيه ميثود وهذه الميثود نحتاج نفسها في كلاس آخر فبدل ان نكتب شفرة هذه الميثود في الكلاس الجديد فأننا سوف نرث من هذا الكلاس وبالتالي صار لدينا الميثود بدون أن نكتب شفرتها مرة أخرى .

الكلاس الأصلي الذي سوف نرث منه يسمى بالأب Parent وبعض الأحيان يسمى بـ Base Class أو Super Class, أما الكلاس الآخر يسمى بالأبن Child أو Derived Class أو Sub-Class . هناك صيغة لهذا النوع من العلاقات حيث تقول الصيغة :

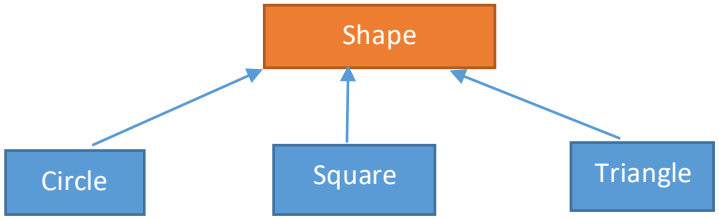
Is-A

يعني على سبيل المثال :

Car Is a vehicle.

السيارة هي مركبة (يعني الذي قبل Is هو الأبن أو الفرع , وبعد a هو الأب او الرئيسي .

على سبيل المثال لدينا مخطط UML التالي :



لدينا كلاس اصلي هو Shape , وكلاسات فرعية ,المثلث والمربع والدائرة , حيث جميع هذه الكلاسات تتوارث من كلاس الشكل .

هذا الكلام نظري طيب كيف سيكون بصورة برمجية :

لنعمل مشروع جديد نسميه Chapter Four ونظيف فيه كلاسين وكالتالي :

الآن داخل الكلاس Shape سوف نعرف خاصيتين واحدة للعرض والأخرى الطول , وكذلك ميثود لعرض أبعاد الشكل :

```
namespace Chapter_Four
{
    public class Shape
    {
        public int Width { get; set; }
        public int Length { get; set; }

        public void Diamenation()
        {
            System.Console.WriteLine($"the diamenation of width is {Width
} , and the length is :{Length}");
        }
    }
}
```

الان لنعمل كلاس آخر خاص بالمثلثات وهذا الكلاس يرث من كلاس الشكل Shape :

```
using System;

namespace Chapter_Four
{
    public class Triangle :Shape
    {
        public string Style { get; set; }
        public double Area()
        {
            return (Width * Length /2);
        }
        public void ShowStyle()
        {
            Console.WriteLine("Traingle is "+ Style);
```

```
        }
    }
}
```

عملية الوراثة برمجيا تكون بوضع بعد تعريف الكلاس (:) ثم اسم الكلاس الأب , في مثالنا هذا فان Triangle سوف يرث خصائص وميثودز الكلاس Shape وكان هذه الخصائص والميثودز قد كتبت فيه , لو عملت كلاس ثالث خاص بالمربعات وجعلته يرث من Shape فهو كذلك سوف يرث الخصائص والميثودز , هل لاحظت ان Length و Width والميثود Diamenation قد قمنا بكتابتها مرة واحدة في الكلاس Shape وقد ورثت منها كلاسين وهذا هو الفائدة الرئيسية من الوراثة وهي أعادة استخدام الشفرات المكتوبة أكثر من مرة بدون كتابتها مرة ثانية .

الان في البرنامج (كلاس التطبيق) سوف نعمل أو بحكت من كلاس المثلثات ونتعامل معه :

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

namespace Chapter_Four
{
    class Program
    {
        static void Main(string[] args)
        {
            Triangle t = new Triangle();
            t.Length = 7;
            t.Width = 4;
            t.Style = "Reight ";
            t.ShowStyle();
            t.Diamenation();
            System.Console.WriteLine("the area of triangle is "+ t.Area()
);
            System.Console.ReadKey();

        }
    }
}
```

العيب في الوراثة أن أي تعديل يصير في الكلاس الأب سوف يؤثر على كل الكلاسات الأبناء (Tightly Couple) .

ثانياً : Composition :

هذا النوع من العلاقات بين الكلاسات يسمح لكلاس أن يحتوي كلاس آخر , مثل لو عندنا كلاس خاص بالسيارة فأنه بالتأكيد سيحتوي على كلاس آخر خاص بالمحرك , كذلك الفائدة منه إعادة استخدام الكود المكتوب أكثر من مرة . طبعاً هذا النوع من العلاقات أكثر مرونة (Flexibility) من النوع الأول لأن التعديل على الكلاس المربوط في لا يؤثر على البقية وبالتالي هذا النوع يسمى بـ..... Loosely-Couple . بينما الوراثة تكون العلاقات هرمية بحيث كلاس واحد ممكن يأثر على الكثير من الكلاسات عكس هذا .

كذلك يمثلون هذا النوع بصيغة لغوية حيث تقول الصيغة :

Is-a-part-of

Example :

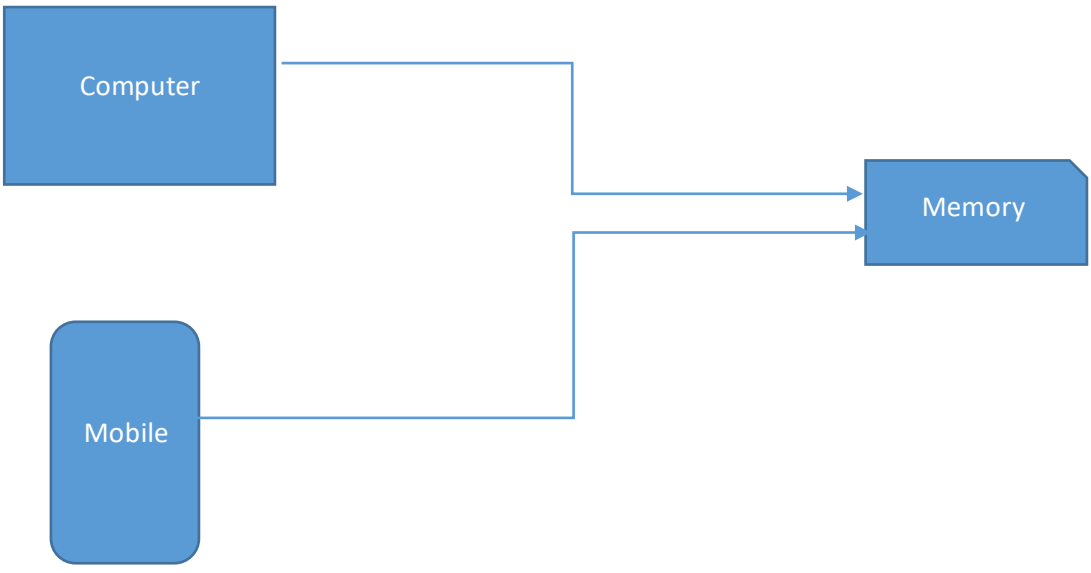
Engine is a part of car

Memory is a part of Computer

Memory is a part of phone

طيب لنأخذ مثال بسيط وهو الميموري هي جزء مهم في الحاسوب وفي الموبايل , بالتالي فأننا سوف نعمل كلاس خاص بالميموري وكلاس للحاسوب وآخر للموبايل :

نبدأ بتمثيل الكلاسات في UML وهذا شي مهم جداً :



برمجياً داخل كلاس الكومبيوتر أو الموبايل نعمل حقل برايفت من نوع Memory ويكون Read-Only وفي الكونستركتر للكلاس نمرر معامل من نوع Memory ثم داخله نسند للحقل قيمة هذا المعامل كما في الشفرة أدناه :

كلاس Memory :

```
namespace Chapter_Four
{
    public class Memory
    {

        public void MemorySize(int size)
        {
            System.Console.WriteLine("The memory size is : " +size + "Mb"
);
        }

    }
}
```

كلاس computer :

```
namespace Chapter_Four
{
    public class Computer
    {
        private readonly Memory _memory;
        public Computer(Memory memory)
        {
            _memory = memory;
        }
        public void ComputerInfo()
        {
           this._memory.MemorySize(1024);
        }
    }
}
```

كلاس Mobile :

```
namespace Chapter_Four
{
    public class Mobile
    {
        private readonly Memory _memory;

        public Mobile(Memory memory)
        {
            this._memory = memory;
        }
```

```
        public void MobileInfo()
        {
            this._memory.MemorySize(512);
        }
    }
}
```

كما تلاحظ في الكلاسين Computer و Mobile عملنا حقل برايفت من نوع ميموري وللقراءة فقط وأسندنا له قيمة داخل الكونستركتر (في الدرس السابق تحدثنا كيف نسند قيمة للحقل الذي يكون للقراءة فقط) ثم عملنا ميثود وداخلها استدعينا ميثود Memorysize الموجودة في الكلاس memory .

اما في الكلاس الرئيسي Program :

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

namespace Chapter_Four
{
    class Program
    {

        static void Main(string[] args)
        {
            var pc = new Computer(new Memory());
            var memory = new Memory();
            var phone = new Mobile(memory);
            pc.ComputerInfo();
            phone.MobileInfo();
            System.Console.ReadKey();

        }
    }
}
```

لاحظ الأوبجكت pc من نوع الكلاس Computer , ولأننا داخل الكونستركتر لهذا الكلاس مررنا معامل من نوع الكلاس memory لذلك عندما نعمل أي انستانس من هذا الكلاس Computer فيجب ان نمرر له أوبجكت من نوع memory , وهنا وضعنا (new Memory()) بينما بعده عملنا أوبجكت من نوع ميموري ومررناه عند تعريف الأوبجكت phone وهي نفس الطريقة (يعني طريقتين لتعريف الكلاسات ولكنهما نفس الشيء) .

لنأخذ مثال نظري حول المقارنة بين Inheritance و Composition :

لنفترض لدينا كلاس رئيسي خاص بAnimals ولدينا كلاسات فرعية Duck , Fish , Dogs , في الكلاس Animal وضعنا مجموعة من الخصائص وبعض الميثودز منها ميثود خاصة بالأكل وأخرى بالنوم طيب ماذا لو أردنا إضافة ميثود خاصة بالسباحة ل Fish , Duck في الكلاس Animal بالتالي هذه الميثود ستكون موجودة في الكلاس Dogs رغم ان هذا الكلاس لا يستفاد منها بشيء . ماذا لو كان لدينا كلاس فرعي رابع خاص بالطيور Brides وأردنا له ميثود للطيران في الكلاس Animal هذا يعني ان Dogs , Fish لن يستفادو من هذه الميثود بشيء وصار وجودها غير منطقي , سنقول عادي لن نضع ميثود الطيران في الكلاس Animal وانما اضعها في كلاس الطيور حل صحيح لكن ماذا لو Duck فهي أيضا تطير وتحتاج لهذه الميثود هل سوف تكتبها داخل هذا الكلاس مرة أخرى ! هذا العيب في الوراثة أن أي شيء تضعه في الكلاس الرئيسي Base Class ترثه بقية الكلاسات وأي تعديل تعمله أيضا سوف يحصل بالبقية بسبب الهرمية في الكلاسات , وأن الكلاسات الفرعية لا يمكن ان ترث من أكثر من كلاس , هذا العيب في الوراثة وهذا لا يعني أنك لا يجب أن تستخدمها وأنما ذلك يعتمد على الحاجة .

بالنسبة Composition عيبها أنها أصعب من الوراثة , لكنها أكثر مرونة كيف ؟ نفس المثال السابق في الكلاس Animal نضع فقط الميثودز المشتركة بين كل الحيوانات , أما بالنسبة لميثود الطيران سوف نعمل لها كلاس جديد وهذا الكلاس نعمل منه أوبجكت في كل الكلاسات الخاصة بالحيوانات التي تطير بالتالي هذا لن يؤثر على كلاسات الحيوانات التي لا تطير , كذلك الحال بالنسبة لميثود السباحة .

غالبا سوف تحتاج أن تستخدمهما كلاهما في التطبيق وأؤكد حسب الحاجة .

Chapter V

Constructor Inheritance,
Polymorphism

Constructor Inheritance:

في الفصول السابقة قلنا أن الكلاس الرئيسي سوف نسميه بـ Base Class ولهذا الكلاس كونستركتر الذي ينفذ دائما أولا , المشكلة التي سوف تواجهنا في الوراثة خصوصا في الكونستركتر للبيس كلاس وهي أن الكونستركتر لا يمكن أن نرثه من الكلاس الأصلي لذلك علينا أن نعمل كونستركتر للكلاس الأبن بمعزل عن كونستركتر للكلاس الرئيسي .

على سبيل المثال نعمل مشروع جديد من نوع Console ونضع فيه بيس كلاس وهو Vehicle وفيه الشفرة التالية :

```
namespace Chapter_Five
{
    public class Vehicle
    {
        public Vehicle()
        {
            System.Console.WriteLine("the vehicle class being initialized
 ");
        }
    }
}
```

الآن نعمل كلاس آخر Car وفيه الشفرة التالية :

```
namespace Chapter_Five
{
    public class Car:Vehicle
    {
            public Car ()
            {
                System.Console.WriteLine("the Car class being initialized
 ...");
            }
    }
}
```

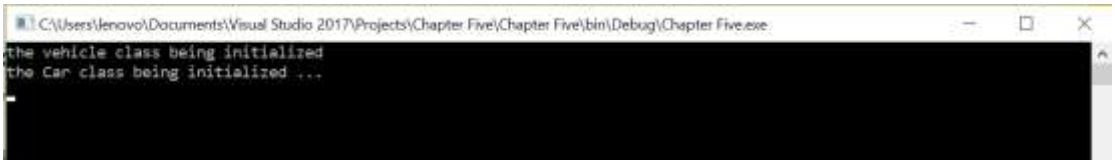
كما تلاحظ في الشفرة أعلاه ان كلاس Car يرث من الكلاس الأب Vehicle .

الان في الكلاس Program نعرف أوبجكت من نوع Car .

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

```
namespace Chapter_Five
{
    class Program
    {
        static void Main(string[] args)
        {
            var car = new Car();
            Console.ReadKey();
        }
    }
}
```

الآن نفذ البرنامج :



لاحظ أن كونستركتر الكلاس الأب نفذ قبل كونستركتر الأبن .

لنرجع الى الكلاس Vehicle ونشيء كونستركتر آخر يتقبل معامل واحد وهو registerNumber :

```
namespace Chapter_Five
{
    public class Vehicle
    {
        private readonly string registerNumber;

        public Vehicle()
        {
            System.Console.WriteLine("the vehicle class being initialized
 ");
        }
        public Vehicle(string registerNumber)
        {
            this.registerNumber = registerNumber;
            System.Console.WriteLine("the vehicle number is {0}",register
Number);
        }
    }
}
```

أما الكلاس Car نعدل على الكونستركتر الخاص به ونضيف اليه كلمة base حتى نستطيع أستدعاء الكوسنتركتر الجديد له .

```
namespace Chapter_Five
{
    public class Car:Vehicle
    {
```

```
            public Car(string registerNumber)
            :base(registerNumber)
            {
                System.Console.WriteLine("the Car number {0}...",register
Number);
            }
    }
}
```

الان في كلاس Program نمرر registerNumber :

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

namespace Chapter_Five
{
    class Program
    {
        static void Main(string[] args)
        {
            var car = new Car("A41102");
            Console.ReadKey();
        }
    }
}
```

فيكون التنفيذ بهذا الشكل :

3- Polymorphism .

الأشكال المتعددة ترجمتها وهي العمود الثالث من برمجة الكائنات الموجهة , لنأخذ مثال بسيط لتوضيح فكرتها وهو الدائرة تعتبر من الأشكال (Shapes) , طيب ماذا عن المربع ؟ كذلك من الأشكال , والمستطيل والمثلث أيضاً بالتالي فأن Shapes تأخذ أشكال متعددة لذلك من هذا المثال نستطيع أعطاء المورفيزم تعريف بسيط وهو أنها الشيء الذي يمتلك أشكال مختلفة .

في البوليمورفيزم نستعمل ميثود تسمى (Overriding) وتعني بالأمكان التعديل على الميثودز الموجودة في الكلاس الأب من داخل الكلاس الأبن . هذه الميثود نعرفها بذكر كلمة virtual مع الأكسس موديفاير لها في الكلاس الأب بينما في الكلاس الأبن نكتب override .

طيب لنأخذ مثال عملي حتى تتوضح الفكرة , حيث أننا سوف ننشيء كلاس جديد باسم Shapes ونضع فيه الشفرة التالية:

```
namespace Chapter_Five
{
    partial class Program
    {
        public class Shapes
        {
            private int _x;
            private int _y;

            public Shapes()
            {

            }
            public Shapes(int x , int y)
            {
                this._x = x;
                this._y = y;
            }
            public virtual void Draw()
            {
                System.Console.WriteLine("Drawing Shape {0} , {1}" ,_x ,_
y);
            }

        }
    }
}
```

في الكلاس أعلاه عملنا أثنين كونستركتر , الأول أفتراضي ليس فيه معاملات والثاني يتقبل معاملين ليسندهما للحقلين , ثم عملنا ميثود Draw والتي أستخدمنا معها كلمة virtual وبالتالي صار بالأمكان التعديل على هذه الميثود من جانب الكلاس الأبن .

طيب لننشيء كلاس جديد باسم Square :

```
namespace Chapter_Five
{
    public class Square:Shapes
    {
        public Square()
        {

        }
        public Square(int x ,int y)
            :base(x,y)
        {

        }
        public override void Draw()
        {
            System.Console.WriteLine("Drawing Sqaure {0} , {1}", _x, _y);
        }

    }
}
```

لاحظ أن كلاس Square يرث من الكلاس الأب Shapes وحتى نستدعي كونستركتر الأب أستخدمنا كلمة base ,

طيب لنرى ميثود Draw كما تشاهد فقد استخدمنا كلمة override في الأكسس موديفاير حتى نستطيع التعديل عليها من

جانب الكلاس الابن لننشيء كلاس باسم Circle ونظيف فيه الشفرة التالية :

```
namespace Chapter_Five
{
    public class Circle:Shapes
    {
        public Circle()
        {

        }
        public Circle(int z ,int r ):base(z,r)
        {

        }
        public override void Draw()
        {
            System.Console.WriteLine("Drawing Circle {0} , {1}", _x, _y);
        }
    }
}
```

إذا تلاحظ في الكلاسين Circle و Square قمنا بالتعديل على الميثود Draw من خلال وضع كلمتي Circle أو

Sqaure في نص WriteLine .

لنعدل على كلاس Program ونرى النتيجة بالتنفيذ

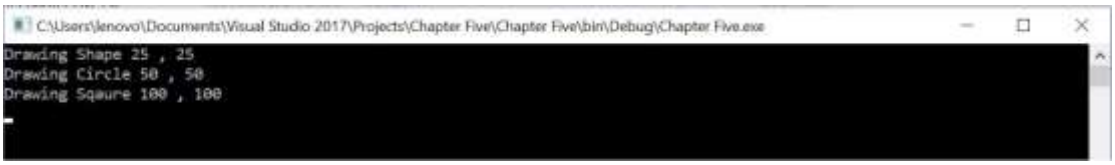
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Chapter_Five
{
    partial class Program
    {
        static void Main(string[] args)
        {
            //Base-Class
            var shapes = new Shapes(25 ,25);
            shapes.Draw();
            //Sub-Class
            var circle = new Circle(50, 50);
            circle.Draw();
            //Sub-Class
            var square = new Square(100, 100);
            square.Draw();

            Console.ReadKey();
        }
    }
}
```

الشفرة أعلاه نستطيع كتابتها بشكل آخر عن طريق تعريف مصفوفة أو بحكات من نوع Shapes على اعتبار ان Shapes يحوي كل أنواع الاشكال ويكون بهذا الشكل :

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

namespace Chapter_Five
{
    partial class Program
    {
        static void Main(string[] args)
        {
            /*  //Base-Class
              var shapes = new Shapes(25 ,25);
              shapes.Draw();
              //Sub-Class
              var circle = new Circle(50, 50);
              circle.Draw();
              //Sub-Class
              var square = new Square(100, 100);
              square.Draw();
              */
```

```
            Shapes[] shapes = new Shapes[3];
            shapes[0] = new Shapes(25, 25);
            shapes[1] = new Circle(50, 50);
            shapes[2] = new Square(100, 100);
            foreach (Shapes shape in shapes)
                shape.Draw();

            Console.ReadKey();
        }
    }
}
```

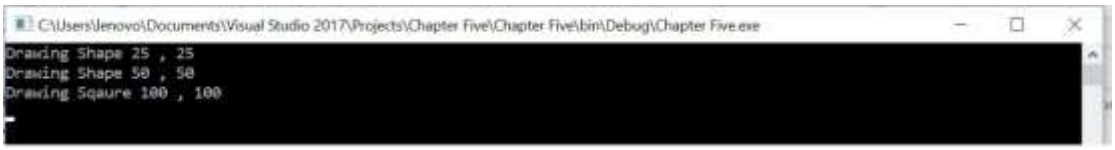
عند التنفيذ :



أعتقد الآن وضحت فكرة البوليمورفيزم .

طيب ماذا لو حذفنا كلمة override من تعريف الميثود في الكلاس الأبن على سبيل المثال من الميثود Draw في الكلاس Circle سوف يظهر لنا تنبيه كما في الصورة :

وعند التنفيذ :



لاحظ السطر الثاني وكأن الميثود Draw لم يتغير منها شيء وهذا ما يخبرنا به التنبيه أعلاه بأن الميثود Draw سوف لن تتغير ما لم نضع Override .

Abstract Class and Member

لنأخذ حالة ممكن تصادفنا في عمل تطبيق ما وهي لو أردنا الميثود في البيس كلاس (الكلاس الأب) لا تنفذ وأنما فقط نعرفها ويجري عليها التعديل والتنفيذ في الكلاس الأبن , يعني مثل ما تعرفون ان الميثود تتكون من جزئين هما الرأس أو التعريف والجسم , في الجسم (Body) سوف نضع الشفرة التي سوف تنفذ .

حتى نفهم هذا الشيء أكثر لنعود الى مثالنا السابق حيث لدينا ميثود Draw في الكلاس Shapes فيها تنفيذ (يعني في شفرة داخل هذه الميثود) وهذا التنفيذ لم نستفد منه بشيء حيث أننا سوف نجري التعديلات من داخل الكلاس Circle و Square وبالتالي صارت شفرة التنفيذ لهذه الميثود في الكلاس الأب زائدة لا داعي لها , في مثل هذه السيناريوهات سوف نستخدم مفهوم جديد أو أكسس موديفاير جديد يسمى ب Abstract .

لدينا مجموعة من القواعد يجب ان نلتزم بها حين نستخدم هذا الأكسس موديفاير , حيث لو أستخدم مع الميثود فيجب ان لاتحتوي الميثود على أي تنفيذ (Implementation) وأنما فقط التعريف ويجب كذلك أن نضع هذا الأكسس موديفاير مع تعريف الكلاس , مثلا الميثود Draw في الكلاس Shapes :

```
namespace Chapter_Five
{

        public abstract class Shapes
        {
            protected int _x;
            protected int _y;

            public Shapes()
            {

            }
            public Shapes(int x , int y)
            {
                this._x = x;
                this._y = y;
            }
        public abstract void Draw();



    }
}
```

لاحظ الميثود Draw فقط التعريف لها كتبناه . أما في الكلاس الأبن Circle فهي نفس السابق

```
public override void Draw()
{
```

```
    System.Console.WriteLine("Drawing Circle {0} , {1}", _x, _y);
}
```

لدينا قاعدة أخرى أيضا تقول اذا كانت الميثود في البيس كلاس من نوع Abstract فيجب ان نضع لها تنفيذ في الكلاس الأبن (يعني نعمل لها override) , أيضا الكلاس الذي يكون من نوع Abstract لا يمكننا أن ننشيء منه أنستانس :

```
Shapes shapes = new Shapes();
//wrong
```

أي خاصية داخل الكلاس ان كانت Abstract فيجب أن يكون الكلاس أيضا Abstract .

حسب الكلام السابق نستطيع فهم أن الغرض من Abstract هو أن يكون لدينا شيء عام وليس مخصص داخل الكلاس الأب سواء ميثود أو خاصية . وأيضا لو كان عندك فريق من المطورين وتريدهم أن يتبعوا التصميم الذي وضعته أنت يكون عن طريق Abstract .

Sealed Class :

لدينا الكثير من الكلاسات الجاهزة في الدوت نت , على سبيل المثال الكلاس Pens الموجود ضمن النيم سبيس Drawing هذا الكلاس كل عناصره من نوع Static , مثلا اللون Pens.Black , مثل هذا الكلاس الأفضل لنا ان نوقف أي وراثه له من خلال أستخدام الكلمة المفتاحية sealed بالتالي لا يمكن لأي كلاس ان يرث منه .

هذا النوع من الكلاسات قليل الاستخدام بل في حالات نادرة نستعمله خصوصا عندما يكون لدينا كلاس كل عناصره Static .

مثلآ :

```
public sealed class Shapes
{
}
```

: Upcasting/Downcasting

التحويل من نوع الى آخر عندنا في السي شارب يكون على نوعين تام Implicit وغير تام Explicit . طيب الوضع الآن يقول ماذا لو أردنا تحويل أو بحكت من نوع كلاس معين الى آخر كيف سيكون ؟

هنا لدينا طريقتين للتحويل وهما :

الأول : Upcasting : ويعني التحويل من الكلاس الأبن الى الكلاس الأب ويكون من نوع تام لأن غالبية ميثود الأبن تكون موجودة عند الأب . مثلا

```
Shapes shape = new Circle(50, 50);
```

الثاني :Downcasting : هذا عكس الأول حيث التحويل يكون من الأب الى الأبن ويكون غير تام , في هذا النوع دائما نستخدم الكلمات المفتاحية as و is (خصوصا مع الشروط) .

لننشيء كلاس جديد باسم Document :

```
namespace Chapter_Five
{
    public class Documents
    {
        public int Width { get; set; }
        public int Height { get; set; }
        public int X { get; set; }
        public int Y { get; set; }
        public void Draw()
        {

        }
    }
}
```

في الشفرة أعلاه عرفنا أربعة خصائص وميثود بسيط باسم Draw الان لننشيء كلاس آخر خاص بالنصوص وكالتالي :

```
namespace Chapter_Five
{
    public class Text : Documents
    {
        public int FontSize { get; set; }
        public string FontName { get; set; }
```

```
    }
}
```

فقط عرفنا خاصيتين واحدة لحجم الخط والثانية لأسم الخط . الان نذهب الى الكلاس Program ونجرب التحويلات :

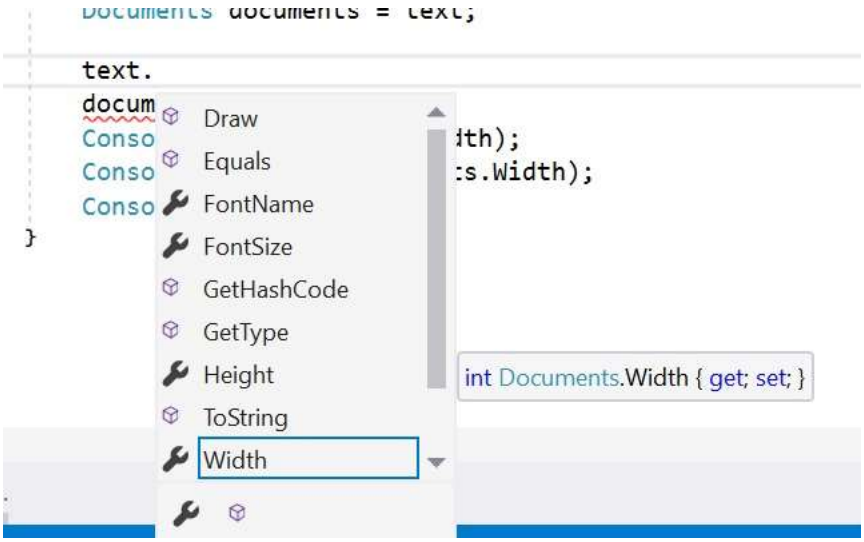
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Chapter_Five
{
    partial class Program
    {
        static void Main(string[] args)
        {
            var text = new Text();
            Documents documents = text;

            text.Width = 100;
            documents.Width = 200;
            Console.WriteLine(text.Width);
            Console.WriteLine(documents.Width);
            Console.ReadKey();
        }
    }
}
```

عند التنفيذ :

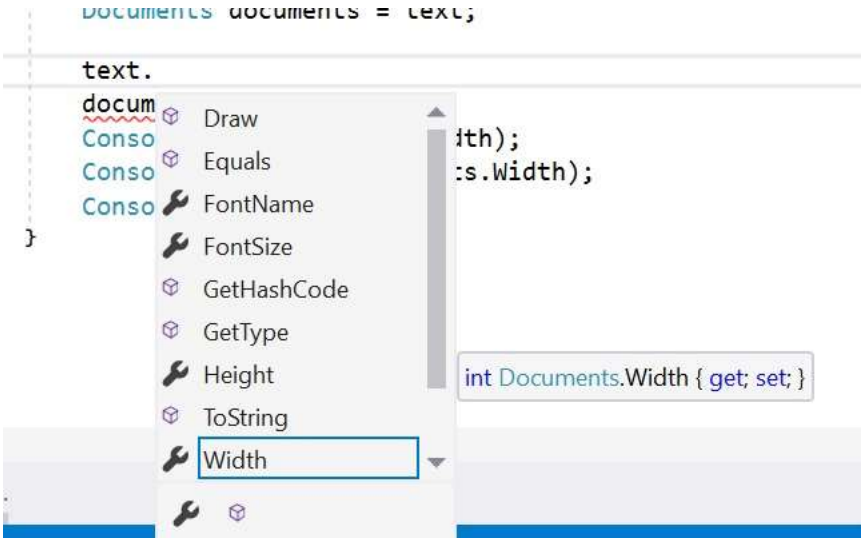
200

200

طيب بالبداية عرفنا أوبجكت من نوع Text , ثم عرفنا أوبجكت ثاني من نوع Documents لكننا قمنا بتحويله الى Text بالتالي هذا النوع implicit لكن بعد أن أسندنا width=100 ثم width=200 عند التنفيذ ظهرت لنا النتيجة 200 . لماذا ؟ كلا الأوبجكتين يشيران الى نفس الريفرينس بالميموري وبالتالي تكون قيمهم (values) متشابهه . دعنا نلاحظ الميثود والخصائص الموجودة في الأوبجكت documents و text :



بينما الأوبجكت documents الذي قمنا بتحويله الى text :



اذا تلاحظ أن الميثود والخصائص الموجودة في documents أقل مما موجودة في text .

طيب لو أردنا التحويل من الكلاس الأب الى الأبن (Downcasting) يكون بالشكل التالي :

```
var documents = new Documents();
 var text = (Text) documents;
```

أو بهذا الشكل :

```
var documents = new Documents();
            var text = documents as Text;
```

Boxing/Unboxing :

كما نعرف أن أنواع البيانات في السي شارب تقسم الى صنفين ,الأولى تسمى ب Value Type والثانية Reference Type ,بالنسبة للنوع الأول مثل int , Boolean , char حيث يخصص جزء لها من الميموري يسمى Stack حجمه يكون صغير وعمر البيانات في الستاك يكون قصير, بينما Ref Type مثل الأوبجكت او المصفوفة . . الخ تخزن في مساحة أكبر من الميموري تسمى Heap ويكون عمرها أطول من الأولى .

بالنسبة Boxing فهو تحويل من متغير نوع value type الى reference (أوبجكت) مثلا :

```
int number = 10;
object obj = number;
//or
 object obj = 10;
```

حيث بالبداية سوف يحجز مساحة بال Stack للمتغير number وكذلك الأوبجكت obj وبعدها القيمة 10 تخزن في heap ويعمل لها reference للأوبجكت الموجود في stack .

بينما Unboxing العملية بالعكس حيث نحول أوبجكت لمتغير من نوع value type :

```
object obj = 10;
 int number = (int)obj;
```

Chapter VI

Interfaces

ماهو Interface :

الأنترفيس مثل الكلاس من ناحية التركيب (Structure) حيث ممكن أن يحتوي على خصائص (Properties) و Methods لكنه يختلف بالوظيفة !

صيغة تعريف الأنترفيس تكون بهذا الشكل :

```
public interface IShapes
   {

   }
```

دائما وخصوصا بالدوت نت يفضل أن يبدأ آسم بحرف I .

في الأنترفيس يكون تعريف الميثود بدون Body وبدون أكسس موديفاير يعني مثل Abstract , وهذا ما يجعل علاقته بالكلاسات تكون من نوع Loosely Coupling .

لنأخذ مثال حول Interface نعمل مشروع جديد ونضيف فيه أنترفيس جديد باسم ITransactions وفيه نعرف أثنين ميثود كالتالي :

```
namespace Chapter_Six
{
    public interface ITransactions
    {
        void showTransaction();
        double getAmount();
    }
}
```

كما تلاحظ في الشفرة أعلاه عرفنا اثنين ميثود بدون body أو اكسس موديفاير وهذا ما يجعل استخدام الإنترفيس أكثر مرونة , للنشيء كلاس جديد باسم Transaction يرث من الإنترفيس وكالتالي :

```
namespace Chapter_Six
{
    public class Transaction:ITransactions
    {
        private string _code;
        private string _date;
        private double _amount;

        public Transaction()
        {
            _code = "";
            _date = "";
```

```
            _amount = 0.0;
        }
        public Transaction(string code ,string date ,double amount)
        {
            _code = code;
            _date = date;
            _amount = amount;
        }
        public double getAmount()
        {
            return _amount;
        }

        public void showTransaction()
        {
            System.Console.WriteLine("Transaction Code : {0}",_code);
            System.Console.WriteLine("Transaction Date : {0}",_date);
            System.Console.WriteLine("Transaction Amount: {0}",_amount);

        }
    }
}
```

بالبداية عرفنا كلاس ثلاث حقول من نوع برايفت وعملنا لهذا الكلاس إثنين كونستركتر الأول افتراضي (Default) والثاني يتقبل ثلاث معاملات , لكن مع هذا سيكون هناك خطأ (خط أحمر يظهر تحت تعريف الكلاس) الخطأ يطلب منك أن تعمل (Implementation) للميثودز الخاصة بالأنترفيس . لذا قمنا بتعريف أثنين ميثود بنفس اسم الميثودز الموجودة بالانترفيس وعملنا لهما Body .

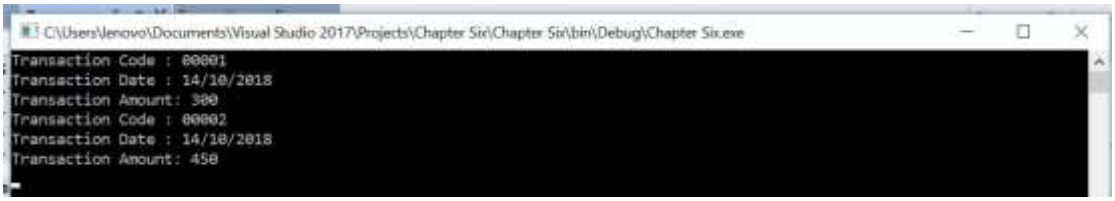
الآن نذهب الى الكلاس Program ونعرف اثنين أوبجكت ونستدعي الميثود ShowTransaction :

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

namespace Chapter_Six
{
    class Program
    {
        static void Main(string[] args)
        {
            var transaction1 = new Transaction("00001", "14/10/2018", 300
.0);
            var transaction2 = new Transaction("00002", "14/10/2018", 450
.0);
            transaction1.showTransaction();
            transaction2.showTransaction();
            Console.ReadKey();
        }
    }
```

```
}
```

أعلاه عندما عرفنا الأوبجكت من نوع Transaction أخترنا الكوسنتركتر الثاني له ومررنا المعاملات .

عند التنفيذ ستكون النتيجة كالتالي :



من الأشياء المهمة في السي شارب هو أن الكلاس لا يرث من أكثر من كلاس وأنما واحد فقط لكن ! ممكن ان يرث من أكثر من Interfaces , بينما في السي بلس بلس ممكن الكلاس ان يرث من أكثر من كلاس .لكن أيضا من الخطأ ان نقول أن الكلاس يرث من الأنترفيس ! وأنما الصحيح ان نقول أن الكلاس ينفذ يتكامل مع الأنترفيس ليش بالعراقي ؟ لأن الكلاس لا يعيد أستخدام شفرات موجودة بالأنترفيس وأنما ينفذها (أي يكمل شفراتها) وهذا لا يحقق مبدأ الوراثة .

```
public class Transaction:ITransactions,IPaid ,IExchange
```

الخاتمة

الأتصال:

1- Tel Number : +964-770 3291 330
2- Email :MohammedHamel@outlook.com
3- Linkedin -www.linkedin.com/in/mohammed-hamil-8816325a/
4- Website : http://mohammedeydan.net/
5- Wordpress : https://mohamediddan.wordpress.com/
6- Tweeter : @EydanMohammed

المصادر :

1- C# 4.0 Unleashed – Bart De Smet
2- C# 6 for Programmers -Paul Deitel ,Harvey Deitel
3- C# 3.0 CookBook -Jay Hilyard ,Stephen Teilpet
4- C# Guide Doc – Microsoft
5- C# Corner – website : www.c-sharpcorner.com
6- Tutorials Point – website : www.tutorialspoint.com
7- Tutorials Teacher – website : www.tutorialsteacher.com
8- Udemy : C# Intermediate Classes Interfaces and OOP -Mosh Hamedani .