

# C#.Net

L4 .Functions ,Delegates  
,Breakpoints , Exception Handling

# Homework1

اكتب برنامج console application يدخل المستخدم من خلاله عدد ما  $n$  فيقوم البرنامج بإظهار القيمة  $2^n$ ، الحل ممكن أن يكون بحلقات ، أو من خلال توابع موجودة مسبقاً في اللغة ، أو من خلال bitwise operator .

```
static void Main(string[] args)
{
    int x = 5;
    Console.WriteLine(1 << x);
}
```

# Homework2

اكتب برنامج console application يدخل المستخدم من خلاله عدد ما  $x$  فيقوم البرنامج بإظهار هل هذا العدد من قوى الـ 2 أم لا . مثلاً العدد 32 هو من قوى الـ 2 لأن  $2^5=32$  ، الحل ممكن أن يكون بحلقات ، أو من خلال توابع موجودة مسبقاً في اللغة ، أو من خلال bitwise operator .

```
static void Main(string[] args)
{
    int x = 8;
    int res = x & (x - 1);
    Console.WriteLine(!Convert.ToBoolean(res));
}
```

# Scope Rules: Local Scope :

النطاق المحلي local scope هو أي نطاق يبدأ ب { وينتهي ب } .

{ ..... }

جسم function يوجد ضمن نطاق ، والـ switch body أيضاً هو نطاق ، .....

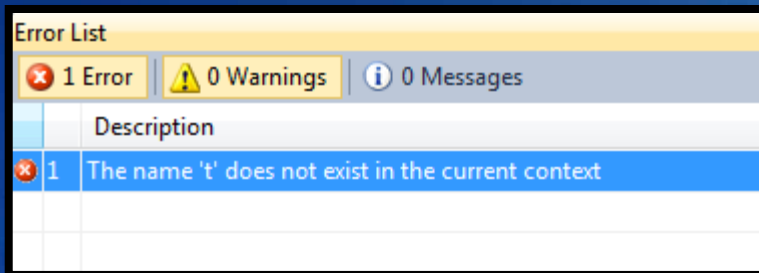
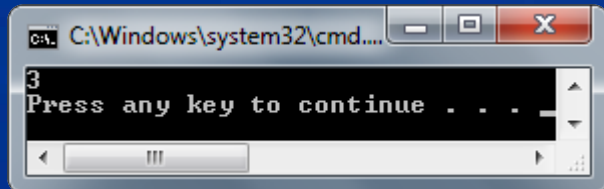
يمكن تعريف نطاق بداخل نطاق .

جميع object التي يتم تعريفها ضمن نطاق ما تكون فترة حياتها ضمن هذا النطاق أي أنها تموت بعد الخروج من هذا النطاق .

# Scope Rules: Local Scope :

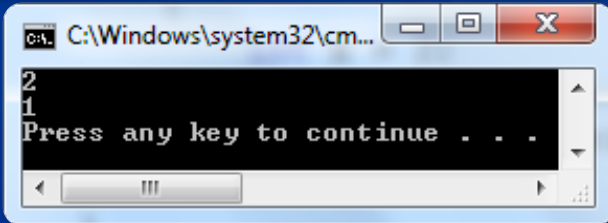
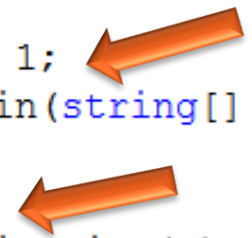
```
class Program
{
    static void Main(string[] args)
    {
        int x = 2, y = 3, z = 4;
        x++;
        Console.WriteLine(x);
    }
}
```

```
static void Main(string[] args)
{
    int x = 2, y = 3, z = 4;
    x++;
    {
        int t = x;
    }
    Console.WriteLine(t);
}
```



# Scope Rules: Global Scope :

```
class Program
{
    static int x = 1;
    static void Main(string[] args)
    {
        int x = 2;
        Console.WriteLine(x);
        Console.WriteLine(Program.x);
    }
}
```



object x الأول هو متحول ساكن

static ضمن الصف وبالتالي

يمكننا الوصول إليه عن طريق

ذكر اسم الصف ثم نقطة ثم اسم

object .

object x الثاني هو متحول محلي

داخل function وتنتهي فترة

حياته عند الانتهاء من استدعاء

function .

# Main function:

هنا تنفيذ عبارة  
عن ملف من النوع  
exe

```
static void Main(string[] args)
{
    int result = 0;
    foreach (string item in args)
    {
        result += Convert.ToInt32(item);
    }
    Console.WriteLine("Result = {0}", result);
}
```

Main function يأخذ دخل وحيد



هنا هو مصفوفة من النمط string .

يمكن الاستفادة منه عندما يتم تنفيذ



ملف الخرج ذو اللاحقة exe . من

خلال command prompt أي من

خلال DOS Command .

```
Administrator: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Mahmoud>C:\Users\Mahmoud\Desktop\ConsoleApplication1\ConsoleApplication
1\bin\Debug\ConsoleApplication1.exe
Result = 0

C:\Users\Mahmoud>C:\Users\Mahmoud\Desktop\ConsoleApplication1\ConsoleApplication
1\bin\Debug\ConsoleApplication1.exe 1 2 7 0 1
Result = 11

C:\Users\Mahmoud>exit
```

# Function Overloading:

يمكننا إجراء ما يسمى بالتحميل الزائد للتوابع أي صنع أكثر من function مشتركين بنفس

الاسم ومختلفين بال parameter list سواء بأنماط متحولات الدخـل أو بترتيبهم أو بعددهم .

نقوم بالعمل على جزء الـ signature (اسم function + متحولات الدخـل).

نقوم بالمحافظة على اسم function .

يمكننا تغيير الـ parameter list كما نشاء .

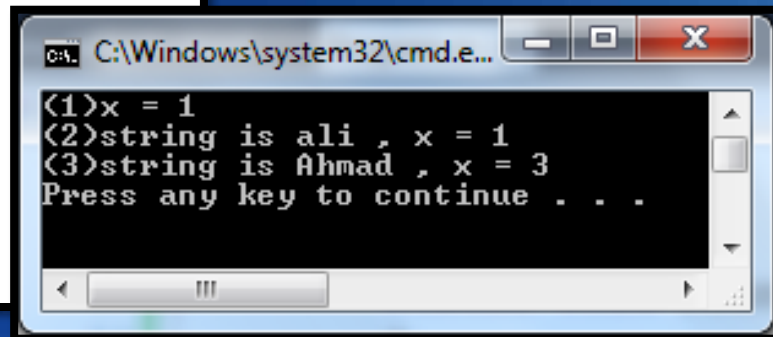
يجب أن يبقى نمط الإرجاع للـ function كما هو !

انتقل للأمثلة التالية حتى يتم إيضاح الفكرة بشكل أكبر .



# Function Overloading:

```
static void Print(int x)
{
    Console.WriteLine("(1)x = {0}",x);
}
static void Print(int x , string s)
{
    Console.WriteLine("(2)string is {0} , x = {1}",s,x);
}
static void Print(string s, int x)
{
    Console.WriteLine("(3)string is {0} , x = {1}", s, x);
}
static void Main(string[] args)
{
    Print(1);
    Print(1, "ali");
    Print("Ahmad", 3);
}
```



A screenshot of a Windows command prompt window. The title bar shows the path "C:\Windows\system32\cmd.e...". The window contains the following text: "(1)x = 1", "(2)string is ali , x = 1", "(3)string is Ahmad , x = 3", and "Press any key to continue . . .". The text is displayed in a monospaced font on a black background.

# Function Overloading:

إذاً يمكننا من استدعاء print function بأكثر من طريقة .

عندها يمكننا القول أنه لدينا overloading لل print function .

يجب أن يكون هنالك تطابق باسم function واختلاف في ال parameters list لهذه التوابع سواء في :

نمط object .

اختلاف ترتيب object .

عدد object .

أحياناً قد نخطئ في عملية ال overloading مما يجعل المترجم أمام أكثر من function يمكن أن يناسبان

الاستدعاء ، عندها سينتج لدينا compiler error .

# Function Overloading:

```
static void Print(int x , float y)
{
    Console.WriteLine("(1)x = {0},y = {1}", x, y);
}
static void Print(float x , int y)
{
    Console.WriteLine("(2)x = {0},y = {1}", x, y);
}
static void Main(string[] args)
{
    Print(1.2f,4);
    Print(1, 4.0f);
}
```

في الاستدعاء الأول تم

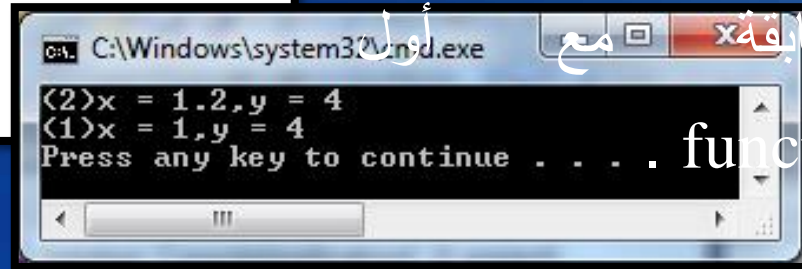
المطابقة مع ثاني

function أما في

الاستدعاء الثاني تم

المطابقة مع أول

function



# Function Overloading:

```
static void Print(int x , float y)
{
    Console.WriteLine("(1)x = {0},y = {1}", x, y);
}
static void Print(float x , int y)
{
    Console.WriteLine("(2)x = {0},y = {1}", x, y);
}
static void Main(string[] args)
{
    Print(1,4);
}
```

في الاستدعاء التالي تتم  
عملية المطابقة مع  
function الأول وأيضاً  
مع function الثاني وهنا  
يحصل تضارب ، فيظهر  
لدينا Compiler Error

Error List

1 Error

0 Warnings

0 Messages

Description

1 The call is ambiguous between the following methods or properties: 'ConsoleApplication1.Program.Print(int, float)' and 'ConsoleApplication1.Program.Print(float, int)'

# Delegates delegate:

delegate هو نوع يمكننا من حفظ مراجع function . 

تخيله كأنه مؤشر على function . 

تعريف delegate يشبه تعريف function إلا أنه لا يحوي جسم . 

بعد تعريف delegate يمكننا أخذ object من هذا delegate . 

بعد ذلك يمكننا تهيئة هذا object ليمثل مرجعاً لأي function له نفس الـ parameters list ونفس نمط 

الارجاع .

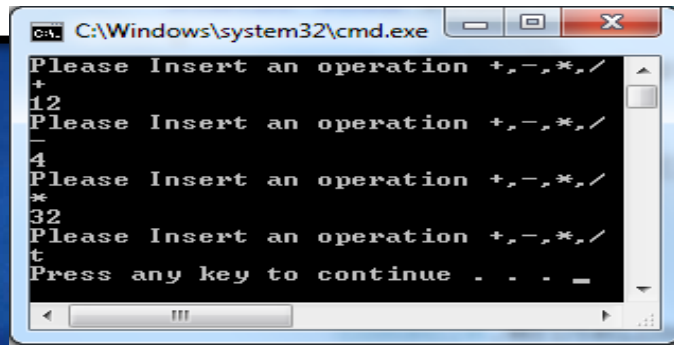
وعندها يمكننا استدعاء هذا function من خلال delegate . 

انتقل للمثال التالي كي تتضح الفكرة : 

# Delegates delegate:

```
static void Main(string[] args)
{
    Operation op;
    while (true)
    {
        Console.WriteLine("Please Insert an operation +,-,*,/ ");
        char c = Convert.ToChar(Console.ReadLine());
        if (c == '+')
            op = new Operation(add);
        else if (c == '*')
            op = new Operation(mult);
        else if (c == '-')
            op = new Operation(sub);
        else if (c == '/')
            op = new Operation(div);
        else
            break;
        int result = op(8, 4);
        Console.WriteLine("result = {0}", result);
    }
}
```

```
delegate int Operation(int x, int y);
static int add(int x, int y)
{
    return x + y;
}
static int sub(int x, int y)
{
    return x - y;
}
static int mult(int x, int y)
{
    return x * y;
}
static int div(int x, int y)
{
    return x / y;
}
```



```
C:\Windows\system32\cmd.exe
Please Insert an operation +,-,*,/
+
12
Please Insert an operation +,-,*,/
-
4
Please Insert an operation +,-,*,/
*
32
Please Insert an operation +,-,*,/
/
12
Press any key to continue . . .
```

# Delegates delegate:

نلاحظ أن delegate ليس له جسم . 

يمكن أخذ object من هذا delegate عن طريق ذكر اسم delegate ثم اسم object . 

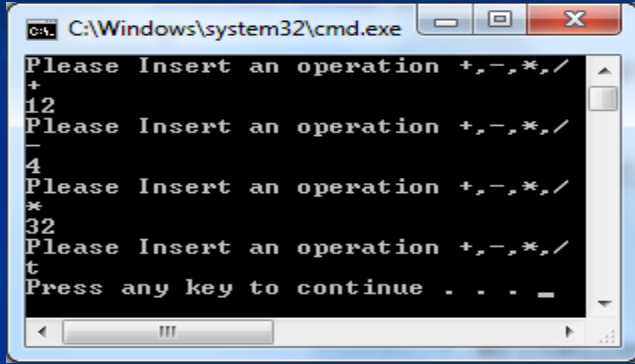
يمكن تخيل أن delegate وكأنه مندوب function ، أو اسم مستعار له . 

يجب أن يتم تطابق بين ترويسة delegate وترويسة delegate function . 

يتم تهيئة object عن طريق new ثم ذكر اسم delegate ثم ندخل اسم function الذي يتطابق دخله وخرجه مع دخل وخرج delegate . 

# Delegates delegate:

بما أنه يمكننا أن نأخذ متحول من delegate فإنه يمكننا تمرير هذا object إلى أي function آخر ، وبالتالي يمكننا القول بأننا تمكنا من جعل function على أنه دخل لـ function آخر .



```
static void Print(Operation op, int x, int y)
{
    Console.WriteLine(op(x, y));
}
```

```
static void Main(string[] args)
{
    Operation op;
    while (true)
    {
        Console.WriteLine("Please Insert an operation +,-,*,/ ");
        char c = Convert.ToChar(Console.ReadLine());
        if (c == '+')
            op = new Operation(add);
        else if (c == '*')
            op = new Operation(mult);
        else if (c == '-')
            op = new Operation(sub);
        else if (c == '/')
            op = new Operation(div);
        else
            break;
        Print(op, 8, 4);
    }
}
```




# Delegates delegate:

لم ننته من delegate بعد ، سنعود إليها لاحقاً ونتحدث عنها بتفصيل أكبر   
عندما نتحدث عن مفهوم الأحداث Events .

Edited by Eng. Ayman Kamhan

# Recursion !

function العودية هي توابع يتم تنفيذها مرة واحدة أو أكثر ويتم الخروج منها عند عدم تحقق شرط ما يدعى شرط التوقف . 

function العودي هو function يقوم باستدعاء نفسه كل مرة إلى أن يصل إلى شرط التوقف فيتوقف عن الاستدعاء ويقوم بتحصيل النتائج المتراكمة عن الاستدعاءات السابقة . 

# Recursion !

```
class Program
{
    static int factorial(int n)
    {
        if (n == 0 || n == 1)
            return 1;
        else
            return n * factorial(n - 1);
    }
    static void Main(string[] args)
    {
        Console.WriteLine(factorial(5));
    }
}
```

# Recursion !

يمكننا كتابة المهمة السابقة عن طريق function عودي بالاستناد إلى 

```
factorial(n) = n * factorial(n-1)

factoria(4) = 4 * factorial(3)
              = 4 * 3 * factorial(2)
              = 4 * 3 * 2 * factorial(1)
              = 4 * 3 * 2 * 1 * factorial(0)
              = 4 * 3 * 2 * 1 * 1 = 24
```

نقوم بالتوقف عن العودية عندما يصبح العدد المدخل للـ function هو 0 أو 1 . 

# Recursion !

كل عملية استدعاء لـ function تكون له متحولاته الخاصة والمستقلة عن الاستدعاء السابق . 

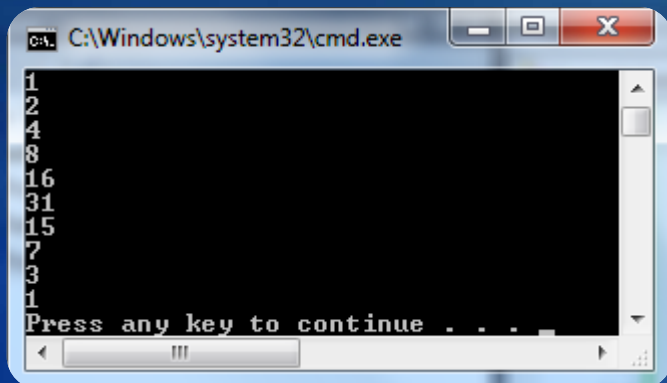
يجب تنفيذ جميع التعليمات الموجودة في الاستدعاء العودي الحالي . 

انتقل للمثال التالي والذي يشرح كامل الفكرة !! 

# Recursion !

```
static void Do(int n)
{
    if (n != 32)
    {
        Console.WriteLine(n);
        n *= 2;
        Do(n);
        n--;
        Console.WriteLine(n);
    }
}

static void Main(string[] args)
{
    Do(1);
}
```



# Exception Handling :

إن أسهل مرحلة من مراحل بناء البرنامج هو مرحلة الـ coding !!!



وتكون المرحلة الأصعب هي عملية بناء model مناسب للبرنامج ودراسة بنيته وعلاقة أجزائه مع بعضها البعض .



إذا قمنا ببناء model قوي هذا سيضمن لنا برنامج ناجح ومتمين .



عملية بناء الـ model تكمن في مرحلتي التحليل والتصميم من مراحل بناء وتطوير البرمجية .



وعلى الرغم من أن مرحلة كتابة الكود Coding هي المرحلة الأسهل إلا أنه أحياناً تظهر لدى المستخدم أخطاء أثناء عمله لذا لا بد من معالجة هذه الأخطاء .



سنعالج هذه الأخطاء بالتعرف على المفهوم الجديد Exception Handling .



# Exception Handling :

```
static int add(int x, int y)
{
    return x + y;
}
static void Main(string[] args)
{
    Console.Write("Please insert first number : ");
    int x = Convert.ToInt32(Console.ReadLine());
    Console.Write("Please insert second number : ");
    int y = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("Success - Reading values");
    Console.WriteLine("Now Calculating add");
    int res = add(x, y);
    Console.WriteLine("res = {0}", res);
    Console.WriteLine("Process Completed Successfully");
}
```

في هذا الكود قمنا بكتابة بعض الأسطر

الإضافية التي تمكننا من معرفة عملية سير

البرنامج هل يسير بشكل صحيح أم لا .

لكن هذا أصبح يضيف أشياء زائدة عن

متطلبات المستخدم والذي يريد فقط نتيجة

عملية الجمع دون معرفة تفاصيل سير

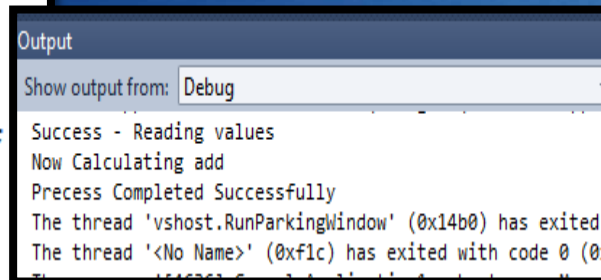
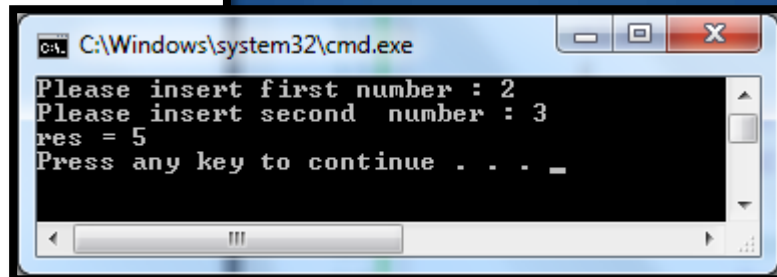
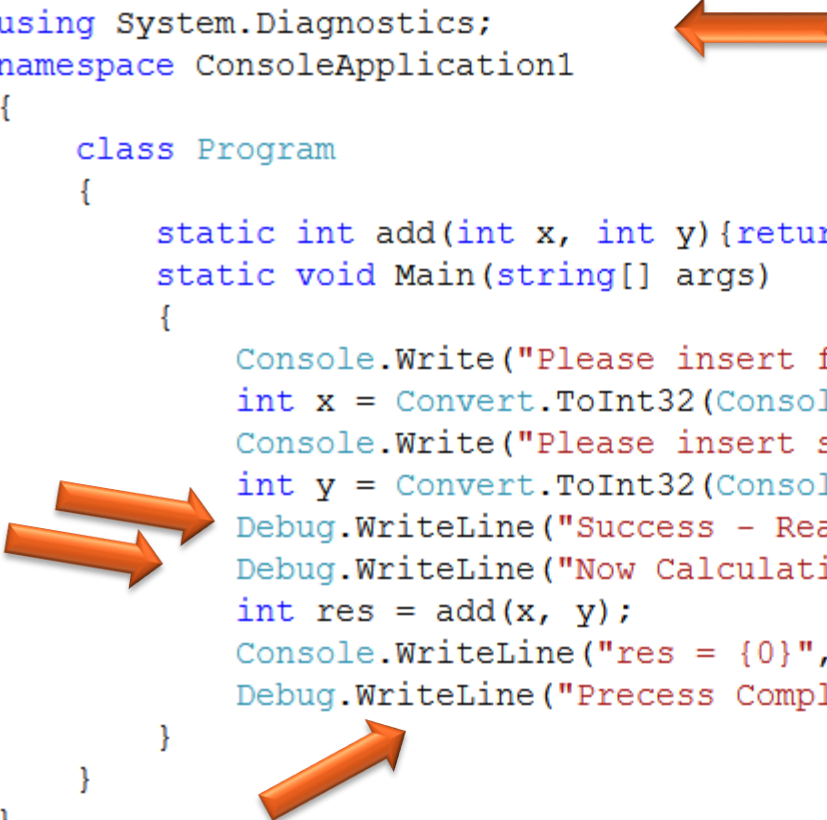
البرنامج إذ لا تهمه هذه الأمور البرمجية.

ويكون الحل في الشريحة التالية .



# Exception Handling :

```
using System.Diagnostics;
namespace ConsoleApplication1
{
    class Program
    {
        static int add(int x, int y){return x + y;}
        static void Main(string[] args)
        {
            Console.WriteLine("Please insert first number : ");
            int x = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("Please insert second number : ");
            int y = Convert.ToInt32(Console.ReadLine());
            Debug.WriteLine("Success - Reading values");
            Debug.WriteLine("Now Calculating add");
            int res = add(x, y);
            Console.WriteLine("res = {0}", res);
            Debug.WriteLine("Process Completed Successfully");
        }
    }
}
```



# Exception Handling :

في البرنامج السابق قمنا باستخدام Debug عوضاً عن Console ولكن يجب تضمين مكتبة الأسماء 

. System.Diagnostics

هذا يمكننا من طباعة تقرير سير عمل البرنامج في شاشة الـ Output وليس على شاشة الـ Console . 

وبالتالي سيحصل المستخدم على متطلباته دون أي زيادة تتعلق بالأمور البرمجية . 

وأيضاً سيقوم المبرمج بمهمته بشكل صحيح من خلال معرفة سير البرنامج ويفيده ذلك في تفحص الأخطاء 

ومعرفة الأماكن التي من الممكن أن يكون فيها أخطاء .

ولكن عند تنفيذ البرنامج يجب تنفيذ البرنامج في وضعية الـ Debug أي بالنقر على الزر start 

.  debugging

# Exception Handling :Break points

نقاط المقاطعة Break Points هي علامات في الـ source code وتتم مقاطعة تنفيذ التطبيق عند

الوصول إليها (أثناء عملية الـ Debugging) أي بالنقر على F5 وليس Ctrl+F5 .

يمكننا إعداد نقطة المقاطعة بعدة حالات أهمها:

- ١ - الدخول في نمط المقاطعة عند الوصول إلى نقطة المقاطعة .
- ٢ - الدخول في نمط المقاطعة عندما يتحقق شرط منطقي عند الوصول إلى نقطة المقاطعة .
- ٣ - الدخول في نمط المقاطعة عندما يتم المرور على نقطة المقاطعة بنفس القيم لعدد معين من المرات.

# Exception Handling :Break points

يتم إنشاء نقطة مقاطعة عن طريق النقر على يسار السطر في المنطقة الرمادية ، أو من خلال النقر 

بالزر الأيمن بالفأرة ثم اختيار Breakpoint ثم اختيار Insert Breakpoint .

يتم عرض نافذة التحكم بنقاط المقاطعة عن طريق Debug-> Windows -> Breakpoints 

# Exception Handling :Break points

يمكننا إعداد نقطة المقاطعة بإحدى الحالات :

١ - الدخول في نمط المقاطعة عند الوصول إلى نقطة المقاطعة .

```
12
13 static void Main(string[] args)
14 {
15     Console.WriteLine("Studying Breakpoints");
16     int[] vec = new int[50];
17     for (int i = 0; i < 50; i++)
18     {
19         vec[i] = i*2;
20     }
21
22 }
```

في المثال التالي نريد أن نشاهد القيم عند أول

دخول للحلقة ، فنضع نقطة مقاطعة بداخلها.

أثناء التنفيذ سيتم مقاطعة البرنامج ليتم التوقف

عند السطر الحاوي على نقطة مقاطعة ويحدث ذلك

أثناء عمل Start Debugging .

# Exception Handling :Break points

يمكننا إعداد نقطة المقاطعة بإحدى الحالات :

١ - الدخول في نمط المقاطعة عند الوصول إلى نقطة المقاطعة .

عند التنفيذ F5 فإنه يتوقف عمل البرنامج عند الوصول إلى نقطة المقاطعة وعند الإشارة بالفأرة على object i يمكننا معرفة القيمة الحالية له عند مقاطعة البرنامج ، هذا الأمر مفيد جداً للمبرمج ليتمكن من كشف أخطاءه .

```
13 static void Main(string[] args)
14 {
15     Console.WriteLine("Studying Breakpoints");
16     int[] vec = new int[50];
17     for (int i = 0; i < 50; i++)
18     {
19         vec[i] = i*2;
20     }
21
22 }
```

# Exception Handling :Break points

يمكننا إعداد نقطة المقاطعة بإحدى الحالات :

٢ - الدخول في نمط المقاطعة عندما يتحقق شرط منطقي عند الوصول إلى

في المثال التالي المقاطع عند القيم عند تحقق

شرط منطقي عند نقطة المقاطعة .

بعد إضافة نقطة المقاطعة نقر عليها بالزر الأيمن

للفأرة ثم نختار condition فتظهر لدينا نافذة

يحتوي مكاناً لكتابة الشرط المنطقي  $i==5$  فإذا

تحقق هذا الشرط المنطقي سيتم مقاطعة البرنامج .

```
12
13 static void Main(string[] args)
14 {
15     Console.WriteLine("Studying Breakpoints");
16     int[] vec = new int[50];
17     for (int i = 0; i < 50; i++)
18     {
19         vec[i] = i*2;
20     }
21 }
```

# Exception Handling :Break points

يمكننا إعداد نقطة المقاطعة بإحدى الحالات :

٢ - الدخول في نمط المقاطعة عندما يتحقق شرط منطقي عند الوصول إلى

```
13 static void Main(string[] args)
14 {
15     Console.WriteLine("Studying Breakpoints");
16     int[] vec = new int[50];
17     for (int i = 0; i < 50; i++)
18     {
19         vec[i] = i*2;
20     }
21 }
```

عند التنفيذ F5 فإن البرنامج يتوقف عن التنفيذ عند

الوصول إلى نقطة المقاطعة وعند الإشارة بالفأرة على

i object يمكننا معرفة القيمة الحالية لها عند مقاطعة

البرنامج ، وهي القيمة 5 لأنه تم مقاطعة البرنامج عند تحقق

الشرط المنطقي  $i==5$  .



# Exception Handling :Break points

يمكننا إعداد نقطة المقاطعة بإحدى الحالات :



٣ - الدخول في نمط المقاطعة عندما يتم المرور على نقطة المقاطعة بنفس القيم لعدد معين من المرات.



في المثال التالي نريد أن نشاهد القيم عندما يتم



المرور على سطر المقاطعة عدد معين من المرات .

بعد إضافة نقطة المقاطعة ننقر عليها بالزر الأيمن



للفأرة ثم نختار hit count فتظهر لدينا نافذة

نختار منها break when the hit count is

. equal to 3

```
13 static void Main(string[] args)
14 {
15     Console.WriteLine("Studying Breakpoints");
16     int[] vec = new int[50];
17     for (int i = 0; i < 50; i++)
18     {
19         Console.WriteLine("using count");
20         vec[i] = i*2;
21     }
22 }
```

# Exception Handling :Break points

يمكننا إعداد نقطة المقاطعة بإحدى الحالات :

٣ - الدخول في نمط المقاطعة عندما يتم المرور على نقطة المقاطعة بنفس القيم لعدد معين من المرات.

عند التنفيذ F5 فإن البرنامج يتوقف عن التنفيذ عند

الوصول إلى نقطة المقاطعة وعند الإشارة بالفأرة على

object i يمكننا معرفة القيمة الحالية لها عند مقاطعة

البرنامج ، وهي القيمة 2 لأنه تم مقاطعة البرنامج عند

المرور على سطر المقاطعة ثلاث مرات وعندها  $i=2$  .

```
13 static void Main(string[] args)
14 {
15     Console.WriteLine("Studying Breakpoints");
16     int[] vec = new int[50];
17     for (int i = 0; i < 50; i++)
18     {
19         Console.WriteLine("using count");
20         vec[i] = i*2;
21     }
22 }
```

# Exception Handling :Break points

يمكننا إعداد نقطة المقاطعة وفق حالات أهمها:



١ - الدخول في نمط المقاطعة عند الوصول إلى نمط المقاطعة .



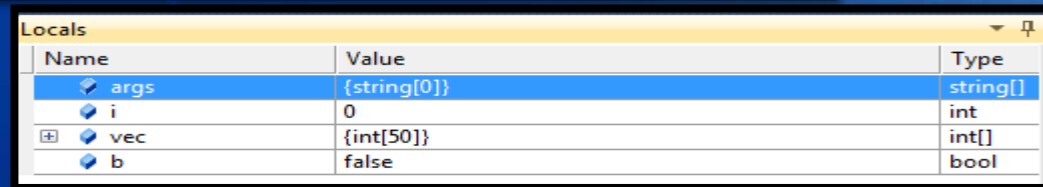
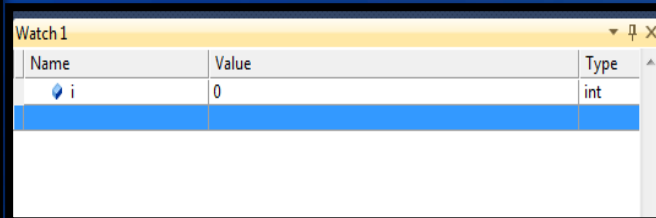
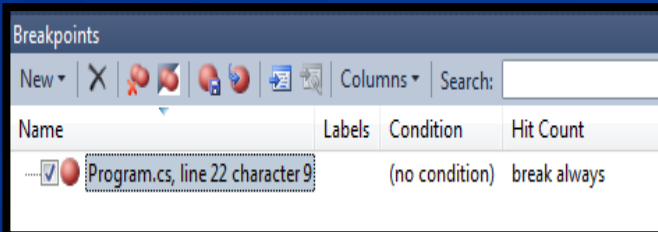
٢ - الدخول في نمط المقاطعة عندما يتحقق شرط منطقي عند الوصول إلى نقطة المقاطعة .



٣ - الدخول في نمط المقاطعة عندما يتم المرور على نقطة المقاطعة بنفس القيم لعدد معين من



المرات.



# Exception Handling :Exceptions

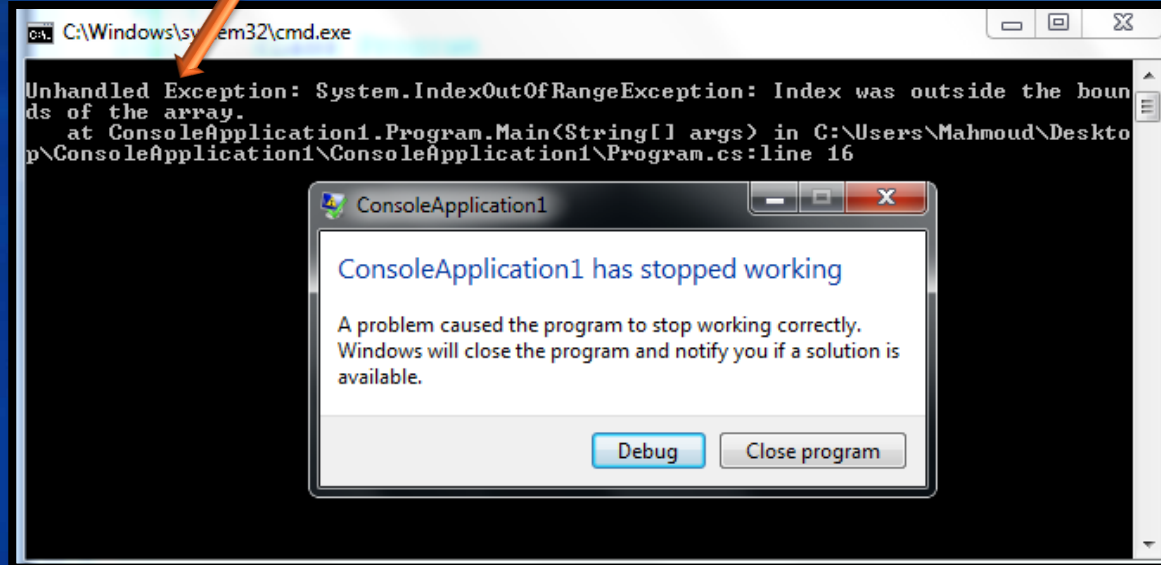
الاعتراض هو خطأ يحدث أثناء تنفيذ التطبيق وهو Runtime Error وليس Compiler Error .



ومثال عليه عندما نحاول الوصول إلى خارج حدود المصفوفة .



```
class Program
{
    static void Main(string[] args)
    {
        int[] vec = new int[3];
        Console.WriteLine(vec[3]);
    }
}
```



لتجنب حدوث مثل هذه الأخطاء

ولتجنب توقف البرنامج عن العمل

يجب معالجة هذه الاعتراضات .

# Exception Handling :Exceptions

الاعتراض هو خطأ يحدث أثناء تنفيذ التطبيق وهو Runtime Error وليس Compiler Error .

ومثال عليه عندما نحاول الوصول إلى خارج حدود المصفوفة .

في الحقيقة الاعتراض هو غرض من صف ما يعبر عن نوع الخطأ الحاصل ففي المثال السابق عند محاولة الوصول إلى خارج حدود المصفوفة فإنه تم إنشاء غرض من الصف `System.IndexOutOfRangeException` .

وبالتالي يكون لكل نوع من الاعتراضات صف خاص يعبر عنه .

# Exception Handling :Exceptions

عند ظهور خطأ في البرنامج يحدث لدينا اعتراض وفق الآلية التالية :

يتوقف تنفيذ function الذي أظهر لدينا الخطأ ولا ي function تنفيذه أبداً .

يتم إنشاء غرض من الصف الذي يعبر عن الخطأ الحاصل فإذا كان الخطأ

الحاصل هو محاولة الوصول إلى خارج حدود المصفوفة فإنه يتم إنشاء

غرض من الصف `System.IndexOutOfRangeException` .

ثم يتم رمي throw هذا الاعتراض (هذا الغرض) .

فإذا لم تتم معالجة هذا الخطأ في الكود فإنه سيتوقف تنفيذ البرنامج وإلا سيتم

م function تنفيذه .

# Exception Handling :Exceptions

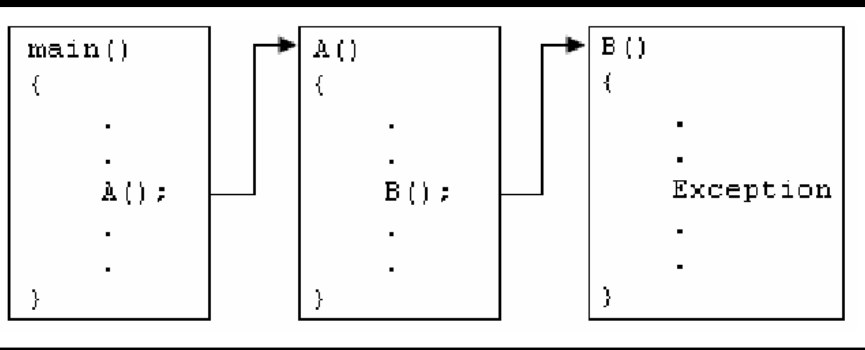


هنا نلاحظ ظهور اعتراض في B function فإذا تم التقاط catch هذا الاعتراض ومعالجته ضمن B function فإنه سيتم معالجة function تنفيذ البرنامج بشكل طبيعي ، وإذا لم يتم التقاط catch هذا الاعتراض ولم تتم معالجته داخل B function فإنه سيتم رمي هذا الاعتراض إلى المستوى الأعلى (أي إلى function الذي استدعى B ألا وهو A function) .



وبالمثل إذا تم التقاط catch هذا الاعتراض ومعالجته ضمن A function فإنه سيتم معالجة function تنفيذ البرنامج بشكل طبيعي ، وإلا سيتم رمي الاعتراض (الغرض) إلى المستوى الأعلى .. وهكذا ، فإذا لم يعالجه

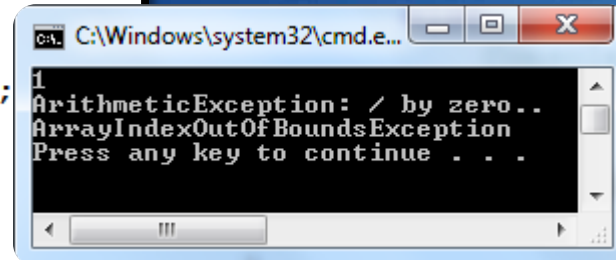
أحد سيتوقف البرنامج وتظهر رسالة الخطأ .



# Exception Handling :Exceptions

```
static void Do(double[] vec, int i)
{
    if (vec != null)
    {
        if (i >= 0 && i <vec.Length)
        {
            if (vec[i] != 0)
                Console.WriteLine(1.0 / vec[i]);
            else
                Console.WriteLine("ArithmeticException: / by zero..");
        }
        else
            Console.WriteLine("ArrayIndexOutOfBoundsException");
    }
    else
        Console.WriteLine("NullPointerException..");
}
```

```
static void Main(string[] args)
{
    double[] vec = new double[3] { 1, 0, 6 };
    Do(vec,0);
    Do(vec,1);
    Do(vec,4);
}
```



```
C:\Windows\system32\cmd.e...
1
ArithmeticException: / by zero..
ArrayIndexOutOfBoundsException
Press any key to continue . . .
```



# Exception Handling :Exceptions

في المثال السابق عالجنا ثلاث أنواع من الأخطاء :

١ - أن يكون المصفوفة غير مهياة أي null .

٢ . أن نحاول الوصول إلى خارج حدود المصفوفة .

٣ - القسمة على صفر .

لكن عملية المعالجة التي قمنا بها ليست قوية بالكفاية وتتطلب جهد من المبرمج ،وتؤدي إلى تعقيد في الكود .

توجد في لغة C# أسلوب متطور لمعالجة الاستثناءات وهو باستخدام try,catch,finally .

# Exception Handling :

الآن سنقوم بمعالجة الاعتراضات عن طريق try , catch , finally .

```
try
{
    //if an exception fired
}
catch (Exception)
{
    //here code for
    //exception handling
}
finally
{
    //code must executed
    //if an error fired or not .
}
```

أولاً : try

هي الكتلة التي ستتضمن الكود الذي من المتوقع أن يسبب

ظهور اعتراض exception بداخله.

فإذا ظهر اعتراض فإنه يتم توليد غرض من هذا

الاعتراض ثم يتم رميه throw من مكان ظهور هذا

الاعتراض أي بداخل الكتلة try إلى خارج الـ try .

# Exception Handling :

الآن سنقوم بمعالجة الاعتراضات عن طريق try , catch , finally .

```
try
{
    //if an exception fired
}
catch (Exception)
{
    //here code for
    //exception handling
}
finally
{
    //code must executed
    //if an error fired or not .
}
```

ثانياً: catch

هي الكتلة التي تتضمن الكود الذي سينفذ من أجل معالجة

الاعتراضات التي تم رميها من داخل try .

فحتى نقوم بالتقاط اعتراض من النمط

System.IndexOutOfRangeException فيجب كتابة داخل

القوسين التي تلي catch نوع الاعتراض.

# Exception Handling :

الآن سنقوم بمعالجة الاعتراضات عن طريق try , catch , finally . 

```
try
{
    //if an exception fired
}
catch (Exception)
{
    //here code for
    //exception handling
}
finally
{
    //code must executed
    //if an error fired or not .
}
```

ثانياً: catch 

من الممكن أن يكون لدينا أكثر من 

catch كل واحدة تقوم بالتقاط نوع

معين من الاعتراضات كي تعالجها

باسلوب معين .

# Exception Handling :

الآن سنقوم بمعالجة الاعتراضات عن طريق try , catch , finally . 

```
try
{
    //if an exception fired
}
catch (Exception)
{
    //here code for
    //exception handling
}
finally
{
    //code must executed
    //if an error fired or not .
}
```

ثانياً: catch 

يمكن التقاط أي اعتراض عن طريق ذكر catch 

(Exception) والتي يمكن فهمها بالتقاط

اعتراض مهما كان نوعه ولكن هنا لن نتمكن من

معالجة هذا الاعتراض بشكل صحيح لأننا لن

نتمكن من معرفة نوعه.

# Exception Handling :

الآن سنقوم بمعالجة الاعتراضات عن طريق try , catch , finally . 

```
try
{
    //if an exception fired
}
catch (Exception)
{
    //here code for
    //exception handling
}
finally
{
    //code must executed
    //if an error fired or not .
}
```

ثالثاً: finally 

إن وجدت فهي تتضمن الكود الذي ينفذ دوماً بالحالات: 

بعد الكتلة try إذا لم يُرمى أي اعتراض . 

بعد الكتلة catch إذا تم رمي اعتراض من داخل 

try ومعالجته من قبل إحدى الـ catch .

قبل ظهور رسالة الخطأ في حال لم يعالج الخطأ . 

# Exception Handling :

الآن سنقوم بمعالجة الاعتراضات عن طريق try , catch , finally . 

```
try
{
    //if an exception fired
}
catch (Exception)
{
    //here code for
    //exception handling
}
finally
{
    //code must executed
    //if an error fired or not .
}
```

ثالثاً: finally 

إذا قمنا بفتح ملف في داخل الكتلة try وسواء نجحنا 

أو فشلنا في عملية فتح الملف فإنه يجب علينا أن نقوم

بإغلاق الملف كي لا تبقى هنالك موارد مستخدمة في

النظام كي لا تحدث مشاكل لاحقة ، وتتم عملية إغلاق


الملف في داخل الكتلة finally إذ أنها تُنفذ دوماً .

# Exception Handling :

```
static void Do(double[] vec, int i)
{
    try
    {
        Console.WriteLine(1.0 / vec[i]);
    }
    catch (DivideByZeroException)
    {
        Console.WriteLine("Sorry : Error division by 0");
    }
    catch (IndexOutOfRangeException)
    {
        Console.WriteLine("Sorry : trying to access out of array bounds");
    }
    catch (Exception)
    {
        Console.WriteLine("Sorry :Error ");
    }
}
```

```
static void Main(string[] args)
{
    double[] vec = new double[3] { 1, 0, 6 };
    Do(vec, 0);
    Do(vec, 1);
    Do(vec, 4);
}
```



اكتب function سمه PrintWithFormat والذي يقوم بعمل يشبه عمل function   
WriteLine والذي يأخذ كدخل أول له تنسيق السلسلة النصية المراد طباعتها أما  
باقي عناصر الدخل تشكل Variable list .

```
int x=2 , y= 3 , z = 14 ;  
Console.WriteLine("{2} add {0} add {1} equals {3}", x, y, z, x + y + z);
```

ينبغي استخدام معالجة السلاسل المحرفية حتى يتم معالجة الدخل الأول . 

ينبغي استخدام params . 

**The End**