

Extension Methods

هي زى أي function تانىه ولكن ليها شروط و syntax مختلف شويه.
ومن اسمها بنسخدمها عشان نعمل extension لكلاس تانى من غير ما نعدل في الكلاس نفسه.

Syntax:

```
public static class int_extensions
{
    public static bool is_even (this int num)
    {
        return num%2==0;
    }
    public static bool is_odd (this int num)
    {
        return !is_even(num);
    }
}
```

Call syntax:

```
static void Main(string[] args)
{
    int s=0;
    Console.WriteLine(s.is_even());
    Console.WriteLine(s.is_odd());
}
```

في المثال اللي فوق ده كنت عاوز اضيف ميزه لكلاس int عشان اعرف هل الرقم زوجي ولا فردي فعملت
كلاس جديد سميته int_extensions ونلاحظ هنا انه مش تابع لكلاس int ولا حتى بيورث منه حاجه
وضفت فيه methods2 الأولى بتتأكد اذا كان زوجي ولا لا والثانية بتتأكد اذا كان فردي او لا.

نلاحظ في ال syntax كل method بتاع كل parameter قبله الكلمة this وده السبب في ان لما عملت call
لله method عملته باستخدام ال s الي من نوع int مش باسم الكلاس وهنا بيجي دور this لأنها
بتعود على ال object اللي بيستخدم ال method

نلاحظ كمان ان عشان اعمل extension Method لازم الكلاس وال static يكونو يكونوا

Nullable & Null operators

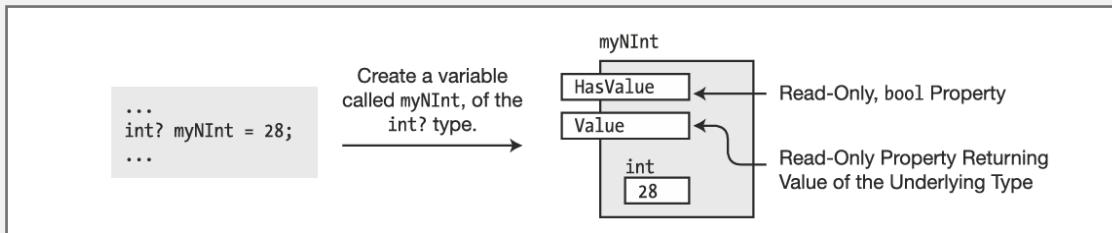
في أوقات كتير بيكون عندي متغير (value type) او reference type ولسبب ما مش يحتاج اخصله قيمة في الوقت الحالي في الحالة دي بيجي وظيفة null reserved word in C# بتعبر عن اللا قيمة ولكن منقدرش نستخدمها الا مع الـ nullable types سواء كانت value او reference

ايه هي الـ nullable types ؟

هي أي حاجه من الممكن ان ميكونش ليها قيمة واللي بيحدد ده هو المبرمج وقت تعريف متغير مثلًا فيضيف علامة استفهام بعد نوع المتغير او بيستعين بكلاس <Nullable> الخ ودي بعض الامثله لتعريف متغير nullable قيمته بـ null:

```
int? num1 = null;
Nullable<int> num2 = null;
Nullable<int> num3 = new Nullable<int>();
Nullable<int> num4 = default(int?);
```

الجدير بالذكر ان لما نعرف لما نعرف nullable type ده بيضيف 2 للـ object اللي عرفناه الأولى اسمها HasValue وظيفتها ترجع قيمة boolean ه تكون true اذا فيه value لو مفيش والثانية بترجع نفسها زي ما متوضعي الصورة:



و زي ما فيه nullable type هبيقى على النقيض من ذلك فيه non وهو الشائع والأكثر استخداما بين المبرمجين زي تعريف متغير int قيمته بـ 5 مثلاً وبما انه non nullable يعني لو استدنا له قيمة null مش هينفع زي المثال ده:

```
int num1 = null;
```

وأخيرا عشان نضمن سلامه الكود واحنا بنتعامل مع الـ nullable types دي ومن المستحسن methods معتمده على قيمة وهي في الأساس ممكن تكون مش موجوده (null) لازم نتأكد الأول هل قيمة المتغير ده مش وده عن طريق شرط بسيط زي المثال ده:

```
char[] word = null;
int? num_of_chars = 5;
if (word != null)
    num_of_chars = word.Length;
Console.WriteLine($"number of characters in the world: {num_of_chars}");
```

ولكن C# اختصرت الشرط ده في ما يسمى بـ the null conditional operator الشرط اللي فوق ده كالتالي:

```
num_of_chars = word?.Length;
Console.WriteLine($"number of characters in the world: {num_of_chars}");
```

تاني وه هو the null coalescing operator باختصار هو شرط اذا المتغير اللي بنستخدمه فيه قيمة هنستخدمها واذا مكاش فيه قيمة يعني قيمته بـ null هنستخدم القيمة الـ default اللي بعد علامتين الاسفهان زي المثال اللي تحت والي ه تكون قيمة num2 فيه في اول مره بـ 1- لان قيمة num1 بـ null وتاني مره بـ 10.

```
int? num1 = null;
int num2 = num1 ?? -1;
num1 = 10;
num2 = num1 ?? -1;
```

Exception Handling

هو الجزء المسؤول عن معالجة الأخطاء اللي ممكن تظهر وقت الـ debugging بداع البرنامج

في حالتين للموضوع ده:

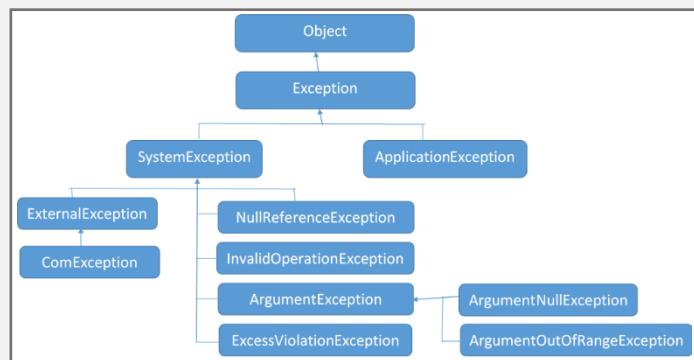
الأولى ان المبرمج هو اللي عاوز يرمي exception لسبب ما وفي الحاله دي بيستخدم throw كالتالي:

```
throw new DivideByZeroException();
```

والثانية ان الـ exception حصل بسبب خطأ في البرنامج وهنا بنستخدم try then catch finally كالتالي:

```
try
{
    int x = 1;
    int y = 0;
    x = x/y;
}
catch (Exception ex)
{
    Console.WriteLine("Divide By Zero exception :");
}
finally {
    Console.WriteLine("End");
}
```

في المثال اللي فوق حصل exception بسبب عملية القسمه على 0 اللي رياضيا تعتبر من المحرمات ولهاذا السبب حطينا الكود اللي بيعمل الـ exception ده في try block عشان اذا حصل نقدر نتعامل مع exception ده في الـ catch block طيب ايه فايدة finally في الحاله دي ؟ عشان نفهم وظيفة finally نعم نعرف أنواع الـ exceptions اللي بتدعمها C# اللي موضحه في الصوره اللي تحت:



في أنواع كتير للـ exceptions منها اللي جاهز تقدر تستخدمه وهو ما يندرج تحت الـ system exception class ومنها اللي تقدر تنشئه وفي الحاله دي بيعتبر application exception بشرط انه يورث من الـ system exception او application class Exception سواء كان فكلهم بيندرج تحت الـ "Exception" super class اي الفايده من ده ؟ لنفترض انت عامل 2 للمثال اللي فوق كالتالي:

```
catch(DivideByZeroException ex)
{
    Console.WriteLine("Divide By Zero exception :");
}
catch (Exception ex)
{
    Console.WriteLine("general Exception handler");
}
```

في المثال اللي فوق ده لو حصل قسمه على الصفر ال الأول(DivideByZeroException) هي
اللي هتشتعل وهي اللي هتعالج exception لكن لو بدلنا أماكن catch blocks اللي تحت
فوق واللي فوق تحت زي كده مثلا :

```
catch (Exception ex)
{
    Console.WriteLine("general Exception handler");
}
catch(DivideByZeroException ex)
{
    Console.WriteLine("Divide By Zero exception :");
}
catch(NullReferenceException ex)
{
    Console.WriteLine("Null Reference Exception :");
}
```

هنا اول catch هيستقبل أي exception هيقابلة والباقي يعتبر من غير فايده ليه ده بيحصل ؟ لانه ال
باقي exception classes فاي exception classes هيعدي عليه هيعتبر نفسه جزء منه
فهيتنفذ فيه مش في غيره عشان كده لازم نرتب ال catch بحسب ترتيبها في شجرة العيله لل
classes بحيث الأبناء يبقوا فوق والآباء يبقوا تحت

يحي دور finally واستخدامها لنفترض ان ولا catch قدر يعالج exception اللي حصل ده
هنا يجي دور finally اللي بتعتبر base case فبتنفذ الكود الأخير في أي حالة exception حصلت سواء
ال catch عالجتها او لا .

Asynchronous programming

المسار الطبيعي لاي برنامج مش يستخدم ال asynchronous programming انه يبدأ بتنفيذ الكود اللي في main امر امر واذا مثلاً قابل call يروح ينفذها ويوقف تنفيذه في ال main لحد ما يكمل function ويرجع يكمل تنفيذ في ال main لكن الأسلوب ده ليه عيوبه لأن لو عندنا 2 مش معتمدين على بعض يعني مفيش واحد مضطره تستنى الثانية عشان تشغله ففي الحاله دي احنا ضيعنا فرصه اننا نخليلهم يشتغلوا مع بعض وهنـا يجي دور ال asynchronous programming في انه بيوفـرـنـا الفرصة دي.

C# بتتوفر ال threads كحل للمشكله اللي فوق ولكنه مش افضل وسـيلـهـ في وجود ال abstractoin منه وهو task كلـهـمـ بيـأـدـوـ نفسـ الغـرضـ باختلافـ الـ syntaxـ وـديـ مـقارـنـهـ بـسيـطـهـ بينـهـمـ:

| WHY TASK OVER THREAD? | | | |
|------------------------|------------------------|-------------|---------------------|
| CRITERIA | THREAD | TASK | ADVANTAGE |
| CONCEPT | LOW LEVEL | ABSTRACTION | LESS DETAILS |
| LEVERAGING THREAD POOL | NO | YES | PERFORMANCE |
| RETURN VALUE | NO | YES | LESS CODE |
| CHAINING | NO | YES | ORDER/READABILITY |
| EXCEPTION PROPAGATION | NO | YES | PARENT CATCH IT |
| TASK TYPE | BACKGROUND/ FOREGROUND | BACKGROUND | PROCESS TERMINATION |
| SUPPORT CANCELLATION | NO | YES | SAVE RESOURCES |

Syntax for both of them:

```
Thread thread = new Thread(() => { Console.WriteLine("hi 1"); });
Task task = new Task(() => { Console.WriteLine("hi 2"); });
thread.Start();
task.Start();
```

output for the above code:

hi 1
hi 2

hi 2
hi 1

لو جربـناـ الكـوـدـ الليـ فوقـ اـكـتـرـ منـ مـرـهـ وـدهـ لـانـ الـ treadـ مـمـكـنـ يـخـتـلـفـ كلـ مـرـهـ وـدهـ لـانـ الـ outputـ والـ taskـ بـيـشـتـغـلـوـ فيـ نفسـ الـوقـتـ والـليـ بـيـخـلـصـ الأولـ بـيـظـهـ النـاتـجـ قـبـلـ التـانـيـ .

لفترض ان ال task شغال على function ولكن ال function دي بتاخذ وقت عشان تتنفذ وانا عاوز البرنامج يكمل ولما اال function تخلص تبقى تطبع الناتج مثلا الحالة دي ممكن نطبقها باكتر من طريقة كالتالي:

Using GetAwaiter() and OnCompleted():

```
static void Main(string[] args)
{
    Task<string> task = Task.Run(getname);
    var awaier = task.GetAwaiter();
    awaier.OnCompleted(() =>
    {
        Console.WriteLine(awaier.GetResult());
    });
}

static string getname()
{
    var name = "ahmed";
    return name!;
}
```

Using ContinueWith():

```
Task<string> task = Task.Run(getname);
task.ContinueWith((x) => Console.WriteLine(x.Result));
Console.WriteLine("wating for the result ...");
```

على النقيض من ذكر في بعض الأحيان بنبقى ببنفذ ال task بشكل Asynchronously ولكن بنحتاج نستنى . await .async .awaiting معينه عشان نقدر نكمل وهنا تيجي أهمية

Example:

```
static async Task Main(string[] args)
{
    await ClcAgeAsync();
    Console.WriteLine("wating for the result ...");
}
static async Task ClcAgeAsync()
{
    int age=0;
    await Task.Run(() => age = System.DateTime.Now.Year -
GetBirthDateYear());
    Console.WriteLine("age is:" +age);
}
```

نلاحظ اننا لازم نستخدم async قبل اسم ال method واذا مش بترجع حاجه هنكتب جنبها Task او اذا بترجع حاجه هنكتب Task<return data type> اما بالنسبة ل await فبتيجي مع ال function او ال task اللي مش عاوزين نتخطاه الا لما يكمل اللي بيعمله .

اللي اسمها ClcAgeAsync هنا بتسمى ب function asynchronous