



# البرمجة الإجرائية Procedural Programming

IPG202

## الفصل الرابع الطرائق Methods

## الكلمات المفتاحية

برمجة إجرائية، تابع، إجراء، طريقة، طريقة مسبقة التعريف، طريقة معرفة من قبل المستخدم، قيمة معادة، void، return، استدعاء، مرجع، عودية، متحول محلي، متحول عام.

## ملخص الفصل

يركز هذا الفصل على مفهوم الطرائق، حيث يعرف بمصطلح البرمجة الإجرائية، ثم يقدم إضاءة حول الطرائق مسبقة التعريف في لغة C#، لينتقل بعدها لتقديم المفاهيم المتعلقة بالطرائق المعرفة من قبل المستخدم، حيث يبين آلية تعريفها واستخدامها ومجالات رؤيتها وتحولاتها والمفاهيم المختلفة المتعلقة بتمرير المعاملات، ليختتم بالتعريف بنوع خاص من الطرائق يدعى الطرائق العودية، مع عدد من الأمثلة التي تدعم اكتساب المهارات الخاصة بالتعامل مع هذا المفهوم.

## أهداف الفصل

بنهاية هذا الفصل سيكون الطالب قادراً على:

- توضيح مفهوم البرمجة الإجرائية.
- تعريف الطرائق في لغة C#.
- استخدام آليات استدعاء الطرائق المختلفة.
- تعريف مجالات الرؤية للمعرفات.
- إجراء التحميل الزائد للطريقة.
- تحقيق العمليات العودية من خلال الطرائق.

## محتويات الفصل

1. مقدمة
2. لغة C# والطرائق.
3. الطرائق مسبقة التعريف.
4. تعريف الطرائق.
5. استخدام الطرائق ( الاستدعاء).
6. استدعاء الطرائق بالمرجع (بالعنوان).
7. المصفوفة كبارامتر للطريقة.
8. استخدام الوسطاء الافتراضية.
9. مجالات الرؤية.
10. التحميل الزائد للطرائق
11. الطرائق العودية.
12. أمثلة حول الطرائق.
13. تمارين وأنشطة.

## 1- مقدمة.

رأينا فيما مضى، أن حل المسألة بلغة C# يتضمن -فيما يتضمن- تحديد وتوصيف العمليات التي تنفذ ضمن البرنامج، ومن ثم ترتيب هذه العمليات وفق خوارزمية تعبر عن حل هذه المسألة. حيث يكون تدفق التحكم في هذه الخوارزمية تسلسلياً sequential أو تفرعياً branching.

إن الكلام السابق يبقى فعالاً وسليماً من أجل كل أنواع المسائل. إلا أننا وجدنا أن هذه الفعالية تتناقص في الكثير من الحالات ومنها:

1. عندما تطول المسألة، إذ يصبح البرنامج عندها صعب التتبع والفهم والتصحيح.
2. عندما نتتقد المسألة ويكون توصيف العمليات المتضمنة فيها مركباً ويحتوي بحد ذاته على عمليات داخلية، إذ يصبح إدراج شيفرة العملية ضمن البرنامج الكامل مركباً وصعباً.
3. عندما يتكرر استخدام العملية مرات عديدة ضمن المسألة، نصبح عندها بحاجة لإعادة استخدام كتل كبيرة من الشيفرة عدة مرات ضمن البرنامج.

لقد وجدت الإشكالية المطروحة حلاً لها من خلال مفهوم البرمجة الإجرائية procedural programming -الذي تعتمد لغة C#- إذ يعتمد هذا المفهوم على مبدأ "فرق تسد divide and conquer" حيث يتم تجزئة المسألة المطروحة، إلى عدد من المسائل الجزئية، حيث يتم التعامل مع كل مسألة جزئية كما لو كانت مسألة مستقلة ويكتب لها برنامج فرعي، ومن ثم يتم توظيف البرامج الفرعية ضمن برنامج واحد رئيسي.

يدعى البرنامج الفرعي الذي يعالج عملية ( أو مسألة ) جزئية باسم الطريقة method ( أو التابع function أو الإجراء procedure).

## 2- لغة C# والطرائق

تضمنت لغة C# - في تقييساتها المختلفة - كماً كبيراً من الطرائق التي تم تعريفها من قبل ناشري اللغة، حيث تم تخزين هذه الطرائق في مكتبات namespaces وأصناف classes قياسية، كل مكتبة خاصة بفئة معينة من الطرائق -حسب الوظيفة والتأثير-، إذ يصبح بإمكان المبرمج استخدامها تلقائياً ( عن طريق استدعائها إلى البرنامج الرئيسي أو أي من البرامج الفرعية ) بمجرد أن يقوم بتضمين المكتبة التي تحتوي هذه الطريقة باستخدام التوجيه للمترجم using ضمن قسم التضمينات من البرنامج.

دعيت هذه الطرائق باسم الطرائق مسبقاً التعريف pre-defined methods.

إلا أن هذا الأمر لم يمنع من قيام لغة C# بإتاحة الإمكانية للمبرمج بأن يقوم هو بنفسه بتعريف طرائقه الخاصة، وذلك لإتاحة المجال أمامه للقيام بتوسيع اللغة نظراً لكون الطرائق مسبقة التعريف في اللغة لا تغطي كافة الوظائف التي قد نواجهها في مسائل العالم الحقيقي.

دعيت هذه الطرائق باسم **الطرائق المعرفة من قبل المستخدم user-defined methods**.

يتم تعريف الطرائق المعرفة من قبل المستخدم إما بشكل مباشر ضمن الصنف الأساسي للبرنامج (الحاوي على الطريقة main) وإما ضمن أصناف أو مكتبات أخرى يتم إنشاؤها من قبل المبرمج، ويصبح فيما بعد بإمكان المبرمج استخدامها بنفس طريقة استخدام طرائق المكتبات القياسية، أي عن طريق استدعائها إلى الطريقة الرئيسية main أو أي من البرامج الفرعية.

يقوم مترجم اللغة في مرحلة الوصل linking باستدعاء الطرائق التي تم تعريفها في مكان آخر مثل المكتبات المعيارية أو المكتبات التي تم بناؤها من قبل المبرمج، وبالتالي يقوم الواصل linker بوصل الملف السابق مع نصوص الطرائق المستدعاة.

### 3- الطرائق مسبقة التعريف

استخدمنا فيما سبق -لدى تعرفنا على أساسيات البرمجة بلغة C#- العديد من الطرائق مسبقة التعريف ضمن اللغة والمخزنة ضمن المكتبة (فضاء الأسماء) System، حيث استخدمنا طرائق مثل ReadLine، WriteLine، Parse.... إلخ، وقد كان استخدامنا لها كما لو أننا نستخدم صندوقاً مقلداً، بحيث كان يكفي أن نقوم بذكر اسم التابع بطريقة ملائمة وإعطائه القيم التي يحتاجها لإنجاز مهمته ليتولى إنجاز هذه المهمة وإعطائنا الناتج النهائي لعمله.

لسنا في وارد سرد جميع الطرائق والمكتبات مسبقة التعريف في لغة C#، ولكننا سنتوقف قليلاً مع طرائق الصنف Math بغية إيضاح مفهوم الطرائق مسبقة التعريف وكيفية استخدامها.

#### الصنف Math

يحتوي الصنف Math (المعرف ضمن فضاء الأسماء System) على مجموعة كبيرة من الطرائق الرياضية (إضافة إلى مجموعة من الثوابت العامة والساكنة).

يمكن استخدام هذه الطرائق عبر استدعائها، فمثلاً يمكن حساب الجذر التربيعي لعدد بكتابة:

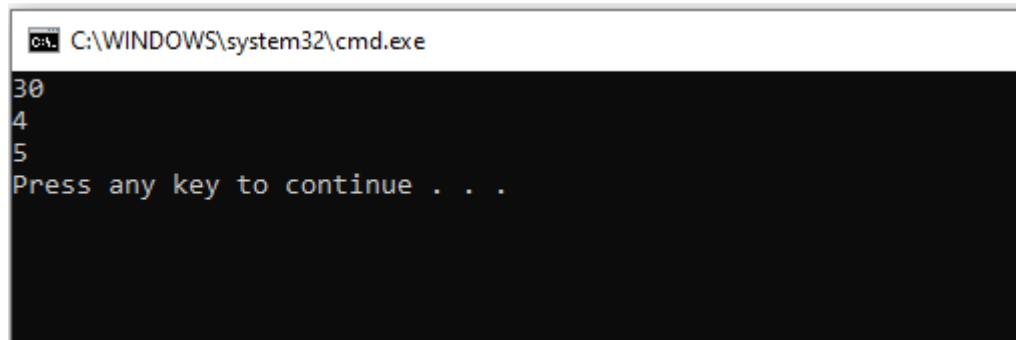
Math.Sqrt( 900.0 );

لقد استخدمنا في الأمر السابق، الطريقة Sqrt (من طرائق الصنف Math) حيث قمنا باستدعائها ملحقة بالإشارة إلى الصنف Math وإعطائها القيمة 900.0 لكي تقوم بإنجاز عملية حساب الجذر التربيعي لها.

## مثال 1:

```
static void Main(string[] args)
{
    const double C = 900.0;
    double x = 16.0;
    double c = 13.0, d = 3.0, f = 4.0;
    Console.WriteLine(Math.Sqrt(C));
    Console.WriteLine(Math.Sqrt(x));
    Console.WriteLine(Math.Sqrt(c + d * f));
}
```

قمنا في المثال 1 بإجراء استدعاءات مختلفة للطريقة Sqrt، يعطي هذا البرنامج عند تنفيذه الخرج التالي:

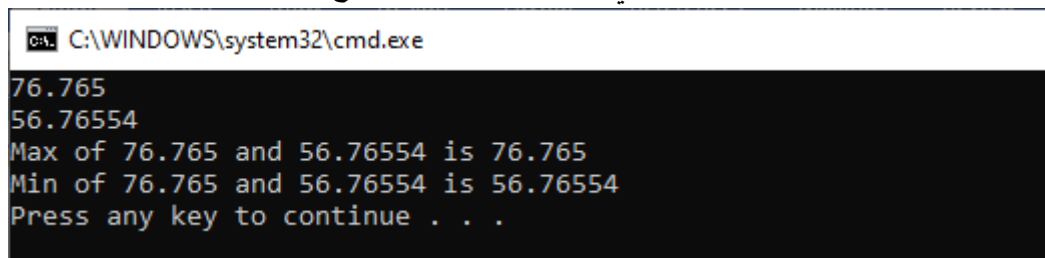


```
C:\WINDOWS\system32\cmd.exe
30
4
5
Press any key to continue . . .
```

## مثال 2:

```
static void Main(string[] args)
{
    double x, y, min, max;
    x = double.Parse(Console.ReadLine());
    y = double.Parse(Console.ReadLine());
    max = Math.Max(x, y);
    min = Math.Min(x, y);
    Console.WriteLine("Max of {0} and {1} is {2}", x, y, max);
    Console.WriteLine("Min of {0} and {1} is {2}", x, y, min);
}
```

تضمن المثال 2 استدعاء للطريقتين Max و Min لحساب القيمة العظمى (والقيمة الدنيا على التوالي) لقيمتين تم إعطاءهما للطرائق عبر الوسيطين x و y، فيما يلي عينة عن تنفيذ هذا البرنامج:



```
C:\WINDOWS\system32\cmd.exe
76.765
56.76554
Max of 76.765 and 56.76554 is 76.765
Min of 76.765 and 56.76554 is 56.76554
Press any key to continue . . .
```

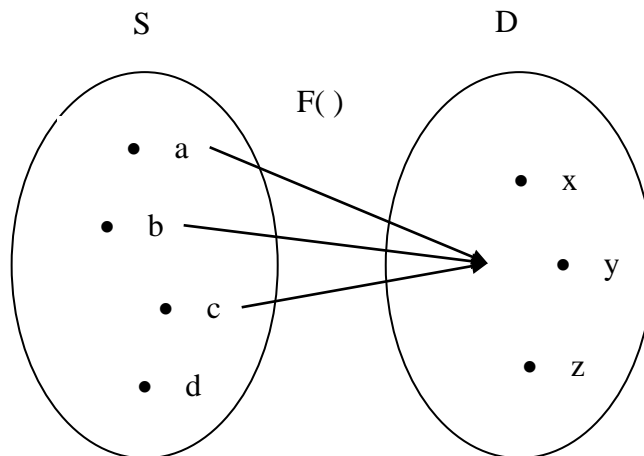
يبين الجدول التالي أهم طرائق الصنف Math:

الطريقة	الوصف	أمثلة
<b>Abs( x )</b>	القيمة المطلقة لـ x	Abs( 23.7 ) is 23.7 Abs( 0 ) is 0 Abs( -23.7 ) is 23.7
<b>Ceiling( x )</b>	التقريب لأصغر عدد طبيعي ليس أصغر من x	Ceiling( 9.2 ) is 10.0 Ceiling( -9.8 ) is -9.0
<b>Cos( x )</b>	جيب x (بالراديان)	Cos( 0.0 ) is 1.0
<b>Exp( x )</b>	الرفع لقوة العدد e	Exp( 1.0 ) is approximately 2.7182818284590451 Exp( 2.0 ) is approximately 7.3890560989306504
<b>Floor( x )</b>	التقريب لأكبر عدد طبيعي ليس أكبر من x	Floor( 9.2 ) is 9.0 Floor( -9.8 ) is -10.0
<b>Log( x )</b>	اللوغاريتم الطبيعي لـ x (القاعدة e)	Log( 2.7182818284590451 ) is approximately 1.0 Log( 7.3890560989306504 ) is approximately 2.0
<b>Max( x, y )</b>	أكبر قيمة	Max( 2.3, 12.7 ) is 12.7 Max( -2.3, -12.7 ) is -2.3
<b>Min( x, y )</b>	أصغر قيمة	Min( 2.3, 12.7 ) is 2.3 Min( -2.3, -12.7 ) is -12.7
<b>Pow( x, y )</b>	x مرفوع للقوة y	Pow( 2.0, 7.0 ) is 128.0 Pow( 9.0, .5 ) is 3.0
<b>Sin( x )</b>	جيب x (بالراديان)	Sin( 0.0 ) is 0.0
<b>Sqrt( x )</b>	الجذر التربيعي لـ x	Sqrt( 900.0 ) is 30.0 Sqrt( 9.0 ) is 3.0

#### 4- تعريف الطرائق

لقد اشتق مفهوم الطريقة (التابع أو الإجرائية) في لغة C# من مفهوم التابع في الرياضيات. حيث يمكن تعريف التابع (رياضياً) على أنه علاقة تربط عنصر أو أكثر من مجموعة المنطلق source بعنصر واحد من مجموعة المستقر destination.

يبين الشكل التالي تمثيل التابع في الرياضيات



الشكل 1- تمثيل التابع في الرياضيات.

يمكن القول بأن على سبيل المثال:

$$y = F(a, b, c) = a + b^2 - c^3$$

حيث أن:

y : تدعى قيمة التابع.

a, b, c : تدعى وسطاء التابع.

F : اسم التابع.

a + b<sup>2</sup> - c<sup>3</sup> : علاقة التابع، ويقصد به العملية التي يقوم بها التابع لتحديد العنصر الذي سيتم الارتباط به في مجموعة المستقر ( الحالة هنا حالة مثال ).

قامت لغة C# بمحاكاة هذا المفهوم في تعريفها للطرائق، حيث يمكن تعريف الطريقة باستخدام الصيغة العامة التالية:

```
access_qualifier return_type method_name(parameter_declaration_list)
{
    statement_list
    return return_value;
}
```

حيث:

- **Access\_qualifier** يعبر عن أسلوب استدعاء الطريقة وسماحية الوصول إليها.
- **return\_type** هو نوع القيمة المعادة (النتيجة) من قبل الطريقة.
- **method\_name** هو معرف يحدد اسم الطريقة.
- **parameter\_declaration\_list** هي لائحة التصريحات لبارامترات الطريقة تفصل بينها فواصل.
- **statement\_list** تتالي التعليمات والأوامر التي تصف سلوك الطريقة.
- **return\_value** القيمة التي تقوم الطريقة بإعادتها.
- **return** كلمة مفتاحية وظيفتها إنهاء التابع وإعادة القيمة الناتجة.

ننوه هنا إلى الأمور الأساسية التالية:

**الأمر الأول:** سنكتفي في كامل هذه الأملية باستخدام الكلمتين المفتاحيتين `public` و `static` في موضع كتابة محدد الوصول `access qualifier` والتين تعنيان أن هذه الطريقة تنتمي للصنف الذي عرفت ضمنه ويوجد منها نسخة واحدة ويمكن الوصول إليها من قبل جميع أجزاء البرنامج.

**الأمر الثاني:** يمكن للقيمة المعادة أن تكون من أي نوع معروف في اللغة أو معرف من قبل المستخدم، كما يمكن للطريقة أن لا يكون لها قيمة معادة وفي هذه الحالة يعبر عن نوع القيمة المعادة باستخدام الكلمة المفتاحية `void`.

**الأمر الثالث:** يمكن للطريقة أن يكون لها أي عدد من البارامترات ومن أنواع مختلفة وفي هذه الحالة يتم التصريح عن هذه البارامترات ضمن القوسين ويفصل بينها بفواصل عادية، كما يمكن أن لا يكون للطريقة أي بارامترات وفي هذه الحالة يتم وضع قوسين فارغين بعد اسم الطريقة.

بالنظر إلى هذه الصيغة العامة نجد أن المحاكاة التي قامت بها لغة `C#` تتمثل في المقابلة بين مكونات تعريف التابع رياضياً وتعريف الطريقة برمجياً كما يلي:

التابع رياضياً	الطريقة برمجياً
اسم مجموعة المستقر	نوع القيمة المعادة
اسم التابع	اسم الطريقة
اسم مجموعة المنطلق وعناصرها.	لائحة التصريحات لبارامترات الطريقة
القيمة الناتجة	القيمة المعادة
علاقة التابع.	تتالي التعليمات

علماً أن تعريف الطريقة برمجياً تضمن بعد التمايزات عن المفهوم الرياضي، إذ يمكن برمجياً -في بعض الحالات- أن لا يكون للطريقة قيمة معادة، لا يكون لها بارامترات، أو أن تكون بارامترات من أكثر من نوع.

نورد فيما يلي مجموعة من الأمثلة حول تعريف طرائق بأشكال مختلفة، ولا نهتم بكيفية استخدامها (لأن ذلك سيرد في الفقرة القادمة).



مثال 1: تعريف طريقة لها قيمة معادة وتملك بارامتراً أو أكثر

```
class MethodsExamples
{
    static void Main(string[] args)
    {
    }
    // returns the maximum of its three integer parameters
    public static int Maximum(int x, int y, int z)
    {
        int maximumValue = x; // assume x is the largest to start
        // determine whether y is greater than maximumValue
        if (y > maximumValue)
            maximumValue = y;
        // determine whether z is greater than maximumValue
        if (z > maximumValue)
            maximumValue = z;
        return maximumValue;
    } // end method Maximum
}
```

بالنظر إلى المثال السابق، نسجل الملاحظات التالية:

- تضمنت الطريقة ثلاثة بارامترات، حيث تم تعريف البارامترات ضمن ترويسة الطريقة محصورة بين قوسين ( ) وتضمن تعريف كل بارامتر تحديد نوعه واسمه، وتم الفصل بين البارامترات بفواصل عادية.
- القيمة المعادة أو الناتجة عن الطريقة هي من النوع int.
- وضعت كامل أوامر الطريقة بعد التصريح عنها ضمن قوسي كتلة { }.
- استخدمت الكلمة المفتاحية return لإعادة ناتج تنفيذ الطريقة (وهي هنا القيمة الكبرى بين ثلاث قيم معطاة).

مثال 2: تعريف طريقة لها قيمة معادة ولا تملك بارامترات

```
class MethodsExamples
{
    static void Main(string[] args)
    {
    }
    // returns the sum of values within range 1 to 50
    public static int SumInterval( )
    {
        int sum = 0;
        for (int i = 1; i<50; i++)
            sum += i;
        return sum;
    } // end method SumInterval
}
```

بالنظر إلى المثال السابق، نسجل الملاحظة الإضافية التالية:

- لم يتضمن تعريف الطريقة أية بارامترات لذلك تركت الأقواس فارغة، ولا يجوز مطلقاً حذف الأقواس عندما لا يكون هناك بارامترات للطريقة.

مثال 3: تعريف طريقة ليس لها قيمة معادة وتملك بارامتراً أو أكثر

```
class MethodsExamples
{
    static void Main(string[] args)
    {
    }
    // printing welcome message
    public static void WelcomeMsg(string name )
    {
        Console.WriteLine("Welcome {0} to Methods in C# ", name);
    } // end method WelcomeMsg
}
```

بالنظر إلى المثال السابق، نسجل الملاحظات الإضافية التالية:

- ليس لهذه الطريقة قيمة معادة لذلك تم استخدام الكلمة المفتاحية void للدلالة على ذلك.
- حذفت العبارة return من الطريقة لأنه ليس هناك قيمة ناتجة عن هذه الطريقة لتتم إعادتها.

مثال 4: تعريف طريقة ليس لها قيمة معادة و لا تملك بارامترات

```
class MethodsExamples
{
    static void Main(string[] args)
    {
    }
    // printing general welcome message
    public static void GeneralWelcomeMsg( )
    {
        Console.WriteLine("Welcome to Methods in C# ");
    } // end method GeneralWelcomeMsg
}
```

## 5- استخدام الطرائق (الاستدعاء)

لو حاولنا تنفيذ أي من البرامج المبينة في الأمثلة الأربعة السابقة، فإن النتيجة ستكون ذاتها .... لا شيء سيتم تنفيذه وستظهر لنا شاشة خرج فارغة، كما يلي:

```
C:\WINDOWS\system32\cmd.exe
Press any key to continue . . .
```

لا يتم تنفيذ شيفرة الطريقة بمجرد أن نقوم بتعريفها، وإنما لكي يتم تنفيذها يجب أن يتم استخدام الطريقة وذلك يتم من خلال استدعائها calling في المكان الذي نرغب بتطبيق تأثيرها فيه وباستخدام القيم المراد تحقيق هذا التأثير عليها.

يمكن أن يتم استدعاء الطريقة في أي مكان من البرنامج، سواء أكان ذلك من قبل الطريقة الرئيسية main() أو من قبل أي طريقة أخرى معرفة من قبل المستخدم.

تتطلب عملية استدعاء التابع السليمة التركيز على مكونات ثلاثة:

- **المكون الأول هو اسم الطريقة :** يستخدم اسم الطريقة كمعرف لاستدعاء ( مناداة ) هذه الطريقة.
- **المكون الثاني هو البارامترات :** حيث يتم تمرير قيم لبارامترات الطريقة بحيث يتم مراعاة مايلي:
  - عدد القيم الممررة يجب أن يساوي عن عدد البارامترات.
  - أنواع القيم الممررة يجب أن يطابق أنواع بارامترات الطريقة.
  - ترتيب القيم الممررة يجب أن يطابق ترتيب بارامترات الطريقة.
- **المكون الثالث هو القيمة المعادة :** ونميز هنا بين حالتين:
  - حالة طريقة لا تعيد قيمة (نوع القيمة المعادة هو void) في هذه الحالة يكتب اسم الطريقة مع تمرير قيم إلى بارامترات بشكل سليم (ولا يجوز إسنادها أو استخدامها ضمن تعبير حسابي ... إلخ).
  - حالة طريقة تعيد قيمة (نوع القيمة المعادة ليس void) في هذه الحالة يجب أن يتم الاستدعاء بما يتناسب مع القيمة المعادة أي ضمن تعبير إسناد، أو تعبير حسابي، أو تعليمة إخراج.

كأمثلة على كل ماسبق فيما يتعلق باستدعاء الطرائق، لنعد إلى الأمثلة المعرفة في الفقرة السابقة ولنحاول استدعاء الطرائق المعرفة في كل مرة بكل الطرق الممكنة.

مثال 1: استدعاء طريقة لها قيمة معادة وتملك بارامتراً أو أكثر

```
class MethodsExamples
{
    static void Main(string[] args)
    {
        int a = 19, b = 23, c = 12;

        //Valid calls for Maximum method
        Console.WriteLine(Maximum(a, b, c));           //V1
        int z = Maximum(a, b, c);                     //V2
        z = Maximum(a, b, c) / 2;                     //V3
        if (Maximum(a, b, c) > Maximum(50, 20, 12))    //V4
            Console.WriteLine("It is OK");

        //Invalid calls for Maximum method
        Console.WriteLine(Maximum(22.4, 33.2, 10.5)); //NV1
        Console.WriteLine(Maximum(a, b));             //NV2
        Maximum(10, 12, 11);                           //NV3
    }
    // returns the maximum of its three integer parameters
    public static int Maximum(int x, int y, int z)
    {
        int maximumValue = x; // assume x is the largest to start
        // determine whether y is greater than maximumValue
        if (y > maximumValue)
            maximumValue = y;
        // determine whether z is greater than maximumValue
        if (z > maximumValue)
            maximumValue = z;
        return maximumValue;
    } // end method Maximum
}
```

بالنظر إلى المثال السابق، نلاحظ أنه قد تم استدعاء الطريقة 4 استدعاءات سليمة و 3 استدعاءات خاطئة كما يلي:

■ الاستدعاءات السليمة:

- الاستدعاء V1 سليم، تم استدعاء الطريقة بذكر اسمها ضمن عبارة إخراج (وهذا يتناسب مع نوع القيمة المعادة) مع تمرير ثلاث قيم من النوع int (وهذا يتناسب مع عدد البارامترات ونوعها وترتيبها).
- الاستدعاء V2 سليم، تم استدعاء الطريقة بذكر اسمها ضمن عبارة إسناد إلى متحول من النوع int (وهذا يتناسب مع نوع القيمة المعادة) مع تمرير ثلاث قيم من النوع int (وهذا يتناسب مع عدد البارامترات ونوعها وترتيبها).

○ الاستدعاء V3 سليم، تم استدعاء الطريقة بذكر اسمها ضمن تعبير حسابي (وهذا يتناسب مع نوع القيمة المعادة) مع تمرير ثلاث قيم من النوع int (وهذا يتناسب مع عدد البارامترات ونوعها وترتيبها).

○ الاستدعاء V4 سليم، تم استدعاء الطريقة بذكر اسمها ضمن عبارة شرطية (وهذا يتناسب مع نوع القيمة المعادة) مع تمرير ثلاث قيم من النوع int (وهذا يتناسب مع عدد البارامترات ونوعها وترتيبها).

#### ■ الاستدعاءات الخاطئة

○ الاستدعاء NV1 خاطئ، السبب عدم التناسب بين أنواع القيم الممررة (double) وأنواع بارامترات الطريقة (int).

○ الاستدعاء NV2 خاطئ، السبب عدم التناسب بين عدد القيم الممررة وعدد بارامترات الطريقة.

○ الاستدعاء NV3 خاطئ، السبب عدم التناسب بين نوع القيمة المعادة وطريقة الاستدعاء.

#### ملاحظات هامة:

✓ يمكن أن يتم تمرير القيم إلى البارامترات بشكل مباشر Maximum(50, 20, 12) أو عبر وسطاء Maximum(a, b, c).

✓ تدعى الاستدعاءات من هذا النوع باسم الاستدعاء بالقيمة call by value وفي هذه الحالة يتم إنشاء نسخ من قيم الوسطاء الممررة وإعطائها لبارامترات الطريقة.

#### مثال 2: استدعاء طريقة لها قيمة معادة ولا تملك بارامترات

```
class MethodsExamples
{
    static void Main(string[] args)
    {
        //Valid calls for SumInterval method
        Console.WriteLine(SumInterval ());           //V1
        int z = SumInterval ();                       //V2
        z = SumInterval () * 10;                     //V3
        if (SumInterval () > 2000)                   //V4
            Console.WriteLine("It is OK");

        //Invalid calls for SumInterval method
        short k = SumInterval() ;                   //NV1
        int k = SumInterval ;                       //NV2
        SumInterval ();                             //NV3
    }
    // returns the sum of values within range 1 to 50
    public static int SumInterval( )
    {
        int sum = 0;
```

```

    for (int i = 1; i<50; i++)
        sum += i;
    return sum;
} // end method SumInterval
}

```

بالنظر إلى المثال السابق، نلاحظ أنه تم استدعاء الطريقة مع الإبقاء على القوسين فارغين.  
نركز فقط على الاستدعاءات الخاطئة:

- الاستدعاءان NV1 و NV3 خاطئان، عدم التناسب بين نوع القيمة المعادة وطريقة الاستدعاء.
- الاستدعاء NV2 خاطئ، السبب حذف الأقواس أثناء الاستدعاء.

مثال 3: استدعاء طريقة ليس لها قيمة معادة وتملك بارامتراً أو أكثر

```

class MethodsExamples
{
    static void Main(string[] args)
    {
        //Valid calls for WelcomeMsg method
        WelcomeMsg ("Ahmad"); //V1
        string myName = "Salem";
        WelcomeMsg (myName); //V2

        //Invalid calls for WelcomeMsg method
        Console.WriteLine(WelcomeMsg ("Ahmad")); //NV1
        int k = WelcomeMsg ("Ahmad"); //NV2
        WelcomeMsg (); //NV3
    }
    // printing welcome message
    public static void WelcomeMsg(string name )
    {
        Console.WriteLine("Welcome {0} to Methods in C# ", name);
    } // end method WelcomeMsg
}

```

ناقش الاستدعاءات السليمة والاستدعاءات الخاطئة على غرار ما سبق.

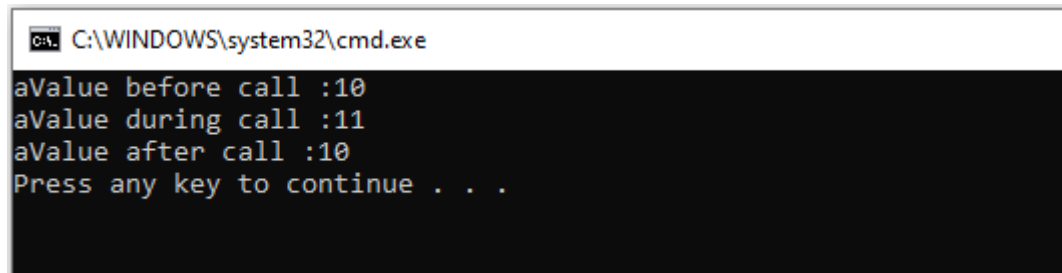
## 6- استدعاء الطرائق بالمرجع (بالعنوان)

رأينا في الفقرة السابقة أن عملية تمرير قيم الوسطاء arguments إلى بارامترات الطريقة parameters ما هي إلا عملية نسخ لقيم هذه الوسطاء إلى بارامترات الطريقة، وبالتالي فإن أي تغييرات تطرأ على قيم هذه الوسطاء داخل الطريقة لا تنعكس على قيمها ضمن الطريقة المستدعية.

يمكن توضيح هذا الأمر من خلال المثال البسيط التالي:

```
static void Main(string[] args)
{
    int aValue = 10;
    Console.WriteLine("aValue before call :" + aValue);
    int bValue = TestValCall(aValue);
    Console.WriteLine("aValue after call :" + aValue);
}
public static int TestValCall(int x)
{
    x++;
    Console.WriteLine("aValue during call :" + x);
    return x;
}
```

يعطي هذا البرنامج على خرجه:



```
C:\WINDOWS\system32\cmd.exe
aValue before call :10
aValue during call :11
aValue after call :10
Press any key to continue . . .
```

نلاحظ أن قيمة المتحول aValue بقيت هي ذاتها قبل وبعد استدعاء الطريقة علماً أن هذه القيمة قد جرت زيادتها بمقدار 1 ضمن الطريقة، السبب في ذلك أنه عند الاستدعاء يتم نسخ قيمة الوسيط aValue إلى البارامتر x وبالتالي فالتغييرات الحاصلة تصيب قيمة x فقط ولا تنعكس إلى خارج الطريقة على قيمة aValue.

يمكن تمثيل أداء الذاكرة المعبر عن هذه العملية كما يلي:

الذاكرة قبل استدعاء الطريقة		الذاكرة لحظة استدعاء الطريقة	
aValue	10	aValue	10
bValue		bValue	
		x	10
الذاكرة قبيل الخروج من الطريقة		الذاكرة بعد استدعاء الطريقة	
aValue	10	aValue	10
bValue		bValue	11
x	11		

يدعى هذا النمط من الاستدعاء للطرائق باسم الاستدعاء بالقيمة **Call by value** ويخلص بما يلي:

- يتم في هذه الحالة تمرير نسخة من قيمة المتغير الممرر كمعامل إلى الطريقة.
- لن يتأثر هذا المتغير بأي تعديلات تقوم بها الطريقة على المعامل في جسم الطريقة.

يحصل في بعض الأحيان أن تتطلب طبيعة المسألة أن تنعكس التغيرات الجارية على قيم الوسطاء داخل الطريقة على قيمها خارج الطريقة.

لأجل التعامل مع هذه الحالة تتيح لغة C# التعامل مع أسلوب آخر في تمرير قيم هذه الوسطاء إلى الطريقة وهو أسلوب التمرير بالمرجع **Call by reference** ويخلص كما يلي:

- يتم في هذه الحالة تمرير مرجع المتغير (عنوان) الممرر كمعامل إلى الطريقة.
- يتأثر هذا المتغير بأي تعديلات تقوم بها الطريقة على المعامل في جسم الطريقة.
- يجب أن يكون المتغير الممرر مهيناً (له قيمة).

يقصد بالمرجع إلى متحول عنوان موقع الذاكرة الذي يحتله هذا المتحول، حيث يتم التعبير عن المرجع إلى متحول من خلال وضع الكلمة المفتاحية **ref** قبل التصريح عن هذا المتحول.

في حال الاستدعاء بالمرجع يتم تمرير عنوان الذاكرة الخاص بالوسيط إلى بارامتر الطريقة، وبالتالي هذا يكافئ حالة إعطاء الوسيط اسماً بديلاً **alias** هو اسم البرامتر.



يمكن تمثيل هذا الأمر من خلال تعديل المثال السابق بحيث يتم تمرير قيمة الوسيط إلى البارامتر بالمرجع كما يلي:

```
static void Main(string[] args)
{
    int aValue = 10;
    Console.WriteLine("aValue before call : " + aValue);
    int bValue = TestRefCall(ref aValue);
    Console.WriteLine("aValue after call : " + aValue);
}

public static int TestRefCall(ref int x)
{
    x++;
    Console.WriteLine("aValue during call : " + x);
    return x;
}
```

بتنفيذ هذا البرنامج يعطي على خرجه:

```
C:\WINDOWS\system32\cmd.exe
aValue before call :10
aValue during call :11
aValue after call :11
Press any key to continue . . .
```

نلاحظ أن قيمة المتحول aValue قد تغيرت بعد استدعاء الطريقة بحيث تعكس الزيادة الحاصلة على هذه القيمة داخل الطريقة، السبب في ذلك أنه عند الاستدعاء يتم إرسال مرجع إلى عنوان الوسيط aValue إلى البارامتر x كما يوضح الشكل التالي:

الذاكرة قبل استدعاء التابع	
aValue	10
bValue	
الذاكرة قبيل الخروج من التابع	
x,aValue	11
bValue	

الذاكرة لحظة استدعاء التابع	
x,aValue	10
bValue	
الذاكرة بعد استدعاء التابع	
aValue	11
bValue	11

يفضل الكثير من المبرمجين استخدام أسلوب الاستدعاء بالمرجع بدلاً من أسلوب إعادة القيمة في الحصول على نتيجة تنفيذ الطريقة (وخاصة عندما يكون هناك حاجة لأن تقوم الطريقة بحساب أكثر من قيمة ناتجة).

لأجل ذلك يمكن في لغة C# استخدام حالة خاصة من الاستدعاء بالمرجع وهي حالة استخدام معامل الخرج out. ويمكن تلخيص هذا الأمر كما يلي:

- يتم في هذه الحالة تمرير مرجع المتغير (عنوان) الممرر كمعامل إلى الطريقة.
- يتأثر هذا المتغير بأي تعديلات تقوم بها الطريقة على المعامل في جسم الطريقة.
- يمكن أن يكون المتغير الممرر غير مهئي (ليس له قيمة).

كمثال، ليكن لدينا البرنامج التالي الذي يقوم بحساب مساحة ومحيط المستطيل:

```
static void Main(string[] args)
{
    int L = 10, W = 15, C,S;
    TestOutCall(L,W,out C, out S);
    Console.WriteLine("C = " + C);
    Console.WriteLine("S = " + S);
}
public static void TestOutCall(int length,int width, out int circum, out int area)
{
    circum = 2 * (length + width);
    area = length * width;
}
```

بتنفيذ هذا البرنامج يعطي على خرجه:

```
C:\WINDOWS\system32\cmd.exe
C = 50
S = 150
Press any key to continue . . .
```

مثال: يبين المثال التالي مختلف الحالات السابقة

```
// uses reference parameter x to modify caller's variable
static void SquareRef(ref int x)
{
    x = x * x; // squares value of caller's variable
} // end method SquareRef

// uses output parameter x to assign a value to an uninitialized variable
static void SquareOut(out int x)
{
    x = 6; // assigns a value to caller's variable
    x = x * x; // squares value of caller's variable
} // end method SquareOut

// parameter x receives a copy of the value passed as an argument,
// so this method cannot modify the caller's variable
static void Square(int x)
{
    x = x * x;
} // end method Square

// call methods with reference, output and value parameters
public static void Main( string[] args )
{
    int y = 5; // initialize y to 5
    int z; // declares z, but does not initialize it
    // display original values of y and z
    Console.WriteLine( "Original value of y: {0}", y );
    Console.WriteLine( "Original value of z: uninitialized\n" );
    // pass y and z by reference
    SquareRef( ref y ); // must use keyword ref
    SquareOut( out z ); // must use keyword out
    // display values of y and z after they are modified by
    // methods SquareRef and SquareOut, respectively
    Console.WriteLine( "Value of y after SquareRef: {0}", y );
    Console.WriteLine( "Value of z after SquareOut: {0}\n", z );
    // pass y and z by value
    Square( y );
    Square( z );
    // display values of y and z after they are passed to method Square
    // to demonstrate arguments passed by value are not modified
    Console.WriteLine( "Value of y after Square: {0}", y );
    Console.WriteLine( "Value of z after Square: {0}", z );
} // end Main
```

يعطي هذا البرنامج على خرجه:

```
C:\WINDOWS\system32\cmd.exe
Original value of y: 5
Original value of z: uninitialized

Value of y after SquareRef: 25
Value of z after SquareOut: 36

Value of y after Square: 25
Value of z after Square: 36
Press any key to continue . . .
```

## 7- المصفوفة كبارامتر للطريقة

يمكن للمصفوفات أن تستخدم كبارامترات للطرائق، وفي هذه الحالة فإن تسلك سلوكاً مميزاً سواء أثناء عملية التصريح عن الطريقة أو أثناء عملية استدعاء الطريقة، نلخص هذا الأسلوب في النقاط التالية:

- عند تمرير مصفوفة كمعامل دخل لطريقة فإن التمرير يكون وفق المرجع بمعنى أن أي تعديل على عناصر المصفوفة ضمن الطريقة سيؤدي إلى تأثير عناصر المصفوفة الممررة.
- عند تمرير عنصر من المصفوفة كمعامل دخل لطريقة، يعامل هذا العنصر مثل أي متغير آخر أي يمكن أن يمرر بالقيمة أو بالمرجع أو أن يكون متغير خرج.
- نقوم في المثال التالي بالتصريح عن الطريقة ModifyArray والتي لها معامل دخل مصفوفة. نقوم في داخل الطريقة بالدوران على عناصر المصفوفة لضربهم بـ 2.
- لاحظ أنه عند استدعاء الطريقة السابقة وتمرير مصفوفة لها. ستتأثر عناصر المصفوفة الممررة بعد الاستدعاء (تضرب بـ 2).
- نقوم في المثال التالي بالتصريح عن الطريقة ModifyElement والتي لها معامل دخل صحيح. نقوم في داخل الطريقة بضرب المعامل الممرر بـ 2.
- لاحظ أنه عند استدعاء الطريقة السابقة وتمرير عنصر من المصفوفة كمعامل لها. لن يتأثر عنصر المصفوفة (معامل قيمة).

```
// Main creates array and calls ModifyArray and ModifyElement
public static void Main( string[] args )
{
    int[] array = { 1, 2, 3, 4, 5 };
    Console.WriteLine(
        "Effects of passing reference to entire array:\n" +
        "The values of the original array are:" );
    // output original array elements
    foreach ( int value in array )
        Console.Write( " {0}", value );
    ModifyArray( array ); // pass array reference
    Console.WriteLine( "\n\nThe values of the modified array are:" );
    // output modified array elements
    foreach ( int value in array )
        Console.Write( " {0}", value );
    Console.WriteLine(
        "\n\nEffects of passing array element value:\n" +
        "array[3] before ModifyElement: {0}", array[ 3 ] );
    ModifyElement( array[ 3 ] ); // attempt to modify array[ 3 ]
    Console.WriteLine( "array[3] after ModifyElement: {0}", array[ 3 ] );
} // end Main
// multiply each element of an array by 2
public static void ModifyArray( int[] array2 )
{
    for ( int counter = 0; counter < array2.Length; ++counter )
        array2[ counter ] *= 2;
} // end method ModifyArray
// multiply argument by 2
public static void ModifyElement( int element )
{
    element *= 2;
    Console.WriteLine("Value of element in ModifyElement: {0}", element );
} // end method ModifyElement
```

يعطي هذا البرنامج على خرجه:

```
C:\WINDOWS\system32\cmd.exe
Effects of passing reference to entire array:
The values of the original array are:
 1  2  3  4  5

The values of the modified array are:
 2  4  6  8 10

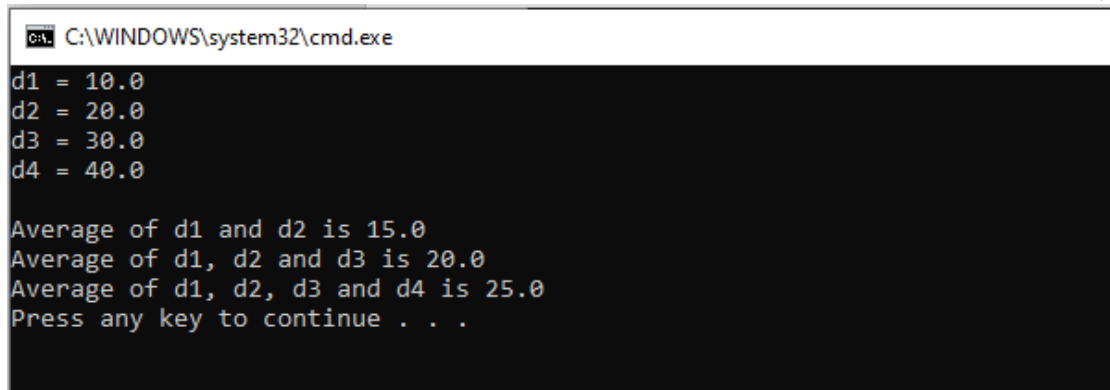
Effects of passing array element value:
array[3] before ModifyElement: 8
Value of element in ModifyElement: 16
array[3] after ModifyElement: 8
Press any key to continue . . .
```

## قائمة المعاملات متغيرة الطول

- تسمح قائمة المعاملات متغيرة الطول بالتصريح عن طرق يمكن أن يكون لها عدد متغير من المعاملات. يتم التصريح عن قائمة من المعاملات متغير الطول باستخدام مصفوفة أحادية كمعامل مسبقة بالكلمة المفتاحية .params
- نصح في المثال التالي عن الطريقة Average مع استخدام مصفوفة أحادية كمعامل دخل مع الكلمة المفتاحية .params
- لاحظ أننا نقوم باستدعاء هذه الطريقة مع عدد متغير من المعاملات في كل مرة.

```
// calculate average
public static double Average( params double[] numbers )
{
    double total = 0.0; // initialize total
    // calculate total using the foreach statement
    foreach ( double d in numbers )
        total += d;
    return total / numbers.Length;
} // end method Average
public static void Main( string[] args )
{
    double d1 = 10.0;
    double d2 = 20.0;
    double d3 = 30.0;
    double d4 = 40.0;
    Console.WriteLine(
        "d1 = {0:F1}\nd2 = {1:F1}\nd3 = {2:F1}\nd4 = {3:F1}\n",
        d1, d2, d3, d4 );
    Console.WriteLine( "Average of d1 and d2 is {0:F1}", Average( d1, d2 ) );
    Console.WriteLine( "Average of d1, d2 and d3 is {0:F1}", Average( d1, d2, d3 ) );
    Console.WriteLine( "Average of d1, d2, d3 and d4 is {0:F1}", Average( d1, d2, d3, d4 ));
} // end Main
```

يعطي هذا البرنامج على خرجه:



```
C:\WINDOWS\system32\cmd.exe
d1 = 10.0
d2 = 20.0
d3 = 30.0
d4 = 40.0

Average of d1 and d2 is 15.0
Average of d1, d2 and d3 is 20.0
Average of d1, d2, d3 and d4 is 25.0
Press any key to continue . . .
```

## 8- استخدام الوسطاء الافتراضية

رأينا في الفقرات السابقة، أشكالاً مختلفة لعمليات استدعاء الطرائق، تعتمد في مجملها على منطق واحد وهو " ذكر اسم الطريقة المستدعاة **called** ضمن الطريقة المستدعية **calling** مع تمرير قيم الوسطاء **arguments** إلى بارامترات الطريقة".

حيث أشرنا إلى أن قيم الوسطاء يجب أن ترد بنفس ترتيب البارامترات في تعريف الطريقة ونفس العدد ومن نفس النوع.

إن أي عملية استدعاء للطريقة تتضمن عدداً من الوسطاء لايتطابق مع عدد البارامترات في نموذج التعريف الطريقة تؤدي إلى توليد خطأ برمجي من قبل مترجم اللغة.

من الممكن في بعض مسائل العالم الحقيقي أن نواجه الحالات التالية:

- 1- بارامتر أو أكثر من بارامترات الطريقة يأخذ القيمة نفسها في أغلب عمليات الاستدعاء.
- 2- بارامتر أو أكثر من بارامترات الطريقة يأخذ قيمة ثابتة.
- 3- بارامتر أو أكثر لا يستخدم في بعض الحالات في تنفيذ العملية الحسابية أو المنطقية التي تقوم الطريقة بتنفيذها.

في جميع الحالات المذكورة، يمكن استخدام الأسلوب التقليدي السابق في الاستدعاء وذلك من خلال تمرير القيم للبارامترات خلال عملية الاستدعاء، إلا أن هذا الأمر قد ينطوي على ممارسة غير مرغوبة أو قد تكون متعبة ومملة وخاصة في حال كانت الطريقة تستدعي عدداً كبيراً من المرات أو كانت تتضمن عدداً كبيراً من البارامترات التي تتمتع بهذه الصفة.

أتاحت لغة C# تقنية أخرى في التعامل مع هذه الحالات، تقوم هذه التقنية على إعطاء قيم افتراضية للوسطاء خلال عملية التصريح عن الطريقة.

في هذه الحالة، في حال قيام الطريقة المستدعية بتمرير قيم صريحة لهذه الوسطاء أثناء عملية الاستدعاء، فإن الطريقة تستخدم هذه القيمة الصريحة في التنفيذ. أما في حال عدم تمرير قيمة صريحة لهذه الوسطاء فإن الطريقة تستخدم القيم الافتراضية الممررة أثناء التصريح لدى تنفيذها للطريقة.

إن هذا الأمر يسهل كثيراً على المبرمج عمليات الاستدعاء للطريقة فهي تتيح له أن لايقوم بتمرير قيم وسطاء لجميع البارامترات أثناء الاستدعاء، كما يمكن أن يقلل من احتمالات الخطأ المحتملة من خلال تمرير قيم غير مرغوبة.

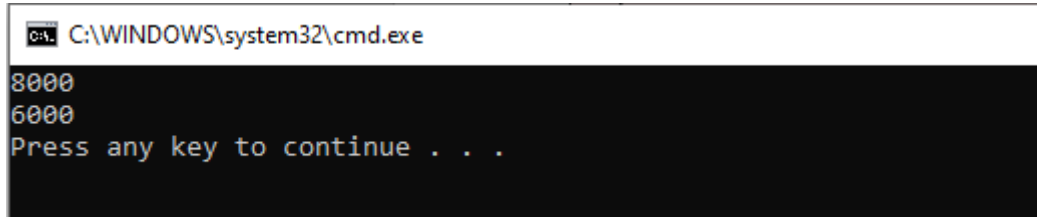
## مثال 1- حساب حجم متوازي المستطيلات

بفرض أن لدينا مجموعة من الكتل الإسمنتية بشكل متوازي مستطيلات أغلبها بارتفاع 40cm، وأن عامل بناء يقوم باستخدام هذه الكتل في بناء جدار ما، حيث يقوم بداية بحساب حجم الكتلة الاسمنتية التي بين يديه ليحدد في أي مكان من الجدار سيقوم بوضعها.  
نرغب بمساعدته في حساب حجم الكتلة الإسمنتية.

```
// calculate volume
public static int block_vol(int length, int width, int height = 40 )
{
    return length * width * height;
} // end method block_vol

public static void Main( string[] args )
{
    // length = 10, width=20, height=40
    Console.WriteLine(block_vol ( 10,20) );
    // length = 10, width=20, height=30 (not 40)
    Console.WriteLine(block_vol ( 10,20,30) );
} // end Main
```

يعطي هذا البرنامج على خرجه:



```
C:\WINDOWS\system32\cmd.exe
8000
6000
Press any key to continue . . .
```

بملاحظة المقطع البرمجي السابق والخرج المتولد نلاحظ أنه في الاستدعاء الأول للطريقة block\_vol، لم يتم تمرير قيمة للبارامتر height وبالتالي كانت قيمة الوسيط التي استخدمتها الطريقة لهذا البارامتر هي القيمة الافتراضية ( أي 40 )، في حين في الاستدعاء الثاني، ام تمرير قيمة ( وهي 30 ) وبالتالي حلت مكان القيمة الافتراضية وتم استخدام القيمة الصريحة الممررة في حساب القيمة المعادة للطريقة.

**ملاحظة 1:** إن ترتيب البارامترات في تعريف الطريقة مهم جداً في حال وجود قيم افتراضية لبعض الوسائط، فعلى سبيل المثال، لو تم تعريف الطريقة السابقة على النحو التالي:

```
public static int block_vol(int length, int height = 40, int width )
{
    return length * width * height;
}
```



فإن مترجم اللغة سيولد خطأ برمجياً، والسبب في ذلك، أنه في حال واجه المترجم استدعاء الطريقة على النحو block\_vol(10,20) فإن القيمة الأولى ستمرر للبارامتر الأول والقيمة الثانية ستمرر للبارامتر الثاني وبالتالي فإن عدد القيم الممررة لا يتطابق مع عدد البارامترات.

تجنباً لوقوع مثل هذا الخطأ، فإن لغة C# فرضت قيداً صارماً في مثل هذه الحالة ينص على أن البارامترات ذات القيم الافتراضية ( في حال وجودها ) يجب أن ترد في نهاية بارامترات الطريقة بحيث لو حذفت أثناء الاستدعاء لا يحصل التباس لدى المترجم في ترتيب القيم الممررة.

**ملاحظة 2-** من الممكن أن تحتوي الطريقة على أي عدد من البارامترات ذات القيم الافتراضية، فعلى سبيل المثال، في المثال السابق، بفرض أن أغلب الكتل الاسمنتية هي بطول 60cm وعرض 50cm وارتفاع 40cm عندئذ يمكن تعديل المثال السابق كما يلي:

```
// calculate volume
public static int block_vol(int length=60, int width=50, int height = 40 )
{
    return length * width * height;
} // end method block_vol

public static void Main( string[] args )
{
    // length = 60, width=50, height=40
    Console.WriteLine(block_vol ( ));
    // length = 30 (not 60), width=50, height=40
    Console.WriteLine(block_vol ( 30 ));
    // length = 30 (not 60), width=10 (not 50), height=40
    Console.WriteLine(block_vol ( 30,10 ));
    // length = 10 (not 60), width=20 (not 50), height=30 (not 40)
    Console.WriteLine(block_vol ( 10,20,30 ));
} // end Main
```

يجب الانتباه في هذه الحالة إلى أن القيم تمرر إلى البارامترات بالترتيب، وبالتالي، ففي حال عدم تمرير جميع القيم فإن البارامترات التي ستحصل على قيمتها الافتراضية هي البارامترات الأخيرة. فمثلاً الاستدعاء block\_vol ( 30,10) سيعطي القيم لـ length و width على الترتيب وسيستخدم height قيمته الافتراضية.

**السؤال :** ماذا لو رغبتنا بأن يكون width هو من سيستخدم قيمته الافتراضية، في هذه الحالات لابد من اللجوء إلى تسمية المعاملات أثناء الاستدعاء، كما يلي:

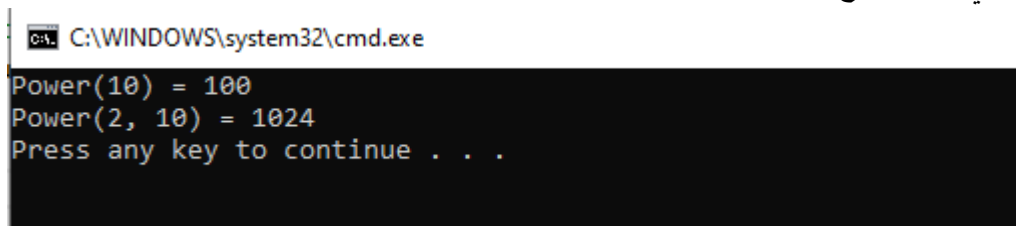
```
Console.WriteLine(block_vol ( length=>30, height=>10) );
```

## مثال 2- حساب رفع عدد إلى قوة صحيحة

نرغب بحساب ناتج رفع عدد إلى قوة بحيث تكون الحالة الأكثر شيوعاً هي حساب مربع العدد.

```
// call Power with and without optional arguments
public static void Main( string[] args )
{
    Console.WriteLine( "Power(10) = {0}", Power( 10 ) );
    Console.WriteLine( "Power(2, 10) = {0}", Power( 2, 10 ) );
} // end Main
// use iteration to calculate power
public static int Power( int baseValue, int exponentValue = 2 )
{
    int result = 1; // initialize total
    for ( int i = 1; i <= exponentValue; i++ )
        result *= baseValue;
    return result;
} // end method Power
```

يعطي هذا البرنامج على خرجه:



## 9- مجال الرؤية (التصريح)

يكون لكل متغير مجال (scope) أي مجموعة التعليمات التي يمكن لها أن تتعامل مع هذا المتغير.

تبين القواعد التالية أساسيات مجال التصريح:

- يكون مجال معاملات الطريقة جسم الطريقة نفسها فقط.
- يكون مجال متغير محلي (local variable) مُصرح عنه ضمن الطريقة كتلة التعليمات (block) التي تحوي التصريح.
- يكون مجال المتغيرات المُصرح عنها ضمن قسم التهيئة الابتدائية في التعليمات for جسم التعليمات for وأي تعابير مستخدمة في ترويضتها.
- يكون مجال الطريقة المُصرح عنها ضمن صف ما كامل جسم الصف هذا الصف.
- يُمكن للطريقة الساكنة (static) أن تتعامل فقط مع حقول الصف الساكنة.
- يُمكن لكل كتلة أن تُصرّح عن متغيراتها.

- يتم العودة للمتغير المُصرّح عنه في الكتلة الأقرب.

نصرّح في المثال التالي عن عدة متغيرات لها نفس الاسم (x):

- نصرّح أولاً في جسم الصف عن المتغير الساكن (x). يمكن لجميع طرق هذا الصف الوصول لهذا المتغير والتعامل معه.
- نصرّح في الطريقة الساكنة Main عن متغير له أيضاً الاسم (x). سيقوم هذا المتغير بحجب المتغير السابق الساكن في جسم الطريقة Main، بمعنى أن أي استخدام للمتغير (x) في جسم الطريقة Main سيعود إلى المتغير (x) المُصرّح عنه ضمن هذه الطريقة.
- نصرّح في الطريقة UseLocalVariable() عن المتغير المحلي (x). سيعود أي استخدام للمتغير (x) ضمن جسم هذه الطريقة إلى هذا المتغير. يعاد تهيئة هذا المتغير المحلي عند كل استدعاء للطريقة. بمعنى أنه في كل مرة نستدعي الطريقة UseLocalVariable() ستكون قيمة المتغير (x) الابتدائية 25.
- نستخدم في الطريقة UseStaticVariable() المتغير (x). وبما أننا لم نصرّح ضمن هذه الطريقة عن متغير باسم (x) سيعود استخدام (x) ضمن هذه الطريقة إلى المتغير (x) المصرّح عنه ضمن جسم الصف.

```
public class Scope
{
    // static variable that is accessible to all methods of this class
    private static int x = 1;
    // Main creates and initializes local variable x
    // and calls methods UseLocalVariable and UseStaticVariable
    public static void Main( string[] args )
    {
        int x = 5; // method's local variable x hides static variable x
        Console.WriteLine( "local x in method Main is {0}", x );
        // UseLocalVariable has its own local x
        UseLocalVariable();
        // UseStaticVariable uses class Scope's static variable x
        UseStaticVariable();
        // UseLocalVariable reinitializes its own local x
        UseLocalVariable();
        // class Scope's static variable x retains its value
        UseStaticVariable();
        Console.WriteLine( "\nlocal x in method Main is {0}", x );
    } // end Main
    // create and initialize local variable x during each call
    public static void UseLocalVariable()
    {
        int x = 25; // initialized each time UseLocalVariable is called
        Console.WriteLine( "\nlocal x on entering method UseLocalVariable is {0}", x );
        ++x; // modifies this method's local variable x
        Console.WriteLine( "local x before exiting method UseLocalVariable is {0}", x );
    }
}
```

```

} // end method UseLocalVariable
// modify class Scope's static variable x during each call
public static void UseStaticVariable()
{
    Console.WriteLine( "\nstatic variable x on entering {0} is {1}",
        "method UseStaticVariable", x );
    x *= 10; // modifies class Scope's static variable x
    Console.WriteLine( "static variable x before exiting {0} is {1}",
        "method UseStaticVariable", x );
} // end method UseStaticVariable

```

يعطي هذا البرنامج على خرجه:

```

C:\WINDOWS\system32\cmd.exe
local x in method Main is 5
local x on entering method UseLocalVariable is 25
local x before exiting method UseLocalVariable is 26

static variable x on entering method UseStaticVariable is 1
static variable x before exiting method UseStaticVariable is 10

local x on entering method UseLocalVariable is 25
local x before exiting method UseLocalVariable is 26

static variable x on entering method UseStaticVariable is 10
static variable x before exiting method UseStaticVariable is 100

local x in method Main is 5
Press any key to continue . . .

```

## 10- التحميل الزائد للطرائق overloading

يمكن في الصف الواحد التصريح عن عدة طرق بنفس الاسم طالما أن لكل منها مجموعة مختلفة من المعاملات (عدد المعاملات وأنماط المعاملات). تُدعى هذه الميزة بالتحميل الزائد للطرق **method overloading**.

عندما يتم استدعاء طريقة محملة، يقوم المترجم بانتقاء الطريقة الموافقة عن طريق فحص عدد المعاملات وأنماطها وترتيبها.

يتم عادةً استخدام التحميل الزائد عندما نحتاج لإنجاز نفس المهمة إنما على أنماط مختلفة. فمثلاً، يكون للطريقة Max في الصف Math حوالي 11 نسخة (تختلف بأنماط المعاملات).

نقوم في المثال التالي بالتصريح عن طريقتين لهما نفس الاسم Square. تقبل النسخة الأولى من الطريقة عدد صحيح وتعيد مربع هذا العدد كعدد صحيح. بينما تقبل النسخة الثانية عدد عشري وتعيد مربع هذا العدد (عدد عشري).

```
// test overloaded square methods
public static void Main( string[] args )
{
    Console.WriteLine( "Square of integer 7 is {0}", Square( 7 ) );
    Console.WriteLine( "Square of double 7.5 is {0}", Square( 7.5 ) );
} // end Main
// square method with int argument
public static int Square( int intValue )
{
    Console.WriteLine( "Called square with int argument: {0}", intValue );
    return intValue * intValue;
} // end method Square with int argument
// square method with double argument
public static double Square( double doubleValue )
{
    Console.WriteLine( "Called square with double argument: {0}", doubleValue );
    return doubleValue * doubleValue;
} // end method Square with double argument
```

يعطي هذا البرنامج على خرجه:

```
C:\WINDOWS\system32\cmd.exe
Called square with int argument: 7
Square of integer 7 is 49
Called square with double argument: 7.5
Square of double 7.5 is 56.25
Press any key to continue . . .
```

**ملاحظة:** لا يمكن التفريق بين الطرق عن طريق نمط القيمة المعادة فقط. بمعنى أنه لا يسمح بتعريف طريقتين لهما نفس الاسم ونفس المعاملات وإنما تختلفان بالقيمة المعادة

في المثال التالي، يعطي المترجم رسالة خطأ تخبرنا بوجود طريقة تحمل نفس الاسم ونفس عدد البارامترات ونوعها:

```
// declaration of method Square with int argument
public static int Square( int x )
{
    return x * x;
} // end method Square
// second declaration of method Square with int argument causes compilation error
public static double Square( int y )
{
    return y * y;
} // end method Square
```

## 11- الطرائق العودية recursion

تضمنت جميع الأمثلة حول استدعاءات الطرائق التي رأيناها حتى الآن طريقة  $f()$  تقوم باستدعاء طريقة أخرى  $g()$  (حيث يكون استدعاء الطريقة  $f()$  غالباً في الطريقة Main)، في حين أن، طريقة  $f()$  يمكن أيضاً أن تقوم باستدعاء نفسها، وهذه الظاهرة تدعى العودية recursion.

نقول عن التابع بأنه عودي recursive إذا كان تعريفه يتضمن مكونين أساسيين:  
**المكون الأول:** هو القيمة البدائية anchor للتابع، ويقصد بها القيمة أو القيم التي يأخذها في حالته الابتدائية.  
**المكون الثاني:** هو الخطوة الدورية induction (أو الخطوة العودية recursive step) وهي التعبير عن قيمة للتابع بدلالة قيمة (أو قيم) سابقة له.

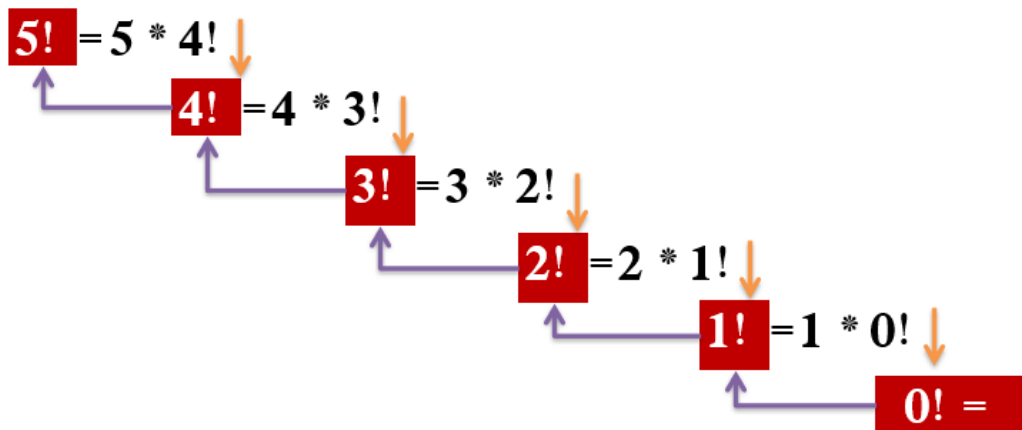
**مثال 1-** من المسائل الشهيرة حول التوابع العودية هي مسألة عاملي عدد صحيح، سنقوم بتوضيحها فيما يلي:

$$n! = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ 1 \times 2 \times 3 \times \dots \times n & \text{if } n > 0 \end{cases}$$

فمثلاً:

$$\begin{aligned} 0! &= 1 \\ 1! &= 1 \\ 2! &= 1 \times 2 = 2 \\ 3! &= 1 \times 2 \times 3 = 6 \\ 4! &= 1 \times 2 \times 3 \times 4 = 24 \\ 5! &= 1 \times 2 \times 3 \times 4 \times 5 = 120 \end{aligned}$$

وبالتالي فالقيمة الابتدائية هي  $0! = 1$  أما الخطوة العودية فهي  $n! = n * (n-1)!$  يمثل الشكل التالي آلية حساب  $5!$  بشكل عودي

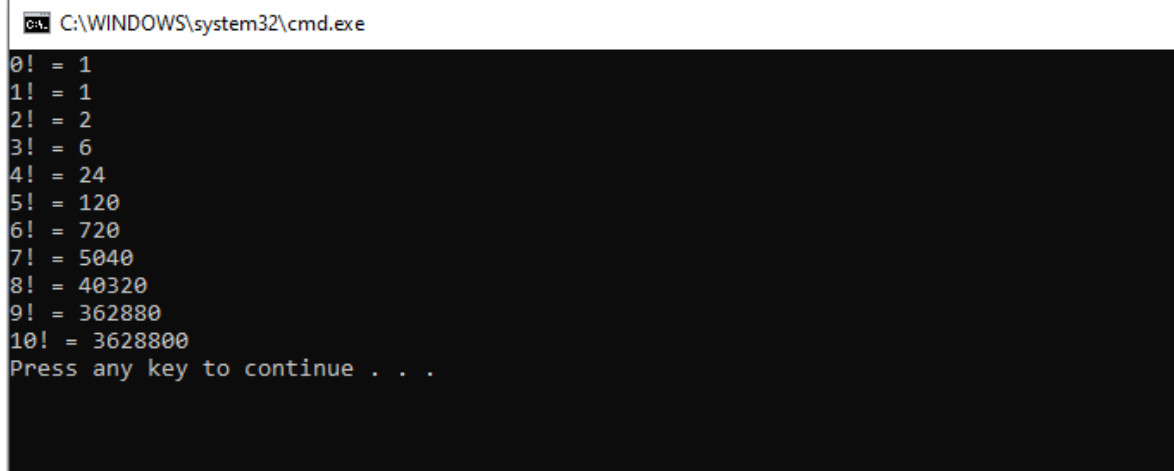


يبين المقطع البرمجي التالي كيفية كتابة طريقة عودية لحساب عاملي عدد:

```
public static long Factorial(long number)
{
    // base case
    if (number <= 1)
        return 1;
    // recursion step
    else
        return number * Factorial(number - 1);
} // end method Factorial

public static void Main( string[] args )
{
    // calculate the factorials of 0 through 10
    for ( long counter = 0; counter <= 10; ++counter )
        Console.WriteLine( "{0}! = {1}",counter, Factorial( counter ) );
} // end Main
```

يعطي هذا المقطع على خرجه:

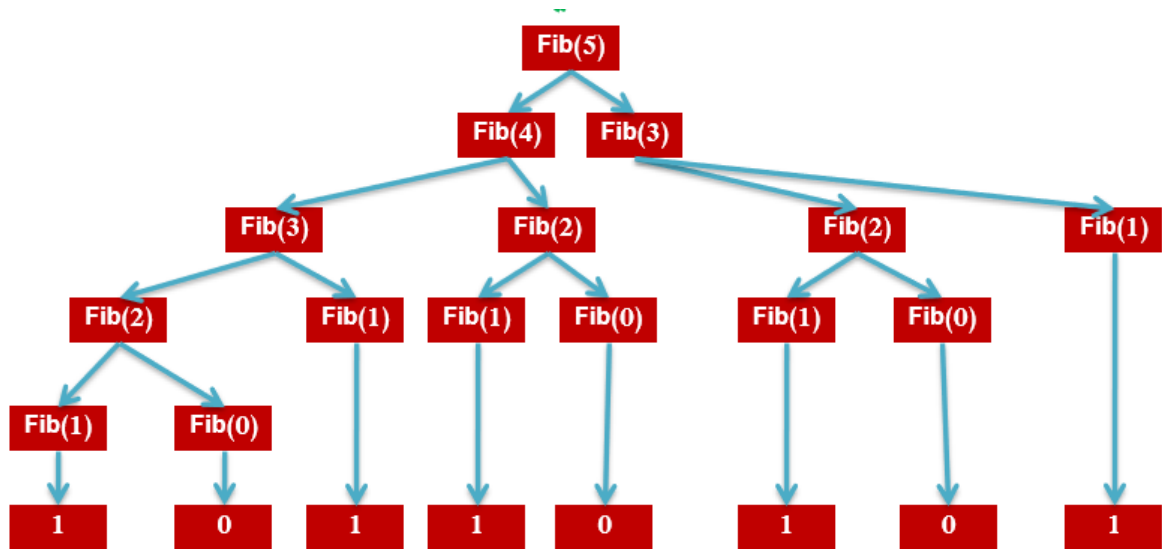


```
C:\WINDOWS\system32\cmd.exe
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
Press any key to continue . . .
```

مثال 2- أيضاً من المسائل الشهيرة حول التوابع العودية هي مسألة حساب حدود متتالية فيبوناتشي Fibonacci series والتي تعرف كما يلي:

$$\text{Fib}(N) = \begin{cases} N & \text{if } N \text{ is 0 or } N \text{ is 1} \\ \text{Fib}(N-1) + \text{Fib}(N-2) & \text{if } N > 1 \end{cases}$$

يوضح الشكل التالي أليه حساب الحد 5 من السلسلة بشكل عودي:



يبين المقطع البرمجي التالي كيفية كتابة طريقة عودية لحساب حدود سلسلة فيبوناتشي:

```

public static int Fib (int number)
{
    // base case
    if (number == 0 || number == 1)
        return number;
    // recursion step
    else
        return Fib (number - 1) + Fib (number-2);
} // end method Factorial

public static void Main( string[] args )
{
    // calculate the factorials of 0 through 10
    for ( int counter = 0; counter <= 20; ++counter )
        Console.WriteLine( "{0} ", Fib ( counter ) );
    Console.WriteLine();
} // end Main

```

ناتج التنفيذ:

```

C:\WINDOWS\system32\cmd.exe
0 1 1 2 3 5 8 13 21 34 55
Press any key to continue . . .

```



## 12- أمثلة حول الطرائق

### المثال الأول:

اكتب بلغة C# برنامجاً لحساب القاسم المشترك الأعظم لعددين صحيحين بطريقتين مختلفتين:

1- طريقة خوارزمية أقليدس إسناد قيمة باقي القسمة الأكبر على الأصغر إلى العدد الأكبر وصولاً إلى الصفر.

2- طريقة خوارزمية الطرح المتتالي للأصغر من الأكبر وصولاً إلى تساوي العددين.

```
static void Main(string[] args)
{
    Console.Write("First Number : ");
    int p = Int32.Parse(Console.ReadLine());
    Console.Write("Second Number : ");
    int q = Int32.Parse(Console.ReadLine());
    if (p * q != 0)
    {
        Console.WriteLine("mgcd of " + p + " and " + q + " = " + mgcd1(p, q));
        Console.WriteLine("mgcd of " + p + " and " + q + " = " + mgcd2(p, q));
    }
    else
        Console.WriteLine("One of the numbers is null !");
}

static int mgcd1(int a, int b)
{
    int r;
    while ((a != 0) && (b != 0))
    {
        if (a > b)
            a = a % b;
        else
            b = b % a;
    };
    r = (a != 0) ? a : b; return r;
}

static int mgcd2 (int p, int q)
{
    while ( p != q)
    {
        if (p > q)
            p -= q;
        else
            q -= p;
    }
    return p;
}
```

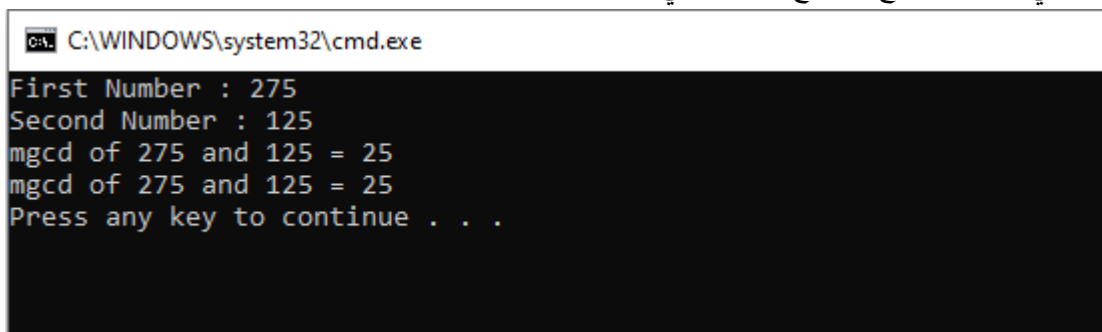
فيما يلي إعادة حل نفس المسألة السابقة ولكن باستخدام الطرائق العودية:

```
static void Main(string[] args)
{
    Console.Write("First Number : ");
    int p = Int32.Parse(Console.ReadLine());
    Console.Write("Second Number : ");
    int q = Int32.Parse(Console.ReadLine());
    if (p * q != 0)
    {
        Console.WriteLine("mgcd of " + p + " and " + q + " = " + mgcd1(p, q));
        Console.WriteLine("mgcd of " + p + " and " + q + " = " + mgcd2(p, q));
    }
    else
        Console.WriteLine("One of the numbers is null !");
}

static int mgcd1(int a, int b)
{
    if (a % b == 0)
        return b;
    else if (b % a == 0)
        return a;
    else if (a > b)
        return mgcd1(a % b, b);
    else
        return mgcd1(a, b % a);
}

static int mgcd2 (int p, int q)
{
    if (p == q)
        return p;
    else if (p > q)
        return mgcd2(p-q,q);
    else
        return mgcd2(p,q-p);
}
```

فيما يلي عينة عن خرج البرنامج وهو ذاته في الحالتين:



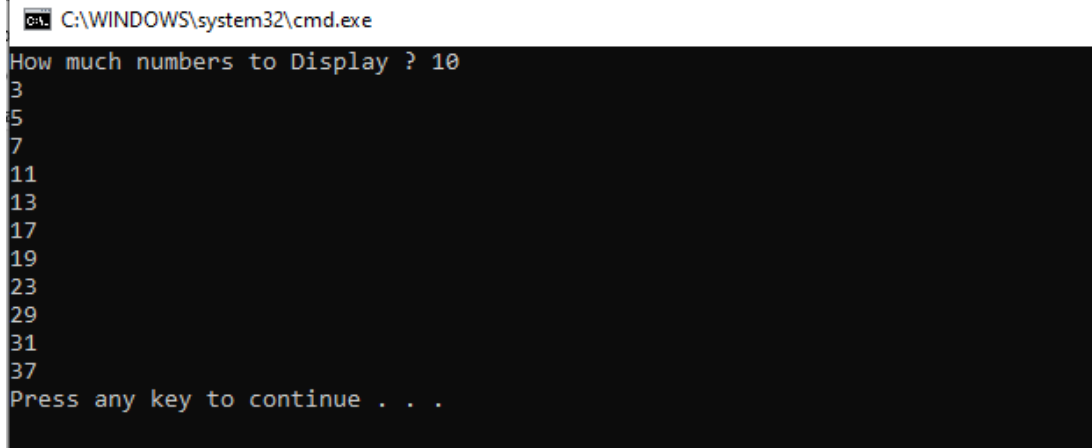
```
C:\WINDOWS\system32\cmd.exe
First Number : 275
Second Number : 125
mgcd of 275 and 125 = 25
mgcd of 275 and 125 = 25
Press any key to continue . . .
```

## المثال الثاني:

اكتب بلغة C# برنامجاً لإظهار أول n عدد أولي من مجموعة الأعداد الصحيحة الموجبة.

```
static void Main(string[] args)
{
    int n,count=0,number=3 ;
    Console.WriteLine("How much numbers to Display ? ");
    n = Int32.Parse(Console.ReadLine());
    while (count <= n)
    {
        if (is_perm(number) == true)
        {
            Console.WriteLine(number);
            count++;
        }
        number++;
    }
}
public static bool is_perm(int x)
{
    bool result = true;
    int i = 2;
    do
    {
        if (x % i == 0)
            result = false;
        else
            i++;
    }
    while ((i <= x / 2) && (result == true));
    return result;
}
```

يكون خرج البرنامج من الشكل:



```
C:\WINDOWS\system32\cmd.exe
How much numbers to Display ? 10
3
5
7
11
13
17
19
23
29
31
Press any key to continue . . .
```

### المثال الثالث:

اكتب بلغة C# برنامجاً للتحقق من أن سلسلة محارف تمتلك صفة "التناظر" PALINDROME، أي أنها تبقى نفسها سواء قرأناها من اليمين إلى اليسار أو من اليسار إلى اليمين.  
مثال: acca، abcd dcba

```
static String inverse(string s)
{
    string r = "";
    int L = s.Length;
    for (int i = 0; i <= L - 1; i++)
        r = r + s[L - 1 - i];
    return r;
}

public static void Main(String[] args)
{
    string s;

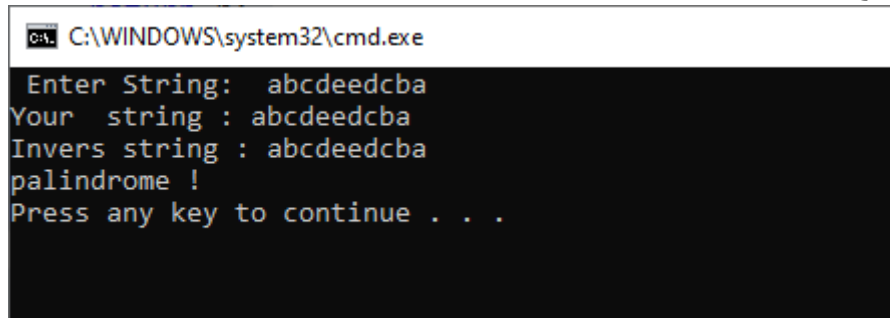
    Console.Write(" Enter String: ");
    s = Console.ReadLine();

    string invs = inverse(s);

    Console.WriteLine("Your string : " + s);
    Console.WriteLine("Invers string : " + invs);

    if (s == invs)
        Console.WriteLine("palindrome !");
    else
        Console.WriteLine("Not palindrome !");
}
```

يكون خرج البرنامج من الشكل:



```
C:\WINDOWS\system32\cmd.exe
Enter String: abcdeedcba
Your string : abcdeedcba
Invers string : abcdeedcba
palindrome !
Press any key to continue . . .
```

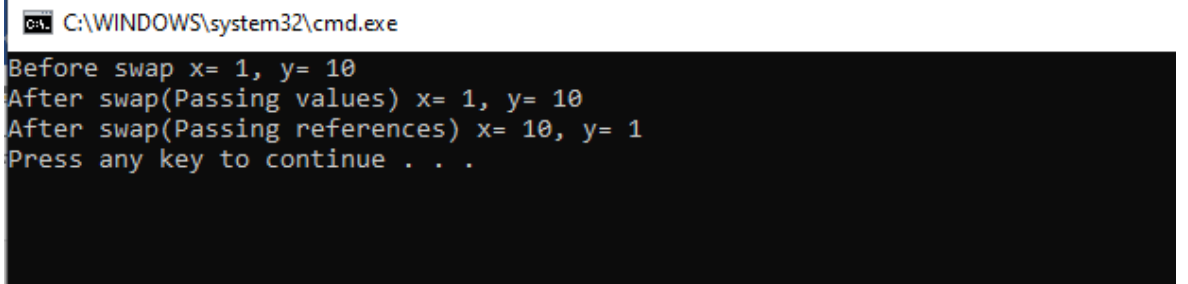
## المثال الرابع:

كتابة طريقة لمبادلة قيمتي عددين ممررين.

ملاحظة: استخدم مفهوم التحميل الزائد ومفهوم الإستدعاء بالمرجع لكتابة نسختين من هذا التابع مع ملاحظة الفرق.

```
static void swap ( int n, int m)
{
    int k;
    k=n;
    n=m;
    m=k;
}
static void swap(ref int n, ref int m)
{
    int k;
    k=n;
    n=m;
    m=k;
}
public static void Main( string[] args )
{
    int x = 1, y = 10;
    Console.WriteLine("Before swap x= {0}, y= {1}", x, y);
    swap(x, y);
    Console.WriteLine("After swap(Passing values) x= {0}, y= {1}", x, y);
    swap(ref x, ref y);
    Console.WriteLine("After swap(Passing references) x= {0}, y= {1}", x, y);
}
```

يكون خرج البرنامج من الشكل:



```
C:\WINDOWS\system32\cmd.exe
Before swap x= 1, y= 10
After swap(Passing values) x= 1, y= 10
After swap(Passing references) x= 10, y= 1
Press any key to continue . . .
```

### المثال الخامس:

كتابة طريقة لإيجاد العدد الأكبر والعدد الأصغر والوسطي لقيمتين.

```
static void MaxMinAvg(double x, double y, out double max, out double min, out
double avg)
{
    if (x > y)
    {
        max = x;
        min = y;
    }
    else
    {
        max = y;
        min = x;
    }
    avg = (x + y) / 2;
}
public static void Main( string[] args )
{
    double a = 10;
    double b = 20;
    double mx, mn, av;
    MaxMinAvg(a, b, out mx, out mn, out av);
    Console.WriteLine("Max= {0}, Min= {1}, Avg={2} ", mx, mn, av);
}
```

يكون خرج البرنامج من الشكل:



```
cmd C:\WINDOWS\system32\cmd.exe
Max= 20, Min= 10, Avg=15
Press any key to continue . . .
```

## المثال السادس:

نُخزّن في المثال التالي نتائج عشرة طلاب في ثلاثة امتحانات وذلك باستخدام مصفوفة مستطيلة.

- تقوم الطريقة OutputGrades بطباعة عناصر مصفوفة مستطيلة (معامل الطريقة مصفوفة مستطيلة).
- تقوم الطريقة GetMinimum بإيجاد أصغر قيمة في مصفوفة مستطيلة (معامل الطريقة مصفوفة مستطيلة).
- تقوم الطريقة GetMaximum بإيجاد أكبر قيمة في مصفوفة مستطيلة (معامل الطريقة مصفوفة مستطيلة).
- تقوم الطريقة GetAverage بإيجاد وسطي علامات طالب (للطريقة معاملين: الأول فهرس الطالب والثاني مصفوفة مستطيلة).
- تقوم الطريقة OutputBarChart بعدّ العلامات في 11 مجال (العلامات المساوية إلى 100، العلامات بين 90 إلى 99، . . . ، العلامات بين 0 و9) ومن ثم إظهار عدد من النجوم لكل مجال وبحيث يكون عدد النجوم مساوي للعلامات المحصورة ضمن المجال.
- تقوم الطريقة ProcessGrades باستدعاء كل من OutputGrades و OutputBarChart.

```
static void ProcessGrades(int[,] grades)
{
    // output grades array
    OutputGrades(grades);
    // call methods GetMinimum and GetMaximum
    Console.WriteLine("\n{0} {1}\n{2} {3}\n",
        "Lowest grade in the grade book is", GetMinimum(grades),
        "Highest grade in the grade book is", GetMaximum(grades));
    // output grade distribution chart of all grades on all tests
    OutputBarChart(grades);
} // end method ProcessGrades

// find minimum grade
static int GetMinimum(int[,] grades)
{
    // assume first element of grades array is smallest
    int lowGrade = grades[0, 0];
    // loop through elements of rectangular grades array
    foreach (int grade in grades)
    {
        // if grade less than lowGrade, assign it to lowGrade
        if (grade < lowGrade)
            lowGrade = grade;
    } // end foreach
    return lowGrade; // return lowest grade
} // end method GetMinimum
```

```

// find maximum grade
static int GetMaximum(int[,] grades)
{
    // assume first element of grades array is largest
    int highGrade = grades[0, 0];
    // loop through elements of rectangular grades array
    foreach (int grade in grades)
    {
        // if grade greater than highGrade, assign it to highGrade
        if (grade > highGrade)
            highGrade = grade;
    } // end foreach
    return highGrade; // return highest grade
} // end method GetMaximum

// determine average grade for particular student
static double GetAverage(int student, int[,] grades)
{
    // get the number of grades per student
    int amount = grades.GetLength(1);
    int total = 0; // initialize total
    // sum grades for one student
    for (int exam = 0; exam < amount; ++exam)
        total += grades[student, exam];
    // return average of grades
    return (double)total / amount;
} // end method GetAverage

// output bar chart displaying overall grade distribution
static void OutputBarChart(int[,] grades)
{
    Console.WriteLine("Overall grade distribution:");
    // stores frequency of grades in each range of 10 grades
    int[] frequency = new int[11];
    // for each grade in GradeBook, increment the appropriate frequency
    foreach (int grade in grades)
    {
        ++frequency[grade / 10];
    } // end foreach

    // for each grade frequency, display bar in chart
    for (int count = 0; count < frequency.Length; ++count)
    {
        // output bar label ( "00-09: ", ..., "90-99: ", "100: " )
        if (count == 10)
            Console.Write(" 100: ");
        else

```



```

        Console.WriteLine("{0:D2}-{1:D2}: ", count * 10, count * 10 + 9);
        // display bar of asterisks
        for (int stars = 0; stars < frequency[count]; ++stars)
            Console.WriteLine("*");
        Console.WriteLine(); // start a new line of output
    } // end outer for
} // end method OutputBarChart

// output the contents of the grades array
static void OutputGrades(int[,] grades)
{
    Console.WriteLine("The grades are:\n");
    Console.WriteLine(" "); // align column heads
    // create a column heading for each of the tests
    for (int test = 0; test < grades.GetLength(1); ++test)
        Console.WriteLine("Test {0} ", test + 1);
    Console.WriteLine("Average"); // student average column heading
    // create rows/columns of text representing array grades
    for (int student = 0; student < grades.GetLength(0); ++student)
    {
        Console.WriteLine("Student {0,2}", student + 1);
        // output student's grades
        for (int grade = 0; grade < grades.GetLength(1); ++grade)
            Console.WriteLine("{0,8}", grades[student, grade]);
        // call method GetAverage to calculate student's average grade;
        // pass row number as the argument to GetAverage
        Console.WriteLine("{0,9:F}", GetAverage(student, grades));
    } // end outer for
} // end method OutputGrades

// Main method begins application execution
public static void Main( string[] args )
{
    // rectangular array of student grades
    int[ , ] gradesArray = { { 87, 96, 70 },
        { 68, 87, 90 },
        { 94, 100, 90 },
        { 100, 81, 82 },
        { 83, 65, 85 },
        { 78, 87, 65 },
        { 85, 75, 83 },
        { 91, 94, 100 },
        { 76, 72, 84 },
        { 87, 93, 73 } };
    ProcessGrades(gradesArray);
} // end Main

```

يكون خرج البرنامج من الشكل:

```
C:\WINDOWS\system32\cmd.exe
The grades are:

Test 1 Test 2 Test 3 Average
Student 1      87      96      70      84.33
Student 2      68      87      90      81.67
Student 3      94     100      90      94.67
Student 4     100      81      82      87.67
Student 5      83      65      85      77.67
Student 6      78      87      65      76.67
Student 7      85      75      83      81.00
Student 8      91      94     100      95.00
Student 9      76      72      84      77.33
Student 10     87      93      73      84.33

Lowest grade in the grade book is 65
Highest grade in the grade book is 100

Overall grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: ***
70-79: *****
80-89: *****
90-99: *****
100: ***
Press any key to continue . . .
```

## المثال السابع:

نقول عن عدد صحيح أنه تام perfect number إذا كان مجموع قواسمه factors بما فيها العدد واحد (ما عدا العدد نفسه) يساوي العدد نفسه، على سبيل المثال العدد 6 عدد تام لأن  $6=1+2+3$  أي أن العدد التام يساوي مجموع قواسمه الأصغر منه. والمطلوب:

أكتب واختبر الطريقة perfect التي تحدد فيما إذا كان العدد المرسل إليها تاماً أم لا واستخدمها لطباعة كافة الأعداد التامة المحصورة بين العددين 1 - 1000 أطبع أيضاً قواسم كل عدد تام.

```
public static void factors(int m,int [] a,out int n)
{
    n=0;
    for(int i=1;i<=m/2;i++)
        if( m%i == 0)
        {
            a[n]=i;
            n++;
        }
}

public static int perfect(int m,int [] a,int n)
{
    int sum=0;
    for(int i=0;i<n;i++)
        sum=sum+a[i];
    if(sum == m)
        return 1;
    else
        return 0;
}

public static void printfactors(int[] a, int n)
{
    for(int i=0;i<n;i++)
        Console.Write(a[i]+" ");
    Console.WriteLine();
}

public static void Main(string[] args)
{
    int[] a = new int[100];
    int n;
    for (int i = 1; i <= 1000; i++)
    {
        factors(i, a, out n);
```

```

int f = perfect(i, a, n);
if (f == 1)
{
    Console.WriteLine(i + " is perfect and its factors are :");
    printfactors(a, n);
}
}
}

```

يكون خرج البرنامج من الشكل:

```

C:\ Select C:\WINDOWS\system32\cmd.exe
6 is perfect and its factors are :1 2 3
28 is perfect and its factors are :1 2 4 7 14
496 is perfect and its factors are :1 2 4 8 16 31 62 124 248
Press any key to continue . . .

```

## 13- تمارين وأنشطة

### التمرين الأول

قم بكتابة الطريقة LCM لحساب المضاعف المشترك البسيط لعددتين صحيحين ثم استخدمه في تطبيق يطلب من المستخدم إدخال عددين صحيحين ثم يقوم بإظهار المضاعف المشترك البسيط لهما. ملاحظة: يمكنك استخدام الطريقة التالية لحساب المضاعف المشترك البسيط: كرر جمع القيمة الأولية للعدد الأصغر حتى يُصبح العددين متساويين.

مثال:

24	36
48	36
72	72

### التمرين الثاني

قم بكتابة عدة نسخ من الطريقة Average بحيث تقبل الطريقة ثلاثة أعداد من النمط int أو long أو double وتُعيد الوسطي الحسابي لها.

### التمرين الثالث

عدّل الطرق السابقة لحساب الوسطي الحسابي. بحيث يُمكن تمرير معامل واحد أو معاملين أو ثلاثة معاملات للطريقة.

### التمرين الرابع

قم بكتابة طريقة تقبل عدد صحيح كمعامل دخل وتُعيد معكوس هذا العدد. مثلاً، إذا كان العدد 7631 فإن الطريقة تُعيد 1367.

### التمرين الخامس

قم بكتابة طريقة عودية لحساب القوة  $n$  لعدد  $x$ . مع ملاحظة:

$$\text{Power}(x, 0) = 1$$

$$\text{Power}(x, n) = x * \text{Power}(x, n-1)$$

### التمرين السادس

قم بكتابة طريقة تُمرر لها مصفوفة أحادية من الأعداد فتقوم بحساب أكبر قيمة، أصغر قيمة، وسطي القيم (استخدم معاملات خرج out). قم باختبار هذه الطريقة في تطبيق.

### التمرين السابع

قم بكتابة طريقة تُمرر لها مصفوفة أحادية من الأعداد فتقوم بإيجاد العنصر ذو التواتر الأكبر في المصفوفة وعدد مرات تواتره (استخدم معاملات خرج out). قم باختبار هذه الطريقة في تطبيق.

### التمرين الثامن

استخدم مصفوفة أحادية لحل مشكلة التكرار التالية: اكتب تطبيق يقوم بقراءة 10 قيم مدخلة من قبل المستخدم ومن ثم طباعة القيم غير المكررة فقط.

### التمرين التاسع

نقول عن عددين أنهما صديقان إذا كان مجموع قواسم الأول يساوي الثاني ومجموع قواسم الثاني يساوي الأول. أكتب تابعاً يقوم باختبار عددين ممررين إليه فيما إذا كانا صديقين أم لا. اختبر التابع في طباعة ثنائيات الأعداد الصديقة المحصورة بين 1 و 100.