مقدمة للبرمجة بالسي شارب

محمود سليمان

مقدمة للبرمجة
بالسي شارب

العالمية للنشر والتوزيع

الكتاب : مقدمة للبرمجة بالسي شارب

المؤلف : محمود سليمان

تصميم الغلاف : تقى عبدالحميد

الطبعة الأولى : أبريل ٢٠١٨

رقم الإيداع : 8345 / 2018

الترقيم الدولي : 978-977-805-066-0

مقدمة للبرمجة
بالسي شارب

محمود سليمان

المحتوى :

عن الكتاب

هذا الكتاب مُعَدّ للمبتدئين لكي يساعدهم على فهم أساسيات لغة السي شارب بطريقة مبسطّة وسهلة، ولا يشترط أن يكون القاريء له أي معرفة مسبقة بالبرمجة، وأهم خطوة بجانب القراءة هي التطبيق العملي المستمر لتحصل على أكبر قدر من الفهم والإستيعاب والتحصيل.

عن اللغة

لغة السي شارب لغة بسيطة جداً وسهلة وحديثة ولها أغراض كثيرة، فهي طُوّرِتْ بواسطة شركة مايكروسوفت (Microsoft) بقيادة أندريس هيلسبرج، اللغة تُمكِنك من عمل تطبيقات للويب والهاتف وسطح المكتب، وتعمل شركة مايكروسوفت على تطويرها دائما، وهي واحدة من أشهر لغات البرمجة وأكثرهم إستخداماً.

بدأوا في تطوير اللغة عام ١٩٩٩ ، وأطلقوا عليها لغة كوول أي :

COOL ----> C Object Oriented Language

ثم غيروا الاسم لسي شارب وأعلنوا عنها عام ٢٠٠٠.

ولغة سي شارب (C#) هي تطوير وإمتداد للغة C++ و C، وهي تعمل بالبرمجة كائنية التوجه بشكل كلي.

إذا اكتشفت أي خطأ في هذا الكتاب رجاءً اعلمني وراسلني على البريد الإلكتروني

Mahmoudsoliman781@yahoo.com

1

مقدمة

إن سي شارب إحدى لغات البرمجة المصممة للعمل على البنية التحتية المشتركة للغات البرمجة تسمى

(Common Language Infrastructure) ←(CLI)

معنى هذا المصطلح أنها تُتيح للمبرمج إمكانية إستخدامها لإنتاج مكتبات تتوافق مع المواصفات والخصائص الشائعة للبنية التحتية أي انه يحتوي على الكود التنفيذي الذي يتيح له استخدام شتى لغات البرمجة عالية المستوى علي منصات عمل مختلفة.

التسلسل الهرمي للبيانات

Bits- هي اصغر وحدة قياس في الكمبيوتر وهي تأخذ قيمتين فقط إما 0 أو 1.

تُعبِرّ الأصفار والآحاد عن إشارات كهربائية، وهي ما يعمل بها الكمبيوتر.

Characters - وهي تتضمن الحروف وتساوي ٢بايت حيث البايت يساوي ٨ بت

ثم تتحول الحروف إلى أصفار وآحاد (لغة الآلة) لكي يفهمها الكمبيوتر ويتم ذلك بـ :

ASCII→(American Standard Code for Information Interchange)

فكل حرف من (A إلى Z) وكل رقم من (0 إلى 9) وكل رمز في الكمبيوتر مثل & ^ $ # @ له عدد عشري يناظره ثم يتحول إلى أصفار وآحاد في النهاية لكي يفهمه الكمبيوتر .

ASCII

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

١- بت = (٠ \ ١)

٢- بايت = ٨ بت

٣- كيلو بايت = ١٠٢٤ بايت

٤- ميجا بايت = ١٠٢٤ كيلو بايت

٥- جيجا بايت = ١٠٢٤ ميجا بايت

٦- تيرا بايت = ١٠٢٤ جيجا بايت

٧- بيتا بايت = ١٠٢٤ تيرا بايت

٨- إكسا بايت = ١٠٢٤ بيتا بايت

٩- زيتا بايت = ١٠٢٤ إكسا بايت

١٠- يوتا بايت = ١٠٢٤ زيتا بايت

Fields (حقول) - تتكون من حروف او أرقام فمثلا تُعبِرّعن اسم الشخص أو عمره.

Records (سجل) - يحتوي على مجموعة من الحقول معاً.

Files (ملفات) - تتكون من مجموعة من السجلات.

Database (قواعد البيانات) - عبارة عن مجموعة من البيانات وقد تكون بيانات ضخمة لكن يسهل الوصول إليها والتعامل معها بسرعة وتتكون من حقول وسجلات وهذه البيانات تُخزَن في جداول بسيطة لها اسم وخصائص.

Big Data (بيانات ضخمة) - كم هائل من البيانات التي تُنتَج في العالم يومياً.

2

إطار العمل

(The .Net Framework)

الـ (دوت نت) Net. هي إطار عمل أو منصة لتنفيذ التطبيقات وتحتوي على مكتبات توفر العديد من الامكانيات التي تستخدم لبناء تطبيقات كبيرة بالسي شارب سواء تطبيقات للويب أو لسطح المكتب أو الهاتف سواء (android ، ios ، windows phone) وذلك من خلال **xamarin** وهي عبارة عن منصة لتطوير تطبيقات الهواتف المحمولة حيث يعمل التطبيق الذي أُنشيء بواسطتها على كل أنظمة تشغيل الهواتف، الـ Net. صُمِمَت لتكون منصة عمل لمختلف التطبيقات، تستطيع إستخدامها في بناء تطبيقات من خلال اللغات الآتية :

C# , C++ , Visual Basic , Python and JavaScript

وهي تحتوي على كل المكتبات المبنية في هذه اللغات وبها كل الموارد التي تمكنك من إنشاء تطبيق من خلالها.

محتويات إطار عمل (Net.) :

CLR (Common Language Runtime) - هي عبارة عن آلة إفتراضية مسئولة عن إدارة الكود مع الذاكرة المؤقتة (RAM) والمعالج (processor) تُحول الكود إلى لغة الآلة، بعدها هذا الكود يستطيع العمل على أي لغة برمجة اخرى، وهو يدير الذاكرة أيضا ويعالج بها الاخطاء ويحتوي على " جامع النفايات ".

GC (Garbage Collector) - يجمع البيانات المهملة والاماكن الغير مستخدمة في الذاكرة ويقوم بحذفها وتنقية الذاكرة فيجعلها تعمل بكفاءة أكثر.

رحلة الكود البرمجي داخل الـ Net. :

أولاً يُعالج الكود فيتحول إلى لغة وسيطة

(MSIL→Microsoft Intermediate Language)

لأن السي شارب من اللغات عالية المستوى فتحتاج إلى لغة وسيطة قبل أن تتحول إلى لغة الآلة.

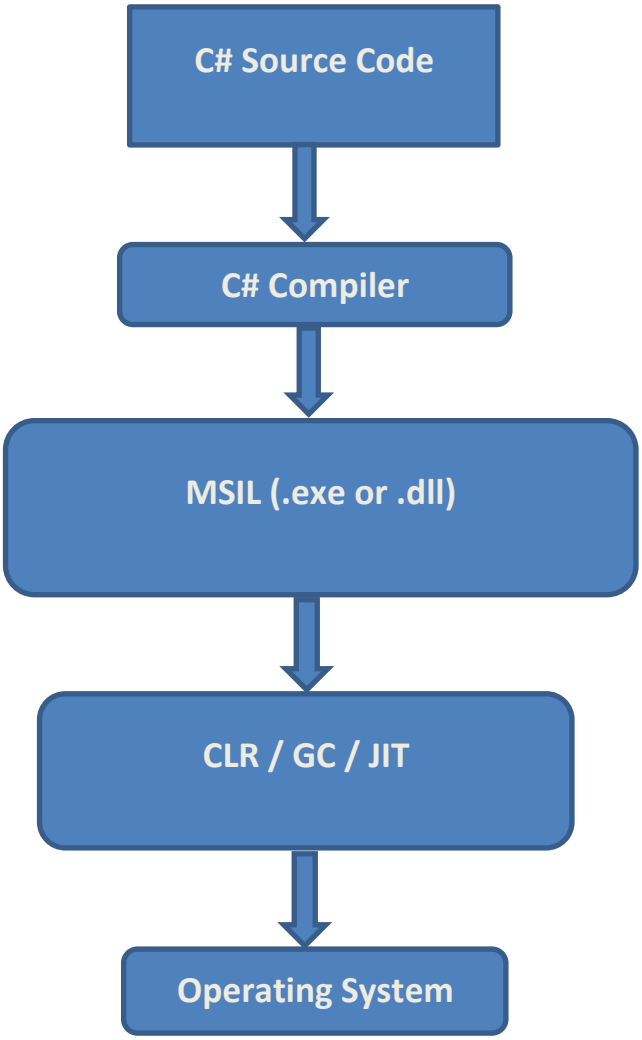
ملاحظة : مكونات تطبيقات اللغة الوسيطة وُضِعت في البرنامج التنفيذي (الملف الذي يُنفِذه الكمبيوتر في نهاية المهام).

ثانيا عندما يُنَفَذ البرنامج، يمر على معالج اخر يُعرف بـ just-in-time (JIT) وهذا أيضاً يكون بداخل CLR

ويعمل على ترجمة اللغة الوسيطة في الملف التنفيذي إلى لغة الآلة (1 , 0).

ملاحظة :

أي أمر يأخذه الكمبيوتر يتحول إلى اللغة التي يفهمها الكمبيوتر ويستطيع التعامل معها وهو لا يفهم غيرها وهي (1 , 0) أي شيء يتحول إلى الأصفار والآحاد.



وتوجد أيضا أداة داخل إطار عمل الـ (.Net) تسمى :

CTS (Common Type System)

وهي جزء من الـ CLR وهي عبارة عن آداة تستطيع تشغيل اكثر من لغة برمجة معاً بمكتبات مختلفة.

وتحتفظ بالاساسيات بداخلها وعندما تعمل أى لغة تعود هذه الآداة لتلك الاساسيات الخاصة بهذه اللغة لكي تعمل على إطار العمل حيث أن كل لغة برمجة يكون لها قواعد خاصة بها في الكتابة ويكون لكل لغة العديد من المكتبات التي تحتوي على تعريف لكل شئ خاص باللغة.

https://msdn.microsoft.com/ar-sa/library/

إن لم تستطع فهم هذه المعلومات فلا بأس، ستفهمها فيما بعد، فمحتوى الكتاب غير معتمد عليها، لكني تعمدّت قولها لكي تعرف كيف يعمل البرنامج الذي تكتب عليه الكود وماذا يحدث مع الكود الذي تكتبه وما يحدث وراء الشاشة وكيف يفهمه الكمبيوتر ويظهر على شكل برنامج له واجهه ويستطيع المستخدم العادي التعامل معه، سنبدأ في بناء هيكل أول برنامج في لغة السي شارب .

إذاً أين يُكتَب الكود؟

يمكننا كتابة الكود على أي محرر نصوص ويكون إمتداد الملف الذي نكتب به الكود (.cs)

أو على بيئة تطوير متكاملة IDE (Integrated Development Environment)

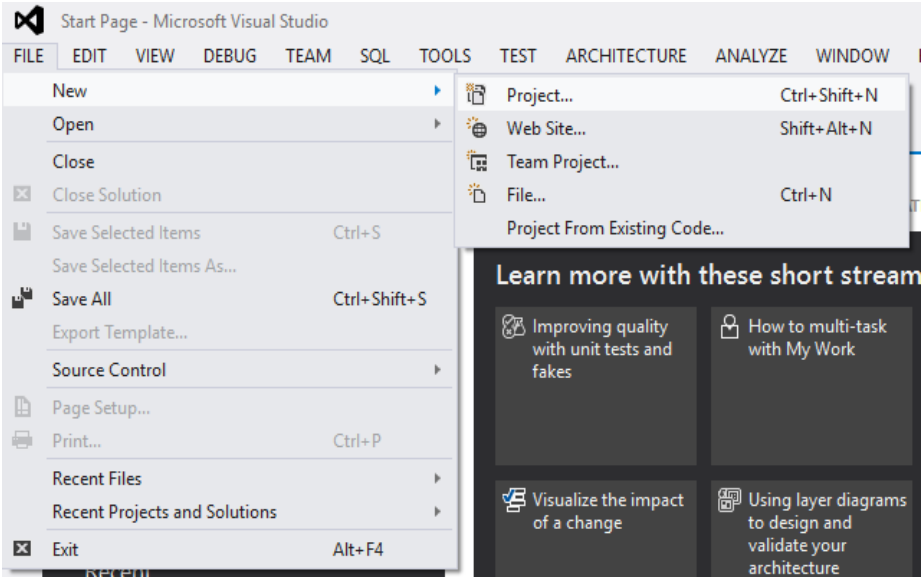
وهذه البيئة (visual studio) هي المعتمدة من شركة مايكروسوفت في التطوير.

يمكنك أن تُحمِلَ أي إصدار منه.

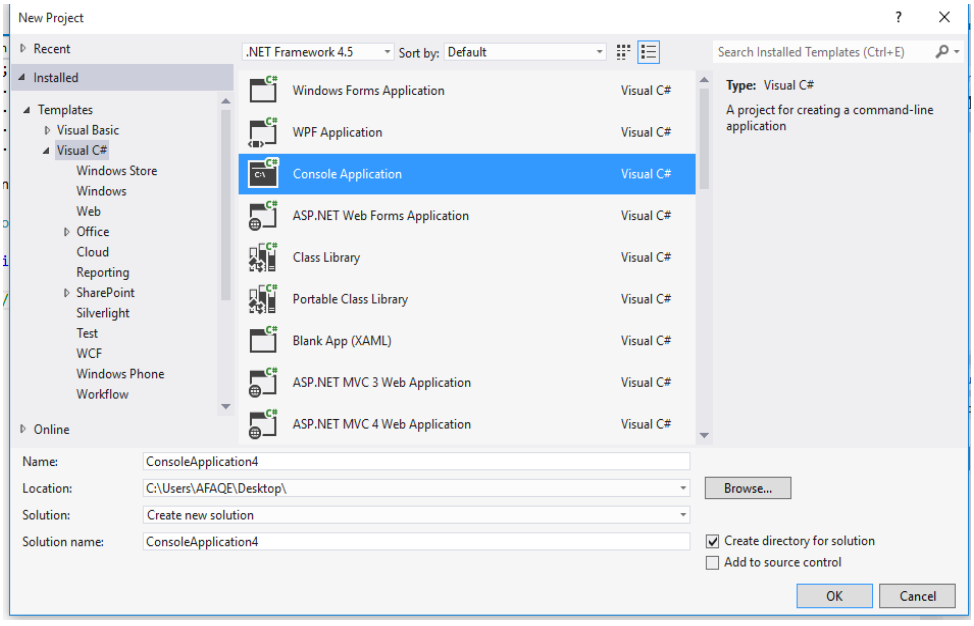
بعد تحميل البرنامج وتثبيته على جهازك ...

اولاً نفتح برنامج visual studio من قائمة Start

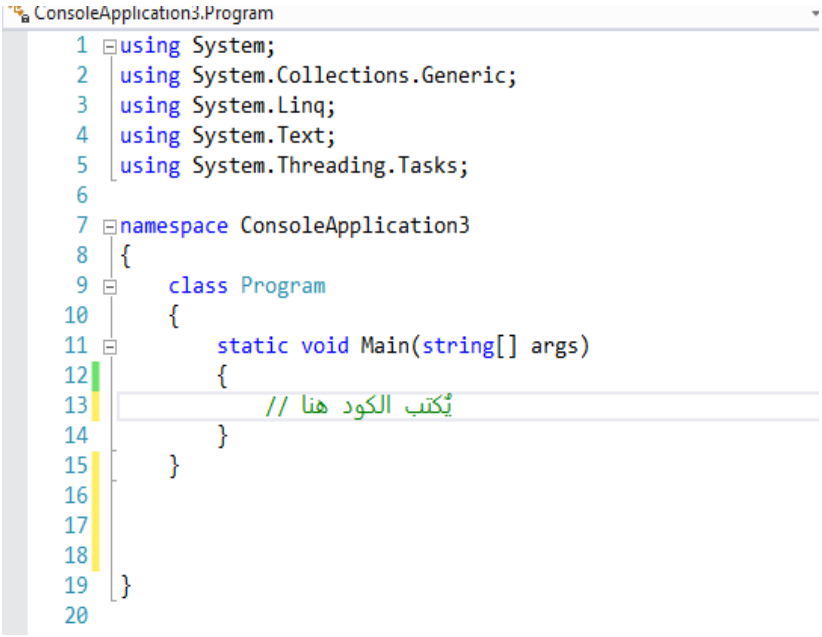
ثم نضغط على File بعدها ندخل إلى New ثم نختار project



بعدها نختار لغة C# ثم Console Application



سيفتح أمامك



using System;

namespace + اسم البرنامج

فمثلاً إذا اسميت البرنامج First_project

سيكون كالتالي:

```
namespace First_project
{
        Class Program
        {
                static void Main(string[] agrs)
                {
                        Console.WriteLine("Hello World");
                }
        }
}
```

بعد الإنتهاء من كتابة الكود سنضغط على كلمة Start لكي نقوم بمعالجة للكود و تنفيذه.

أو يمكننا الضغط على (ctrl+f5)

ملاحظة : بعض إصدارات برنامج visual studio تفتح الشاشة السوداء (Console) وتغلق بسرعة قبل رؤية نتيجة الكود،

يمكنك أن تثبِّتَ الشاشة عن الإغلاق بكتابة (; ( )Console.ReadKey)

أو (; ( )Console.ReadLine)

وتُكتَب هذه الجملة في آخر سطر في الكود في دالة الـ ( )Main ...

```
namespace First_project
{
        Class Program
        {
                static void Main(string[] agrs)
                {
                        Console.WriteLine("Hello World");
                        Console.ReadKey( );
                }
        }
}
```

لكي نفهم هذا البرنامج يجب أن نحلله أولاً.

ما معني هذه الكلمات...؟

(using , namespace , class , Main)

اتفقنا في البداية على أن لغة السي شارب (C#) إنما هي مبنية على اللغتين C++ و C وتعمل بشكل كلي بالبرمجة كائنية التوجه.

إذاً ما معنى البرمجة كائنية التوجه؟ وماهو إختلاف السي شارب عن السي؟ وما هي تقنيات البرمجة؟

لنتعرف على تقنيات البرمجة اولاً :

١- البرمجة الخطية (Linear Programming)

وهي كانت عبارة عن أن الكود يُكتب في صفحة واحدة وبعض الاجزاء في الكود قابلة للتكرار بدون شئ يجعلنا نستغني عن هذا التكرار في جزء معين من الكود، فمثلاً أنك تكتب بعض الاسطر من الكود لتنفيذ عملية معينة ثم تُكمل الكود وبعدها تقابلك مشكلة تحتاج إلى نفس العملية السابقة لكن في جزء مختلف من البرنامج، حينها لا تستطيع إلا أن تعيد كتابة هذا الجزء مرة أخرى، وأيضا عملية البحث عن الاخطاء كانت صعبة جداً وتأخذ الكثير من الوقت، كود يحتوي على آلاف الاسطر فلم يكن من السهل التعامل مع الكود في عمليات التعديل وكان الكود غير مرتب، وُجِدَت فكرة البرمجة الخطية لحل المشاكل المتعلقة بالرياضيات ولتسهيل حلها.

٢- البرمجة الهيكلية (Structure Programming)

ظهر بعض التطوير في هذه الوقت واخترعوا الدوال أو الوظائف (Functions) وهي تُمكِنك من كتابة (عملية معينة) عدد من الأسطر مرة واحدة وتستدعيهم في كل مره احتجت إليهم دون الحاجة إلى إعادة كتابة هذه الاسطر(العملية)، فقط نكتب اسم الدالة، لكن أيضا وُجِدَ بعض التقصير فيها، حيث يصعب فصل البيانات عن العمليات، ولا توجد حماية للبيانات من حدوث الأخطاء، والتعديل بها ينتج عنه أخطاء في أجزاء أخرى من الكود، ومن اللغات التي تعمل بالبرمجة الهيكلية لغة C.

٣- البرمجة كائنية التوجه (Object Oriented Programming)

وأخيرا ظهرت تقنية جديدة في كتابة الكود تُمكِنَك من تنظيم الكود وأختصرت جداً في حجم الكود وأصبح مرتباً وسهلّ عمليات البحث عن الأخطاء وساعد المطورين على العمل بأريحية ووفرت الكثير من الوقت والجهد أيضاً، وأصبح كل شيء منفصلاً عن الآخر حيث تتم عملية التعديل والتغيير في الكود بدون حدوث أخطاء مترتبة على هذا التغيير، وأُضيفت الكثير من التقنيات التي تحافظ على البيانات وتغليفها وإخفاء البيانات والتحكم الكامل بها، ومن اللغات التي تعمل بالبرمجة كائنية التوجه (البرمجة الشيئية) لغة C++ و C# و Java و Python ومعظم لغات البرمجة الحديثة.

Namespace

لنتخيل أننا كتبنا مجموعة من الأسطر البرمجية في دالة (Function)، وأنشأنا أكثر من دالة، إذاً تحتاج إلى شيء يدير هذه الدوال معاً، اسمه class.

لا تشغل تفكيرك بمعنى كلمة class، سنفهمه فيما بعد، كل ما أريدك أن تعرفه عن الـ class أنه يمكن أن يحتوي على أكثر من دالة ويمكن إنشاء أكثر من class في نفس المشروع، إذاً نحتاج إلى شئ ليجمع كل هذه الـ classes مع بعضها، سنضعهم بداخل Namespace فيكون الكود مرتب أكثر.

إذاً فالـ (namespace)هي عبارة عن حاوية لمجموعة من الـ classes، ويمكن أن يحتوي المشروع على أكثر من namespace.

ولنستطيع رؤية وإستخدام الـ classes التي بداخل الـ namespace

يجب أن نستدعي الـ namespace بإستخدام كلمة using .

Main

تسمى الدالة الأساسية التي يبدأ منها البرنامج في تنفيذ الكود، فمثلاً إذا كتبت مئات العمليات والإجراءات خارج الدالة الاساسية فلن يتم تنفيذهم أولاً بل يبدأ البرنامج في تنفيذ ما بداخل الدالة الأساسية.

```
static void Main(string[ ] agrs)
{
        Console.WriteLine(" C# ") ;
}
```

سنفهم معنى كلمة string[ ] args في الفصل الخاص بالدوال، يُمكن الإستغناء عنها في دالة الـ Main، نستطيع كتابة أقواس الدالة بدون كتابتها.

تكون بالشكل التالي :

```
static void Main( )
 {
        Console.WriteLine(" C# ") ;
 }
```

3

الإدخال والإخراج

(Input and Output)

عمليات الإدخال والإخراج هي التي تُمكِنّ المستخدم من التعامل مع البرنامج أو التطبيق، على سبيل المثال إذا كنت من مستخدمي موقع الفيس بوك فإنك أولاً تُدخِل البريد الإلكتروني وكلمة المرور فهذه تُعَد عملية إدخال لك كمُستخدم للموقع، والبيانات التي تظهر فيما بعد فهذه عملية إخراج.

```
using System ;
namespace First_project
{
        Class Program
        {
                static void Main(string[ ] agrs)
                {
                        Console.ReadLine( );
                }
        }
}
```

كلمة Console عبارة عن class بداخل Namespace (System)

ReadLine( )- عبارة عن دالة تُستخدم لعمليات الإدخال وتستقبل نصوص فقط.

ويجب ان نضع فاصلة منقوطة في نهاية كل جملة في البرنامج (;) وإلا سيحدث خطأ.

أنواع الأخطاء في الكود :

١- خطأ في طريقة كتابة الكود (Syntax error) وذلك يكون خطأ في قواعد كتابة الكود.
٢- خطأ منطقي (Logic error) وفي هذه الحالة يكون الكود مكتوب بطريقة صحيحة كقاعدة ولكنه غير منطقي مثل قسمة واحد على صفر.
٣- خطأ أثناء التشغيل (Runtime error) هذا الخطأ يحدث أثناء تشغيل البرنامج، مثل وجود فائض في البيانات المُدخَلَة.

نسيان الفاصلة المنقوطة يُعد خطأ في طريقة الكتابة أي (Syntax error).

```
using System ;
namespace First_project
{
        Class Program
        {
                static void Main(string[ ] agrs)
                {
                        Console.WriteLine(" C# ") ;
                }
        }
}
```

WriteLine( ) - هي دالة لطباعة المخرجات

Line - تُكتب لكي يُنهي السطر الحالي ويبدأ في سطر جديد.

ويمكن أن نكتب فقط ( )Write لكن الأسطر التي تليها في عملية الطباعة ستكون متلاصقة بها.

ويوجد طريقتان للطباعة :

١- (Placeholder)

وتكون كالتالي :

```
Console.WriteLine(" Hello World {0} " , 5 , 20 ) ;
```

وفي هذه الحالة سيطبع Hello World + أول مُدخَل للدالة بعد الفاصلة

وهي القيمة (5)

وإذا أردنا طباعة ثاني مُدخل سنكتب {1} وحينها لن يعرض أول مُدخل وسيعرض القيمة 20

حيث أن أول ترقيم (index) يبدأ بالقيمة صفر { 0 }.

ويُمكنك أن تطبع أي شيء بالشكل الذي تريده فمثلاً :

```
Console.WriteLine(" My Name is {0} and my age is {1}" , " Ali " , 50 ) ;
```

في هذه الحالة سيطبع

My Name is **Ali** and my age is **50**

مثال ٢ :

```
Console.WriteLine(" x =  {2} , y =  {0} and z = {1} " , 10 , 20 , 30 ) ;
```

سيطبع

x = 30 , y = 10 and z = 20

٢- (Concatinatoin) الدمج

```
Console.WriteLine(" Hello World " + 5 +20);
```

لكن هذه الطريقة ثابتة وماهو مكتوب أولاً سيعرض أولاً بعكس الطريقة السابقة انت من يتحكم فيما سيظهر أولاً وسيعرض الكل .

سيكون الناتج....
Hello World 520

سيعرضها بهذا الشكل ولن يجمع الرقمين فهذه الأداة (+) بعد النص في الطباعة ليس أداة جمع إنما هي أداة للدمج بين النصوص فهو يتعامل مع الأرقام التي بعد النص على أنها نص وليست رقم فمنطقياً لا يمكن إضافة رقم على نص، لكن إذا أتى الرقمان قبل جملة الطباعة فسيجمع الرقمين أولاً ويقوم بالعملية الحسابية بطريقة عادية جداً.

```
Console.WriteLine( 5 + 20 + "Hello World" ) ;
```

سيطبع : 25Hello World

وعندما ننتهي من الكود ونريد تشغيل البرنامج نضغط Ctrl+F5.

توجد بعض العلامات تسمى " Escape Sequence " التي قد تساعدك في عملية الطباعة لتخرج النص بشكل منسق أو إنذار عند عملية ما خطأ.

\'	لطباعة علامة تنصيص فردية
\"	لطباعة علامة تنصيص زوجية
\\	لطباعة علامة \
\a	لعمل إنذار
\n	للنزول إلى سطر جديد
\t	لعمل tab (مسافات فارغة)

مثال :

```
Console.WriteLine(" \' Hello " ) ;

Console.WriteLine(" \" Hello " ) ;

Console.WriteLine(" \\ Hello " ) ;

Console.WriteLine(" \a Hello " ) ;

Console.WriteLine(" \n Hello " ) ;

Console.WriteLine(" \t Hello " ) ;
```

4

أنواع البيانات

(Data Types)

البيانات التي تُخزن في الذاكرة يجب أن يكون لها نوع حتي يتعرف عليها مترجم اللغة، فمثلاً تكون نصوص أو حرف أو عدد صحيح أو عدد عشري ويجب أن نحدد نوعها أولاً وإلا لن يتعرف عليها المترجم وسيحدث خطأ، يوجد العديد من الانواع ولكن بالتأكيد لن تسخدم كل شيء، فقط ما تحتاج إليه في برنامجك...

نوع البيانات	النوع الموافق في منصة الدوت نت	الحجم	الوصف
bool	System.Boolean	true أو false	قيمة منطقية صح أو خطأ
sbyte	System.Sbyte	من – ١٢٨ الي ١٢٧	رقم صحيح موجب أوسالب مساحته ٨ بت
byte	System.Byte	٠ الى ٢٥٥	رقم صحيح موجب مساحته ٨ بت
short	System.Int16	–٣٢,٧٦٨ الى ٣٢,٧٦٧	رقم صحيح سالب أو موجب مساحته ١٦ بت
ushort	System.UInt16	٠ الى ٦٥,٥٣٥	رقم صحيح موجب مساحته ١٦ بت

int	System.Int32	٢,١٤٧,٤٨٣,٦٤٨– الى ٢,١٤٧,٤٨٣,٦٤٧	رقم صحيح سالب أو موجب مساحته ٣٢ بت
uint	System.UInt32	٠ الى٤,٢٩٤,٩٦٧,٢٩٥	رقم صحيح موجب مساحته ٣٢ بت
long	System.Int64	– ٩,٢٢٣,٣٧٢,٠٣٦,٨٥٤,٧٧٥,٨٠٨ الى ٩,٢٢٣,٣٧٢,٠٣٦,٨٥٤,٧٧٥,٨٠٧	رقم صحيح موجب أوسالب مساحته ٦٤ بت
ulong	System.UInt64	٠ الى ١٨,٤٤٦,٧٤٤,٠٧٣,٧٠٩,٥٥١، ٦١٥	رقم صحيح موجب مساحته ٦٤ بت
char	System.Char	U+0000 الى U+ffff	يقبل حرف مساحته ١٦ بت
float	System.Single	٠ الى 1.844674407370955e+16	يقبل رقم كسري مساحته ٣٢ بت
double	System.Double	٣.٤٠٢٨٢٣– E+38 الى 3.402823E+38	يقبل رقم كسري مساحته ٦٤ بت
decimal	System.Decimal	– ١.٧٩٧٦٩٣١٣٤٨٦٢٣٢ E+308 الى ١.٧٩٧٦٩٣١٣٤٨٦٢٣٢E+308	يقبل رقم سالب أو موجب مساحته ١٢٨ بت
string	System.String	مقيد بحسب الذاكرة	يمثل سلسلة نصية من جدول اليونكود
object	System.Object	يقبل أي نوع من الأنواع السابقة	

إذا مما تتكون البيانات التي تُخزن في الذاكرة؟

تتكون من (نوع البيانات + المتغير الذي يحتوي على البيانات + القيمة)

Data type + variable + value

5

المتغيرات

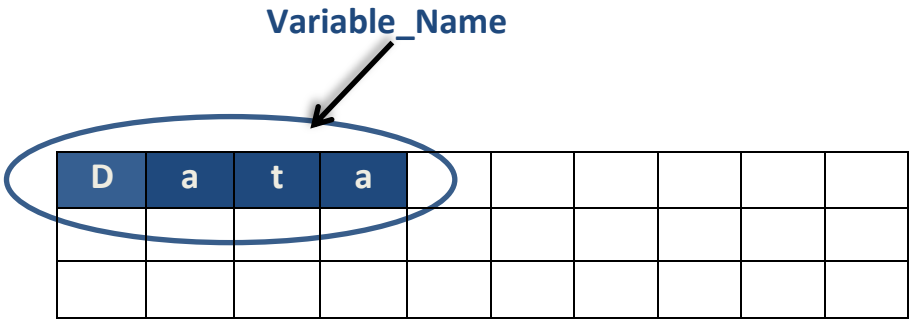
(Variables)

المتغير هو عبارة عن اسم لمكان تخزين البيانات في الذاكرة، والذاكرة عبارة عن خلايا وكل خلية تُعادل ١ بت، وتُحَدَد المساحة التي تأخذها البيانات في الذاكرة من خلال نوعها وتتغير قيمة المتغير بإستمرار كلما أدخلنا له قيمة جديدة، كما يمكننا تسمية المتغير بأي اسم نريده لكن لا نستطيع تسمية متغيرين بنفس الأسم، ويجب أن تكون القيمة التي ستُخزَن تتناسب مع نوع المتغير.

تتم عملية تعريف المتغير بهذا الشكل :

القيمة = اسم المتغير + نوع البيانات التي تخزن في المتغير

Data type Variable_Name = value ;



ونُسمي هذا المكان في الذاكرة لكي نستطيع الوصول إلى البيانات مرة أخرى بسهولة، فهو بمثابة صندوق توضع به هذه البيانات .

```
using System;
namespace First_program
{
        class Program
        {
                static void Main( )
                {
                        int num = 5;
                        string name = "123Mahmoud";
                        float num2 = 34.5f;
                        char c = 'a';
                        double num3 = 34.5;
                }
        }
}
```

في هذا المثال عرفّنا متغير من النوع (int) ووضعنا به القيمة 5

ومتغير من النوع (string) ووضعنا به نص داخل علامتين تنصيص ويجب أن يوضع النص أو الكلمة بين علامتين تنصيص، حتى إذا كانت القيمة عبارة عن أرقام لكنها بداخل علامتين تنصيص فإنه يتعامل معها على أنها نص وليست أرقام.

ومتغير من النوع (float) رقم يحتوي على علامة عشرية ونكتب في آخره (f) لكي يتعرف المترجم على أنه float، لأن القيمة الافتراضية للأعداد الكسرية هي double.

ثم عرفنا متغير من النوع (char) يأخذ حرف ويوضع بين علامة تنصيص فردية.

يجب أن نعلم أن لغة السي شارب حسّاسة من ناحية الحروف أي أن A لا تساوي a

فيجب أن ننتبه عند الكتابة.

☒ Int : خطأ

✓ int : صح

فمثلاً إذا كان :

```
int a=7;
int A =7;
```

فإنهما في هذه الحالة متغيران مختلفان، ويجب مراعاة الدقة في كتابة نوع المتغير والتعامل معه بشكل صحيح وإلا سيحدث خطأ في البرنامج ولن يتعرف على نوع المتغير.

```
using System;
namespace First_program
{
        class Program
        {
                static void Main( )
                {
                int num1=5 ;
                int num2=10 ;
                int sum ;
                sum = num1 + num2 ;
                Console.WriteLine("sum of two numbers = "
                                        +sum);
                }
        }
}
```

وإذا كنت تريد تعريف أكثر من متغير من نفس النوع فمن الممكن أن نعرفّه بهذه الطريقة...

```
int num1=5 , num2=10 , sum ;
```

مثال :

```
static void Main( )
{
        int x = 10 ;
        x = 5 ;
        x = 20 ;
}
```

أدخلنا ثلاث قيم للمتغير، تكون قيمة المتغير الجديدة هي آخر قيمة دخلت له.

إذاً تكون قيمة المتغير x تساوي ٢٠.

ملاحظة :

لا يجوز تعريف المتغير أكثر من مرة في نفس المكان من الكود.

فمثلاً :

```
static void Main( )
{
   int x =1 ;
   int x =5 ;
}
```

خطأ لأن هذا المتغير مُعرفّ مُسبقاً.

تسمية المتغير

يجب أن يكون اسم معبر عن نوع البيانات التي ستُخزن فمثلاً إذا كان اسم او عُمر شخص.

```
string Name;
int Age;
```

ولا يجوز أن يبدأ المتغير بأي من الرموز الآتية :

! # $ % ^ & - = +) (?

لكن ممكن أن يبدأ بـ _

صحيح → int _name ;

يمكننا أن نسمي المتغير بأي اسم نريده، لكن هناك بعض الكلمات المحجوزة في اللغة التي لا نستطيع تسمية المتغير بنفس اسمها ...

abstract	as	base	bool
break	byte	case	catch
char	checked	class	const
continue	decimal	default	delegate
do	double	else	enum
event	explicit	extern	false
finally	fixed	float	for
foreach	goto	if	implicit

in	in (generic modifier)	int	interface
internal	is	lock	long
namespace	new	null	object
operator	out	out (generic modifier)	override
params	private	protected	public
readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc
static	string	struct	switch
this	throw	true	try
typeof	uint	ulong	unchecked
unsafe	ushort	using	using

			static
virtual	void	volatile	while
add	alias	ascending	
async	await	descending	
dynamic	from	get	
global	group	into	
join	let	nameof	
orderby	partial (type)	partial (method)	
remove	select	set	
value	var	when (filter condition)	
where (generic type constraint)	where (query clause)	yield	

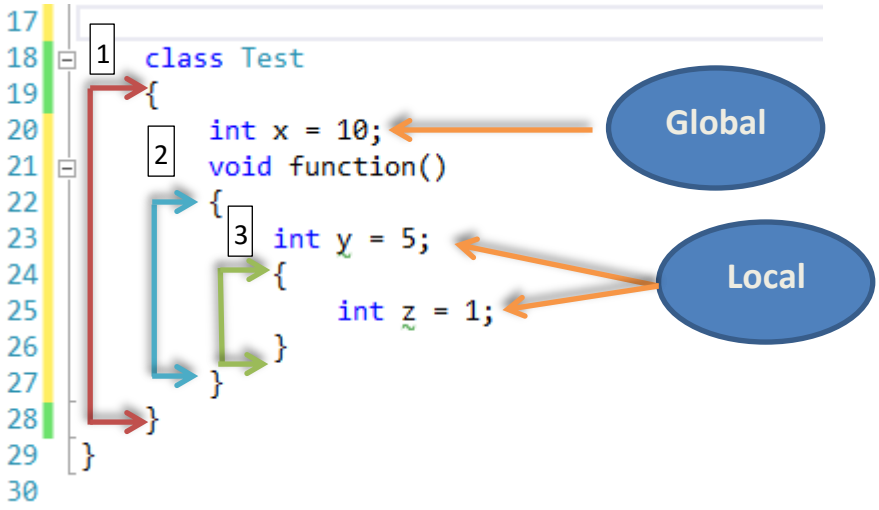
فمثلاً لا نستطيع تعريف متغير هكذا int void
هذا غير صحيح وسيحدث خطأ، يجب الابتعاد عن هذه الكلمات في تسمية المتغيرات فهي كلمات محجوزة في اللغة ولها استخدامات معينة.

كل متغير نعرفّه يكون له نطاق ومجال يُرى فيه في الكود فإما أن يكون :

Local (محلي) : ويكون ذلك في نطاق محدد ونحن نحدد مدى رؤية مترجم اللغة له في باقي الكود من خلال قوسين المجموعة { }

Global (شامل أو عام) : ويكون مرئي في كل الكود داخل الـ class أي يكون في بداية الـ class.

ملاحظة : عندما تعطي قيمة إبتدائية للمتغير فإنه يستخدمها فقط في حالة أنك لم تُدخل له قيمة أخرى.
مثال :



في هذه الصورة يكون المتغير x عام، أي مرئي في كل الكود الذي في هذا الـ class الذي يُسمى Test

فإذا كُتِبَ حرف x في أي مكان في الصورة تحت اسم الـ class سيظهر لك المتغير x لكن بشرط أن يكون داخل مجال الـ class أي أنه لا يقع بين خارج أول قوس وآخر قوس للـ class كما يشير السهم (1) في الصورة. ولا يجوز إعادة تعريف متغير آخر بنفس أسم المتغير x لأنه مُعرفّ مُسبقاً حيث أنه متغير عام داخل الـ class.

أما المتغير y فإنه أقل شمولية من x حيث أنه لا يُتعرفّ عليه خارج قوسين المجموعة للدالة، يُرى بأي مكان داخل قوسين المجموعة الخاصين بالدالة كما يشير السهم (2). ويجوز تعريف متغير آخر بنفس اسمه خارج أقواس الدالة الخاصة به.

المتغير z هو الأقل شمولية فيهم حيث أنه لا يُتَعَرف عليه خارج القوسين المحيطين به كما يشير السهم (3).

كل قوس مجموعة هو عبارة عن مرحلة ومستوى كلما صعد المتغير أعلى خارج الأقواس كلما زادت شموليته ومعرفته في أجزاء أخرى في الـكود.

الثابت (Constant)

الفرق بين الثابت والمتغير أن الثابت قيمته ثابته لن تتغير طوال البرنامج

أما المتغير فيمكن تغيير قيمته، إذا حاولنا تغيير الثابت سيحدث خطأ في البرنامج.

ويكون بالشكل الآتي :

```
const + data type + name + value ;
const string name="Mahmoud" ;
```

```
const int number = 1 ;
```

ويمكن أن نستخدمه في القيم الرياضية والفيزيائية الثابتة.

تمارين٢ :

أحسب مساحة دائرة نصف قطرها ٤

```
using System;
namespace Second_program
{
    class Program
      {
            static void Main()
              {
                    const float pi = 3.14f ;
                    int radius = 4 ;
                    float area = pi * (radius * radius);
                    Console.WriteLine(area);
                    }
            }
}
```

6

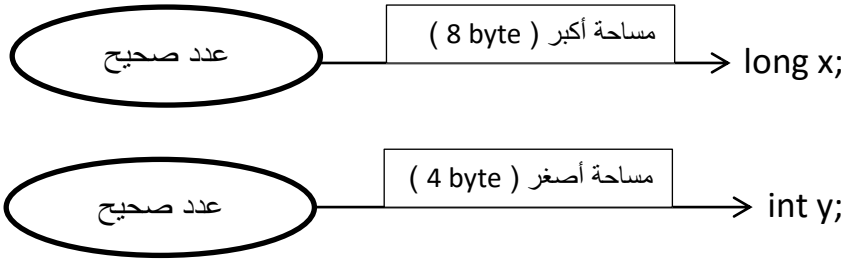
تحويل أنواع المتغيرات

(Conversions)

١- تحويل ضمني (implicit)

وهي أن تأخذ متغير ما لتضعه في متغير أكبر منه في الحجم أو يساويه ونفس نوع البيانات بدون أن يحدث أي فقدان أو إضاعة لجزء من البيانات التي بداخله.

مثال :

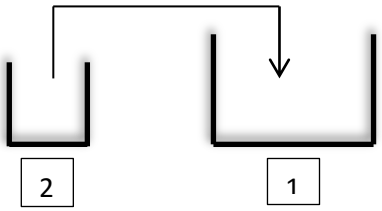



```
x = y ;
```

في هذه الحالة وضعنا قيمة المتغير y داخل المتغير x .

حيث أن حجم المتغير (x) ٦٤ بت وحجم المتغير (y) ٣٢ بت فيمكن للمتغير ذو الحجم الأكبر أن يحتوي على المتغير ذو الحجم الأقل بدون أي حدوث عملية فقد للبيانات. أما إذا حاولت فعل العكس في الحالة الطبيعية فسيحدث خطأ ويحدث فقد للبيانات.

إذا حاولت وضع الصندوق في الشكل (2) في الصندوق في الشكل (1) لن يحدث أي خطأ لأن الصندوق في الشكل (1) أكبر من الصندوق الآخر، لكن في حالة عكس هذه العملية سيحدث خطأ لأن الصندوق في الشكل (2) أصغر في الحجم.



٢- تحويل صريح (Explicit or Casting)

وهي عكس الحالة السابقة، وتحدث عندما يوجد إحتمال فقدان في البيانات أثناء عملية تمرير البيانات إلى المتغيرات الجديدة، ويمكننا وضع بيانات المتغيرات ذي الاحجام المختلفة والانواع المختلفة داخل بعضها بشرط تغيير نوع المتغيرالضيف إلى نفس نوع المتغير المُضيف.

وفي حالة الأعداد يمكننا وضع القيم الكسرية float أو double بداخل متغيرات نوعها int أو long بشرط تغيير نوعها .

مثال ١:

```
long x;

int y;

y = ( int ) x;
```

مثال ٢:

```
int x = 10 ;

float y = 3.0f ;

x = ( int ) y ;
```

أو يمكننا إستخدام هذه الطريقة :

```
x = Convert.ToInt32( y );
```

وفي هذه الحالة فإن المتغير المُضيف يستطيع إستقبال المتغير الضيف.

الفرق بين :

Convert.ToInt16	Convert.ToInt32	Convert.ToInt64
أي أنها تحول المتغير إلى النوع short حيث أن النوع short حجمه ١٦ بت أي ٢ بايت	تحول المتغير إلى النوع int حيث أن النوع int حجمه ٣٢ بت أي ٤ بايت	تحول المتغير إلى النوع long حيث أن النوع long حجمه 64 بت أي ٨ بايت

تذكرّ أن :

إذا جاء حرف u قبل هذه الأنواع من البيانات

(short , int , long , float , double)

معنى ذلك أن هذه الأنواع تستطيع أخذ قيمة موجبة فقط، وحرف u إختصار لـ unsigned.

توجد بعض الدوال لتحويل النصوص ...

مثل :

`decimal`	ToDecimal(String)
`float`	ToSingle(String)
`double`	ToDouble(String)
`short`	ToInt16(String)
`int`	ToInt32(String)
`long`	ToInt64(String)
`ushort`	ToUInt16(String)
`uint`	ToUInt32(String)
`ulong`	ToUInt64(String)

وتوجد أيضا الدالة ---> ( )Parse

لتحويل أي شيء من النوع (string)

فمثلاً إذا أردنا إدخال قيمة لمتغير نوعه int من خلال المستخدم :

```
using System;
namespace First_program
{
        class Program
        {
                static void Main( )
                {
                        int num;
                        num = int.Parse(Console.ReadLine( ));
                }
        }
}
```

هذا الكود معناه أنه يوجد متغير من النوع (int) وسيأخذ قيمته من المستخدم بواسطة الدالة ( )ReadLine لكن هذه الدالة لا تقبل إلا نصوص، لذا سنجعلها تقبل أعداد صحيحة من خلال دالة ( )Parse.

وآخراً سنضع كل هذا داخل المتغير الذي نريد إدخال القيمة له كما في المثال السابق.

أو

```
num = Convert.ToInt32(Console.ReadLine( ) ) ;
```

7

التعليقات

(Comments)

التعليقات هي عبارة عن شرح موجز لجزء معين من الكود البرمجي فيجعل الكود سهل القراءة والفهم، ويتجاهل المترجم التعليقات ولا يتعامل معها نهائياً فهي مجرد ملاحظات أو تنويه يكتبه المبرمج لنفسه أو لمن يعمل بعده، إذا وُجِدَ فيها خطأ نحوي أو منطقي فلا يهم فلن يحدث أي خطأ.

أنواع التعليقات :

تعليق لسطر فردي (//)

مثال :

```
using System;
namespace First_program
{
    class Program
      {
          static void Main( )
           {
               int num;    // this is the first variable
           }
       }
}
```

تعليق متعدد الأسطر (/* ..... */)

أي أنه يمكن أن نكتب ملاحظات في أكثر من سطر، فيكون هذا الجزء من الكود غير مرئي بالنسبة للمترجم.

مثال :

```
using System;
namespace First_program
{
        class Program
        {
                static void Main()
                {
                        int num;
                        /* this is the first variable
                        this is the first variable
                        this is the first variable
                        this is the first variable */
                }
        }
}
```

سيتجاهله المترجم تماماً.

أيضا لا يجب أن ننشغل بالتعليقات كثيراً، تُكتب التعليقات في الاجزاء المهمة التي تحتاج إلى توضيح فقط.

8

المعاملات الرياضية والمنطقية

(Arithmetic and Logic operators)

المعاملات الرياضية

الجمع (+)

الطرح (-)

الضرب (*)

القسمة (/)

باقي القسمة (%)

الزيادة للمتغير بمقدار ١ (++)

النقصان للمتغير بمقدار ١ (--)

مثال :

بفرض أن ....

A = 5 and B = 2

A + B = 7

A – B = 3

A * B = 10

A / B = 2.5

A % B = 1
A++ = 6
A-- = 4

يمكن أن نضع علامتين الجمع بعد المتغير أو قبله.
لكن انتبه...!
A++ لا تساوي ++A
لأنه عندما نضعهم قبل المتغير فإن الزيادة في مقدار المتغير تحدث أولاً ثم تحدث العملية التي تليها على المتغير، أما إذا وضعنا العلامتين بعد المتغير فإن العملية ستحدث أولاً ثم بعد ذلك تحدث الزيادة في القيمة.

مثال :

```
using System;
namespace First_program
{
    class Program
    {
        static void Main()
        {
            int num1 = 10;
            int num2 = 8;
            int sum;
            sum=(num1++) + ( ++num2);
            Console.WriteLine(sum);
            //result=19
```

```
                    Console.WriteLine(num1);
                    //result =11
                    Console.WriteLine(num2);
                    //result =9
                }
            }
        }
```

سيطبع ١٩ فقط، لماذا؟
أول متغير لم يحدث له تغيير ولم تزد قيمته بعد، لأنه يقوم بالعملية على المتغير أولاً ثم الزيادة ثانياً، فقام بعملية الجمع وبعدها زادت قيمته بمقدار ١، أما المتغير الثاني فزادت قيمته أولاً ثم قام بعملية الجمع، وإذا قمنا بطباعة المتغيرين بعد العملية ستجدهم قد زادا بمقدار ١، والطرح كذلك أيضاً.

مثال :

```
using System;
namespace First_program
  {
    class Program
      {
          static void Main()
              {
                 int A=10;
                 int B=8;
                 int sum;
```

```
            sum= ( A++ ) + ( ++B ) + ( --A ) + ( B-- ) +( B++ )
                        + ( A ) - ( B ) ;
            // ( 10 ) + ( 9 ) + ( 10 ) + ( 9 ) + ( 8 ) + ( 10 ) - ( 9 )
                الناتج = ٤٧ // ; ( Console.WriteLine( sum
        }
    }
}
```

إذا وُجِدَ متغيران نوعهما string وبينهما علامة الجمع فإنها ستدمج النَصَين مع بعضهما.

مثال :

```
string x = " Hello " ;

string y = " World " ;

Console.WriteLine( x + y ) ;
```

سيطبع :

Hello World

معاملات المقارنة :

يساوي (==)
لا يساوي (=!)
أكبر من (>)
أقل من (<)
أكبر من أو يساوي (=>)
أقل من أو يساوي (=<)

مثال :
بإفتراض أن ...

A = 10 , C = 10 and B = 5

A == C

A > B

A != B

B < A

A >= C

B <= A

ملاحظة :

علامة = ليست علامة يساوي إنما تسمى تعيين أو تخصيص (assignment).

فإذا قولنا أن :

A = 5

أو

A = B

معنى هذا أن قيمة الجانب الأيمن للعلامة (=) تنتقل إلى الجانب الأيسر.

القيمة 5 تنتقل إلى المتغير A

والمتغير B يوضع بداخل المتغير A وتصبح قيمة المتغير A نفس قيمة المتغير B وقيمة المتغير B لا تتأثر وتظل كما هي.

مثال :

```
using System;
namespace First_program
{
        class Program
        {
                static void Main( )
                {
                        int num1 = 10;
                        int num2 = 8;
                        num1 = num2;
                        Console.WriteLine( num1 );
                        //الناتج = ٨
                        Console.WriteLine( num2 );
                        // الناتج = ٨
                }
        }
}
```

المعاملات المنطقية

(&&) Logical AND : يجب أن تكون القيمتين صحيحتين

(||) Logical OR : يجب أن تكون إحدى القيمتين صحيحة

(!) Logical NOT : يعكس القيمة، فإذا كانت صحيحة في الأصل تنفيها الأداة وتصبح خطأ والعكس.

وسنفهم المعنى أكثر عندما نصل إلى " الجمل الشرطية ".

المعاملات الثنائية التي تتعامل مع البت (Bit)

&

|

^

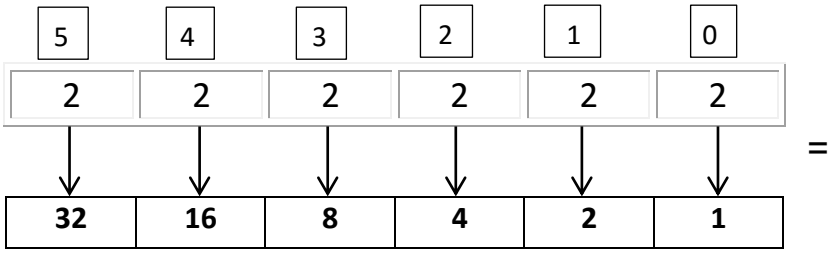
أي أنها تتعامل مع البيانات على شكل النظام الثنائي (Binary System).

ملاحظة :

أي حرف أو رقم يتم تحويله داخل الكمبيوتر إلى النظام الثنائي أي إلى أصفار وآحاد (0111001010010110) فكما ذكرت فهذه تسمى لغة الآلة والكمبيوتر لا يفهم غيرها.

يتم تحويل الأرقام العشرية إلى النظام الثنائي بهذه الطريقة :

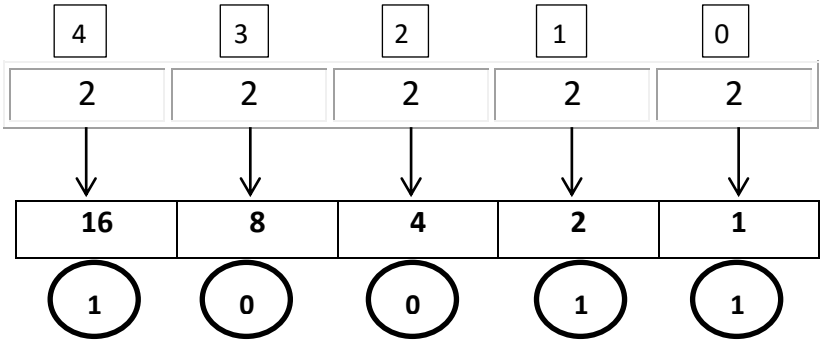
٢ أُس صفر إلى ٢ أُس ما لا نهاية، لذا سُميَ بالنظام الثنائي.



ويتم جمع كل القيم التي تُكوِنّ القيمة المُعطاة، والقيمة التي نأخذها نضع مكانها 1 والتي لم نستخدمها نضع مكانها 0. لا يشترط تكرار معين للأساس ٢، فقط ما تحتاج إليه لإخراج القيمة بالنظام الثنائي، أو يُمكنك أن تقسم القيمة المُعطاة على العدد ٢ وباقي القسمة يُكوِنّ القيمة بالنظام الثنائي.

مثال١ :

حوِل القيمة ١٩ إلى النظام الثنائي.



ثم نبحث عن الأرقام التي تُكوِنّ الرقم ١٩.

فتكون (16,2,1) فنضع 1 تحت كل قيمة قد أخذناها و0 تحت كل قيمة لم نستخدمها في تكوين الرقم العشري.

أو نقسم القيمة ١٩ على ٢ ونخرج باقي القسمة

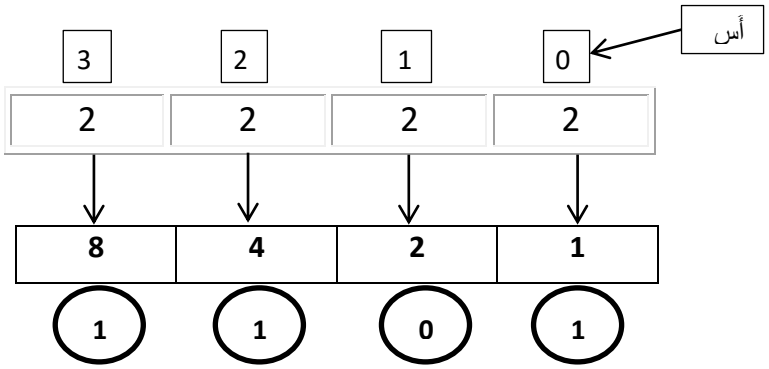
وتكون بهذا الشكل :

القيمة	الأساس	باقي القسمة
١٩	٢	١
٩	٢	١
٤	٢	٠
٢	٢	٠
١	٢	١
١		

فتكون القيمة (١٩) بالنظام الثنائي هي ---> 10011

مثال٢ :

حول القيمة ١٣ إلى النظام الثنائي.



نبحث عن الأرقام التي تُكوِنّ الرقم ١٣.

فتكون (8,4,1) فنضع 1 تحت كل قيمة قد أخذناها و0 تحت كل قيمة لم نستخدمها في تكوين الرقم العشري.

فتكون القيمة (١٣) بالنظام الثنائي هي ---> 1101

شرح هذه الجزئية كان فقط لتفهم كيفية عمل هذه المعاملات (" & | ^ ") مع الأرقام.

مثال :

- في حالة (&)

A = 5 and B = 20

A & B =

00101

&

10100

_____ =

00100

عند جمع 1 و 0 بإستخدام & فإن الناتج يكون 0، ويكون الناتج 1 في حالة واحدة... إذا كان الجمع بين 1 و 1.

إذاً A & B = 4

- في حالة (|)

A | B =
00101
|
10100
=_____
10101

إذاً A | B = 21

عند جمع 1 و 0 بإستخدام | فإن الناتج يكون 1، ويكون الناتج 0 في حالة واحدة... إذا كان الجمع بين 0 و 0.

في حالة (^)

A ^ B =
00101
^
10100
=_____
10001

إذاً A ^ B = 17

عند جمع 1 و 0 بإستخدام ^ فإن الناتج يكون 1، ويكون الناتج 0 في حالة أن القيمتان متشابهتان 1 و 1 أو 0 و 0

A	B	A & B	A | B	A ^ B
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

معاملات النقل أو الإسناد (Assigment):

=
+=
-=
*=
/=
%=
&=
^=
|=

مثال :

A = 3 and B = 2

A += B تعني أن المتغير A ستُجمع قيمته على المتغير B ثم سيوضع ناتج الجمع بداخل A.

إنما هي إختصار لهذا النمط :

A = A + B

A -= B → A = A- B

المتغير الثاني سيُطرَح من المتغير الأول ثم ستُوضع قيمة الطرح في الجانب الأيسر.

(A*=B → A=A*B) حاصل ضرب القيمتين يُوضع في الجانب الأيسر.

(A/=B → A=A/B) حاصل قسمة القيمتين يُوضع في الجانب الأيسر.

(A%=B → A=A%B) حاصل باقي قسمة القيمتين يُوضع في الجانب الأيسر.

(A&=B → A=A&B) ناتج العملية يُوضع في الجانب الأيسر.

(A^=B → A=A^B) ناتج العملية يُوضع في الجانب الأيسر.

(A|=B → A=A|B) ناتج العملية يُوضع في الجانب الأيسر.

أولويات العمليات الحسابية :

( ) يُنَفَذّ ما بداخل الأقواس أولاً.

% / * تُحسَب من الشمال إلى اليمين وأيهم يأتي أولاً فتحدث عمليته أولاً.

+ - تُحسَب من الشمال إلى اليمين وأيهم يأتي أولاً فتحدث عمليته أولاً.

=< < => > تُحسَب من الشمال إلى اليمين وأيهم يأتي أولاً فتحدث عمليته أولاً.

=! == تُحسَب من الشمال إلى اليمين وأيهم يأتي أولاً فتحدث عمليته أولاً.

= القيمة التي بالجانب الأيمن تنتقل للجانب الأيسر.

مثال :

1 – 7 % 3 * (5 + 2 / 6)

الناتج : 2

4 / 2 + 6 * 3 - 5 % 5 - 1 * (3 - 1)

الناتج : 8

توجد أيضاً بعض العوامل التي نستخدمها في الكود بإستمرار مثل:

النقطة (دوت) –> (.) تستخدم للوصول إلى الدوال والمتغيرات والـ classes

عندما نقول ...

```
System.Console.WriteLine( );
```

System هي عبارة عن namespace و للوصول إلى ما بداخلها من classes نكتب نقطة (.)

Console هي عبارة عن class و للوصول إلى ما بداخله من دوال ومتغيرات نكتب نقطة (.)

وكذلك دالة ( )WriteLine و ( )ReadLine.

[ ] تُستخدم مع المصفوفات وللوصول إلى فهرس المصفوفة.

( ) تُستخدم مع الدوال، أي دالة يجب أن تحتوي على هذين القوسان أثناء الإنشاء والإستدعاء، وأيضاً نستخدمها عند تحويل أنواع البيانات.

تمارين :

أكتب برنامج ليبدِلّ قيمتين لمتغيرين مع بعضهما :

مثال لتبسيط الفكرة :

إذا كان معك كوب ماء وكوب آخر من الشاي واردت أن تبدل الماء بالشاي والعكس، ستأتي بكوب ثالث فارغ لكي يحتوي على احد النوعين إلى أن تبدل الكوبين الآخرين.

```
using System;
namespace First_program
{
        class Program
```

```
        {
                static void Main( )
                {
                        int num1=10;
                        int num2=15;
                        int temp;
                        temp = num1;
                        num1 = num2;
                        num2 = temp;
                        Console.WriteLine(num1);
                        //num1=15
                        Console.WriteLine(num2);
                        //num2 =10
                }
        }
}
```

تمارين :

أكتب برنامج لحساب المتوسط الحسابي لخمسة أرقام يدخلها المستخدم.

```
using System;
namespace Third_program
{
    class Program
      {
        static void Main( )
         {
            double n1 , n2 , n3 , n4 , n5 , average ;
            Console.WriteLine(" enter five numbers ");
            n1 = double.parse(Console.ReadLine( ) ) ;
```

```
            n2 = double.parse(Console.ReadLine( ) ) ;
            n3 = double.parse(Console.ReadLine( ) ) ;
            n4 = double.parse(Console.ReadLine( ) ) ;
            n5 = double.parse(Console.ReadLine( ) ) ;
            average = ( n1 + n2 + n3 + n4 + n5 ) /5 ;
            Console.WriteLine( average ) ;
        }
    }
}
```

9

الجُمل الشرطية

(Conditional Statements)

if

هي عبارة عن قرارات قائمة على شروط، فإذا تحقق الشرط يُنفذ الكود الذي بداخل جملة التحكم (if).

مثال:

إذا كانت درجتك أكبر من ٥٠% أنت ناجح.

إذا كان تقديرك أكبر من ٨٥% فيكون امتياز.

وهكذا ...

```
if ( الشرط )
        code
```

مثال :

```
if ( grade>=85 )
   Console.WriteLine(" Excellent ") ;
```

نكتب if ثم نفتح قوسين لكتابة الشرط بداخلهما ونكتب بداخلها الكود المراد تنفيذه.

وهي تنفذّ سطر فردي تحتها فقط في حالة عدم وضع قوسين مجموعة.

أما إذا أردنا كتابة أكثر من أمر وأكثر من سطر نكتب الكود بداخل { }.

```
if ( your_name=="Mahmoud")
{
    Console.WriteLine("Hello Mahmoud");
    your_name="Mahmoud"+"Soliman" ;
}
```

يمكننا إجراء وتنفيذ أي كود نريده طالما هذا الشرط صحيح.

في المثال السابق كلمة grade هي عبارة عن متغير ثم الشرط =>

ومعنى الكود أنه إذا كانت درجتك أكبر من أو تساوي ٨٥ سيُنَفَذ الكود الذي بداخل جملة if وفي هذه الحالة سيطبع " Excellent "، فيما عدا ذلك لن يُنفذها.

ثم نكتب الكود بداخل هذين القوسان { }

ويجب كتابة الشرط بداخل هذين القوسان ( )

ونستطيع كتابة أكثر من شرط بداخلها، فمثلاً إذا كنا نريد فحص بيانات شخص كاملة، مثل (اسمه ، عمره ، جنسه ، رقم تليفونه)

نحدد إذا وُجِدَ خطأ في أحد هذه الشروط يكمل تنفيذ الكود أو لا.

وذلك بواسطة :

(&& , || , ^ , !)

المعاملات المنطقية (&&) وتعني (AND) تتحقق من الجانب الأيسر والأيمن إذا وُجد خطأ في احدهما فلن يتم تنفيذ أي منها ويكون ناتجهما خطأ ولتعطي ناتجاً صحيحاً يجب أن يكون الجانبان صحيحين.

المعاملات المنطقية (||) وتعني (OR) تتحقق من الجانب الأيسر والأيمن ولن تعطي خطأ إلا إذا كان الجانبان شرطهما خطأ، إذا كان هناك جانب خطأ وآخر صحيح فستتجاهل الخطأ وتعطي ناتج صحيح لأنه يوجد على الأقل أحدهما صحيح.

المعاملات المنطقية (^) وتعني (XOR) أي أنه لكي يتم تنفيذ الكود الذي بداخل if يجب أن يكون الجانبان مختلفين أي أن أحدهما صحيح والآخر خطأ، إذا كان ناتج الشرط في الجانبين خطأ أو كان ناتج الشرط في الجانبين صحيح فلن يتم تنفيذ الكود أيضاً. يجب أن يكونا مختلفين .

A	B	A & B	A | B	A ^ B
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

جانب أيمن && جانب أيسر

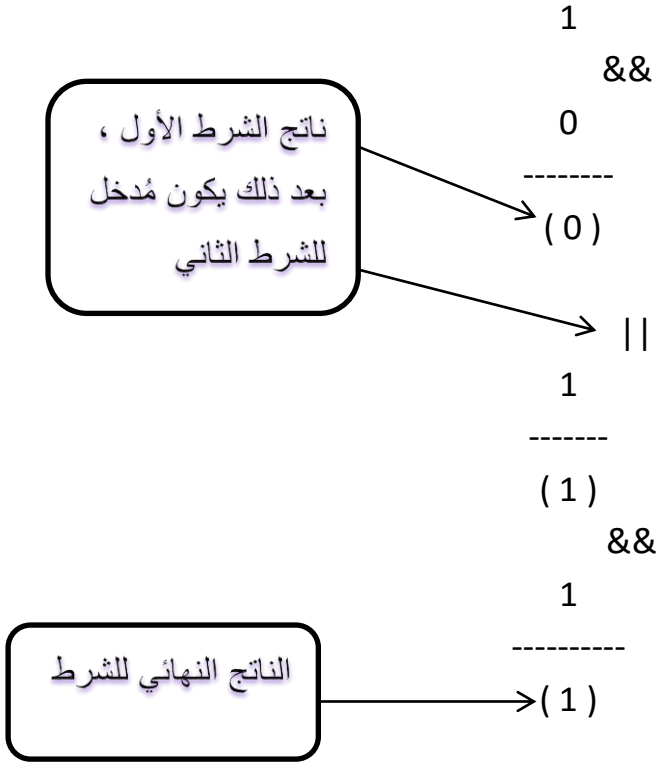
مثال :

```
if ( name == "Mahmoud" && age == 30 || gender == "Male"||
    phone == "00000000" )
```

عندما يوجد أكثر من شرط فإنه يُقسِّم كل شرطين معاً وبعد الإنتهاء من الشرطين يكون ناتجهما مُدخَل للشرط الذي يليه.

على سبيل المثال نفرض أن name قيمته صحيحة و age قيمته أيضاً صحيحة

إذاً فناتج الشرط يكون صحيحاً أما إذا كانت أحدهما خطأ فيكون ناتج الشرط خطأً، ثم هذا الناتج يكون مُدخل للشرط الذي بعده أي يكون يسار الـ (||) صحيحاً ويسارها أياً يكن فيكون ناتج الشرط صحيح لكليهما لأن (||) تحتاج جانب واحد صحيح على الأقل فيكون الناتج النهائي صحيحاً لهذا الشرط، يتبقى آخر فحص وهو الـ phone وناتج الشرط الذي على يساره صحيح وبما أن أداة الشرط (||) فيكون الناتج النهائي لجملة if صحيح فيتم تنفيذ ما بداخلها.



إذا كانت جملة الشرط عبارة عن متغير من النوع (bool) وكتبنا الجملة بدون وضع أي علامات تحدد الشرط وكتبنا المتغير الذي يحمل القيمة فقط فستظل هذه القيمة ثابتة فلو كانت true فإن if دائماً ستُنفَذ، أما إذا كانت false فلن يتم تنفيذ ما بداخل if أبداً.

مثال :

```
bool variable=false;
if( variable )
    {
        Console.WriteLine(" C# ");
    }
```

جملة الشرط تلقائياً تكون بهذا الشكل (variable == true) عندما يكون المتغير من النوع (bool) لو لم نكتب قيمة true أو false في الشرط.

وإذا أردنا النفي في جملة الشرط نضع علامة تعجب قبل المتغير المراد نفي قيمته (!).

```
bool variable = false ;
if( variable )
    {
        Console.WriteLine(" Hello ") ;
    }
```

في هذه الحالة لن يُنَفذ الكود الذي بداخل القوسين لأن ناتج الشرط true والقيمة التي يحملها المتغير في الاصل false.

أما إذا أردت أن يتم تنفيذ الكود في حالة أن المتغير يحمل القيمة false فيجب أن تكون جملة الشرط هكذا...

```
if( ! variable )
```

أو

```
if( variable == false )
```

إذا كتبنا if(true)
{
//code
}
في هذه الحالة سينفذ الكود الذي بداخل if لأننا حددنا أن الشرط صحيح في كل الحالات.

else

تُنَفذّ هذه الجملة عندما يكون الشرط الذي في جملة if خطأ، لكن إن كان الشرط صحيحاً فلن تُنفذ جملة else.

مثال :

```
int num=5;
if( num == 2 )
        Console.WriteLine("this is true");
else
        Console.WriteLine("this is false");
```

في هذه الحالة الشرط خطأ إذاً سينفذ جملة else

إذا أردنا كتابة أكثر من سطر داخلها فيجب وضع قوسين { }.

جُمل شرط متداخلة (Nested if)

يمكننا كتابة أكثر من شرط بداخل بعضهم البعض فيمكننا كتابة أكثر من if بداخل if أو كتابة if بداخل جملة else

مثال:

```
int number = 5 ;
if ( number == 2 )
{
// else الشرط خطأ فلن يُنَفذّ وسينفذ ما بداخل
        Console.WriteLine("this is true");
}
else
        {
                if ( number > 1 )
                {
                        number += 3 ;
                        if ( number < 10 )
                          {
                                number += 5 ;
                          }
                        else
                          {
                                number -= 10 ;
                          }
                }
                Console.WriteLine(number);
        }
// ١٣ سيطبع
```

معنى هذا الكود أنه يفحص قيمة المتغير أولاً فإذا كانت ٢ سيطبع this is true إذا كانت غير ذلك فإنه يدخل إلى else فيجد بداخلها جملة شرط أخرى ويفحص القيمة التي بداخل المتغير هل القيمة أكبر من ١؟
إذا كانت كذلك فيدخل إلى if وتزيد قيمة المتغير بمقدار ٣ ثم يفحص المتغير مرة أخرى هل قيمته أقل من ١٠؟ إذا كانت أقل فينفذّ ما بداخلها وتزيد قيمة المتغير بمقدار ٥ ولو لم تكن قيمته أقل من ١٠ فلن ينفذ جملة if هذه وينزل إلى جملة else وتنقص قيمة المتغير بمقدار ١٠ وبعد ذلك يطبع قيمة المتغير في أول جملة else التي بداخلها كل جُمل الشرط هذه.

if …..else if

يمكننا إختبار أكثر من شرط معاً من خلال else if

```
using System;
namespace First_program
{
    class Program
      {
         static void Main( )
           {
                Console.WriteLine("enter your grade");
                int grade = int.Parse(Console.ReadLine( ));
                if ( grade == 50 )
                    Console.WriteLine(" your grade=50");
                else if ( grade == 60 )
                    Console.WriteLine("your grade=60");
                else if ( grade == 70 )
                    Console.WriteLine("your grade=70");
                else if ( grade == 80 )
                    Console.WriteLine("your grade=80");
                else if ( grade == 90 )
                    Console.WriteLine("your grade=90");
                else
```

```
                    {
                      Console.WriteLine("you didn't success if
                      your grade less than 50 !");
                    }
                }
            }
        }
```

سيفحص أول متغير يُدخله المستخدم فإذا تحقق أول شرط سينفذه ولن ينفذ أي شيء آخر وإذا لم يتحقق سيتجه إلى الشرط الذي يليه ثم الذي يليه إلا أن يتحقق الشرط، إذا كانت كل الشروط وكل الاحتمالات خطأ سيتجه إلى else وينفذ ما بداخلها.

تمارين :

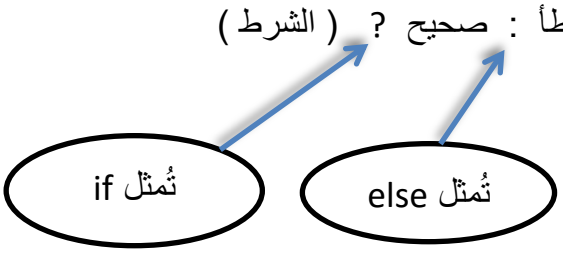
اكتب برنامج يفحص الرقم الذي يُدخله المستخدم ويعرض نوع الرقم إذا كان عدداً زوجياً أم فردياً.

```
using System;
namespace First_program
{
   class Program
    {
        static void Main( )
          {
             int number = int.parse(Console.ReadLine( ));
             if( number%2 == 0 )
                {
                   Console.WriteLine("this number is Even");
                }
```

```
            else
              {
                Console.WriteLine("this number is Odd");
              }
        }
    }
}
```

Inline Condition

يوجد نوع آخر من الجمل الشرطية وتكتب في سطر واحد وتعمل عمل if :



مثال :

```
int x =10 ;

( x > 5 && x <20 ) ? " true" : " false " ;
```

هذا الكود معناه أن في حالة أن x أكبر من ٥ وأقل من ٢٠ أطبع true ما عدا ذلك اطبع false.

ويمكننا إستقبال القيمة الناتجة من هذه العملية في متغير من نفس نوع البيانات الخارجة، في هذا الشرط تكون النواتج من النوع string فيمكننا إستقبالها في متغير نوعه string

فيمكن أن تكون بهذا الشكل :

```
string str = ( x > 5 && x < 20 ) ? " True " : " False " ;

Console.WriteLine( str );
```

الناتج :

True

أيا كان الناتج true أو false سيُخزن بداخل المتغير str.

يمكننا كتابة أي جملة نريد تنفيذها لا يشترط جملة طباعة.

مثال ٢ :

```
int num = 20 ;

num = ( num > 10 && num < 35 ) ? num +5 : num -1 ;

Console.WriteLine( num );
```

الناتج :

25

switch

هي جملة شرطية تُستخدم للتأكد من المُدخلات ما إذا كانت صحيحة أم خاطئة مثل if

لكن هناك بعض الفروق البسيطة بينهما :

Switch	If
لا تستطيع التعامل إلا مع عملية واحدة بداخل الشرط لا نستطيع استخدام == > أو < أو & % بداخلها فهي محددة .	تستطيع التعامل مع أكثر من عملية بداخل الشرط الواحد .
تكون سهلة في عملية القراءة والفهم للكود أكثر من if	تكون صعبة في عمليات القراءة وفهم الكود في حالة الشروط الكثيرة جداً وعند إستخدام كم كبير من الارقام للفحص عنها يُفضل إستخدام switch
أسرع من if في حالة إستخدام الارقام المحددة لانها تتوجه مباشرة إلى الشرط الصحيح وتنفذه .	يفضل إستخدامها في حالة العمليات البسيطة والتي لا تحتوي على أشياء محددة فهي شمولية أكثر ، على كلٍ هي الشائعة في الإستخدام للجُمل الشرطية .

وتكون صيغتها بهذا الشكل ...

```
switch( variable)
{
case قيمة :
        statement;
        break; تستخدم لتوقف الفحص عندما يتحقق
        الشرط
case قيمة :
```

الجملة المراد تنفيذها إذا كان statement;
الشرط صحيح
break;
default:
تنُفذ هذه الجملة في حالة عدم وجود شرط صحيح
تُشبه else في جملة if
statement;
الجملة المراد تنفيذها إذا كان الشرط صحيح
break;
}

مثال :

```
int x = 10 ;
switch( x )
{
 case 5:
        Console.WriteLine(" this is number 5 ");
        break;
case 3:
        Console.WriteLine(" this is number 3 ");
        break;
case 10:
        Console.WriteLine(" this is number 10 ");
        break;
case 2:
        Console.WriteLine(" this is number 2 ");
        break;
default:
```

```
        Console.WriteLine(" false ");
        break;
}
```

يمكننا الإستغناء عن كلمة break في بعض الحالات.

إذا كنا نريد الكشف عن أكثر من قيمة في نفس الحالة، مثلا عند القيمة كذا أو كذا... نفذّ.

مثال:

أكتب برنامج للكشف عن تقدير الطلبة، يُدخله المستخدم ويطبع له التقدير الكلي .

```
int grade=int.parse(Console.ReadLine());
switch(grade)
{
case 50:
        Console.WriteLine("Accepted");
        break;
case 65:
        Console.WriteLine("Good");
        break;
case 76:
        Console.WriteLine("Very Good");
        break;
case 85:// لا يجب كتابة أي كود في هذه الحالة
case 95:
        Console.WriteLine("Excellent");
        break;

default:
        Console.WriteLine("Didn't pass");
```

```
break;

}
```

معنى الكود أن :

في حالة إذا كان التقدير ٥٠ سيطبع مقبول وإذا كان ٨٥ **أو** ٩٥ سيطبع امتياز

هذه الكود غير منطقي ويُفضلّ أن يُكتَب بـ if.

كان هذا للتوضيح فقط.

لكن في جملة الـ default يجب علينا كتابة break.

إذ لم تُكتب في آخر جملة سيحدث خطأ في الكود.

لأن عند عدم كتباتها فهذا معناه أنه في هذه الحالة أو التي تليها "إذا كان الشرط صحيح" نفذّ. فيحدث خطأ لأنه يتوقع وجود حالة بعدها يكشف عنها ، فكيف يُنفذ ولا توجد حالة بعدها؟!

إذاً فلا يجوز تجاهلها ويجب كتابتها في آخر جملة.

10

حلقات التكرار

(Loop Statements)

هي عبارة عن تكرار جزء معين من الكود لعدد معين من المرات، وهي مهمة جداً لانها توفر الوقت والمجهود، تنفيذ هذه الجملة يعتمد على شرط وطالما الشرط يتحقق فهي تُنَفَذّ وتنتهي ثم تبدأ مرة أخرى إلى أن تصل إلى نهاية الشرط، ويتم إستخدامها عن طريق بعض الجُمل التي تستخدم في عملية التكرار.

إحدى أشكال الحلقات هي :

while

وتكون جملة التكرار بالشكل التالي :

```
while ( شرط )
{
    الكود المراد تكراره
}
```

ومعنى الكود أنه طالما أن هذا الشرط يتحقق نفِذّ.

مثال :

```
using System;
namespace First_program
{
    class Program
    {
        static void Main( )
        {
            int i = 1;
            while( i < 10 )
            {
                Console.WriteLine( i );
```

لكن إذا كتبناها بدون عملية الزياده فستظل تعمل بدون توقف لأن الشرط دائما صحيح طالما القيمة لن تتغير، وستكون حلقة لا نهائية .
فيجب أن نزود قيمة المتغير بالمقدار الذي نريده، فمثلا إذا قولنا

++i سيزيد بمقدار ١
i+=2 سيزيد بمقدار ٢
i+=3 سيزيد بمقدار ٣
وهكذا....

```
            }
        }
    }
}
```

لنكتبه بصورة صحيحة :

```
using System;
namespace First_program
{
    class Program
    {
```

```
        static void Main( )
        {
            int i = 1;
            while( i < 10 )
            {
                Console.Write( i ) ;
                i++;
            }
        }
    }
}
```

سيطبع :

1 2 3 4 5 6 7 8 9

ثم سيتوقف عمل الحلقة لأنها وصلت لنهاية الشرط، حيث أن الشرط لن يتحقق لأن ١٠ ليست أقل من ١٠ عندها سيتوقف الشرط وتتوقف حلقة التكرار.

يمكننا وضع الشرط الذي نريده وبالزيادة التي نريد وكتابة أي كود بداخل جملة التكرار وسيتم تنفيذه طالما أن الشرط صحيح.

إذاً تتكون جملة التكرار من :

قيمة إبتدائية للمتغير + while (الشرط) + مقدار الزيادة (العداد).

تمارين :

أكتب برنامج يطبع حروف الهجاء من A : Z

```
using System;
namespace First_program
{
```

```
        class Program
        {
                static void Main( )
                {
                        char alphapet = 'A';
                        while( alphapet<= 'Z' )
                        {
                                Console.WriteLine(alphapet);
                                alphapet++;
                        }
                }
        }
}
```

أو يمكننا كتابتها بهذا الشكل :

```
using System;
namespace First_program
{
        class Program
        {
                static void Main( )
                {
                        char alphapet = 65;
                        while(alphapet<= 90)
                        {
                                Console.WriteLine(alphapet);
                                alphapet++;
                        }
                }
        }
}
```

هذا في حالة إذا أردنا حروف الهجاء في حالة الحروف الكبيرة أما إذا أردناها في حالة الحروف الصغيرة سنبدأ بـ ٩٧ ونضع الشرط أقل من أو يساوي ١٢٢.

سيكون الناتج :

a

b

.

.

z

do….while

وجملة التكرار هذه تختلف عن السابقة في أنها تنفذ الكود الذي بداخلها أولاً بعد ذلك تفحص الشرط وتتأكد ما إذا كان صحيحاً أم خطأً حتى إذا كان الشرط غير صحيح فإنها تنفذ الكود مرة واحدة فقط على الأقل.

وتكون بالشكل التالي:

```
using System;
namespace First_program
{
        class Program
        {
                static void Main( )
                {
                        int i = 1;
                        do
```

```
                    {
                        Console.WriteLine( i );
                        i++;
                    }
                    while( i < 10 ) ;
                }
            }
        }
```

إذاً تتكون جملة التكرار هذه من :

قيمة إبتدائية للمتغير + do + مقدار الزيادة (العداد) + while (الشرط).

تمارين :

أكتب برنامج يأخذ الأرقام والرموز والحروف ويطبع الرقم العشري المقابل لها في الـ ascii.

```
using System ;
namespace ConsoleApplication1
{
   class Program
    {
        static void Main( )
        {
                char check ;
                do
                 {
                        Console.Write( " enter character : " ) ;
                        check = char.Parse( Console.ReadLine( ) ) ;
                        int decim = check ;
                        Console.WriteLine( decim ) ;
                }
                while ( true ) ;
        }
    }
}
```

for

هي شكل من أشكال حلقات التكرار وهي الشائعة في الإستخدام.

وتكون بهذا الشكل :

for (قيمة إبتدائية للعداد ; الشرط ; مقدار الزيادة أو العداد)

وتُكتب الجملة في سطر واحد بخلاف الجمل السابقة التي كانت يُكتب كل جزء منها في مكان مختلف في الكود.

تُستخدم في الجُمل التي نعرف عدد مرات تكرارها، فيفضل إستخدامها.

والفرق بينها وبين while

while تستخدم عندما لا نعرف عدد مرات التكرار.

مثال:

```
using System;
namespace First_program
{
        class Program
        {
                static void Main( )
                {
                        bool question=true;
                        int choice;
                        while(question==true)
                        // ( ==true ) يمكننا الإستغناء عن
                لانها تلقائياً تكون true في الشرط إذا لم نغيرها بـ false
                        {
```

```
                    choice=int.parse(Console.ReadLine());
                        switch(choice)
                        {
                         case 1:
                                statement;
                                break;
                        case 2:
                                statement;
                                break;
                        case 3:
                                statement;
                                break;
                        default:
                        question=false;
                        break;
                        }
                    }
                }
        }
}
```

مثال على for لتوضيح الفرق بينهما

```
using System;
namespace First_program
{
    class Program
        {
            static void Main( )
             {
              for( int number=1 ; number<=10 ; number++ )
                 {
```

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

أكتب برنامج لجمع الارقام من ١ إلى ١٠٠

```
using System;
namespace First_program
{
   class Program
    {
        static void Main( )
          {
            int sum = 0;
            for( int number=1 ; number<=100 ; number++ )
                {
                    sum += number;
                }
            Console.WriteLine( sum );
                }
        }
}
```

سيطبع ٥٠٥٠

ويمكننا إستخدام بداخلها جملة تكرار أخرى، وتكون حلقات متداخلة ويمكننا إستخدام بداخلها جملة if إذا أحتجنا إليها.

يمكننا أيضا كتابة جملة for بهذا الشكل

بشرط أن نكتب القيمة الإبتدائية والشرط والعداد بداخلها ويمكنا كتابة القيمة الإبتدائية فوقها، وإذا لم نكتب شيء وتركناها هكذا فستكون حلقة لا نهائية لا تتوقف.

for(; ;)

مثال:

أكتب برنامج يبحث عن رقم يدخله المستخدم من بين الارقام من ١ إلى ١٠٠٠ إذا وجده يطبع هذا الرقم.

```
using System;
namespace First_program
{
   class Program
     {
        static void Main( )
          {
                int search = int.Parse(Console.ReadLine( ));
                for( int i = 1 ; i<= 1000 ; i++ )
                  {
                     if ( i == search )
                      {
                        Console.WriteLine("the number is "+i);
                      }
                  }
          }
     }
}
```

توجد بعض الكلمات التي تستخدم بداخل جملة التكرار مثل :

break : تستخدم لإيقاف حلقة التكرار.

continue : تستخدم لتخطي بعض الخطوات وتجاهلها وعدم تكرارها.

مثال :

```
using System;
namespace First_program
{
        class Program
        {
                static void Main( )
                {
                        for( int i = 1 ; i<=10 ; i++ )
                        {
                                if( i == 4 )
                                        break;
                                Console.WriteLine( i );
                        }
                }
        }
}
```

عندما تكون قيمة المتغير بـ ٤ سيخرج من حلقة التكرار ولن يكمل وسيطبع فقط...

١

٢

٣

أما إذا كتبنا..

```
using System;
namespace First_program
{
        class Program
        {
                static void Main( )
                {
                        for( int i = 1 ; i<=10 ; i++ )
                        {
                                if( i==4 )
                                        continue;
                                Console.Write( i );
                        }
                }
        }
}
```

هنا سيتجاهل هذه الخطوه وسيتخطاها ويطبع :

1 2 3 5 6 7 8 9 10

إذاً كلمة break تُكتب مع switch وتُكتب أيضاً مع if لكن يجب أن تكون بداخل حلقة تكرار.

Nested for (حلقات تكرار مُتداخلة)

تُكتَب for بداخل for

سنفهم معناها وكيفية عملها بمثال.

مثال :

أطبع جدول الضرب من ١ إلى ١٢

```
using System;
namespace First_program
{
        class Program
        {
                static void Main( )
                {
                        for( int i = 1 ; i<=12 ; i++ )
                        {
                                for( int j = 1; j<=12 ; j++ )
                                {
                                Console.WriteLine( i * j );
                                }
                        }
                }
        }
}
```

في هذا الكود سيدخل بداخل أول for عند قيمة i = 1 بعدها سيجد for أخرى بداخلها أيضاً فيظل ينفذ الكود الذي بداخل for الداخلية إلى أن تنتهي أي يكون الشرط خطأ ثم

يصعد إلى for الأولى يزود قيمة i بمقدار ١ وطالما أن الشرط صحيح فإنه سيظل يكرر هذه العملية إلى أن يكون شرط for الأولى خطأ.

في المثال السابق :

عندما تكون قيمة i=1 سيدخل إلى for الثانية وتكون قيمة j=1 فيطبع ناتج ضرب i , j ثم يزود قيمة j بمقدار ١ فتكون ٢ ويضرب i*j وهكذا إلى أن تكون قيمة j=12 فيطبع ناتج الضرب لأننا تضمنا في الشرط (أقل من أو يساوي) وبعدها تزيد قيمة j=13 فيكون الشرط في هذه الحالة خطأ لأن ١٣ ليست أقل من أو تساوي ١٢.

فتتوقف هذه الحلقة ويصعد إلى for الأولى وتزيد قيمة ثم يدخل ينفذ ١٢ عملية تكرار أخرى ويظل يكرر إلى أن تكون قيمة i = 13 حينها سيكون الشرط خطأ وسيتوقف.

1*1 , 1*2 , 1*3 , 1*4 , 1*5 , 1*6 , 1*7 , 1*8 , 1*9 , 1*10 , 1*11 , 1*12

2*1 , 2*2 , 2*3 , 2*4 , 2*5 , 2*6 , 2*7 , 2*8 , 2*9 , 2*10 , 2*11 , 2*12

وهكذا....

أي أن كل عملية واحدة لأول for تعادلها ١٢ عملية لـ for التي بداخلها في هذا الكود.

أطبع هذا الشكل :

```
*
**
***
****
*****
******
```

سنستخدم nested for لطباعة هذا الشكل.

for الخارجية تعبر عن عدد الصفوف و for التي بداخلها تعبر عن عدد الأعمدة، في هذا المثال يوجد ٦ أعمدة و ٦ صفوف، لا يشترط أن يكون عدد الأعمدة يساوي عدد الصفوف. نلاحظ أن الشكل يبدأ بعمود واحد ثم يزيد بمقدار ١ (" * ") في كل صف جديد.

```
using System;
namespace First_program
{
        class Program
        {
                static void Main( )
                {
                        for( int i = 1 ; i<=6 ; i++ )
                        {
                                for( int j = 1 ; j<= i ; j++ )
                                {
                                        Console.Write( "*" );
                                }
                                Console.WriteLine( );
                        }
                }
        }
}
```

في ذلك المثال حددنا له الشرط في جملة التكرار الثانية ألا تتكرر إلا إذا كان عدد الأعمدة أقل من أو يساوي عدد الصفوف، فمثلاً الصف الأول يطبع نجمة واحدة والصف الثاني يطبع نجمتين وفي الصف الثالث يطبع ثلاث نجوم وهكذا إلى أن تتوقف حلقة التكرار الخارجية.

أثناء الطباعه نجعل الأعمدة تُطبع بجانب بعضها حتى يخرج لي هذا الشكل أي أننا لا نطبع في دالة ( )WriteLine بل نطبع داخل الحلقة الخاصة بالأعمدة بـ ( )Write فقط وعندما يُنهي الشرط في الحلقة الخاصة بالأعمدة يخرج منها فيجد دالة ( )WriteLine كي ينزل إلى صف جديد لأن هذا الصف قد أنتهى.

مثال آخر :

أطبع هذا الشكل .

```
******
*****
****
***
**
*
```

نلاحظ أن عدد الأعمدة مكتمل ثم ينقص بمقدار ١ مع كل صف جديد.

يمكننا كتابة الكود بأكثر من طريقة فمثلاً يمكننا كتابته بهذا الشكل ...

```
using System;
namespace First_program
{
        class Program
        {
                static void Main( )
                {
                        for( int i = 1 ; i<=6 ; i++ )
                        {
```

```
                    for( int j=i; j<= 6 ; j++ )
                    {
                        Console.Write( "*" );
                    }
                    Console.WriteLine( );
                }
            }
        }
```

أو بتلك الطريقة...

```
using System;
namespace First_program
{
    class Program
    {
        static void Main( )
        {
            for( int i = 1 ; i<= 6 ; i++ )
            {
                for( int j = 6 ; j>= i ;  j - - )
                {
                    Console.Write( "*" );
                }
                Console.WriteLine( );
            }
        }
    }
}
```

مثال :

أطبع هذا الشكل ...

```
*
* *
* * *
* * * *
* * * * *
* * * * * *
* * * * * * *
* * * * * * * *
```

في هذا الشكل نحتاج إلى طباعة مسافات على الجانب الأيسر أولاً ونلاحظ أن هذه المسافات تقل بمقدار ١ مع كل صف جديد، إذاً سنطبع مسافات وأعمدة وصفوف، سنحتاج إلى for للصفوف بداخلها حلقتين تكرار أحدهما للمسافات والأخرى للأعمدة.

```
using System;
namespace First_program
{
        class Program
        {
                static void Main( )
                {
                        for( int i = 1 ; i<=8 ; i++ )
                        {
                                for( int k = 1 ; k<=8 –i ; k++ )
                                {
                                        Console.Write( " " );
                                }
                                for( int j = 1 ; j<=i ;  j++ )
                                {
```

```
                    Console.Write( "*" );
                }
                Console.WriteLine( );
            }
        }
    }
}
```

مثال :

أطبع هذا الشكل...

```
          *
         ***
        *****
       *******
      *********
     ***********
    *************
   ***************
```

هو نفس المثال السابق لكننا سنضاعف عدد النجوم فقط .

```
using System;
namespace First_program
{
    class Program
```

```
{
    static void Main( )
    {
        for( int i = 1 ; i<=8 ; i++ )
        {
            for( int k = 1 ; k<=8 - i ; k++ )
            {
                Console.Write( " " );
            }
            for( int j = 1; j<= i*2-1 ; j++ )
            {
                Console.Write( "*" );
            }
            Console.WriteLine( );   /*
*/ لكي ينزل إلى سطر جديد مع نهاية كل صف قد أكتملت دورته
        }
    }
}
}
```

مثال :

أطبع هذا الشكل...

```
using System;
namespace First_program
{
        class Program
        {
                static void Main( )
                {
                        for( int i = 1 ; i<=12 ; i++ )
                        {
                                for( int k = 1 ; k < =12 - i ; k++ )
                                {
                                        Console.Write( " " );
                                }
```

```
                for( int j = 1; j < = i*2-1 ;  j++ )
                {
                        Console.Write( "*" );
                }
                Console.WriteLine( );
        }
        for( int i = 8 ; i<=12 ; i++ )
        {
                for( int k = 1 ; k < =12 - i ; k++ )
                {
                        Console.Write( " " );
                }
                for( int j = 1 ; j< = i*2-1 ;  j++ )
                {
                        Console.Write( "*" );
                }
                Console.WriteLine( );
        }
    }
  }
}
```

في هذا المثال أردنا طباعة مثلث متساوي الساقين لذا ضاعفنا الأعمدة، فجعلنا حلقة التكرار الخاصة بالأعمدة تدور إلى ضعف عدد الصفوف. ثم أنقصنا منها ١ ليكون رأس المثلث نجمة واحدة وليست ٢.

ثم بدأنا من منتصف الصفوف بعدما أنتهينا من المثلث متساوي الساقين لتتكون الشجرة.

هذه الأمثلة فقط لتجعلك متمكن من إستخدام حلقات التكرار بداخل بعضها.

foreach

تستخدم أيضاً للتكرار لكن هذه الجملة تستخدم في حالة وجود مجموعة من البيانات المرتبطة معاً ومُخزنة معاً بداخل مُجَمَّع من البيانات مثل المصفوفة " Array ".

ويكون تعريفها بهذا الشكل :

foreach (نوع البيانات + الكلمة المحجوزة **in** + المصفوفة أو مجموعة البيانات)

```
foreach( DataType variable in data )
{   code ;   }
```

مثال:

```
using System;
namespace First_program
{
        class Program
        {
                static void Main( )
                {
                   int[ ]arr = new int[ 5 ] { 1 , 2 , 3 , 4 , 5 } ;
                   foreach( int i in arr )
                        {
                                Console.WriteLine( i );
                        }
                }
        }
}
```

هذا مجرد مثال للتوضيح فقط وسأشرح المصفوفات لاحقاً.

11

المصفوفات

(Arrays)

(المصفوفة الاحادية أو الفردية)

المصفوفة عبارة عن مجموعة من المتغيرات وتكون من نفس نوع البيانات وتكون متتالية في الذاكرة، أي أنها تحجز خلايا متتالية في الذاكرة المؤقتة (RAM) ويكون عنوان هذه الخلايا متتابع في الذاكرة، ويكون لها حجم ثابت أي أننا يجب أن نحدد عدد العناصر التي سنحتاج استخدامها، فائدة المصفوفة أنها تمكننا من التعامل مع أكثر من عنصر في نفس الوقت ونُعدِلَّ ونغير من قيمته وتحدث عليه أي عملية بطريقة سهلة جداً وبسيطة، يمكننا أن نغير قيمة آلاف البيانات في سطر واحد، وذلك بإستخدام المصفوفة.

تعريف المصفوفة يكون بالشكل التالي :

; [حجم المصفوفة] + نوع البيانات + new = اسم المصفوفة [ ] نوع البيانات .

Data Type + [ ] + Array_Name =new DataType [size];

يجب أن نحدد حجم المصفوفة أثناء التعريف، وكل عنصر هو عبارة عن متغير بداخل المصفوفة.

أُضيفت كلمة new بداخل السي شارب وهي تعبر عن أن المصفوفة ديناميكية أي يجوز تغيير حجمها، فعند تغيير حجمها تأخذ نسخة من المصفوفة القديمة وتنشيء

مصفوفة جديدة بنفس الحجم الذي حددته، حيث أن المصفوفة سابقاً كانت حجمها ثابت غير قابل للتعديل، أما حالياً يجوز تغيير حجمها متى أردنا.

لكي نشير إلى عنصر معين بداخل المصفوفة نكتب اسم المصفوفة وموضع العنصر بداخل المصفوفة، نكتب موضع العنصر بداخل قوسين [ ].

ويبدأ ترقيم المصفوفة من صفر حيث يكون أول عنصر في المصفوفة ترقيمه صفر.

مثال :

```
int [ ] arr = new int [10];
```

ويمكن أن تأخذ حجم المصفوفة من المستخدم بالطريقة العادية.

```
int size = int.Parse(Console.ReadLine( ) ) ;

int [ ] arr = new int[ size ];
```

معني هذه الجملة أننا أنشأنا مصفوفة بداخلها عشرة عناصر (متغيرات) من النوع (int)، ويمكننا إجراء أي عملية عليهم، جمع أو طرح أو قسمة أو ضرب أو أي عملية، يمكننا أن نعطي لعناصر المصفوفة قيم إبتدائية بهذه الطريقة...

```
int [ ] arr = new int [10] { 1 , 4 , 12 , 65 , 14 , 16 } ;
```

أو بهذه الطريقة مباشرةً ..

```
int [ ] arr={ 1 , 4 , 12 , 65 , 14 , 16 } ;
```

سيضع هذه القيم في أول ٦ خلايا من المصفوفة، أما باقي العناصرمن المصفوفة سيضع لها قيمة إفتراضية صفر.

arr[0]	1
arr[1]	4
arr[2]	12
arr[3]	65
arr[4]	14
arr[5]	16
arr[6]	0
arr[7]	0
arr[8]	0
arr[9]	0

دائماً ما يكون موضع آخر عنصر في المصفوفة = حجم المصفوفة - ١

مثال :

إذا أردنا إجراء عملية جمع على العنصر الخامس في المصفوفة ونضيف على قيمة العنصر القيمة ٢٠، إذاً سيكون العنصر رقم ٤ لأن المصفوفة تبدأ بصفر.

فتحدث العملية بالشكل التالي...

```
arr[ 4 ] += 20 ;
```

ويمكننا أيضا إضافة عنصر بداخل المصفوفة إلى عنصر آخر بداخلها

أو نسخ قيمة عنصر إلى عنصر آخر

مثال:

```
arr[2] = arr[1];
```

معنى هذه الجملة أن قيمة العنصر الثاني في المصفوفة نُسِخَت في العنصر الثالث، وأصبحت قيمة العنصر الثالث بنفس قيمة العنصر الثاني.

```
arr[5] += arr[2];
```

في هذه الجملة أُضيفت قيمة العنصر الثالث إلى قيمة العنصر السادس ووُضع ناتج الجمع بداخل العنصر السادس.

```
arr[3] /= arr[4];
```

في هذه الجملة قَسَمَ قيمة العنصر الرابع على العنصر الخامس ووضع ناتج القسمة بداخل العنصر الرابع.

```
int sum = arr[2]+arr[4];
```

جُمِعَت قيمة العنصر الخامس مع قيمة العنصر الثالث ووُضع ناتج الجمع بداخل المتغير sum.

arr[0] موضع العنصر الاول في المصفوفة.

arr[1] موضع العنصر الثاني في المصفوفة.

arr[2] موضع العنصر الثالث في المصفوفة.

وهكذا ...

نستطيع تغيير حجم المصفوفة بهذه الطريقة...

نفترض أننا أنشأنا مصفوفة بهذه الطريقة ونريد تغيير مساحتها :

```
int [ ] numbers = new int [ 10 ];
```

يحدث التعديل بهذا الشكل :

Array.Resize(ref numbers , 20) ;

كلمة Array هي عبارة عن class يحتوي بداخله على دالة تستطيع إعادة تعيين حجم المصفوفة فتنسخ محتوى المصفوفة القديمة وتُدخله في المصفوفة الجديدة، حيث تنشيء مصفوفة جديدة بالحجم الجديد.

Resize : هي دالة التعديل وإعادة تعيين حجم المصفوفة، وهي دالة جاهزة في اللغة.

ref : هي اختصار لـ reference ومعناها أن هذه التغيير سيحدث فعلياً في هذا المكان بداخل الذاكرة (RAM).

نمرر لهذه الدالة اسم المصفوفة والحجم الجديد.

ويتم التعامل مع العناصر داخل المصفوفة من خلال حلقات التكرار for أو foreach أو while.

مثال :

أنشيء مصفوفة مكونة من ١٠ أرقام و ضعْ قيم أولية بداخلها ثم اطبع هذه القيم.

```
using System ;
namespace First_program
{
        class Program
        {
        static void Main( )
        {
        int [ ] num=new int[ 10 ] { 11  ,2 , 3 , 4 , 5 ,
                                    12 , 14 , 16 , 17 , 9 } ;
        for( int counter = 0 ; counter<num.Length ; counter++ )
```

```
            {
                Console.WriteLine( num[ counter ] );
            }
        }
    }
}
```

length : هي خاصية تُرجع أو تحسب لنا حجم المصفوفة، كان من الممكن أن نكتب المساحة بشكل مباشر لكن من المفضل أن نكتب الجملة بهذا الشكل حتى إذا تغيرت المساحة فلن نضطر إلى تغييرها في كل الشروط المتعلقة بهذه المصفوفة، فهذه الخاصية تحسبها تلقائياً. وسنعرف ماهي الخصائص فيما بعد.

أو يمكننا إستخدام foreach فهي أبسط في الكتابة من for وتأتي بنفس النتيجة.

```
foreach ( int count in num)

    Console.WriteLine( count ) ;
```

أنشأنا عداد لكي يدور على عناصر المصفوفة، ثم يطبع محتوى المصفوفة.

مثال :

أنشيء مصفوفة مكونة من 15 عنصر وأجعل المستخدم يدخل قيم لهذه العناصر، ثم أضف القيمة 5 إلى قيمة العنصر العاشر وأطبع كل قيم العناصر.

```
using System;
namespace First_program
{
    class Program
      {
        static void Main( )
          {
```

```
            int [ ] num = new int[15];
            for( int counter = 0 ; counter<num.Length ; counter++ )
              {
                    num[counter]=int.Parse(Console.ReadLine));
              }
            for( int counter = 0 ; counter<num.Length ; counter++ )
             {
                    if ( num[counter]==10 )
                          num[counter]+=5;
             }
            for( int counter = 0 ; counter<num.Length ; counter++ )
             {
                 Console.WriteLine( num[counter] );
             }
        }
    }
}
```

ملاحظة— لا يجوز أن تبدأ العداد بصفر وأن تجعله يزيد بمقدار ١ ثم تكتب في الشرط أقل من أو يساوي (قيمة حجم المصفوفة)، لانك إذا كتبت أو يساوي فأنت تقول أن المصفوفة تحتوي على عدد العناصر الموجودة +١ وفي هذه الحالة سيحدث خطأ لأنك إذا بدأت مصفوفة مكونة من ١٠ عناصر بصفر وانتهيت بالقيمة ١٠ سيكون عدد العناصر ١١ وليس ١٠ وسيحدث خطأ اثناء وقت التشغيل، يجب أن تبدأ بصفر وتنتهي بـ ٩

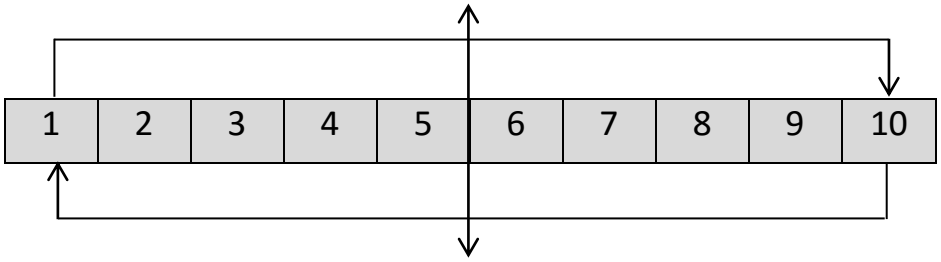
counter < = num.Length----------**x** (خطأ)

تمارين :

ادخل ١٠ عناصر في مصفوفة ثم اعكس هذه المصفوفة، أي أن تجعل العنصر الأخير يكون في مقدمة المصفوفة والعنصر الأول ينتقل إلى آخرها.

```
using System;

namespace First_program

{

   class Program

    {

        static void Main( )

          {

              int [ ] arr=new int[10];

              for( int counter = 0 ; counter<arr.Length ; counter++ )

                 {
                   arr[counter]=int.Parse(Console.ReadLine));
                 }

             for( int counter = 0 ; counter<arr.Length/2 ; counter++ )

                 {
                         int temp;
                         temp = arr[counter];

                         arr[counter] = arr[arr.Length-1-counter];

                         arr[arr.length-1-counter] = temp;
                 }
          }
     }
}
```

فكرة البرنامج أننا سنجري عملية تبديل بين كل عنصرين متقابلين، وسنظل نُجري هذه العملية إلى أن نصل إلى منتصف المصفوفة، أي أنه ستحدث عملية التبديل 5 مرات وليس 10 لأننا لو أجرينا عملية التبديل 10 مرات سترجع المصفوفة إلى ما كانت عليه في السابق.



١ مع ١٠

٢ مع ٩

٣ مع ٨

٤ مع ٧

٥ مع ٦

لو جعلنا الـ for تدور إلي ما بعد منتصف المصفوفة فإنها ستبدلهم مرة أخرى.

لكن ماذا لو كان طول المصفوفة عدداً فردياً مثل 11؟

لن يؤثر ذلك في شيء لأن النوع int لا يقبل إلا أعداد صحيحة فخارج القسمة سيكون عدد صحيح وسيتجاهل الاعداد العشرية فناتج قسمة 12 على 2 يساوي 5، وسيكون في منتصف المصفوفة رقم على يمينه 5 أرقام وعلى يساره 5 أرقام، سنبدل اليمين باليسار والمنتصف لن يتأثر لأنه في كل الحالات سيظل في المنتصف.

مثال :

٣ ٢ ١ ٠ -١ -٢ -٣

إذا أردت أن تبدل الثلاثة أرقام التي على يمين الصفر بالثلاثة التي على يساره، هل سيتأثر مكان الصفر؟

الجواب لا، نفس الفكرة لن يتأثر مكان العنصر الذي في منتصف المصفوفة ذات الحجم الفردي.

وأنشأنا متغير ليحمل القيمة مؤقتاً حتى تتم عملية التبديل ووضعنا به أول عنصر ثم وضعنا آخر عنصر في مكان أول عنصر، بعدها وضعنا القيمة التي بداخل المتغير المؤقت في مكان آخر عنصر وهكذا.

مثال ٢:

انشيء مصفوفة و أدخل بها ١٠ عناصر ثم رتب العناصر تصاعدياً.

```
using System;
namespace First_program
{
        class Program
        {
                static void Main( )
                {
                        int [ ] arr = new int[ 10 ] ;
                        for( int i= 0 ; i<arr.Length ; i++ )
                        {
                        arr[i]=int.Parse(Console.ReadLine));
                        }
                        for( int i = 0 ; i <arr.Length-1 ; i++)
                        {
                                for( int j = i ; j <arr.Length ; j++)
```

```
                    {
                        if( arr[ i ] > arr[ j ] )
                        {
                            int temp;
                            temp = arr[ i ] ;
                            arr[ i ] = arr[ j ] ;
                            arr[ j ] = temp ;
                        }
                    }
                }
            }
        }
    }
```

فكرة هذا البرنامج أننا نقارن كل عنصر بالذي يليه فإذا كان ما يليه أقل منه فإنه سيبدِل القيمتين ببعضهما.

وكما عرفنا، تتم عملية التبديل بين قيمتين عن طريق وجود متغير ثالث يحمل قيمة أحدهما حتى تتم عملية التبديل.

تمارين :

أكتب كود يجعل المستخدم يبدل بين قيمتين من خلال فهرس هذه القيم في المصفوفة، المصفوفة مكونة من ١٠ عناصر.

```
using System ;

namespace ConsoleApplication

{

  class Program
```

```
{
    static void Main ( )
    {
        int [ ] arr = new int [ 10 ] { 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 } ;
        int index1 = int.Prase( Console.ReadLine( ) );
        int index2 = int.Parse( Console.ReadLine( ) );
        int temp ;
        for ( int i = 0 ; i < 10 ; i++ )
        {
            for ( int j = i ; j < 10 ; j++ )
            {
                if ( i == index1 && j == index2 )
                {
                    temp = arr [ index1 ] ;
                    arr [ index1 ] = arr [ index2 ] ;
                    arr [ index2 ] = temp ;
                }
            }
        }
    }
}
}
```

المصفوفة ثنائية الابعاد

هي شكل من أشكال المصفوفات ويكون لها بُعدين ويتم التخزين بها عن طريق إستخدام رقم الصف والعمود.

يجب أن تحدد عدد الأعمدة والصفوف التي بها فمساحتها تساوي عدد الصفوف في عدد الأعمدة.

مثال :

	عمود 0	عمود 1	عمود 2
صف 0	12	54	6
صف 1	76	23	3
صف 2	4	1	8

هذه المصفوفة تحتوي على ٣ صفوف و٣ أعمدة وأيضا يبدأ ترقيم الصف أو العمود من صفر.

وتتم عملية تعريف المصفوفة بهذا الشكل:

```
DataType + [ , ] + name = new DataType [ rows , cols ];
```

```
int [ , ] arr = new int [ 5 , 4 ] ;
```

معنى هذا الكود أننا أنشأنا مصفوفة ثنائية اسمها arr وبها ٥ صفوف و ٤ أعمدة.

فمثلاً إذا أردنا تخزين قيمة في مكان محدد في المصفوفة فإننا نحدد مكان الصف والعمود.

مثال :

في حالة تخزين القيمة ١٠٠ بداخل هذه المصفوفة، نحدد بأي صف و بأي عمود وبتلاقي الصف مع العمود نضع القيمة، تماما كالرسم البياني للمحورين (س ، ص).

لنحدد مثلا الصف الثالث والعمود الثاني.

فتكتب بهذا الشكل :

```
arr[ 2 , 1 ] = 100 ;
```

	عمود 0	عمود ١	عمود ٢	عمود 3
صف 0				
صف ١				
صف ٢		**100**		
صف 3				
صف 4				

تتم عملية الإدخال لهذه المصفوفة بتلك الطريقة :

سنفرض أن مساحة المصفوفة هي ٤ صفوف و٣ أعمدة وسنحفظ بداخلها أسماء الطلاب.

```
using System;
namespace First_program
{
        class Program
        {
                static void Main( )
                {
                        string [ , ] arr = new string [ 4 , 3 ] ;

                        for( int i = 0 ; i<4 ; i++ )
                        {
                                for( int j = 0 ; j<3 ; j++ )
                                {
                                arr[ i , j ]= Console.ReadLine( );
                                }
                        }
                }
        }
}
```

في عملية تعريف المصفوفة يجب أن نحدد لها عدد الصفوف والأعمدة أو أن نأخذ قيمتهما من المستخدم.

مثال :

```
using System;
namespace First_program
{
        class Program
        {
          static void Main( )
           {
              int row = int.Parse( Console.ReadLine( ) ) ;
             int col = int.Parse( Console.ReadLine( ) ) ;
```

```
                string [ , ] arr=new string[ row , col ];
                for( int i = 0 ; i< row ; i++ )
                    {
                        for( int j =0 ; j< col ; j++ )
                            {
                                arr[ i , j ] = Console.ReadLine( );
                            }
                    }
            }
        }
}
```

تمارين :

أنشيء مصفوفة ٣ * ٣ لتخزين مرتبات موظفين ثم أطبع أعلى مرتب وأقل مرتب.

```
using System;
namespace First_program
{
   class Program
     {
         static void Main( )
           {
              int row=3;
              int col=3;
              int max_value = 0 ;
              int mini_value ;
              int [ , ] arr = new int[ row , col ];
              for( int i = 0 ; i< row ; i++ )
                  {
                    for( int j = 0 ; j< col ; j++ )
```

```
            {
              arr[ i , j ]=int.Parse(Console.ReadLine());
            }
        }
    mini_value = arr[ 0 , 0 ];

        فرضاً أن أول قيمة في المصفوفة هي أقل قيمة //

    for( int i = 0 ; i< row ; i++ )
        {
           for( int j = 0 ; j< col ; j++ )
             {
                 if ( arr[ i , j ] > max_value)
                   {
                          max_value = arr[ i , j ] ;
                    }
                 if ( arr[ i , j ] < mini_value)
                 {
                          mini_value = arr[ i , j ] ;
                 }
           }
        }
Console.WriteLine("the maximum salary="+max_value);
Console.WriteLine("the minimum salary="+mini_value);
```

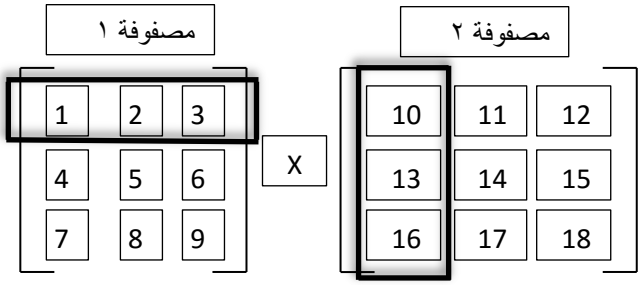
```
            }
        }
}
```

تمارين :

أنشيء مصفوفتين مساحتهما ٣ * ٣ ثم أضرب المصفوفتين في بعضهما وخزن ناتج الضرب في مصفوفة جديدة.

ملاحظة :

عند ضرب مصفوفتين فإن كل صف في المصفوفة الأولى (١) يُضرب في كل أعمدة المصفوفة الثانية (٢).



فتُضرَب المصفوفتان بهذه الطريقة :

1*10+2*13+3*16	1*11+2*14+3*17	1*12+2*15+3*18
4*10+5*13+6*16	4*11+5*14+6*17	4*12+5*15+6*18
7*10+8*13+9*16	7*11+8*14+9*17	7*12+8*15+9*18

```
using System;
namespace ConsoleApplication1
{
   class Program
   {
      static void Main( )
      {
        int multi = 1 , temp = 0 ;
        int [ , ]arr = new int[ 3 , 3 ] ;
        int [ , ]arr2 = new int[ 3 , 3 ] ;
        int [ , ]arr3 = new int[ 3 , 3 ] ;
        Console.WriteLine("---------Enter values of array 1--------");
         for( int i = 0 ; i<3 ; i++ )
          {
              for( int j = 0 ; j<3 ; j++ )
               {
                    Console.Write(" [ " + i + " , " + j + " ] = ");
                    arr[ i ,  j ] = int.Parse(Console.ReadLine( ));
               }
          }
        Console.WriteLine("---------Enter values of array 2--------");
         for( int i = 0 ; i<3 ; i++ )
          {
            for( int j = 0 ; j<3 ; j++ )
              {
                   Console.Write(" [ " + i + " , " + j + " ] = ");
                   arr2[ i ,  j ] = int.Parse(Console.ReadLine( ));
              }
           }
         for( int k = 0 ; k<3 ; k++ )
          {
             for( int i = 0 ; i<3 ; i++ )
              {
                 for( int j = 0 ; j<3 ; j++ )
                  {
```

```
                multi = arr[ k , j ] * arr2[ j , i ];
                temp += multi;
              }
            arr3[ k , i ] = temp;
            temp = 0 ;
          }
        }
      Console.WriteLine(" multiplication Result of array1 and
                          array2 : ");
        for( int i = 0 ; i<3 ; i++)
        {
            for( int j = 0 ; j<3 ; j++ )
            {
                Console.Write( arr3[ i , j ] + "\t" );
            }
            Console.WriteLine( );
        }
    }
  }
}
```

تمارين :

أكتب برنامج بإستخدام المصفوفات لتنفيذ لعبة **X O**.

X-O

X	**X**	**O**
X	**O**	**X**
O	**X**	**O**

```
using System;
namespace ConsoleApplication7
{
   class Program
   {
      static void Main(string[] args)
      {
         const char player1 = 'x', player2 = 'o', empty = ' ';
         char[,] game = new char[3, 3];
         char winner = ' ';
         int pos1;
         int pos2;
         bool choice = true ;
         for (int i = 0; i < 3; i++)
         {
            for (int j = 0; j < 3; j++)
            {
               if (choice == true)
               {
                  Console.WriteLine(" X Playing ");
                  Console.WriteLine(" enter your positions ");
                  pos1 = int.Parse(Console.ReadLine());
                  pos2 = int.Parse(Console.ReadLine());
                  if (pos1 > 2 || pos2 > 2)
                  {
                     Console.WriteLine("\t \t wrong position");
                     j--;
                  }
                  else if (game[pos1, pos2] == 'o' || game[pos1, pos2] =='x')
                  {
```

```
            Console.WriteLine("\t \t wrong position");
            j--;
        }
        else
        {
            game[pos1, pos2] = 'x';
            Console.WriteLine();
            choice = false ;
        }
    }
    else
    {
        Console.WriteLine(" O Playing ");
        Console.WriteLine(" enter your positions ");
        pos1 = int.Parse(Console.ReadLine());
        pos2 = int.Parse(Console.ReadLine());
        if (pos1 > 2 || pos2 > 2)
        {
            Console.WriteLine("\t \t wrong position");
            j--;
        }
        else if (game[pos1, pos2] == 'x' || game[pos1, pos2] =='o')
        {
            Console.WriteLine("\t \t wrong position");
            j--;
        }
        else
        {
            game[pos1, pos2] = 'o';
            Console.WriteLine( );
```

```
            choice = true ;
        }
    }
    for (int a = 0; a < 3; a++)
    {
        for (int b = 0; b < 3; b++)
        {
            switch (game[a, b])
            {
                case player1:
                    Console.Write("X");
                    break;
                case player2: Console.Write("O");
                    break;
                case empty:
                    Console.Write(" ");
                    break;
            }
            if (b < 2)
                Console.Write(" | ");
        }
        if (a < 2)
        {
            Console.WriteLine();
            Console.WriteLine(" ------");
        }
    }
    Console.WriteLine(); Console.WriteLine();
    for(int k=0;k<3;k++)
                {
```

```
for(int g=0;g<1;g++)
{
  if (game[k , g]=='x' && game[k , g+1] == 'x'
        && game[k , g+2] == 'x')
      {
        Console.WriteLine("\t \t X Wins ");
        winner = player1;
        break;
      }
else if (game[g , k] == 'x' && game[g+1 , k]== 'x'
     && game[g+2 , k] == 'x')
      {
        Console.WriteLine("\t \t X Wins ");
        winner = player1;
        break;
      }
  else if(game[0,0]=='x' && game[1,1]=='x' &&
  game[2,2]=='x')
      {
        Console.WriteLine("\t \t X Wins ");
        winner = player1;
         break;
      }
      else if(game[2,0]=='x' &&
         game[1,1]=='x' && game[0,2]=='x')
      {
        Console.WriteLine("\t \t X Wins ");
        winner = player1;
        break;
      }
```

```
        }
        if (winner == player1 || winner == player2)
            break;
}
for(int k=0;k<3;k++)
  {
      for(int g=0;g<1;g++)
      {
          if(game[k,g]=='o' && game[k,g+1]=='o'
                  && game[k,g+2]=='o')
          {
             Console.WriteLine("\t \t O Wins ");
             winner = player1;
             break;
          }
          else if(game[g,k]=='o' &&
        game[g+1,k]=='o' && game[g+2,k]=='o')
          {
             Console.WriteLine("\t \t O Wins ");
             winner = player1;
             break;
          }
        else if(game[0,0]=='o' && game[1,1]=='o'
             && game[2,2]=='o')
          {
             Console.WriteLine("\t \t O Wins ");
             winner = player1;
             break;
          }
        else if(game[2,0]=='o' && game[1,1]=='o'
```

```
                                && game[0,2]=='o')
                            {
                                Console.WriteLine("\t \t O Wins ");
                                winner = player1;
                                break;
                            }
                        }
                        if (winner == player1 || winner == player2)
                            break;
                    }
                if (winner == player1 || winner == player2)
                    break;
    }
    if (winner == player1 || winner == player2)
        break;
}
if (winner != player1 && winner != player2)
    Console.WriteLine("\t \t Draw ");
Console.WriteLine(); Console.WriteLine();

 for (int i = 0; i < 3; i++)
    {
       for (int j = 0; j < 3; j++)
       {
          switch (game[i, j])
          {
             case player1:
                Console.Write("X");
                break;
             case player2:
```

```
                Console.Write("o");
                break;
            case empty:
                Console.Write(" ");
                break;
          }
          if (j < 2)
            Console.Write(" | ");
        }
        if (i < 2)
        {
          Console.WriteLine( );
          Console.WriteLine("------");
        }
      }
      Console.WriteLine( );
    }
  }
}
```

يُمكن تنفيذ اللعبة بشكل أبسط من هذا لكن لم نتطرق لهذه الأجزاء إلى الآن لذا فهذا كان مجرد تطبيق على المصفوفات فكل ما يهمنا هو إتقانها أكثر ومعرفة كيفية عملها.

12

النصوص

(Strings)

النصوص هي عبارة عن متسلسلة أو مصفوفة من الحروف، فيمكنك فعل ما تشاء في النص من حذف أو إضافة أو إقتصاص.
فيمكنك تعريف النص بهذه الطريقة :

```
string var_name = " value ";
```

القيمة أو النص يجب أن يُوضع بين علامتين تنصيص وبداخلها يمكنك أن تكتب أي شيء حتى إذا كانت أرقام أو أو مسافة خالية أو نص مدمج مع رقم فإنه سيتعامل معها على أنها نص لأنها وُضِعَت بين علامتين تنصيص.
مثل :

```
string name = "123    mahmoud";
Console.WriteLine( name );
```

الناتج :

```
123   mahmoud
```

ويُمكنك أن تضيف أي قيمة من النوع string أو char على قيمة المتغير الذي يكون من النوع string

```
string name1 = "mahmoud" , name2="soliman", name3;
name3 = name1 + name2;
```

أو يمكننا أن نضيف النص أو الحرف الجديد على القديم بطريقة مباشرة.

مثال :

```
string name ="Mahmoud";
name +=" Soliman";
Console.WriteLine( name );
```

الناتج :

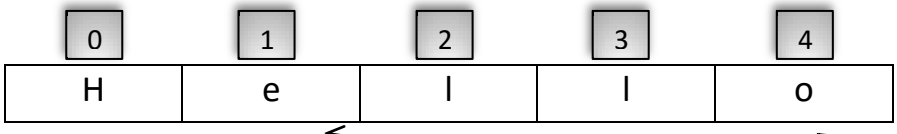
Mahmoud Soliman

ويُمكنك أن تأخذ القيم من المستخدم بالطريقة العادية.

أول حرف في النص يأخذ الترقيم (index) 0 كما في المصفوفة لأنه كما قلت هو عبارة عن مصفوفة من الحروف.

مثال :

```
string x = "Hello";
```




```
Console.WriteLine(" the result {0} {1}  ", x [ 1 ] , x [ 4 ] );
```

الناتج :

e o

- لمعرفة عدد حروف نص معين نستخدم الخاصية (Length).

```
Console.WriteLine( x.Length );
```

الناتج :

5

مثال :
أبحث عن الحرف m في نص يُدخله المستخدم.

```
static void Main( )
{
        string search = Console.ReadLine( );
        bool check = false;
        for (int i = 0; i < search.Length; i++)
        {
            if ( search[ i ] == 'm' )
                    check = true;
        }
        if (check == true)
              Console.WriteLine("exist ");
        else
              Console.WriteLine("not exist");
}
```

في هذا الكود عرفنا متغير نوعه string ثم أخذنا قيمته من المستخدم، وعرفنا متغير نوعه bool يأخذ القيمة true أو flase وقد مررت له قيمة إبتدائية false أي أن الحرف الذي نبحث عنه غير موجود إلا أن نجده، ثم أنشأنا حلقة بإستخدام for وعداد يبدأ من القيمة صفر إلى أقل من القيمة search.Length والتي تُمثل حجم النص أو عدد حروفه. من الممكن أن تسأل نفسك لماذا لا نكتب عدد حروف النص بشكل مباشر؟! ببساطة لأنه في بعض البرامج لا نعرف عدد حروف النص لأنها تأتي من المُستخدم فلا يُمكن التوقع بحجم النص، ويُفضلّ دائماً إستخدام هذه الخاصية في حالة المصفوفات.

وبداخل الحلقة for نستخدم شرط أنه إذا وُجِدَ الحرف الذي نبحث عنه نجعل قيمة المتغير check = true

ولو لم يجده فتظل قيمته بـ false كما كانت وهذه فائدة أن تمرر قيمة إبتدائية بـ false.

وبعد أن تنتهي الحلقة نستخدم شرط آخر لنتأكد هل تغيرت قيمة check أي هل الحرف الذي نبحث عنه موجود أم لا.

مثال :

استقبل نص من المستخدم ثم أعكس هذا النص.

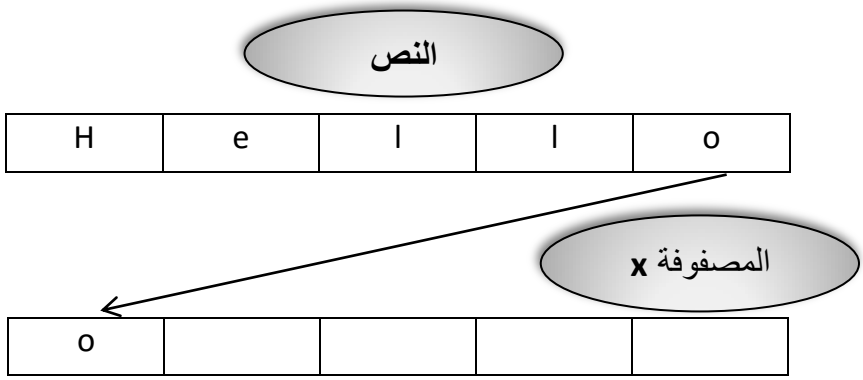
Hello ------> olleH

```
static void Main( )
{
        string str = Console.ReadLine( );
       string str2 = string.Empty; ;
        char[ ] x = new char[ str.Length ];
        for (int i = 0; i < str.Length; i++)
          {
             x[ i ] = str[ str.Length - 1 - i ];
             str2 += x[ i ];
          }
        Console.WriteLine( str2 );
}
```

أخذنا القيمة من المستخدم ثم أنشأنا مصفوفة نوعها char أي من النوع حرف ويكون حجمها نفس حجم النص ولذلك أستخدمنا الخاصية str.Length لكي ترجع لنا حجم النص، ثم أنشأنا حلقة وجعلنا كل حرف من نهاية النص يُخزَن في بداية المصفوفة.
وأنشأنا متغير آخر نوعه string لكي نُعيد فيه تخزين الحروف التي تأتي من المصفوفة.

ملاحظة :

string.empty هذه لتفريغ المتغير من أي قيم موجودة بهذا المكان مُسبقاً.



أو يمكننا الإستغناء عن المصفوفة التي من النوع char لكني تعمدّتُ كتابتها بهذه الطريقة لتعرف طريقة التعامل مع النص والمصفوفة التي من النوع char وأنهما متماثلان.

```
static void Main( )
{
        string str = Console.ReadLine( );
       string str2 = string.Empty; ;
        for (int i = 0; i < str.Length; i++)
              str2 += str[ str.Length - 1 - i ];
        Console.WriteLine( str2 );
}
```

تمارين :

أمامك مصفوفة مكونة من [1, 4, 5, 8, 8, 9, 9, 0] أطبع أكبر عدد مكون من رقمين متجاورين.

في هذا المثال يكون الناتج 99

```
static void Main( )
    {
      int[ ] arr = { 1, 4, 5, 8, 8, 9, 9, 0 };
      int max = arr[0];
```

```
    string res = string.Empty;
    int num = 0;
    for (int i = 0; i < arr.Length - 1; i++)
    {
       res = arr[ i ].ToString( ) + arr[ i + 1 ].ToString( );
       num = Convert.ToInt32( res );
       if (num > max)
           max = num;
    }
    Console.WriteLine(max);
}
```

في هذا الكود عرّفنا المتغير max نوعه int لكي يحمل أكبر قيمة موجودة، ثم عرّفنا المتغير res ونوعه string لكي يحمل كل عنصرين متجاورين لأننا سنضطر إلى دمج كل عنصرين متجاورين ولا يمكن الدمج إلا في النصوص لذا سنستقبل كل متغير والذي يليه ثم سنحولهم إلى نصوص من خلال الدالة ( )ToString وبذلك يتحول الرقمين إلى نص ثم ندمجهم ونخزنهم في المتغير res، بعد ذلك نُحولّ النص المكون من الرقمين المدمجين إلى أعداد صحيحة مرة أخرى لكي نستطيع مقارنتهم بألاعداد الأخرى، لأنه لا يمكن مقارنة نص بعدد صحيح.
توجد الكثير من الخصائص والدوال الجاهزة الخاصة بالنصوص.

Length	خاصية تحسب عدد حروف النص
string.Concat( )	لدمج أكثر من نص مع بعضهم
Contain( )	للبحث عن مقطع معين داخل النص وتُعيد true أو false
EndsWith( )	للتأكد من نهاية النص هل ينتهي بمقطع أو حرف معين انت تحدده أم لا فهي تعيد true أو false
StrartsWith( )	للتأكد من بداية النص هل يبدأ بمقطع أو حرف معين أنت تحدده أم لا وتعيد true أو false
ToUpper( )	لتحويل النص إلى حالة الحروف الكبيرة (capital)

ToLower( )	لتحويل النص إلى حالة الحروف الصغيرة (small)
Substring( )	لقطع حرف أو أكثر من النص
Replace( )	لإستبدال حرف أو مقطع في النص بقيمة جديدة

ويوجد الكثير من الدوال الأخرى الخاصة بالنصوص.

مثال :

```
string x = "C# is a programming language" ;
string y = " and it's a high level language";
Console.WriteLine( x.Length );   ------> 28
Console.WriteLine( string.Concat( x , y ) );
C# is a programming language and it's a high level language
Console.WriteLine(x.EndsWith("uage")); ------> true
Console.WriteLine(x.StratsWith("is")); ------> false
Console.WriteLine(x.ToUpper());
C# IS A PROGRAMMING LANGUEAGE

Console.WriteLine(x.ToLower());
c# is a programming language

Console.WriteLine( x.Substring( 6 , 13 ) );
```

في هذه الدالة نحدد بداية ونهاية المقطع الذي نريده من النص.

```
a programming
```

أو يمكننا تحديد البداية فقط وتكون نقطة النهاية هي نهاية النص.

```
Console.WriteLine( x.Substring( 6 ) );
a programming language

Console.WriteLine(x.Replace("pro","PRO"));
C# is a PROgramming language
```

مثال :

أكتب برنامج يحوِل النص إلى النظام الثنائي (Binary System).

```
using System;
namespace Test
{
   class Program
   {
      static void Main( )
      {
         string str = "Mahmoud soliman";
         string result ;
         int bin = 0;
         Console.WriteLine( str );
         for (int i = 0; i < str.Length; i++)
         {
            result = string.Empty;
            bin = str[ i ];
            Console.Write(str[ i ] + " = " + bin + " ---> ");
            while ( bin != 0 )
            {
               result = ( bin % 2 ).ToString( ) + result;
               bin /= 2;
            }
            Console.WriteLine(result + "  ");
         }
      }
   }
}
```

في هذه العملية المتغير bin يأخذ القيمة العددية للحرف المحدد من النص.

عملية تحويل الرقم إلى النظام الثنائي من خلال أخذ **باقي** القسمة له على 2 ثم قسمته على 2

كل حرف يكون له قيمة عددية صحيحة مناظرة له كما ذكرت في مقدمة الكتاب والنص هو عبارة عن مجموعة من الحروف حتى المسافة الفارغة تُعد حرفاً فإننا نأخذ القيمة العددية لكل حرف ونحولها إلى النظام الثنائي ونطبع قيمة أول حرف بالنظام الثنائي ثم نفرغ المتغير الذي نجمع فيه القيم التي تكون بالنظام الثنائي ليكون مُهيَّأ لإستقبال قيمة الحرف الذي يليه.

13

البرمجة كائنية التوجه

(Object Oriented Programming)

(OOP)

البرمجة كائنية التوجه هي عبارة عن طريقة لكتابة الكود لكنها جعلت الكود أسهل وأختصرت فيه كثيراً ووفرت من الوقت والمجهود بشكل كبير جداً فهي تتعامل مع كل شيء على أنه كائن وهذا الكائن له صفات ووظائف فمثلاً الإنسان هو كائن له صفات تميزه ووظائف يقوم بها.

فمثلاً من صفاته (لون البشرة : لون العين : الطول :)

ومن أفعاله (يأكل : يمشي : ينام :)

وهكذا....

إذاً فكل كائن من نفس النوع يمتلك نفس اسم الصفات والأفعال لكن بتفاصيل مختلفة، أي تختلف بيانات هذه الصفات والأفعال لأن لكل كائن صفاته وأفعاله الخاصة به.

فالبرمجة كائنية التوجه قامت على هذا الأساس، فمثلاً إذا كنت تريد إنشاء حساب على موقع الفيس بوك، عند إدخال بياناتك هل يكتب لك مصمم الموقع كود برمجي خاص بصفحتك عند إنشائك لحساب على الموقع؟ بالطبع لا، عندما تنشيء حساباً لك فإنك تأخذ نسخة من الموقع الأصلي لكن البيانات مجردة أي بدون بيانات وتملأها أنت ببياناتك الخاصة،

في ذلك الوقت أنت أخذت كائن (نسخة) من الموقع وأدخلت فيه بياناتك أنت، وشكل الموقع ووظائفه متشابهه تماماً عند كل الناس والبيانات المجردة أيضا متشابهه عند كل الناس فمثلاً هو يطلب منك تحديد الجنس والعمر وهكذا وأنت تدخل بياناتك، هذه هي البيانات المجردة أي أنها لا تزال لا تحتوي على قيم، وأي شخص جديد يريد إنشاء حساب فإنه سيأخذ نفس النسخة من الموقع ويملأها ببياناته الخاصة.

هذه مجرد أمثلة تشبيهية فقط.

وهذه هي الطريقة التي بنيت عليها لغة السي شارب كما قولنا في مقدمة الكتاب.

إذاً أين يكتب الكود الذي يأخذ منه الكائن نسخة من البيانات التي بداخله؟

يكتب بداخل class والـ class هو البنية التحتية للبرمجة كائنية التوجه.

وتكمن قوة الـ class في عملية تسمى عملية تغليف البيانات، أي يحدد من يستطيع الوصول إلى البيانات والتعامل معها.

وسنتحدث عن كل شيء يخص البرمجة كائنية التوجه والـ class. لكن أولا سنتحدث عن طرق الوصول للبيانات أو التغليف.

14

التغليف

(Encapsulation)

توجد ٥ أنواع لتغليف البيانات في لغة السي شارب، أو تسمى طرق الوصول للبيانات.

قبل البداية في شرح هذه الأنواع سأُذكرّكم أن المشروع أو البرنامج هو عبارة عن مجموعة من namespaces والـ namespaces عبارة عن مجموعة من الـ classes.

يمكننا إنشاء أكثر من namespace في المشروع ويمكننا إنشاء أكثر من class بداخل نفس الـ namespace، ويكون بداخل كل class الخصائص والدوال (الصفات والوظائف) الخاصة به.

في هذه الحالة نحتاج إلى حماية بعض البيانات التي تكون بداخل الـ class مثلاً من يستطيع رؤيتها والتعامل معها بداخل المشروع، فمن الممكن أن تكون بعض البيانات لـ class مرئية في class آخر وبعض البيانات غير مرئية ولا يمكن التعامل معها إلا بداخل الـ class الذي كُتِبتْ بداخله فقط، المبرمج يحدد من يستطيع رؤية البيانات بداخل المشروع سواء class في نفس الـ namespace أو class في namespace أخرى بداخل المشروع.

أول طريقة للتغليف هي :

1 - public

البيانات التي تكون من هذا النوع فإنها ستكون مرئية في كل المشروع ويستطيع أي class داخل namespace أو أي namespace أخرى رؤيتها والتعامل معها.

طريقة تعريف الدوال أو الخصائص تكون بهذا الشكل :

طريقة الوصول + نوع البيانات + الأسم

مثال :

public int variable;

```
using System;
namespace First_program
{
        class Program
        {
                static void Main( )
                {
                        Program2 pro = new Program2( );
                        pro.x = 10;
                }
        }
        class Program2
        {
                public int x;
        }
}
```

كل ما عليك معرفته في هذا الكود أننا استطعنا إسناد قيمة للمتغير X في الـ class الآخر Program، وأستطعنا رؤيته والتعامل معه. ويتم الوصول إلى محتوى الـ class من خلال كائن أو نسخة (object)من هذا الـ class ثم النقطة (.) (dot)

النوع الثاني وهو :

2 – private

هذا النوع يجعل إستخدام البيانات التي من نفس نوعه تقتصر على الـ class المكتوبة بداخله فقط، أي أنه يحرّم إستخدام البيانات التي من نوعه إلى أي class آخر حتى لو class في نفس الnamespace ويمكننا التعامل مع البيانات التي من هذا النوع بداخل الـ class الذي كُتبت به فقط، ولو حاولنا تنفيذ الكود السابق على النوع private فإن المترجم سيخرج لك خطأ أنه لا يمكن الوصول لهذا المتغير بسبب الحماية التي يمتلكها.

```
namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            pro p = new pro();
            p.x = 10;
        }
    }
    class pro
    {
        private int x;
    }
}
```

int pro.x

Error:
'ConsoleApplication3.pro.x' is inaccessible due to its protection level

النوع الثالث :

protected – 3

وهذا النوع معناه أنه لا يمكننا رؤية البيانات ولا التعامل معها في أي class آخر إلا إذا وُرِثَ الـ class الذي كُتِبتْ بداخله البيانات التي من هذا النوع. وسنفهم مفهوم الوراثة لاحقاً.

```
using System;
namespace First_program
{
        class Program : Prog
        {
                static void Main( )
                {
                        Program  p = new Program( ) ;
                        p.x = 10 ;
                }
        }
        class Prog
        {
                protected int x ;
        }
}
```

النقطتان تعني أن الـ class " Program " قد ورث الـ class " prog " ويمكنه التعامل مع البيانات التي بداخله كأنها كُتِبت بداخله لكن ليس كل البيانات، فقط المسموح به، مثل public و protected، إلى الآن، وعندما نصل إلى الوراثة سنفهم كل شيء عنها.

هذه الخطوة --> Program p=new Program( );

تعني أننا أخذنا نسخة (كائن) من هذا الـ class (Program)

كائن (object) -----> p

Program <----- class

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace ConsoleApplication1
8  {
9      class Program : pro
10     {
11         static void Main(string[] args)
12         {
13             Program p=new Program();
14             p.x = 10;
15         }
16     }
17     class pro
18     {
19         protected int x;
20     }
21 }
22
```

النوع الرابع :

internal – 4

وهي تعني أننا نستطيع رؤية البيانات التي من هذا النوع في كل الـ assemply للبرنامج

الـ assemply يمكن أن تحتوي على أكثر من namespace

assemply---> ملف الـ DLL أو الـ exe

النوع الخامس :

protected internal – 5

نستطيع رؤية البيانات والتعامل معها في كل الـ assemply لكن يجب أن يكون الـ class الذي كُتِبَت بداخله موروث للـ class الذي سنستخدمها فيه.

protected internal int x;

15

(Static)

المتغيرات والدوال التي بداخل الـ class ويكون نوعها static تكون ثابتة أي أنه لا يمكن التعامل معها إلا من خلال اسم الـ class نفسه.

نستخدم static عندما نريد التعامل مع الدالة أو الحقل خارج الـ class بدون الحاجة إلى أخذ نسخة منه (object) فمثلاً الدالة ( )WriteLine هي دالة static داخل Console والـ Console هو الـ class الخاص بدوال الإدخال والإخراج. فعند إستدعاء هذه الدوال نستخدم اسم الـ class مباشرةً بدون أن نأخذ منه نسخة.

والدالة الرئيسية يجب أن تكون من النوع static، وأي دالة نريد إستدعائها مباشرة في دالة من النوع static يجب أن تكون static. بإختصار أي نوع static لا نستطيع التعامل معه مباشرة إلا من خلال نوع static آخر، لكن يمكن التعامل مع المتغيرات التي من نوع static من خلال الدوال، حتى لو لم تكن الدالة static لكن العكس خطأ، أي أننا لا نستطيع إستخدام متغير ليس static مع دالة نوعها static.

مثال :

```
using System;
namespace First_program
{
        class Program
        {
                int x;
                static void Main( )
                {
                        x = 10;
```

```
                                خطأ لأن الدالة من النوع static
                وطالما الدالة static يجب أن يكون المتغير static أيضاً
                        }
                }
}

using System;
namespace First_program
{
        class Program
        {
                static int x;
                static void Main( )
                {
                        x = 10;
                }
        }
}
```

في هذه الحالة يجوز إسناد قيمة للمتغير والتعامل معه بشكل طبيعي لأن نوعه static والدالة نوعها static.

```
using System;
namespace First_program
{
      class Program
        {
                static int x;
                void name( )
                {
                        x = 4;
```

```
يجوز إسناد قيمة للمتغير الذي من النوع static من خلال الدالة في الحالة العادية //
                }
                static void Main( )
                {

                }
        }
}
```

مثال ٢ :

```
using System;
namespace First_program
{
        class A
        {
                public static int x ;
        }
        class B
                {
                        public static void variables()

                        {

                        A.x = 5 ;
                        }
                }
}
```

16

الدوال

(Methods or Functions)

تحدثنا في مقدمة الكتاب عن الدوال أو الوظائف وما عملها، قبل وجود فكرة الدوال كنا إذا أردنا إستخدام جزء معين من الكود الذي كتبناه مرة أخرى فكان يجب كتابته ثانيةً كلما أحتجنا إليه، فكان يأخذ وقت ومجهود، فلما لا نكتب جزء معين من الكود مرة واحدة ونستخدمه متى أحتجنا إليه؟!

إذاً ما الكود الذي يُكتب بداخل الدالة؟!

نكتب الكود الذي نريد إستخدامه أكثر من مرة، أي كود نريد تنفيذه. والدوال تُستخدم لإعطاء قيم لخواص الـ class والتعامل معها. وُجِدَتْ فكرة الدوال قبل وجود البرمجة كائنية التوجه، والدوال عامل رئيسي في البرمجة فلا يمكن الإستغناء عنها.

أي دالة يكون لها قيمة عائدة لِما كُتِبَ بداخلها، ما عدا نوع واحد من الدوال (void) لا يعيد قيمة، معنى قيمة عائدة أن الدالة كلها يكون لها قيمة واحدة يمكن إستخدامها وإدخالها في عمليات أخرى، سنفهم معنى القيم العائدة بمثال. أولاً سنعرف طريقة تعريف الدالة.

يمكننا تسمية الدالة بأي اسم نريده.

```
( ) + اسم الدالة + القيمة العائدة من الدالة + طريقة الوصول

{

}
```

Access_Modifier + Return_Type + Name_of_function + ( )

```
{
        نكتب الكود داخل القوسين هنا . //
}
```

طريقة الوصول تعني صلاحيات إستخدام الدالة، أين نستخدمها ومن يستطيع إستخدامها.

قد تكون :

public , private , protected , internal , protected internal.

اسم الدالة :

توجد طرق لكتابة أسماء أعضاء الـ class سواء كان متغير أو دالة، هي مجرد طرق متعارف عليها، يمكننا التغاضي عن هذه الطرق وكتابتها بالشكل الذي نريد، لكن يُفضل كتابتها بهذه الطرق.

أول طريقة :

Pascal case - 1

يكون أول حرف من الاسم حرف كبير (Capital) وإذا كانت الكلمة مكونة من مقطعين فيكون أول حرف من المقطع الأول كبير وأول حرف من المقطع الثاني كبير مثل :

UserName

أو User_Name

ثاني طريقة :

Camel case – 2

وفي هذه الحالة يكون أول حرف من المقطع الأول صغير و أول حرف من المقطع الثاني كبير مثل :

userName

passWord

وهكذا...

ثالث طريقة :

UpperCase – 3

هذه الحالة تكون كل حروف الاسم كبيرة وتستخدم مع الاسماء التي تمتلك حروف قليلة مثل حرفين أو ثلاث ومع الثوابت.

مثال :

```
const string NAME = "mahmoud" ;
```

```
int GPA;
```

وهكذا ...

يجب أن يكون اسم الدالة متصل ولا تفصله مسافة فمثلاً لا نستطيع تسمية دالة أو متغير بهذا الشكل user name

لانه توجد مسافة تفصل الاسم وهذا لا يجوز، وفي حالة الأسماء التي تتكون من أكثر من مقطع مثل First Name ... نستخدم (_) فتكون First_Name.

أنتهينا من تسمية الدالة.

نوع الدالة يكون :

int , float , double , long , short , string , char , bool

هذه الأنواع تعني أن نوع الدالة يكون من نفس نوع القيمة العائدة منها فمثلاً إذا كانت الدالة int إذا يجب أن تعيد قيمة صحيحة ولو كانت string يجب أن تعيد نص أو كلمة، float يجب أن تعيد قيمة كسرية. bool تعيد true أو false وهكذا...

سنفهم معنى القيمة العائدة للدالة لكن مؤقتاً حاول أن تتعلم كيفية بناء هيكل الدالة وماذا يُكتَب بداخلها وكيف تُستَخدَم.

بعد كتابة الدالة والإنتهاء منها كيف نستخدمها وكم مرة نستخدمها؟

عن طريق إستدعاءها بإسمها فقط ونستدعيها بعدد المرات التي نريد، نستطيع إستدعاءها وإستخدامها ملايين المرات بدون إعادة كتابة الكود المكتوب بداخلها.

اتفقنا أن الدالة يكون لها قيمة واحدة عائدة ما عدا نوع واحد فما هو؟

هذا النوع هو **void** أي لا ترجع أي شيء ولا يكون لها قيمة نستطيع إستخدامها في عمليات أخرى، أي عندما ننتهي من تنفيذ الدالة لا نريد منها أي شيء آخر.

ملاحظة :

سنستخدم الـ class الرئيسي وسنكتب بداخله الدوال التي نريدها. سنُطَبِّق عليه "مؤقتاً". ويكون بداخله الدالة الرئيسية التي يبدأ تنفيذ الكود منها. يمكننا كتابة الدالة الجديدة فوق الدالة الرئيسية (Main) أو تحتها، لا يوجد فرق.

" تُكتَب الدوال بداخل class "

مثال :

```
using System;
namespace AnyThing
{
        class Program
                {
```

```
                سنكتب كل الدوال هنا
                static void Main( )
                {

                }
        }
}
```

هذا هو الشكل الذي سنعمل عليه .

مثال:

أكتب دالة تطبع Hello World.

قبل البدء في كتابة الكود يجب أن نحلل السؤال أولاً، بعدها نكتب الخوارزميات لهذه المسألة " الخوارزميات أي نحدد خطوات الحل خطوة بخطوة ونحل المسألة بطريقة جبرية "، إذا سرنا على هذا النمط فلن نواجه أي صعوبة في كتابة أي كود.

لنحلل هذه المسألة ...

نريد إنشاء دالة تطبع Hello World هل نحتاج قيمة لهذه الدالة فيما بعد؟ هل سنجري عليها أي عملية؟ جمع أو طرح أو قسمة أو سنأخذ قيمتها ندخلها في دالة أو متغير آخر؟ إذا كان الجواب لا ... فلن نحتاج أن يكون لها قيمة عائدة وستكون نوع الدالة void. ونسميها بالاسم الذي نريد لكن يفضل أن يكون اسم معبر عن الوظيفة التي تقوم بها الدالة، مثلاً إذا كانت دالة طباعة.. نسميها مثلاً print أو show أو view أي اسم نريده فلن يحدث هذا فارق في أي شيء.

ملاحظة :

طريقة الوصول لن تصنع فارق هنا، لأننا لن نستخدم الدالة في class خارجي إنما سنستخدمها بداخل الـ class المكتوبة بداخله فلن يفرق ما إذا كانت public أو private أو protected الكل سيان في هذا الوقت. اتفقنا أن هذا لمجرد التطبيق مؤقتاً فقط وبعدما ننتهي من تعلم كتابة الدوال والتعامل معها سنتعلم كيف نستخدمها في أكثر من class آخر.

```
using System;
namespace ConsoleApplication1
{
 class Program
 {
        public void print()
          {
                    Console.WriteLine("Hello World") ;
          }
 }
}
```

إذاً أنتهينا من كتابة الكود الخاص بالدالة كيف سننفذها وأين؟

عندما تفتح visual studio فأنت تجد class مكتوب والدالة الرئيسية التي يبدأ تنفيذ الكود منها مكتوبة بداخله.

فنستدعي الدالة التي أنشأناها بواسطة كتابة اسمها في الدالة الرئيسية وبما أن الدالة الرئيسية نوعها static فيجب أن تكون الدالة التي سنكتبها نوعها static أيضاً حتى تستطيع الدالة الرئيسية رؤيتها، يمكننا إستدعائها بطريقة أخرى عن طريق إنشاء كائن من هذا الـ class لكن عندما نصل إلى شرح الـ class وكيفية التعامل معه وطريقة إنشاءه سنتعلم كل شيء يخص الدوال وكل طرق التعامل معها بداخل الـ class.

حالياً أي دالة نوعها static يجب أن يكون ما سنستدعيه بداخلها سواء دالة أخرى أو متغير من نفس نوعها.. أي static.

وسيتم تعريفها بهذا الشكل :

طريقة الوصول + static + القيمة العائدة من الدالة + اسم الدالة ( ) { الكود }

```
Access_Modifier + static + Return_Type + Name_of_function + ( )

{ The code }
```

مثال :

```
using System;
namespace ConsoleApplication1
{
 class Program
 {
        public static void print()
         {
               Console.WriteLine("Hello World");
         }
        static void Main( )
         {
               print ( ) ;
         }
 }
}
```

عند كتابة اسم الدالة ونُشغِل البرنامج سيذهب إلى المكان الذي كتبنا فيه كود الدالة ويتم تنفيذ ما بداخلها.

سيطبع **Hello World**

في عملية إنشاء الدالة إذا كنا سنستخدم متغيرات في هذه الدالة فيمكن أن نمرر هذه المتغيرات للدالة أثناء التعريف وتسمى في هذا الوقت (Parameters) لا يوجد حد معين لعدد مدخلات دالة " parameters " فقط ما سنستخدمه وما نحتاج إليه، وهذه المدخلات عند إستدعاء الدالة سنضطر إلى تمريرقيمها إلى الدالة حتى تحدث العملية التي نريدها من الدالة، وإعطاء مُدخَل للدالة هو أمر إختياري لكن هناك بعض الدوال يُفضَل تمرير مُدخلات لها لإجراء عمليات عليها وتمييز كل دالة عن الأخرى.

فمثلاً عند تسجيل الدخول لحساب فيس بوك فإنك تدخل البريد الالكتروني وكلمة المرور، فهذه البيانات التي تدخلها تكون مُدخل لدالة تفحص كلمة المرور والبريد الالكتروني الخاصين بك المُسجلين على قواعد البيانات.

ويتم تعريف المُدخل بالطريقة العادية لانه متغير عادي.

Access_Modifier + static + Return_Type + Name_of_function +

(parameter1,parameter2,……)

مثال :

انشيء دالة لجمع رقمين ثم اطبع ناتج الجمع.

```
using System;
namespace ConsoleApplication1
{
 class Program
 {
        public static void Print_Sum( int num1 , int num2 )
          {
                int sum = num1 + num2 ;
                Console.WriteLine( sum ) ;
          }
        static void Main( )
         {
                int number1 , number2 ;
                number1 = int.Parse(Console.ReadLine() ) ;
                number2 = int.Parse(Console.ReadLine() ) ;
                print( number1 , number2 ) ;
         }
 }
}
```

في هذا الكود أنشأنا الدالة وأنشأنا لها مُدخلين وتمت عملية الجمع بداخل الدالة، ثم أستدعينا دالة جمع الرقمين في الدالة الرئيسية، لكن الدالة التي أنشأناها تأخذ مُدخلين لذا سنضطر إلى تعريف متغيرين وتمريرهما إليها وقد أخذنا قيم المتغيرات من المستخدم، عند تشغيل البرنامج وعندما يصل المترجم إلى السطر الذي استدعينا به الدالة يذهب إلى هذه الدالة أياً كان موقعها في الكود وينفذ ما بداخلها.

الدالة التي من النوع void لا نستطيع عمل أي عمليات عليها فلا يمكن أن ندخلها في أي عملية أخرى ولا نستطيع إجراء عملية طباعة عليها بإستخدام دالة Console.WriteLine( )

ولا نستطيع فعل ذلك :

```
int temp = print( number1 , number2 ) +2;
```

لا نستطيع تمريرها إلى متغير أو فعل أي شيء فيها لأنها ليس لها قيمة عائدة.

ماذا إذا أردنا أن نستخدم قيمة ناتج الجمع ونسندها إلى متغير آخر؟

لنحلل المسألة ...

ناتج جمع رقمين من النوع int ماذا سيكون نوع ناتج الجمع؟

عدد صحيح أيضاً.

إذا سيكون نوع القيمة العائدة من هذه الدالة int فيكون نوع الدالة int.

ويجب في هذه الحالة أن نستخدم الكلمة المحجوزة في اللغة return في نهاية الكود نكتب القيمة العائدة من هذه الدالة، وكل دالة يكون لها قيمة واحدة فقط عائدة.

return + القيمة

مثال :

```
using System;
namespace ConsoleApplication1
{
```

```
class Program
{
        public static int Print_Sum( int num1 , int num2 )
          {
                int sum = num1 + num2 ;
                return sum ;
          }
        static void Main( )
          {
                int number1,number2 ;

                number1 = int.Parse(Console.ReadLine( )) ;

                number2 = int.Parse(Console.ReadLine( )) ;

                int temp = print( number1 , number2 ) + 100 ;

                Console.WriteLine( temp ) ;

          }

}

}
```

في هذا الكود أنشأنا الدالة وأصبح نوعها int أي القيمة العائدة منها int الدالة كاملة في ذلك الوقت هي عبارة عن قيمة واحدة يمكن أن نفعل بها أي شيء، وأستدعيناها في الدالة الرئيسية ومررنا لها المُدخلات التي عرفناها وأخذنا قيمهم من المستخدم، بعد ذلك مررنا الدالة كاملة إلى متغير، إذا هذا المتغير حالياً يحمل قيمة الدالة. ثم أضفنا إلى قيمة الدالة ١٠٠، فمثلاً إذا أدخل المستخدم ٢٠ و٣٠ سيكون ناتج الدالة ٥٠ وسيضيف عليها ١٠٠ ثم سيضع القيمة الجديدة في المتغير temp.

ماذا لو كانت العملية التي ستتم في الدالة هي قسمة وليست جمع إذا فيوجد إحتمال أن يكون ناتج القسمة عدد كسري، فيجب أن نغير نوع الدالة لنفس نوع القيمة العائدة منها. أي يكون نوعها float أو double.

ملاحظة :

مُدخلات الدالة أثناء الإنشاء تسمى " Parameters " أما إثناء إستدعاء الدالة وتمرير قيم لها فتُسمى المُدخلات " Arguments ".

مثال :

أكتب دالة تأخذ رقمين ثم تعيد الرقم الأكبر فيهما.

```
using System;
namespace ConsoleApplication1
{
 class Program
 {
        public static int Max_Value( int num1 , int num2 )
          {
                if ( num1 > num2 )
                        return num1;
                else if ( num2 > num1 )
                        return num2 ;
                else
                        return 0 ;
          }
        static void Main( )
         {
                int number1 , number2;
                number1 = int.Parse(Console.ReadLine( ));
                number2 = int.Parse(Console.ReadLine( ));
                Console.WriteLine(Max_Value( number1 , number2 ) ;
```

```
        }
 }
}
```

تمارين :

أكتب دالة تأخذ من المستخدم الأس والأساس وتخرج الناتج.

مثلاً الاساس ٢ والأس ٥ = ٣٢

الأساس ٣ والأس ٤ = ٨١

```
using System;
namespace ConsoleApplication1
{
 class Program
 {
        public static void Power( int Base_num , int power_num )
          {
                int result = 1;
                for ( int i = power_num ; i > 0 ; i-- )
                {
                        result *= Base_num ;
                }
                Console.WriteLine( result );
          }
        static void Main( )
          {
                int Base = int.Parse(Console.ReadLine( ) );
                int Power = int.Parse(Console.ReadLine( ) );
```

```
            Power ( Base,Power ) ;     // ( 2,6 )
        }
    }
}
```

الناتج :

64

وتوجد دالة جاهزة بداخل اللغة تسمى pow داخل class يسمى Math

```
Math.Pow ( 2 , 6 ) ;
```

سيخرج لك نفس النتيجة لكن يجب أن تعتمد على نفسك في مرحلة التعلم وتتعلم كيف تنشئ دوال بنفسك .

تمارين :

أكتب دالة تأخذ رقم من المستخدم وتعيد الجذر التربيعي لهذا الرقم إن وُجد.

```
using System;

namespace ConsoleApplication1

{
 class Program
 {
        public static void SQRT( int num )
          {
```

```
            for ( int i = num ; i > 0 ; i-- )
            {
                if ( i * i == num )
                {
                    Console.WriteLine ( i );
                }
            }
            Console.WriteLine ( result );
        }
    static void Main( )
     {
            int number=int.Parse ( Console.ReadLine ( ) );
            SQRT ( number );   // ( 25 )
     }
  }
}
```

الناتج :

5

تمارين :

أكتب دالة تأخذ من المستخدم مصفوفة ورقم معين ثم ترجع عدد مرات تكرار هذا الرقم في تلك المصفوفة.

```
using System;
namespace ConsoleApplication1
{
 class Program
 {
        public static int count_item( int [ ]arr, int num )
          {
```

```
            int count = 0;
            for ( int i = 0 ; i < arr.Length ; i++ )
            {
                if ( num == arr [ i ] )
                {
                    count++;
                }
            }
            return count;
        }
    static void Main( )
        {
            int [ ] x = { 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 1 , 4 , 7 , 9 , 4 , 5 }

            int num = int.Parse ( Console.ReadLine( ) );

            Console.WriteLine ( " item " + num + " repeated " + count
            ( x , num ) + " times " );

        }

    }

}
```

تمارين :

أكتب دالة لتنفيذ العمليات الحسابية، تأخذ القيمة الأولى والعملية التي تتم سواء ضرب أو جمع أو قسمة ثم تأخذ القيمة الثانية وتعيد ناتج العملية التي ستتم وتحدد من قِبل المُستخدم.

```
using System;
namespace ConsoleApplication1
{
  class Program
```

```
{
    static float Math_Operation(float operand1, char operation,
                                        float operand2)
    {
    float result = 0.0f;
    switch ( operation )
      {
       case '+':
          result = operand1 + operand2;
          break;
       case '-':
          result = operand1 - operand2;
          break;
       case '/':
          result = operand1 / operand2;
          break;
       case '*':
          result = operand1 * operand2;
          break;
       case '%':
          result = operand1 % operand2;
          break;
      }
    return result;
   }
  static void Main( )
  {
    while ( true )
    {
       Console.WriteLine("Enter operand1 : operation : operand2 :");
```

```
            float op1 = float.Parse(Console.ReadLine( ));
            char op = char.Parse(Console.ReadLine( ));
            float op2 = float.Parse(Console.ReadLine( ));
            Console.WriteLine(Math_Operation(op1 , op , op2));
        }
    }
  }
}
```

الدوال التي تعيد قيم يمكن تمرير القيمة العائدة منها إلى دالة أخرى .

مثال :

```
using System;
namespace ConsoleApplication6
{
   class Program
   {
      static int a(int x , int y)
      {
         if (x > y)
            return x;
         else
            return y;
      }
      static int b(int x, int y)
      {
         return x + y;
      }
      static int c(int x, int y)
      {
         return x - y;
      }
      static int d(int x, int y)
      {
         return x * y;
```

```
    }
    static void sum(int a,int b,int c)
    {
      Console.WriteLine(a + b + c);
    }
    static void Main( )
    {
      sum(a(7, b(3, c(5, d(2,4) ) ) ), 5, 2);
    }
  }
}
```

الناتج : ١٤

في هذا الكود أنشأنا ٥ دوال، كل دالة لها وظيفة مختلفة عن الأخرى، فالدالة "a" تأخذ قيمتين ثم تعيد القيمة الأكبر فيهما والدالة "b" تأخذ قيمتين ثم تعيد ناتج جمعهما، والدالة "c" تأخذ قيمتين وتعيد ناتج طرحهما والدالة "d" تأخذ قيمتين وتعيد ناتج ضربهما، والدالة "sum" تأخذ ثلاث قيم وتطبع ناتج جمعهم.

وفي دالة الـ Main أستدعينا الدالة sum وهي تأخذ ثلاث قيم كما ذكرت، ثم مررنا لها الدالة a وبعد ذلك مررنا للدالة a القيمة ٧ والقيمة الأخرى كانت عبارة عن الدالة b ومررنا للدالة b القيمة ٣ والقيمة الأخرى كانت عبارة عن الدالة c ومررنا لها القيمة ٥ والقيمة الأخرى كانت عبارة عن الدالة d ومررنا للدالة d القيمتين ٢ و٤.

عند تنفيذ هذا الكود فإن الدوال الداخلية هي التي تُنفَذ أولاً وبما أن هذه الدوال ذات قيمة عائدة فإن كل دالة تُعيد قيمة واحدة فقط. وتكون هذه القيمة هي المُدخل الآخر للدالة التي تسبقها .

في هذا المثال الدالة d تأخذ القيمتين ٢و٤ ووتعيد ناتج ضربهما أي ٨ إذاً تكون القيمتان اللتان ممرناهما للدالة c هما ٥ و ٨ ، ثم تعيد الدالة c ناتج طرحهما فتعيد القيمة -٣ ، وبذلك تكون القيم الممررة إلى الدالة b هي ٣ و -٣ ، ثم تعيد الدالة b ناتج جمعهما فتعيد القيمة صفر، وبذلك تكون القيم الممررة للدالة a هي صفر و ٧ وتعيد الدالة a القيمة الأكبر فيهما فتعيد القيمة ٧ ،

وبذلك تكون القيم التي مُررت للدالة sum هي : (2, 5 , 7)

وتطبع ناتج جمعهم فيكون ١٤

ref

عندما نمرر قيم للدالة فإننا نأخذ نُسخة من هذه القيم ويحدث التغيير فيها أما القيم الأساسية في ذاكرة الكمبيوتر لم يحدث بها أي تغيير ولم تتأثر.

فمثلاً إذا أردنا تنفيذ عملية تبديل بين قيمتي متغيرين عن طريق دالة فإنه سيبدل القيم لكن سيبدلها في النُسخة فقط وليس في المتغيرات الأصلية.

```
using System;
namespace ConsoleApplication1
{
 class Program
 {
        public static void Swap( int num1 , int num2 )
          {
                int temp=num1;
                num1=num2;
                num2=temp;
          }
        static void Main( )
         {
                int number1 = 10 , number2 = 20;
                Swap ( number1 , number2 ) ;
                Console.WriteLine("number1 = " + number1);
                Console.WriteLine("number2 = "+number2);
         }
 }
}
```

الناتج :

number1 = 10;

number2 = 20;

لن يحدث أي تبديل فعلي في القيم لأننا غيرّنا في النُسخ فقط. لكن ماذا لو أردت إحداث تغيير فعلي في المتغيرات الأصلية في الذاكرة ؟

نستخدم الكلمة المحجوزة (**ref**)

```
using System;
namespace ConsoleApplication1
{
 class Program
 {
        public static void Swap(  ref int num1 ,ref  int num2 )
          {
                int temp = num1;
                num1 = num2;
                num2 = temp;
          }
        static void Main( )
         {
                int number1 = 10 , number2 = 20;
                Swap ( ref number1 , ref number2 ) ;
                Console.WriteLine("number1 = " + number1);
                Console.WriteLine("number2 = "+number2);
         }
 }
}
```

الناتج :

number1 = 20;

number2 = 10;

```
using System;
namespace ConsoleApplication1
{
 class Program
 {
        public static void Swap(  ref  int num1 ,ref  int num2 )
          {
                num1 = 40 ;
                num2 = 30 ;
          }
        static void Main( )
         {
                int number1=10,number2=20;
                Swap ( ref number1 , ref number2 ) ;
                Console.WriteLine("number1 = " + number1);
                Console.WriteLine("number2 = "+number2);
         }
 }
}
```

الناتج :

number1 = 40;

number2 = 30;

القيم الجديدة لن تؤثر في المتغيرات لأننا مررنا للمتغيرات قيم مرجعية في الدالة والقيم وُضعت في المتغيرات الأصلية لا النُسخ.

تمارين :

```
using System;
namespace ConsoleApplication1
{
 class Program
 {
        public static int Swap(  ref  int num1 )
          {
                num1 = num1*num1 ;
                return num1;
          }
        static void Main( )
         {
                int number1 = 10;
                Swap ( ref number1  ) ;
                Console.WriteLine("number1 = " + number1);
         }
 }
}
```

الناتج :

number1 = 100

out

إذا أنشأنا دالة تأخذ مُدخلين ووأردنا إستدعاء هذه الدالة في دالة أخرى أو أي مكان في الكود، ومررنا متغيرات للدالة في الاستدعاء لكننا لم نعطي قيم لهذه المتغيرات ونريد أن نأتي بالقيم من الدالة نفسها، في الحالة العادية هذا لا يجوز ويجب أن نعطي قيم للمتغيرات التي سنمررها كمُدخلات للدالة حتي إذا أدخلنا لهذه المتغيرات قيم بتدائية في الدالة.

```
namespace ConsoleApplication3
{
    class Program
    {
        public static void fun(int num1,int num2)
        {
            num1 = 200;
            num2 = 100;
        }
        static void Main(string[] args)
        {
            int number1, number2;
            fun(number1, number2);
            Console.WriteLine(" number1 = " + number1);
            Console.WriteLine(" number2 = " + number2);
        }
    }


}
```

لإستخدام القيم التي أستخدمناها في الدالة يجب أن نحدد له أنه سيأتي بقيم لمُدخلات الدالة من الدالة نفسها.

مثال :

```
using System;
namespace ConsoleApplication1
{
 class Program
 {
        public static void fun_name(  out  int num1 ,out  int num2 )
          {
                num1 = 200 ;
                num2 = 100 ;
          }
        static void Main( )
         {
                int number1 , number2;
                fun_name( out number1 , out number2 ) ;
                Console.WriteLine("number1 = " + number1);
                Console.WriteLine("number2 = "+number2);
         }
 }
}
```

الناتج :

number1 = 200

number2 = 100

حتى لو أدخلنا قيم لـ number1 , number2 فإنه سيتجاهلها ويستخدم القيم الخارجية التي تأتي من الدالة.

```
using System;
namespace ConsoleApplication1
{
 class Program
 {
        public static void fun_name(  out  int num1 ,out  int num2 )
          {
                num1 = 200 ;
                num2 = 100 ;
          }
        static void Main( )
         {
                int number1 = 60 , number2 = 30 ;
                fun_name ( out number1 , out number2 ) ;
                Console.WriteLine("number1 = " + number1);
                Console.WriteLine("number2 = "+number2);
         }
 }
}
```

الناتج :

number1 = 200

number2 = 100

ref تتطلب أن تدخل قيم للمتغيرات التي ستمررها للدالة لحظة الإستدعاء.

```
namespace ConsoleApplication3
{
    class Program
    {
        public static void fun(ref int num1,ref int num2)
        {
            num1 = 200;
            num2 = 100;
        }
        static void Main(string[] args)
        {
            int number1, number2;
            fun(ref number1, ref number2);
            Console.WriteLine(" number1 = " + number1);
            Console.WriteLine(" number2 = " + number2);
        }
    }


}
```

الخط الذي تحت المتغيرين لحظة إسنادهما للدالة يُعني أنه يجب أن نعطي لهما قيم وقت تعريفهما.

مثال :

أكتب دالة لتنفيذ عملية المضروب لرقم يدخله المستخدم.

مثال لمضروب الـ ٤ :

٤ * ٣ * ٢ * ١ = ٢٤

```
using System;
namespace ConsoleApplication1
{
 class Program
 {
        public static void factorial ( int num )
          {
                int fact=1;
                if( num<=0 )
                        Console.WriteLine( 0 );
                else
                {
                        for(int i=1 ; i<=num ; i++)
                        {
                                fact *= i ;
                        }
                        Console.WriteLine(fact);
                }
          }
        static void Main( )
          {
                int number = int.Parse( Console.ReadLine( ));
                factorial(number) ;
          }
    }
 }
```

params

تستخدم كلمة params عندما لا نعلم عدد مُدخلات (parameters) الدالة وعدد المُدخلات في هذه الدالة يُعبر عن حجم المصفوفة.

مثال :

```
using System;
namespace ConsoleApplication1
{
 class Program
 {
        public static int Max_Value_in_array ( params  int [ ] arr )
         {
               int max=arr[0] ;
               for( int i =0 ; i < arr.Length ; i++)
                     if ( arr [ i ] > max )
                           max = arr[ i ];
               return max ;
         }
        static void Main( )
         {
                Console.WriteLine ( Max_Value_in_array ( 3 , 4 , 5 , 6 , 7
                                    , 8 , 1 , 2 , 3 , 4 , 5 ) );
         }
 }
}
```

الناتج :

٨عدد مُدخلات الدالة يكون غير محدود أي يمكننا تمرير العدد الذي نريده لكن مع كل مُدخل نضيفه يزيد حجم المصفوفة بمقدار ١، فمثلاً حجم المصفوفة في هذا الكود سيكون ١١، وسيطبع القيمة الأكبر في هذه المصفوفة.

17

الدالة ذاتية الإستدعاء

(Recursion function)

يوجد نوع من الدوال يسمى الدالة العائدة، أي التي تُستدعى بداخل نفسها
إذا أُستدعت دالة بداخل نفسها فإنها تظل تستدعي نفسها إلى أن توقفها بشرط، لذا فيجب تحديد شرط لها لكي تتوقف.
سننفذ نفس برنامج المضروب بها، بدون إستخدام loop

```
using System;
namespace ConsoleApplication1
{
 class Program
 {
        public static int factorial ( int num )
          {
                if( num<=0 )
                        return 0 ;
                else if ( num==1 ) // لتوقف الدالة عند هذا الرقم
                        return 1 ;
                else
                        return ( num * factorial(num-1));
          }
        static void Main( )
         {
```

```
            int number = int.Parse( Console.ReadLine ( ) );
            Console.WriteLine ( factorial ( number ) ) ;
        }
}
}
```

الناتج :

١٢٠

يجب أن يكون نوع الدالة أي نوع غير void فلا يُمكن أن يكون void، لأن طريقة عمل هذا النوع من الدوال هو أنها تستدعي نفسها مرة أخرى بداخل نفسها فإذا كان نوعها void فإننا لن نستطيع عمل أي عملية عليها فنحن هنا نحتاج إلى أن نضرب الرقم في قيمة الدالة ونطرح ١ من الرقم إلى أن يصل الرقم المُدخل إلى ١.

فإذا أدخل المستخدم صفر أو قيمة أقل من الصفر بالسالب فإنها سترجع صفر،

وإذا أدخل ١ فإن الدالة سترجع ١

وإذا أدخل رقم ٥ فإنها ستضرب الرقم ف قيمة الدالة ناقص ١ وتستدعي نفسها مرة أخرى فيكون الرقم ٤ ثم ٣ ثم ٢ وتتوقف وتعيد ناتج ضرب هذه القيم مع بعضها فتكون قيمة الدالة كلها تساوي ٥ * ٤ * ٣ * ٢ أي ١٢٠.

مثال :

أحسب مجموع الأرقام من رقم يدخله المستخدم إلى رقم أكبر يحدده المستخدم أيضاً بدون إستخدام loop.

مثلاً من ٤٠ إلى ١٢٠

```
using System;

namespace ConsoleApplication1
```

```
{
 class Program
 {
        public static int sum ( int start , int end)
         {
               if( start>end ) //  لتوقف الدالة عند هذا الرقم
                     return 0 ;
               else
                     return ( start + sum(start+1 , end));
         }
        static void Main( )
         {
               int start = int.Parse( Console.ReadLine( ));
               int end = int.Parse( Console.ReadLine( ));
               Console.WriteLine ( sum ( start , end ) ) ;
         }
   }
}
```

مثال :

أنشيء دالة تأخذ مصفوفة كمُدخل وتخرج أكبر قيمة بهذه المصفوفة بدون إستخدام حلقات تكرار.

```
using System;
namespace ConsoleApplication19
{
   class Program
   {
      static int max = int.MinValue;
      static void fun(int[ ] arr, int low)
```

```
    {
        if (low < arr.Length)
        {
            if (arr[low] > max)
                max = arr[low];
            fun(arr, ++low);
        }
        else
            Console.WriteLine(max);
    }
    static void Main(string[ ] args)
    {
        int[ ] x = {-123,-565,-3555,-293,234,987,265};
        fun(x, 0);
    }
  }
}
```

الناتج :

987

في هذا المثال عرفنا متغير يسمى max ووضعنا به قيمة إبتدائية وهذه القيمة تأتي من خلال (int.MinValue) وتكون هذه القيمة هي أصغر قيمة يمكن أن يحملها متغير من النوع int، يُمكنك وضع قيمة إبتدائية صفر لكن بإفتراض أن الأعداد المتواجدة في المصفوفة كلها سالبة ففي هذه الحالة لن يكون هناك رقم سالب أكبر من الصفر ويكون الناتج صفر وهذه النتيجة خاطئة، ثم أنشأنا دالة void وكان من الممكن أن تكون int ثم تُعيد قيمة الـ max من خلال كلمة return، وهذه الدالة تأخذ مصفوفة نوعها int وتأخذ أيضاً أصغر index في المصفوفة التي سنعمل عليها ودائماً ما يكون الصفر، ونكتب الشرط الذي نريد أن تتوقف الدالة عنده، متى تتوقف الدالة ؟! عندما نصل إلى آخر index في المصفوفة أي أن المتغير الذي يسمى low عندما يصل إلى آخر index فإنها تتوقف بعد هذه العملية. فمثلاً إذا كان حجم المصفوفة 7 فتكون آخر قيمة

يصل إليها المتغير low هي 6 ، ثم بعد ذلك نتحقق من القيم الموجودة بالمصفوفة ما إذا كانت أكبرمن قيمة الـ max أم لا، وأي قيمة أكبر من القيمة الحالية للـ max إذا ستكون قيمة الـ max هي هذه القيمة الجديدة ثم نستدعي الدالة داخل نفسها لكن نجعل المتغير low يزيد بمقدار 1 حتى ينتقل إلى الـ index الذي يليه وهكذا إلى أن يكون الشرط الخارجي خطأ وبذلك تتوقف الدالة عن إستدعاء نفسها ويتم تنفيذ ما بداخل else وتطبع آخر قيمة وصل إليها المتغير max.

مثال :

أنشيء دالة تأخذ مصفوفة كمُدخل وتخرج أكبر قيمة بهذه المصفوفة.

```
using System;
namespace ConsoleApplication1
{
 class Program
 {
        public static int Max_Value_in_array ( int [ ] arr )
          {
                int max=arr[0] ;
                for(int i = 0 ; i < arr.Length ; i++)
                        arr[i]=int.Parse(Console.ReadLine( ));
                for( int i =0 ; i < arr.Length ; i++)
                        if ( arr [ i ] > max )
                                max = arr[ i ];
                return max ;
          }
        static void Main( )
          {
                int [ ]x=new int[10] ;
                Console.WriteLine ( Max_Value_in_array ( x ) ) ;
          }
```

```
 }
}
```

يمكن للدالة أن تأخذ مصفوفة ويتم التعامل معها بطريقة عادية جداً. يتم إدخال المصفوفة كـ (Parameter) مُدخل للدالة، في هذه الحالة نحن لا نعلم حجم المصفوفة لذا توجد خاصية لإيجاد حجم المصفوفة **Length.** فنستخدمها وحينما ننشيء المصفوفة في دالة الـ Main ونمررها للدالة يجب أن نحدد الحجم وحينها هذه الخاصية ستحسب حجم المصفوفة في الدالة بعدها تُمِرر اسم المصفوفة التي أنشأتها في دالة الMain كمُدخل للدالة، نُمرِر اسم المصفوفة فقط.

تمارين :

أنشيء دالة تأخذ مصفوفتين وتنسخ أحدهما في الأخرى.

```
using System;
namespace ConsoleApplication1
{
 class Program
 {
        public static void Copy_array ( int [ ] arr ,int [ ] arr2)
          {
                for(int i = 0 ; i < arr.Length ; i++)
                        arr2 [ i ] = arr [ i ] ;
          }
        static void Main( )
          {
                int [ ]x = new int[10]{ 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 } ;
                int [ ]y=new int[ 10 ];
                Copy_array   ( x , y ) ;
          }
  }
}
```

18

(Object , Var , Dynamic)

يوجد ثلاثة أنواع من البيانات يمكنهم إستقبال أي بيانات من أي نوع، سواء كانت البيانات رقم صحيح أو كسر أو نص أو حرف أو أي نوع آخر، لكن يوجد فرق بين الثلاثة أنواع.

أول نوع هو :

object

كل أنواع البيانات هي عبارة عن (object) فيمكنه أن يأخذ بيانات من أي نوع.

وهو الـ class الرئيسي لكل أنواع البيانات ويكون في قاعدة هرم أنواع البيانات.

لكن (object) لا يفهم معنى البيانات التي بداخله، فلا يمكنه إجراء أي عملية على ما بداخله مع متغير آخر.

مثال :

```
object x = " aaa " ;
```

string y = x → خطأ

لا يمكن لأي نوع بيانات آخر من الأنواع المعروفة أن يحمل قيمة متغير من النوع object لأن نوعه يعتبر غير معلوم، هو فقط يحمل بيانات بداخله لكن لا يعلم نوعها. فلا يمكنه التعامل إلا مع object مثله. لكن ماذا لو أردنا تخزين القيمة التي تأتي من متغير نوعه object يجب في ذلك الوقت أن نحولها لنفس النوع الذي ستخزن فيه، فمثلا لو أردنا تنفيذ نفس العملية السابقة سنضطر إلى تحويل الـ x إلى نفس نوع الـ y.

```
object x = " aaa " ;

string y = ( string ) x ;
```

لكن يمكننا فعل العكس بدون تحويل.

```
string y = " aaa " ;

object x = y ;
```

لأننا أتفقنا أن الـ object يمكنه أن يحمل أي نوع آخر.

ويقبل أن نغير نوع البيانات التي بداخله بدون أن يحدث خطأ.

فمثلاً :

```
object x = " aa ";

x = 4;

Console.WriteLine( x ) ;
```

الناتج :

4

يمكن أن يكون القيمة العائدة من دالة من النوع object إذا كنا لا نعرف النوع الراجع تحديداً.

ويمكن للمدُخل أيضاً أن يكون من النوع object. في حالة أننا لا نعرف القيمة التي سيدخلها للدالة هل هي نص أم حرف أم رقم.

```
using System;
namespace ConsoleApplication1
{
 class Program
```

```
{
        public static object fun ( params  object [ ] arr )
          {
                return arr[3] ;
          }
        static void Main( )
          {
                Console.WriteLine ( fun ( 3 , 4 , "aa" , 's' , true , 8 , 1 , 3.44
                                        , 3 , 4 , 5 ) );
          }
}
}
```

الناتج :

S

المصفوفة نوعها object لذا يمكنها أن تحتوي على أي نوع بداخلها، والدالة تعيد قيمة من نفس نوع المصفوفة وبما أن المصفوفة نوعها object فالقيمة العائدة منها إذاً ستكون من النوع object .

أكتب دالة لطباعة هذه المصفوفة التي أدخلنا فيها البيانات سابقاً.

```
using System;
namespace ConsoleApplication1
{
 class Program
 {
        public static void fun ( params  object [ ] arr )
          {
                foreach( object i  in arr )
                        Console.WriteLine ( i ) ;
          }
        static void Main( )
```

```
        {
            fun ( 3 , 4 , "aa" , 's' , true , 8 , 1 , 3.44 , 3 , 4 , 5 ) ;
        }
}
}
```

is

هي كلمة محجوزة للتأكد من نوع البيانات .

مثال :

```
object x =10 ;
objext y = " aa " ;
if ( x is int )
{
   Console.WriteLine(" Yes ");
}
```

في هذا المثال سينفذ جملة الطباعة التي بداخل if لأن الشرط صحيح.

أما إذا قولنا :

```
if ( x is string )
{
   Console.WriteLine(" Yes ");
}
```

فلن ينفذ لأن المتغير x عدد صحيح وليس نصاً، إذاً فالكلمة المحجوزة is تتأكد من نوع البيانات هل هو متطابق أم لا.

var

هي مثل object لكن يوجد بعض الإختلافات بينهم.

فمثلاً عند تعريف متغير نوعه var لا يمكن تغيير القيمة التي بداخله لنوع آخر.

مثال :

```
var x = 3 ;
```

خطأ ; " aaa " = x

هو يقبل نوع واحد فقط فعندما مررنا له القيمة ٣ فهو يتعامل مع المتغيرعلى أنه int ويجب أن نمرر له قيمة أثناء تعريف المتغير.
فلا يجوز أن نعرف المتغير من النوع var بدون تمرير قيمة له في مرحلة التعريف.

☒ خطأ ; var x

يمكنه أن يأخذ أي قيمة. لكنه يتميز عن أن object في أنه عندما نُعرفّ متغير من نوعه فإنه يعرف نوع القيمة التي بداخله ويمكننا أن ندخلها في متغير آخر من نفس نوعها.

مثال :

```
var x = " aaa " ;

string y = x ;
```

ويمكننا إجراء أي عملية عليه طالما أننا نستخدمه مع متغيرات من نفس نوعه.
لكن إذا أدخلناه في متغير من نوع آخر فيجب أن نحول نوعه أولاً لنفس النوع الجديد.
مثال :

```
var x = 3.4f ;

int y = ( int ) x;
```

ملاحظة : القيم الكسرية يكون نوعها تلقائياً double، إذا أردناها أن تكون float فإننا نكتب حرف **f** بجانب القيمة.

لا يستخدم لتعريف الدوال أو حتى المصفوفات، فهو يستخدم لتعريف المتغيرات فقط بداخل الدوال.

dynamic

هو عبارة عن نوع بيانات يستطيع أن يخزن أي شئ بداخله وتستطيع أن تغير قيمته لأي نوع تريده وأن تدخله في أي عملية تريدها. هذا النوع لا يتعرف على ما بداخله إلا وقت التشغيل.

```
dynamic c =" ttt " ;
string s = c + " aaa " ;
c = 6 ;
int n = c-3 ;
Console.WriteLine( s +"   "+ n );
```

الناتج :

```
tttaaa    3
```

حتى إذا وُجد خطأ فإنه لا يتعرف عليه إلا عند التشغيل لأنه يترجم ما بداخله في وقت التشغيل فقط.

مثال :

```
dynamic x = ' a ' ;

string y =  x ;
```

لن يظهر الخطأ في الكود لكنه سيظهره عند تشغيل البرنامج.

يمكننا أن نستخدمه في تعريف دالة أو مصفوفة بدون مشكلة.

تمارين :

أكتب دالة لتسجيل عملية دخول إلى نظام أو موقع له بريد إلكتروني وكلمة سر

بإفتراض أن البريد الإلكتروني المُسجل على قاعدة البيانات هو
Mahmoud@gmail.com

وكلمة السر هي mahmoud12345

```
using System;
namespace ConsoleApplication1
{
 class Program
 {
        public static void Login ( string _Email , string _Password  )
          {
                if ( _Email == " Mahmoud@gmail.com " && _Password
                   ==" mahmoud12345 ")
                      Console.WriteLine(" welcome ");
                else
                      Console.WriteLine( " invalid data ");
          }
        static void Main( )
                {
                      string E_Mail = Console.ReadLine ( );
                      string password= Console.ReadLine ( );
                      Login ( E_Mail , password );
                }
        }
}
```

19

(Classes)

ذكرت مسبقاً نبذة عن البرمجة كائنية التوجه وعن ماهية الـ class سنتعلم كيف نُنشيء الـ class.

عندما نُنشيء class فإننا نُنشيء نوع من البيانات التى نستطيع أخذ عدد غير محدود من النسخ من هذا النوع الواحد، وقد شرحت مثال لموقع الفيس بوك، فمثلاً صفحتك الشخصية تكون عبارة عن class به كل بياناتك الشخصية، وعند تسجيل الدخول يكون class به دوال لتنفيذ عملية الدخول، كل شخص يكون له بياناته الخاصة أي أن كل شخص قد أخذ نسخة من هذا الـ class وملأها ببياناته الخاصة.

إذا كنت تريد إنشاء برنامج أو موقع ويوجد عليه مستخدمون وكل مستخدم له بيانات بأنواع مختلفة مثل العمر والاسم وتاريخ الميلاد ...، كيف ستربط كل هذه البيانات بمستخدم واحد؟!

هذه التقنية جعلت البرمجة عامةً سهلة ومرنة.

توجد قاعدة في البرمجة الكائنية تقول أن كل شيء هو عبارة عن كائن.

طريقة التعريف :

نكتب كلمة class ثم اسم الـ class، يُمكن أن نسميه بأي اسم نريده، لكن يُفضَل أن يكون اسم يُعبِرّ عما يقوم به هذا الـ class

```
class Name
{
}
```

ويُكتب بداخل namespace، طريقة الوصول لأي class تكون internal تلقائياً ويُمكنك تغييره إلى public، والبيانات التي تكون بداخل ال class تكون private تلقائياً ما لم نغيرها إلى نوع آخر.

إذاً ماذا نكتب بداخل ال class؟

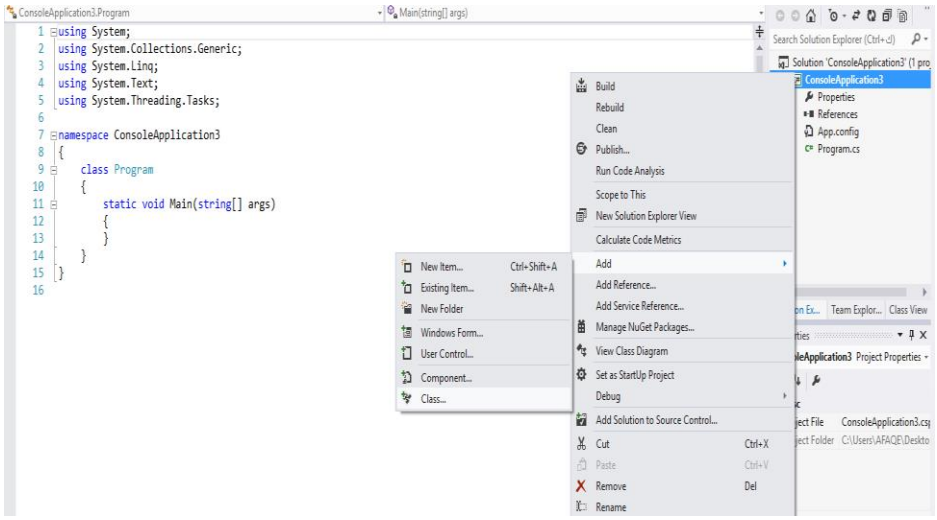
نكتب بداخله كل الدوال التي نحتاجها لتنفيذ عملية معينة، والمتغيرات التي سنستخدمها، المتغيرات التي بداخل الـ class غالباً ما تكون private ونستخدمها من خلال الدوال، ويكون نوع الدالة public. الطريقة المُتعارف عليها أن يُكتب أول حرف من اسم الـ class بحرف كبير، مثل class Login

مثال :

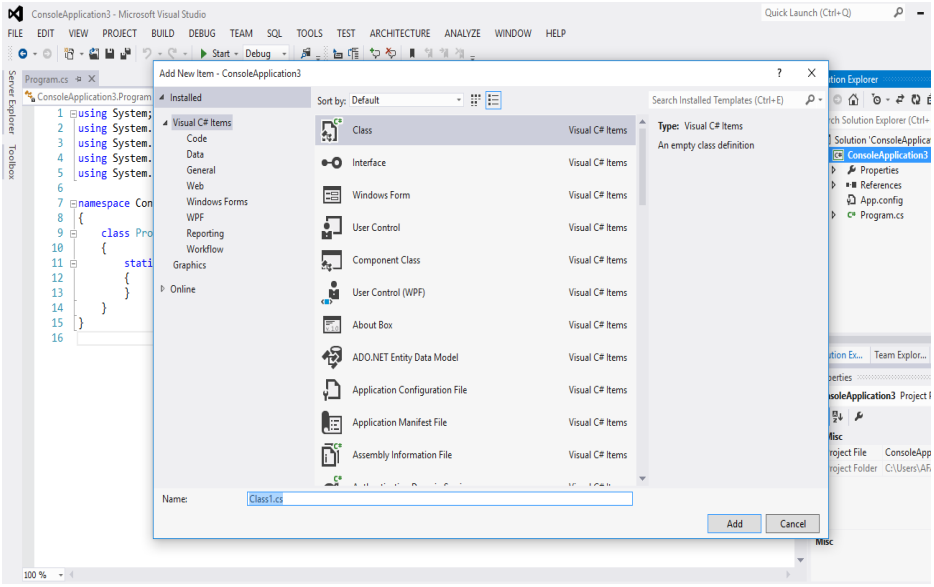
```
using System;
namespace ConsoleApplication1
{
  class First
    {
    }
  class Program
  {
    static void Main( )
          {
          }
    }
}
```

يمكننا إنشاؤه عن طريق كتابته مباشرةً كما فعلت، لكن يُفضل أن يكون كل class في صفحة مستقلة لكي يكون شكل الكود منظم وسهل في القراءة والتعديل.

نذهب إلى اسم المشروع على الفيجوال ستوديو ثم نضغط ضغطة يمين بالمؤشر فتظهر قائمة بها كلمة Add ونضغط على class ونسميه ونضغط enter، هكذا نكون قد أنشأنا class.



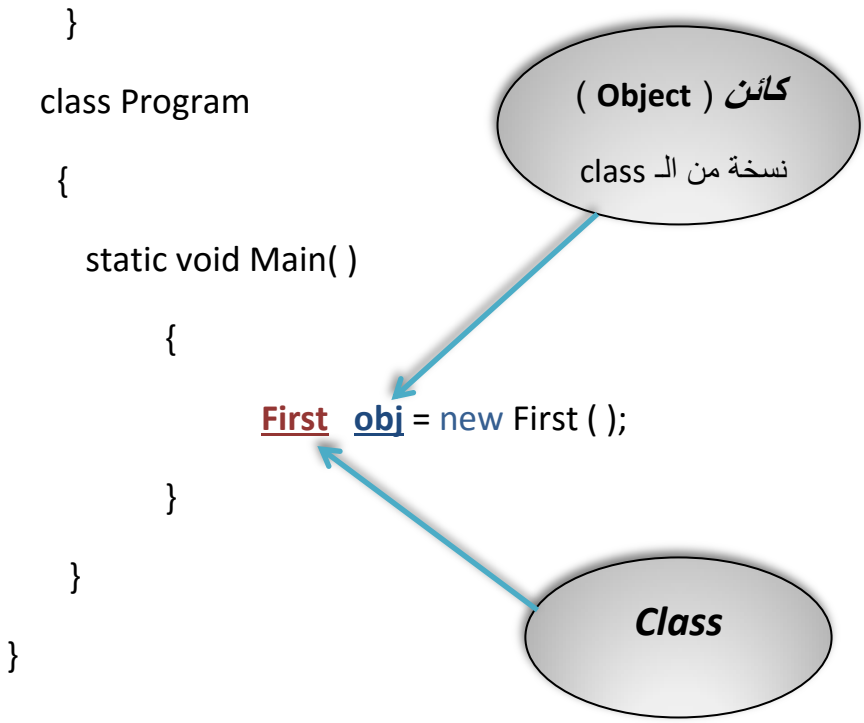
(١)



(٢)

لكي نأخذ نسخة من هذا ال class بكل البيانات التي بداخله " المسموح الوصول إليها فقط " نكتب اسم الـ class ثم اسم المتغير، لكن في هذه الحالة يسمى كائن (object) وليس متغير.

```
using System;

namespace ConsoleApplication1

{

  class First

   {

     public int age ;

     string name ;
```



تستخدم الكلمة المحجوزة **new** لخلق كائن جديد، واسم الـ class يتبعه قوسين يسمى constructor (" مُشيد ") سنعرف ماهو المُشيد لاحقاً، كل ما عليك معرفته مؤقتاً أن هذه طريقة تعريف الـ class وأخذ نسخة منه أو إنشاء كائن جديد منه.

عندما نُنشيء كائن (object) من class فإنه في ذلك الوقت يحتوي على كل خواص هذا الـ class المسموح له رؤيتها.

- نستطيع الوصول إلى كل خصائص الـ class من خلال "اسم الكائن (object) متبوعاً بنقطة " (.)

بعدها يمكننا أن نمرر له القيمة.

مثال :

أكتب class لموظف يحمل البيانات الآتية :

(" الاسم ، العمر ، المرتب ، رقم الموبايل ").

```
using System;
namespace ConsoleApplication1
{
class Employee
{
        public string Name;
        public int Age;
        public int Salary;
        public string Mobile_number;
}
 class Program
     {
        static void Main( )
                {
                        Employee emp = new Employee( );
                        emp.Name = " ali " ;
                        emp.Age = 30 ;
                        emp.Salary = 3000 ;
                        emp.Mobile_number = " +201111111111 " ;
                }
        }
}
```

أولاً أنشأنا class بعد ذلك عرفَنا الخصائص التي بداخله وإذا أردنا رؤيتها خارج هذا الـ class يجب أن يكون نوعها public كما عرفنا مسبقاً،

ثم أخذنا نسخة "كائن" (object) من هذا الـ class ومررنا قيم لهذه البيانات المجردة.
كما اتفقنا البيانات التي تكون بداخل الـ class تكون مجردة أي أنها لا تحتوي على قيمة.

this

الكلمة المحجوزة this تستخدم بداخل الـ class وتستطيع رؤية كل ما بداخل الـ class وتعمل كأنها كائن من هذا الـ class فنستخدمها بداخل الدوال التي بداخل الـ class لكن غير مسموح إستخدامها بداخل دوال من النوع static ولا تستطيع رؤية المتغيرات أو الحقول التي تكون من النوع static.

سنُعيد نفس المثال السابق لكن بإستخدام الدوال.

```
using System;
namespace ConsoleApplication1
{
class Employee
{
        string Name ;
        int Age ;
        int Salary ;
        string Mobile_number ;
        public void add_employee ( string name , int age , int salary ,
        string mobile )
        {
        this.Name = name ;
        this.Age = age ;
        this.Salary = salary ;
```

```
        this.Mobile_number = mobile ;
        }
}
 class Program
     {
        static void Main( )
                {
                        Employee emp = new Employee( );
                        emp.add_employee ( "ali", 30 , 5000 ,
                        "+201111111111");
                        // لإضافة موظف آخر
                        emp.add_employee ( "john" , 34 , 6000
                        ,"+201222222222");
                        //وهكذا
                }
        }
}
```

أنشأنا دالة ويجب أن يكون نوعها public لكي تُرى خارج الـ class ولكي نستطيع التعامل معها، وبها المُدخلات التي سنمرر قيم من خلالها لخصائص الـ class أي للموظف كما في المثال.

بعدها أنشأنا كائن من هذا الـ class ثم أستدعينا الدالة التي تضيف البيانات ومررنا لها القيم بشكل مباشر. يمكنك أن تأخذ البيانات من المُستخدم بأن تعرفّ متغيرات في الدالة الرئيسية ثم تأخذ قيم لها من المستخدم ثم تمرر هذه المتغيرات للدالة.

```
string name = Console.ReadLine( );
int age = int.Parse(Console.ReadLine( ));
int salary = int.Parse(Console.ReadLine( ));
string mobile = Console.ReadLine( );
emp.add_employee ( name , age , salary , mobile );
```

وهكذا....

Constructor (المُشيدّ)

هو عبارة عن دالة ويكون بنفس اسم الـ class لكنه لا يرجع أي بيانات
أي لا يكون له قيمة عائدة " Return value " مثل int أو String ولا حتى void.
تعمل هذه الدالة تلقائياً عندما نأخذ نسخة من هذا الـ class يُكتب في هذه الدالة قيم إبتدائية نريد أن نمررها لمتغير ما ومن الممكن أن نمرر له مدخلات أثناء الإنشاء كالدالة تماماً، ويوجد أكثر من نوع للـ constructor.

أول نوع هو :

١- الـ **default " التلقائي "** :

وهذا النوع لا يأخذ أي مُدخلات أثناء الإنشاء ويمكننا إعطاء قيم إبتدائية للحقول التي تكون في الـ class، وحتى لو لم نكتب الـ default constructor في الـ class فإنه يُنشأ تلقائياً، ويعطي للحقول التي من النوع int قيمة إبتدائية صفر وللحقول التي من النوع string قيم بتدائية ب null، و null تُعبرّ عن اللا شيء.

طريقة تعريف المُشيد التلقائي :

```
public + Class_Name( )
{

}
```

مثال :

```
class Test
{
        public int id ;
        public string name;
public Test( )
  {
        this.id = 1;
        this.name = "ali";
```

```
    }
}
class Program
{
        static void Main( )
         {
                Test T = new Test( ); // Default constructor.
                Console.WriteLine("My id is "+T.id + "and my name is " +
                                        T.name );
                // My id is 1 and my name is ali
         }
}
```

٢- مُشيدّ له مُدخلات (Parameterized constructors)

وهو يأخذ مُدخلات مثل الدالة تماماً وعند أخذ نسخة من هذا الـ class فيجب تمرير مُدخلات له حتى يتم تنفيذه.

طريقة تعريف المُشيد الذي يأخذ مُدخلات :

```
public + Class_Name( المُدخلات )
{

}
```

مثال :

```
class Employee
{
        string Name ;
        int Age ;
```

```
        int Salary ;
        string Mobile_number ;
        public Employee( ) // default constructor
        {  }
        public Employee( string name , int age )
        {       // Parameterized constructors
                Name = name ;
                Age = age ;
        }
        public void add_employee ( string name , int age ,
                                        int salary,string mobile )
        {
        this.Name = name ;
        this.Age = age ;
        this.Salary = salary ;
        this.Mobile_number = mobile;
        }
}
```

هذا المُشيد عندما تأخذ نسخة من الـ class فإنه يعمل تلقائياً ويطلب منك أثناء تعريف الكائن أن تضع قيم إبتدائية للبيانات التي أدخلتها في المُشيد أثناء إنشاؤه لكن إذا لم تدخل له مدخلات فلن يطلب منك أي شيء، وبالطبع هذه القيم الإبتدائية هي مجرد قيم مؤقتة يمكنك تغييرها، فمثلاً إذا أستدعيت الدالة التي تُدخل البيانات فالبيانات الجديدة التي أدخلتها هي التي ستُخزن، البيانات التي بداخل المُشيد هي مؤقتة وتُخزن فقط في حالة أنك لم تدخل أي بيانات أخرى للمتغير.

```
Employee Emp = new Employee ( " Mahmoud ", 30 ) ;
```

هكذا يكون شكل المُشيد أثناء إستدعاؤه أو أثناء تعريف كائن جديد.

٣- (**Copy Constructor**)

وهذا النوع وظيفته هي أنه ينسخ قيم من (Object) كائن إلى (Object) كائن آخر من نفس الـ class، وأثناء تعريفه يأخذ object كائن أو نسخة من الـ class كمُدخل.

مثال :

```
using System;
namespace Copy_Constructor
{
   class Program
   {
      static void Main(string[ ] args)
      {
         employee e = new employee(10, "John", 3000);
         employee em = new employee( e );
         em.print( );
         // Employee id : 10 , Employee name : John ,
         // Employee salary : 3000
      }
   }
   class employee
   {
      int id;
      string name;
      int salary;
      public employee(int _id, string _name, int _salary)
      {
         this.id = _id;
         this.name = _name;
         this.salary = _salary;
      }
```

```
    public employee(employee emp)
    {
      this.id = emp.id;
      this.name = emp.name;
      this.salary = emp.salary;
    }
    public void print()
    {
      Console.WriteLine(" employee id is : "+id+" \n Employee name :
      "+name+" \n Employee salary : "+salary);
    }
  }
}
```

Destructor (الهدّام)

الهدّام عكس المُشيد، هو أيضاً عبارة عن دالة تعمل تلقائياً بعد إنتهاء البرنامج، ووظيفته أنه يهدم أي كائن لم يُستَخدَم أو كائن أُستخدِمَ لكن لم يعد له حاجة أو كائن لم يُستَخدَم، وهو ليس كالمُشيد تحتاج أن تستدعيه. هو يعمل فور إنتهاء البرنامج دون إستدعاء ولا يحتاج إلى (Access modifier)، وكل class يكون له destructor واحد فقط.

طريقة التعريف :

~ Class_Name ( ) { }

هذه العلامة (~) تُسمى تيلدا .

مثال :

```
class Employee

{
```

```
        string Name;
        int Age;
        int Salary;
        string Mobile_number;
        public Employee( ) // default constructor
        {  }
        public Employee( string name , int age )
        {
                Name = name ;
                Age = age ;
        }
        public void add_employee ( string name , int age , int salary ,
        string mobile )
        {
        this.Name = name ;
        this.Age = age ;
        this.Salary = salary ;
        this.Mobile_number = mobile ;
        }
        ~Employee ( )
        {
                GC.Collect( ) ;
         }
}
```

GC.Collect يشير إلى ---> Garbage Collector الذي تحدثنا عنه في بداية الكتاب في الجزء الخاص بإطار العمل.

20

الخصائص

(Properties)

get , set

الموصلّات (Accessors)

الحقول التي تكون بداخل الـ class تكون محمية ومخفية على المستخدم ولا يستطيع الوصول إليها، لكن ماذا لو كنا نريد تمرير قيم لهذه الحقول من خارج الـ class بدون أن نتعامل مع الحقول بشكل مباشر أو بدون دوال؟

في هذا الوقت سنستخدم set و get وهما عبارة عن كلمتين محجوزتين في اللغة حيث تُمكنك set من وضع قيم لهذه الحقول أي للكتابة فقط، و get تُمكنك من قراءة البيانات التي تكون بداخل الحقل أي تكون للقراءة فقط.

يوجد ثلاثة طرق للوصول إلى هذه الموصلّات :

Public

Private

Protected

الموصلّات تعامل معاملة الدوال.

فنحن نتحكم مَن يستطيع تمرير قيم للحقول ومَن لا يستطيع، مَن يمكنه رؤيتها ومَن لا يمكنه.

وتُستخدم set في تحديد القيمة التي يُدخلها المستخدم، فمثلاً إذا كنت ستُدخِل قيمة لمرتب لكن هذه القيمة غير منطقية مثل مرتب موظف ١ جنية في الشهر فأنت يمكنك أن تتحكم في مدى القيم الداخلة ويجب أن يُدخل المستخدم قيم في المدى المطلوب لكي تُخزن بداخل الحقل " المتغير في الـ class " فالقيم تأتي أولاً عن طريق set وإذا كان الشرط خطأ فلن تُخزن في الحقل، يُمكنك تحديد شرط قبل إستقبال القيم بداخل set أو لا تضع شروط، فالشرط ليس إضطرارياً لكن للتأكد من القيم فقط قبل تخزينها.

مثال :

```
class Employee
{
        private string  Employee_Name ;
        private int Employee_Age ;
        private int Employee_Salary ;
        public string Name
        {
                set
                {
                        Employee_Name = value;
                }
                get
                {
                        return Employee_Name;
                }
        }
        public int Salary
        {
                set
                {
                        if ( value <1000 )
                        {
                                Console.WriteLine (" invalid data ") ;
                        }
```

```
                else
                {
                        Employee_Salary = value ;
                }
        }
        get
        {
                return Employee_Salary ;
        }
    }
}
```

معنى الكود أن :

أولاً أنشأنا class ثم عرفّنا فيه الحقول (" الأسم ، العمر ، المرتب ") وطرق الوصول إليهم private، ثم أنشأنا خصائص لكي نتعامل مع هذه الحقول ونمرر لها قيم أو نقرأ منها هذه القيم.

الخصائص (" الأسم ، المرتب ") يتم إنشاء الخصائص تبعاً للحقول التي نريد للمستخدم التعامل معها بشكل غير مباشر فننشيء خاصية لكل حقل على حدة، وتسمى الخاصية بأي اسم نريده لكن يُفضل بأن تكون باسم قريب من اسم الحقل لكي نميزها.

ثم ننشيء الموصلّات (set , get)، ولا يشترط كتابة set و get معاً في نفس الخاصية، بل يمكننا جعل هذا الحقل للقراءة فقط وعندها نستخدم get فقط، أو للكتابة فقط فنستخدم set أو كليهما.

نمرر القيمة للحقل من خلال كلمة value وهي كلمة محجوزة في اللغة تأخذ القيمة التي تأتي من المستخدم وتمررها للحقل المشار إليه ويجب أن تكون القيمة من نفس نوع الحقل او المتغير.

في خاصية الـ Salary أنشأنا شرطاً لكي نتأكد من القيمة التي تأتي من المستخدم هل هي في المدى المطلوب وفي حالة كانت في المدى المطلوب تُخزن بداخل الحقل أما إذا كانت خارج المدى فلا تُخزن.

وفي get نَسترجع القيمة التي بداخل الحقل من خلال الكلمة المحجوزة return.

21

الوراثة

(Inheritance)

الوراثة بالمعنى العامي هو أن يرث الشخص بعض الصفات أو الأشياء الخاصة بأبيه أو أحد أقاربه، كذلك الأمر في البرمجة، عندما يرث class ما class آخر فإنه يرث منه كل الدوال والمتغيرات أو الصفات الخاصة بهذا الـ class ويصبح الـ class الوارث يسمى أبن والـ class الموروث يسمى أب.

والوراثة هي من أهم مباديء السي شارب وهي تسهل وتختصر علينا الكثير من الوقت والجهد، فعندما نرث class ما فإننا نستخدم كل شئ بداخل هذا الـ class بدون الحاجة إلى كتابتها مرة أخرى في الـ class الجديد.

ملاحظة :

نرث كل شئ مسموح لنا من قِبل الـ class الأب لأن عملية التغليف (Encapsulation) تلعب دوراً هاماً في الوراثة فأنت تحدد من يستطيع رؤية البيانات في class آخر ومن لا يستطيع، فمثلاً يوجد class اسمه employee و class آخر اسمه manager

employee يوجد به حقل للأسم والعمر والعنوان والمرتب وبعض الدوال، وفي manager نحتاج أيضاً هذه الحقول بالإضافة إلى بعض الحقول الأخرى، فهل سنكرر نفس الحقول؟

الوراثة جعلتنا نستغنى عن هذا التكرار ويمكننا وراثة الـ class الذي نحتاج الصفات التي بداخله ونستخدمها كيفما نشاء.

لغة الـ C# لا تدعم الوراثة المتعددة، الوراثة المتعددة تعني أن يرث الـ class الأبن أكثر من class آخر.

لكن يوجد شئ آخر يعوض هذا القصور يسمى interface

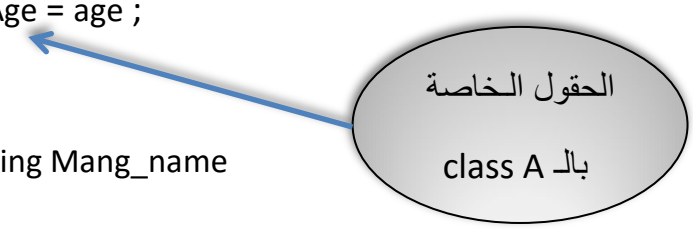
وسنتطرق لفهم هذا المعنى لاحقاً.

تتم وراثة class لـ class آخر عن طريق النقطتين (:)

مثال :

```
using System ;
namespace ConsoleApplication1
{
   Class Employee
   {
     public string _Name ;
     public int _Salary ;
     public int _Age ;
     public void show_data ( )
      {
        Console.WriteLine( " Show ");
      }
  }
  Class Manager : Employee
   {
     public Enter_Data ( string name , int salary , int age )
     {
```

```
            _Name = name ;
            _Salary = salary ;
            _Age = age ;
    }
    public string Mang_name
    {
            get
            {
                    return _Name ;
            }
    }
}
class Product : Manager
 {
   Console.WriteLine(" show product data ");
 }
class Program
{
   static void Main ( )
     {
        Manager child = new Manager( ) ;
        string n=Console.ReadLine( ) ;
        int ag = int.Parse( Console.ReadLine ( ) ) ;
        int sal = int.Parse( Console.ReadLine( ) ) ;
        child. Enter_Data ( n , sal , ag ) ;
        Console.WriteLine ( child.emp_name ) ;
        Product  p = new Product ( ) ;
```



```
        }
    }
}
```

معنى هذا الكود أن :

أنشأنا class Employee وعرفّنا به الحقول الخاصة به ثم أنشأنا class Manager وورثنا الـ Employee وأنشأنا دالة تأخذ القيم من المستخدم وتضعها في هذه الحقول ثم أنشأنا الخاصية get لتعيد الأسم.

في الـ class Product ورثنا Manager فأصبح الـ Product يرى كل ما بداخل الـ Manager والـ class الذي يرثة الـ Manager أيضاً. فيمكننا إستخدام كل الدوال والحقول التي بداخل هذه الـ classes من خلال الـ Product.

يمكن للـ class الأبن رؤية الخواص التي بداخل الـ class الأب التي يكون الوصول إليها public أو protected أو internal أو protected internal ماعدا private.

عندما تكون خواص الوصول في الـ class الأب من النوع protected فإنه لا يمكن لأي class آخر رؤيتها سوى الـ class الذي يرثه فقط.

إخفاء الدالة

إذا كان يوجد class اسمه A ثم ورثه class آخراسمه B وكان يوجد في الـ class A دالة اسمها print( ) وتريد إنشاء دالة بنفس الأسم في الـ class B فيمكنك إنشاء الدالة وإستخدام الكلمة المحجوزة new، لو لم تستخدم new وكتبت نفس الدالة بنفس الاسم لن يحدث خطأ لكن سيظهر لك تحذير ويكتب لك البرنامج أن تستخدم كلمة new في الدالة التي تكون في الـ class الأبن التي تكون بنفس أسم الدالة التي في الـ class الأب حتى تُخفي الدالة التي في الـ class الأب وتعمل على الدالة الجديدة.

مثال :

```
class A
{
    public void print( )
      {
          Console.WriteLine(" Hello ");
      }
}
class B : A
{
    public new void print( )
      {
          Console.WriteLine(" C# ");
      }
}
```

أما في حالة أنك أردت إستخدام نفس الدالة التي في الـ class A بالكود الذي بداخلها في الـ class B نكتب نفس الدالة وكلمة new ثم نكتب بداخل الدالة الكلمة المحجوزة base.print( )، الكلمة المحجوزة base تعمل على الـ class الأب، تأخذ نسخة منه

"object" أي أنها ترى كل ما بداخل الـ class الأب ما عدا الحقول أو الدوال التي يكون الوصول إليها private.

مثال :

```
class A
{
      public void print( )
        {
             Console.WriteLine(" Hello ");
        }
}
class B : A
{
      public new void print( )
        {
            base.print( );
        }
}
class Program
{
          static void Main( )
              {
                  B child = new B( );
                  child.print( );
              }
}
```

سيطبع : Hello

22

(Interface)

ذكرنا الـ interface سابقاً في الوراثة، الـ interface هو عبارة عن حاوية لمجموعة من الخصائص أو الدوال المكتوب رأسها فقط بدون كتابة جسم الدالة أي بدون كتابة كود داخلها، فقط اسم الدالة فتكون مجردة، يمكننا وراثة هذا الـ interface لكن عند إستدعاءه يتطلب تنفيذ وكتابة الأكواد بداخل كل الدوال.

طريقة الوصول للـ interface تكون public تلقائياً فلا يجب كتابة public بجانب الدوال لكن نكتبها في الـ class حتى نستطيع رؤيتها خارج الـ class.

يمكننا إنشاء الـ interface من خلال الكلمة **interface** وهي كلمة محجوزة في اللغة.

يمكننا إنشاء أكثر من interface ويمكننا أيضاً وراثة أكثر من interface في نفس الوقت فهو يدعم الوراثة المتعددة.

كيفية تعريفه :

الطريقة المتعارف عليها في تعريف الـ interface أن أول حرف من اسم الـ interface يكون حرف i ثم يتبعه باقي الاسم الذي تريده، هذا غير ضروري لكنها الطرق المتداولة والمتعارف عليها في تعريفه.

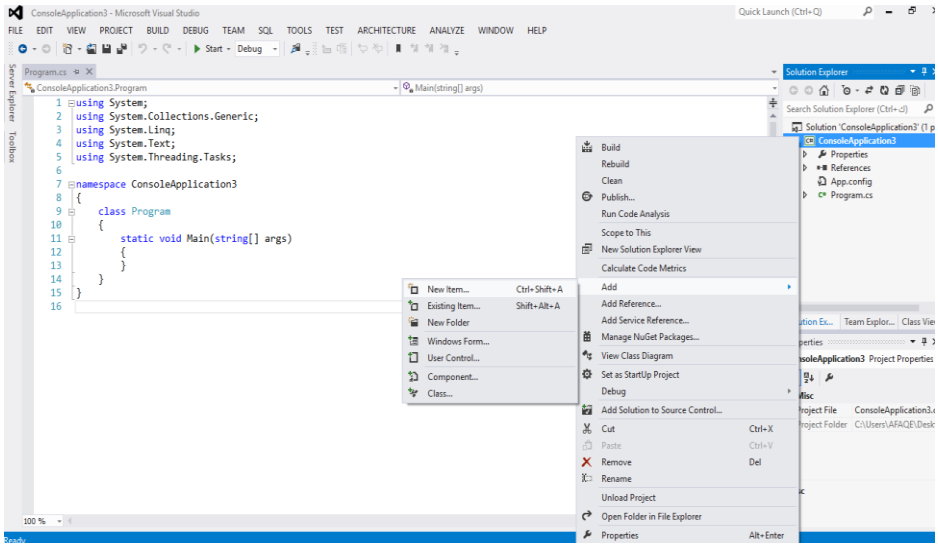
يمكننا كتابة كلمة interface + اسمه

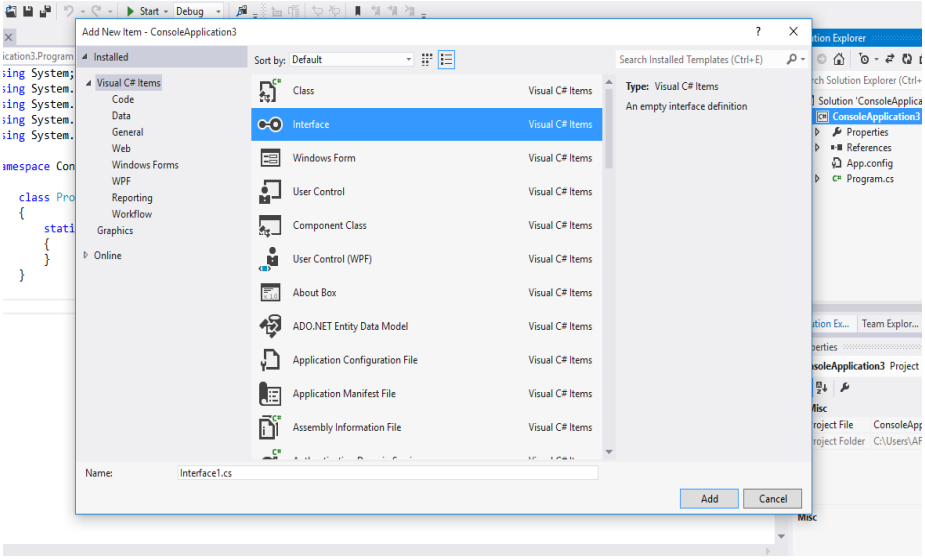
```
interface emp1
  {
    void print1 ( ) ;
    int sum ( int x , int y ) ;
```

```
   }
interface emp2
{
    void print2 ( ) ;
    int multi ( int x , int y ) ;
}
class A : emp1 , emp2
{
    public void print2 ( )
      {
         Console.WriteLine( " employee 2 " ) ;
      }
    public void print1 ( )
      {
         Console.WriteLine( " employee 1 " ) ;
       }
    public int sum ( int x , int y)
      {
         return x + y ;
       }
    public int multi ( )
    {
         return x * y ;
    }
}
```

أو يمكننا إنشاؤه من خلال الضغط على اسم المشروع ونضغط على كلمة Add ثم نختار New item ونختار Interface

ونكتب به الدوال التي نريد.


ConsoleApplication3 - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM SQL TOOLS TEST ARCHITECTURE ANALYZE WINDOW HELP
Program.cs
ConsoleApplication3.Program
Main(string[] args)
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace ConsoleApplication3
{
class Program
{
static void Main(string[] args)
{
}
}
}
Solution Explorer
Solution 'ConsoleApplication3' (1 p
ConsoleApplication3
Properties
References
App.config
Program.cs
Build
Rebuild
Clean
Publish...
Run Code Analysis
Scope to This
New Solution Explorer View
Calculate Code Metrics
Add
Add Reference...
Add Service Reference...
Manage NuGet Packages...
View Class Diagram
Set as StartUp Project
Debug
Add Solution to Source Control...
Cut Ctrl+X
Paste Ctrl+V
Remove Del
Rename
Unload Project
Open Folder in File Explorer
Properties Alt+Enter
New Item... Ctrl+Shift+A
Existing Item... Shift+Alt+A
New Folder
Windows Form...
User Control...
Component...
Class...
100 %
Ready


Start Debug
Add New Item - ConsoleApplication3
Installed
Visual C# Items
Code
Data
General
Web
Windows Forms
WPF
Reporting
Workflow
Graphics
Online
Sort by: Default
Search Installed Templates (Ctrl+E)
Class Visual C# Items
Interface Visual C# Items
Windows Form Visual C# Items
User Control Visual C# Items
Component Class Visual C# Items
User Control (WPF) Visual C# Items
About Box Visual C# Items
ADO.NET Entity Data Model Visual C# Items
Application Configuration File Visual C# Items
Application Manifest File Visual C# Items
Assembly Information File Visual C# Items
Type: Visual C# Items
An empty interface definition
Name: Interface1.cs
Add
Cancel
Solution Explorer
ConsoleApplication3
Properties
References
App.config
Program.cs

مثال :

```
interface functions
{
    void eat( );
   void drink( );
}
class Human : functions
{
   public void eat( )
   {
   Console.WriteLine( " eating " ) ;
   }
  public void drink( )
  {
  Console.WriteLine(" drinking ");
  }
}
```

يجب تنفيذ كل الدوال في الـ class التي أُنشِئتْ بداخل الـ interface

أهمية الـ interface أنه أمكننا إستخدام نفس الدالة بنفس الأسم في أكثر من class وبأكثر من تنفيذ من خلال وراثة الـ interface.

23

(Polymorphism)

الـ polymorphism تُعد من أهم مفاهيم البرمجة كائنية التوجه، والمعنى العام لها أننا يمكننا إعادة إستخدام الدالة بنفس الأسم داخل نفس الـ class أو class آخر بعدد المرات التي نريدها.

وهي تنقسم إلى :

١- OverLoading

هي عبارة عن إنشاء أكثر من دالة بداخل نفس الـ class بنفس الأسم، لكن بشروط معينة.

لنفترض أنه توجد دالة أسمها print ولا تأخذ مُدخلات ونريد إنشاء دالة أخرى اسمها print في هذه الحالة يجب أن نضيف لها مُدخل واحد على الأقل مثل (print (int x حتى نميزها عن الأخرى في وقت الإستدعاء ولا يحدث تشتت للمترجم، الدالتان بنفس الأسم فكيف يفرق بينهما؟!

فهو يعتبر الدالة التي تستدعيها مجهولة لأنه لم يعرف أي منهما تقصد، لكن عند إضافة مُدخل فحينها سيذهب إلى التي قصدتها أهي التي بمُدخل أم لا.

لكن لن نضيف مُدخل لن نستخدمه بغرض إضافة دالة جديدة بنفس الأسم فهذا غير صحيح، تضيف فقط ما تحتاج إليه فعلياً في الكود وما تستخدمه فقط.

أما إذا في حالة أن الدالتين يأخذان مُدخل فيجب أن نغير نوع أحدهما، فمثلاً إذا كانت الدالة (sum (int x فيجب أن يكون نوع المُدخل في الدالة الجديدة أي نوع غير int، من الممكن أن يكون float أو double أو long.

بإختصار لا يجب أن يكون هناك دالتان متماثلتان في نفس الـ class، يجب أن يوجد إختلاف في مُدخلات أحدهما على الأقل.

ملاحظة : لا تحدث عملية الـ overloading لدالة لا تأخذ مُدخلات.

```
Program.cs*
ConsoleApplication3.Program                    Main(string[] args)
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Linq;
 4  using System.Text;
 5  using System.Threading.Tasks;
 6
 7  namespace ConsoleApplication3
 8  {
 9      class Program
10      {
11          static void Main(string[] args)
12          {
13              A ob = new A();
14              ob.print(
15          }   ▲ 1 of 2 ▼ void A.print()
16      }
17      class A
18      {
19          public void print(int x)
20          {
21              Console.WriteLine("Hello");
22          }
23          public void print()
24          {
25              Console.WriteLine("Thanks");
26          }
27      }
28  }
29
```

في وقت إستدعاء الدالة و فور فتح قوس الدالة أثناء الإستدعاء فإن برنامج الفيجوال ستوديو يوضح لك أنه توجد دالتان بنفس الأسم لكن أحدهما لا تأخذ مُدخلات بينما الأخرى تأخذ مُدخل ويجب أن تمرر لها المُدخل عند إستدعائها في حالة أنك أخترت إستدعاء من تأخذ مُدخل ويجب أن يكون من نفس النوع الذي حددته، ونختار بينهما من خلال الأسهم لأعلى ولأسفل.

مثال :

```
using System ;
namespace ConsoleApplication1
{
class Calculator
{
    public void sum ( int x , int y )
            {
               Console.WriteLine( x + y );
            }
   public void sum ( int x , float y )
            {
               Console.WriteLine( x + y );
            }
}
class Program
{
    static void Main( )
      {
          Calculator ca = new Calculator ( );
          ca.sum( 3 , 4 );
          ca.sum( 5 , 4.2f );
                                  //. float أضفنا حرف f لأن المُدخل
     }
 }
}
```

٢- Overriding

تتم عملية الـ override في الوراثة، في حالة وجود دالة ما في class اسمه employee وورثت هذا الـ class في class اسمه manager وتريد إنشاء دالة بداخل manager بنفس الأسم فهذه تسمى override.

وتستخدم الكلمة المحجوزة في اللغة " virtual " في الدالة التي في الـ class الأب، والكلمة المحجوز " override " في الدالة التي بنفس الأسم داخل الـ class الأبن.

وتكون الدالتان متماثلتين تماماً، في الأسم وعدد المُدخلات ونوع المُدخلات، وجسم الدالة أو الكود الذي بداخل الدالة في الـ class الأب أو الأبن تكتب به ما تشاء أو الوظيفة التي تريد تنفيذها بالدالة.

مثال :

```
using System ;

namespace ConsoleApplication1

{

    class Employee

     {

         public virtual void print ( )
              {
                 Console.WriteLine ( " Show data of employee " );
              }
     }
   class Manager : Employee
     {
        public override void print ( )
```

```
        {
            Console.WriteLine( " Show data of manager " );
        }
    }
  class Program
    {
        static void Main (  )
        {
        Manager M = new Manager ( );
        M.print ( );    // show data of Manager
        Employee E = new Employee ( );
        E.print ( );     // show data of employee
        }

    }

}
```

24

(Abstract , Sealed , Partial)

abstract

هي عبارة عن كلمة محجوزة تأتي مع الـ class أو الدالة (Method) وعندما تأتي مع الـ class فلا يمكننا أن نأخذ كائن (object) أو نسخة من هذا الـ class يكون فقط للوراثة. أُنشيء لكي يخدم classes أخرى.

مثال :

```
using System ;
namespace ConsoleApplication1
{
    abstract class Emp
    {
      public int x ;
    }
    class Man : Emp
    {
      public int num
        {
          set
          {
                x = value ;
          }
          get
          {
```

```
            return x ;
        }
    }
}
class Program
 {
    static void Main ( )
                {
                    Emp e = new Emp ( ) ;
                    // خطأ ، لا نستطيع أخذ نسخة منه
                    Man m = new Man ( ) ;
                    m.num = 2 ;
                    Console.WriteLine( m.num ) ; // سيطبع ٢
                }
    }
}
```

إذاً الـ class الذي يكون abstract ممنوع التعامل المباشر معه فيما عدا الوراثة.

النوع الثاني الذي يكون abstract هي الدوال :

الدالة التي تكون abstract تكون أيضاً بداخل abstract class، وتكون الدالة مجردة أي رأس الدالة فقط بدون الكود داخلها، لكن بشرط أن من يرث هذا الـ class يجب أن يكتب الكود بداخل هذه الدالة الـ abstract، هي تشبه للـ interface إلى حد ما. لكن لا تخلط بين الأثنين لأن تلك دالة (method) بينما هذا interface.

وعند كتابة كود الدالة في الـ class الأبن ... نكتب نفس رأس الدالة ونكتب override عوضاً عن abstract ثم نكتب الكود.

الفرق بين virtual و abstract :

virtual : يجب أن نكتب كود الدالة التي تكون virtual أي لا تكون دالة مجردة.

abstract : يجب ألا نكتب كود الدالة التي تكون abstract أي تكون مجردة.

مثال :

```
using System ;
namespace ConsoleApplication1
{
   abstract class Emp
     {
          public abstract void print ( ) ;      // abstract method
          public virtual void view ( )         // virtual method
            {
                      Console.WriteLine( " Hello " ) ;
            }
     }
   class Manager : Emp
     {
         public override void print ( )
          {
             Console.WriteLine( " Show " ) ;
          }
       public override void view ( )
          {
                  Console.WriteLine( " Hi " ) ;
          }
     }

}
```

sealed

تأتي مع تعريف الـ classes والدوال (methods). عندما تأتي مع الـ class فهذا يعني أنه ممنوع وراثة هذا الـ class، لكن يمكننا التعامل معه بطريقة عادية ونستطيع أخذ نسخ منه أو كائن (object).

مثال :

```
sealed class A
  {

  }
class B : A
 {

 }
```

سيحدث خطأ أثناء معالجة الكود لأنه لا يمكن وراثة الـ class الذي يكون sealed

أما في حالة الدوال :

فالدالة التي تكون sealed فهذا يعني أنه لا يمكن عمل override لهذه الدالة في class آخر.

مثال :

```
class A
{
    public sealed override void print ( )
    {
        Console.WriteLine( " C# " ) ;
    }
}
class B :  A
```

```
{
    public override void print ( )
     {
         Console.WriteLine( " Hello " ) ;
     }
}
```

سيحدث خطأ أثناء معالجة الكود لأنه لا يمكن عمل override لدالة sealed.
فائدة sealed أنها تمنع إعادة إستخدام محتويات الـ class في class آخر أو إعادة إستخدام دالة بنفس الأسم ونفس المُدخلات في class آخر.

partial

هي كلمة محجوزة في اللغة وتأتي كلمة partial مع الـ classes وأيضاً مع الـ interface ومع struct وسنتطرق لشرح الـ struct لاحقاً.

معناها أنه يمكننا تقسيم الـ class أو الـ interface على أكثر من ملف. أي يمكننا كتابة class بأسم و class آخر بنفس الأسم وبدوال مختلفة، بالطبع هذا لا يجوز في الحالة العادية لكن يكون متاحاً عندما نستخدم partial.

في حالة إستخدامها مع الـ class :

نكتب كل class في مكان أو ملف مستقل عن الآخر وبنفس الأسم وبدوال مختلفة لكن عند أخذ نسخة من هذا الـ class فإننا في هذا الوقت نتعامل مع class واحد أي أنه يستدعي كل الدوال التي تكون في كل الملفات معاً.

تُكتب الكلمة قبل الـ class أو الـ interface.

مثال :

```
using System ;
namespace ConsoleApplication1
{
```

```
partial class Circle
  {
      public void Show_Diagonal ( float R )
        {
            Console.WriteLine( R * 2 ) ;
        }
  }
partial class Circle
  {
     public float Circle_Area ( float R )
        {
            return 3.14f * R * R ;
        }
  }
class Porgram
  {
   static void Main ( )
        {
            Circle c = new Circle ( ) ;
            c.Show_Diagonal ( 3.5 ) ;
            c.Circle_Area ( 2.3 ) ;
        }
  }
```

ستظهر كل الدوال والحقول التي تكون في الملفين.

ونستطيع كتابة الـ class الواحد على أكثر من ملف.

أما في حالة الـ interface فإنه أيضاً ينقسم على أكثر من ملف لكن عند وراثة الـ interface فإنه يجب أن ننفذ ونكتب كل الأكواد للدوال التي بداخل الـ interface الموجودة في كل الملفات، لكن نكتبها في الـ class الذي ورث الـ interface ليس في الـ interface نفسه لأنه لا يجوز كما عرفنا.

مثال :

```
partial interface Ifunction
 {
     void print ( ) ;
 }
partial interface Ifunction
{
     void view ( ) ;
}
 partial interface Ifunction
{
     void show ( ) ;
 }
class Test : Ifunction
{
     public void print ( )
      {
         Console.WriteLine( " One " ) ;
      }
      public void view ( )
      {
         Console.WriteLine( " Two " ) ;
      }
      public void show ( )
     {
         Console.WriteLine( " Three " ) ;
     }
}
```

تمارين :

أكتب برنامج لتخزين بيانات موظفين (id, name , age , salary).

```
using System;
namespace Test
{
class employee
  {
  employee[ ] emp;
  int size;
  int index;
  int id;
  int age;
  string name;
  int salary;
 public employee( ){ }
 public employee(int size)
  {
     this.size=size;
     index = 0;
     emp=new employee[size];
  }
 public void add_employee(int id,string name,int age,int salary)
  {
    bool check = false;
    for(int i=0;i<index;i++)
      {
        if(emp[i].id==id)
           {
                check=true;
                break;
```

ننشيء class اسمه employee وبما أننا سنخزن بيانات أكثر من موظف أي أكثر من أوبجيكت لهذا الكلاس فإننا سنستخدم الـ array لأن الـ objects كلهم من نفس النوع أي كلهم class واحد وسنأخذ منه أوبجيكت، فنحدد حجم المصفوفة من خلال الـ constructor، وعرّفنا المتغير index حيث أنه يعبرّ عن الـ index الذي سنضيف بداخله.

هذه دالة الإضافة حيث أن كل index يكون بداخله أوبجيكت يحمل id,name,age,salary أي البيانات الخاصة به، وبعدما نضيف بيانات موظف جديد نزيد الـ index بمقدار 1 ليتحرك إلى المكان الذي يليه في المصفوفة، لكن أولاً نتحقق ما إذا كان الـ id موجود مسبقاً أم لا.

```
                    }
                }
  if (check == true)
  {
     Console.WriteLine(" THis id is exist !! ,  please use an new id");
  }
  else
  {
     emp[index] = new employee();
     emp[index].id = id;
     emp[index].age = age;
     emp[index].name = name;
     emp[index].salary = salary;
     ++index;
  }
 }
public void search(int id)
 {
    bool check = false;
    int emp_index = 0;
    for(int i = 0; i<index ; i++)
       {
          if(emp[i].id==id)
             {
                    check=true;
                    emp_index=i;
                    break;
             }
       }
    if(check==true)
```

هذه الدالة للبحث عن موظف معين من خلال الـ id الخاص به فإذا كان موجود فإنها تطبع البيانات الخاصة به

```
        {
          Console.WriteLine("\t This employee is exist");
          Console.WriteLine("\t Employee id : "+emp[emp_index].id);
          Console.WriteLine("\t Employee name :
                                "+emp[emp_index].name);
          Console.WriteLine("\t Employee age :
                               "+emp[emp_index].age);
          Console.WriteLine("\t  Employee salary :
                              "+emp[emp_index].salary);
        }
      else
        {
          Console.WriteLine("\t \t  This employee not exist");
        }
}
public void delete(int id)
  {
     bool check=false;
     int emp_index=0;
     for(int i=0;i<index;i++)
       {
          if(emp[i].id==id)
             {
                   check=true;
                   emp_index=i;
                   break;
             }
       }
     if (check == true)
     {
```

هذه الدالة لحذف بيانات موظف من المصفوفة من خلال الـ id الخاص به

```
        for (int i = emp_index; i < index - 1; i++)
        {
           emp[i] = emp[i + 1];
        }
        --index;
      }
      else
        Console.WriteLine("this id not exist");
   }
public void print( )
 {
   for(int i = 0 ; i< index ; i++)
    {
        Console.WriteLine("\t Employee id : "+emp[i].id);
        Console.WriteLine("\t Employee name : "+emp[i].name);
        Console.WriteLine("\t Employee age : "+emp[i].age);
        Console.WriteLine("\t Employee salary : "+emp[i].salary);
        Console.WriteLine( ); Console.WriteLine( );
    }
 }
}
class Program
  {
     static void Main( )
     {
          employee emp = new employee( 5 );
          emp.add_employee(1, "aaa", 25, 1500);
          emp.add_employee(2, "bbb", 27, 2000);
          emp.add_employee(3, "ccc", 32, 4000);
          emp.add_employee(4, "ddd", 30, 3500);
```

هذه الدالة لطباعة بيانات كل الموظفين

ثم آخراً نختبر هذا الكود في دالة الـ Main.

```
            emp.search(4);  // سيطبع بيانات هذا الموظف لأنه موجود
            emp.delete(4);
            emp.search(4); // سيطبع أن هذا الموظف غير موجود، حيث أنه تم حذفه
            emp.print( );
        }
    }
}
```

0	1	2	3	4
1,aaa,25,1500	2,bbb,27,2000	3,ccc,32,4000	~~4,ddd,30,3500~~	

25

(Struct)

هو عبارة عن بناء نوع جديد من البيانات، يشبه الـ class في التعريف لكن توجد بعض الإختلافات بينهما.

struct	class
- لا يدعم الوراثة - يكون value type وليس reference type مثل الـ class ، عندما إستخدامه لا تحتاج إلى كلمة new . وبما أنه value type فهو أسرع من الـ class لأنه يُخزن inline أو بنظام الـ stack . - لا يمكن إعطاء قيم إبتدائية للحقول التي بداخله public int x = 5 ; (هذا غير مقبول). - يكون كل أعضاؤه public لإنك لو أخفيتهم لن تستطيع إستخدامهم . - لا يمكننا عمل override للدوال التي بداخله .	- يدعم الوراثة - يكون reference type وعند أخذ نسخة منه نستخدم الكلمة المحجوزة new . - يمكن إعطاء قيم إبتدائية للحقول التي بداخله int x = 10 ; (مقبول).

تستخدم الـ struct عندما تريد إنشاء برنامج لا يحتاج إلى خواص الـ class، برنامج تُخزن به بعض البيانات وربطها ببعضها، وهو يُعَدّ بناء نوع جديد من البيانات.

مثال :

```
struct Test
{
   public int x ;
   public int y ;
   public float z ;
}
class Program
{
     static void Main( )
     {
        Test t ;
        t.x = 5;
        t.y =7 ;
        t.z = 9.9f ;
        Console.WriteLine( t.x + t.y + t.z ) ;
     }

}
```

مثال ٢:

```
using System ;
namespace ConsoleApplication1
{
    class TestClass
    {
       public int num1 ;
    }
```

```
struct TestStruct
{
   public int num2 ;
}
class Program
{
   static void Main ( )
    {
            TestStruct s1 = new TestStruct( ) ;
            TestStruct s2 = s1 ;
            s1.num2 = 100 ;
            s2.num2 = 200 ;
            TestClass c1 = new TestClass ( ) ;
            TestClass c2 = c1 ;
            c1.num1 = 300 ;
            c2.num1 = 400 ;
            Console.WriteLine( s1.num2 ) ;
            Console.WriteLine( s2.num2 ) ;
            Console.WriteLine( c1.num1 ) ;
            Console.WriteLine( c2.num1 ) ;
    }
}
}
```

الناتج :

100

200

400

400

لم تتغير القيم في حقول الـ struct فكأنه يأخذ نسخة من القيم الحقيقية ويغير بها أما التغيير الحقيقي في القيم في الذاكرة العشوائية لم يحدث. بينما تغيرت القيم في حقول الـ class لأنه يعمل بـالـ reference (المرجع) أي تتغير القيم الأصلية وليست النسخة منها.

26

(Namespace)

هي عبارة عن مجموعة من الـ classes يمكننا أن نستخدم الـ namespace لكي ننظم الكود أكثر ويكون مرتب ومفهوم، يمكن أن يتواجد أكثر من namespace في نفس المشروع، ويمكن أن يوجد class في namespace بنفس اسم class آخر في namespace أخرى.

يتم إستخدامها عن طريق الكلمة المحجوزة namespace ثم الأسم الذي نريده ونكتب بداخلها كل الـ classes التي نريد.

ويتم إستدعائها عن طريق الكلمة المحجوز using.

في بداية كل كود نكتب using System، وكلمة System هي عبارة عن namespace تحتوي على العديد من الـ classes، مثل Console الذي نستخدمه في عملية الإدخال والإخراج.

ويتم إستدعاء ما بداخل كل namespace عن طريق النقطة (.)

فمثلاً إذا أردنا إستدعاء كل الـ classes التي بداخل System وإستدعاء دالة الإخراج أيضاً من Console فيكون الإستدعاء بهذا الشكل :

```
System.Console.WriteLine( ) ;
```

مثال :

```
using System ;
using Demo ;
namespace Test
{
    class Program
      {
            static void Main ( )
                  {
                        Demo.Prog1 p1 = new Demo.Prog1 ( ) ;
                        p1.view( ) ;
                        Asd a = new Asd( ) ;
                        a.show( ) ;
                        Prog1 p2 = new Prog1 ( ) ;
                        p2.view ( ) ;
                  }
       }
    class Prog1
     {
      public void view( )
        {
            Console.WriteLine( " I'am Namespace Test Class Prog1 " ) ;
        }
    }
}
namespace Demo
 {
   class Prog1
    {
       public void view ( )
```

```
        {
            Console.WriteLine( " I'am Namespace Demo Class Prog1 " ) ;
        }
    }
    class Asd
     {
       public void show ( )
        {
             Console.WriteLine(" I'am Namespace Demo Class Asd " ) ;
        }
     }
}
```

في أول سطر داخل دالة الـ Main كتبت Demo.Prog1 لكي يذهب إلى الـ class في الـ namespace التي أحددها لأنه يوجد class بنفس الأسم في الـ namespace الرئيسية، فلو لم أحدد الـ namespace كان سيذهب إلى الـ class الذي كان في

الـ namespace الرئيسية، وبما أن الـ class الذي اسمه Asd لا يوجد class آخر بنفس اسمه فأستدعيته مباشرةً بدون الحاجة إلى تحديد الـ namespace التي تحتويه.
يمكن أن نكتب namespace كاملة بداخل namespace بنفس الطريقة.
ونستدعيها بهذا الشكل :

using + اسم الـ namespace الأولى + (.) اسم الـ namespace الداخلية ;

مثال :

```
using System ;
using Test1.Test2 ;
namespace Test1
{
    class Program
     {
```

```
        static void Main ( )
        {
          Console.WriteLine( " Main Function " ) ;
        }
    }
namespace Test2
{
    class Prog2
      {
          public void Show( )
            {
                  Console.WriteLine( " Hello World " );
            }
      }
}
}
```

27

الإستثناءات والأخطاء

(Errors and Exceptions)

هي تلك الأخطاء التي يقع بها المستخدم أو تحدث في البرنامج أثناء تشغيله، ولها العديد من الأشكال، لكن يمكننا أن نتحكم في ما سيحدث عندما يقع أي خطأ، فمثلاً عند قسمة أي رقم على صفر فهذا غير صحيح وسيحدث خطأ، أو عند إدخال قيمة لمتغير من نوع مختلف عن نوع المتغير مثل إدخال حرف أو نص في متغير نوعه int أو إضافة قيم بداخل مصفوفة أكثر من عدد عناصرها فكل هذه أخطاء، توجد العشرات من الأخطاء.

يجب أن يكون الكود تحت تحكمنا الكامل به ولا نترك مجالاً لإحتمالية حدوث أي خطأ.

حماية الكود من الأستثناءات التي تحدث يكون عن طريق : try , catch.

try

نكتب بداخلها الكود أو البرنامج المراد تنفيذه ودائماً ما يتبعها catch معناها في حالة حدوث أي خطأ أو إستثناء نفذّ ما بداخل الـ catch. يمكنك أن تشبهها بـ if / else حتى يتضح لك كيفية عملها فقط.

catch

تلتقط الأخطاء في حالة حدوثها ويمكننا كتبة أي كود نريد تنفيذه في حالة حدوث خطأ ما.

ويمكن أن نستخدم أكثر من جملة catch متتالية.

finally

نكتب بداخلها كود نريد تنفيذه في حالة حدوث خطأ أم لا، فما بداخلها سيُنَفذّ تحت أي ظرف ويمكننا كتابتها أو الإستغناء عنها، فلا يشترط كتابتها في الكود مثل catch بعد try.

يجب ألا يخلو أي كود لنا بدون كتابة try , catch حتى نتجنب أي أخطاء.

توجد العديد من الـ classes الخاصة بالأخطاء في الـ namespace " System "

مثل :

Exception	يظهر لك الخطأ الذي حدث أثناء تنفيذ البرنامج
DivideByZeroException	يظهر لك الخطأ عندما تقسم أي رقم على صفر
StackOverflowException	يحدث هذا الخطأ عندما تستدعي دوال كثيرة داخل بعضها
OutOfMemoryException	يحدث عندما تمتلئ ذاكرة الكمبيوتر ولا يوجد مكان في الذاكرة للبرنامج
OverflowException	وجود فائض في العمليات الحسابية أو في التحويل من نوع بيانات لنوع آخر . أو إدخال قيمة أكبر من حجم المتغير .
ApplicationException	يحدث عند وجود خطأ في البرنامج أثناء تشغيله
FormatException	يحدث هذا الخطأ عندما نوع معامل الدالة أثناء الإستدعاء يختلف عن نوع المعامل أثناء إنشاؤه

يمكنك كتابة Exception في فيجوال ستوديو وسيظهر لك كل الـ classes الخاصة بالأخطاء وإذا وقفت بالمؤشر عليها سيظهر لك وظيفة كل class منهم.

مثال :

```
using System ;
namespace ConsoleApplication1
{
    class Program
     {
          static void Main ( )
           {
              try
                 {
                    int x = 5 ;
                    int y = 0 ;
                    Console.WriteLine( x / y ) ;
                 }
             catch ( Exception Ex )
                {
                    Console.WriteLine( Ex.message ) ;
                }
              finally
                {
                    Console.WriteLine( " finished " );
                }
          }
     }
}
```

في هذا البرنامج كتبنا الكود في try ثم كتبنا catch وأخذنا كائن أو نسخة من الـ class الذي يسمى Exception ثم نطبع الرسالة التي بداخله عندما يحدث خطأ، هذا الـ class شمولي أكثر فهو يعمل على أي خطأ يحدث بعكس الـ classes الأخرى تظهر الأخطاء التي تعبر عنها فقط.

فمثلاً DivideByZeroException يظهر الخطأ الذي يحدث عند القسمة على صفر فيما عدا ذلك لن يعمل. أما Exception يظهر أي خطأ يحدث في البرنامج.

يمكنك ألا تأخذ نسخة من أي class وتطبع رسالة خاصة بك.

مثال :

```
try
{
    int x = 8 ;
    int y = 9 ;
    Console.WriteLine( x + y ) ;
}
catch
{
    Console.WriteLine( " Error " ) ;
            تطبع أي شئ تريده أن يظهر عند حدوث أي خطأ //
}
```

مثال :

```
using System ;
namespace ConsoleApplication1
{
   class Program
    {
        static void Main ( )
                {
                    try
                      {
                            int x = 8 ;
                            int y = 3 ;
                           Console.WriteLine( x / y ) ;
                     }
                     catch ( DivideByZeroException D )
                       {
                            Console.WriteLine( D.message ) ;
                       }
                     catch (StackOverflowException S )
                       {
                            Console.WriteLine( S.message ) ;
                       }
                     catch (OverflowException O )
                       {
                            Console.WriteLine( O.message ) ;
                       }
                     catch (Exception E )
                       {
                            Console.WriteLine( E.message ) ;
                       }
                }
    }
}
```

28

(Collections)

الـ Collections هي عبارة عن مجموعة من الـ classes التي تساعدنا في التحكم في البيانات وكيفية تخزين وإسترجاع هذه البيانات، قبل ذلك كان كل شيء خاص بالبيانات مُقيداً والتعامل معه مُحكَم.

فمثلاً كانت المصفوفة يجب أن نحدد لها الحجم أثناء إنشائها وإذا أكتملت عناصرها وأمتلأت فلا نستطيع إضافة عنصر آخر وأنه يجب تحديد نوع المصفوفة وتكون كلها من نفس النوع أي أنه إما تكون كل العناصر int أو string أو float وهكذا. أما في الـ collections نستطيع إضافة أي عنصر من أي نوع وحجم المصفوفة يكون ديناميكي أي أنه يزيد تلقائياً كلما أضفنا عنصر جديد.

لكي تستخدم الـ collections يجب أن تستدعي الـ namespace التي تسمى Collections وهي موجودة في الـ namespace " System ".

يكون الإستدعاء بهذا الشكل :

```
using System.Collections ;
```

أحد هذه الـ classes هو :

Array List

هو عبارة عن class يخلق مصفوفة " Array " لكن يزيد حجمها كلما أضفنا عنصر جديد بها، ويمكنه إستقبال أي نوع من البيانات دون الحاجة إلى تعريف نوعه ويمكن المزج بين هذه البيانات من حيث النوع فيمكننا إضافة string ثم العنصر الذي يليه يكون int وهكذا، ويتم إستدعاء هذا الـ class بالطريقة العادية.

إذا أردنا إنشاء مصفوفة ديناميكية أي يتغير حجمها كلما أضفنا عنصر نستخدم ArrayList ونأخذ منه نسخة ونعمل على الدوال التي بداخله التي تمكننا من التحكم في المصفوفة بشكل كامل.

أولاً طريقة إنشاء مصفوفة ديناميكية :

```
ArrayList arr = new ArrayList ( ) ;
```

أخذنا من الـ class نسخة أي أنشأنا مصفوفة جديدة واسمها " **arr** ".

ومن الممكن أن نحدد حجم المصفوفة أيضاً من خلال تمرير القيمة للـ constructor فيعطي للمصفوفة قيمة إبتدائية فمثلاً إذا حددت حجم المصفوفة 10 وأمتلأت وأردت أن تضيف العنصر جديد سيزيد حجمها تلقائياً وذلك من خلال كلمة new فهي تنشيء مصفوفة جديدة أكبر وتنسخ بها كل بيانات المصفوفة القديمة ثم تُحذف المصفوفة القديمة بالـ Garbage Collector الذي يعمل تلقائياً.

بعكس لغات مثل C++ يجب أن تحذفها بنفسك بإستخدام كلمة delete.

تحديد حجم المصفوفة يكون كالتالي :

```
ArrayList arr = new ArrayList ( 10 ) ;
```

أو من خلال الخاصية Capacity فهي تُمثِل الـ set تأخذ القيمة وتضعها كحجم المصفوفة كالآتي :

```
ArrayList arr = new ArrayList ( ) ;
arr.Capacity = 10 ;
```

- لإضافة عنصر في المصفوفة نستخدم الدالة " ( )Add ".

مثال :

```
using System ;
using System.Collections ;
namespace Test
{
        class Program
          {
              static void Main ( )
              {
                      ArrayList arr = new ArrayList ( ) ;
                      arr.Add( " Hello " ) ;
                      arr.Add( 100 ) ;
                      arr.Add( ' R ' ) ;
                      arr.Add( 2,71 ) ;
                      foreach ( object i in arr )
                       {
                         Console.WriteLine( i ) ;
                       }
              }
      }
}
```

- لعرض محتويات المصفوفة نستخدم **foreach** فأحد الأغراض الأساسية للـ foreach هو إستخدامها مع الـ collectios. وبما أن العناصر التي بداخل المصفوفة

ليست من نفس النوع فلا يمكننا تحديد نوع العدّاد الذي يدور على العناصر بداخل المصفوفة " i " بنوع مثل int أو string، فنحدده بأنه object أو var أو dynamic.

- لمعرفة عدد عناصر المصفوفة الحالية في حالة أنك لم تحدد لها حجم معين، نستخدم الخاصية Count فهي تُمثِل get وتُرجِع حجم المصفوفة.

سنعمل على نفس المثال السابق :

```
Console.WriteLine( arr.Count ) ;
```

- في حالة أنك أردت إدخال عنصر في مكان محدد في المصفوفة نستخدم الدالة " Insert( ) " وهذه الدالة تأخذ المكان الذي تحدده للعنصر وقيمة العنصر.

```
arr.Insert( 3 , " Hi " ) ;
```

لكن يجب أن يكون المكان الذي تحدده موجود فعلياً في المصفوفة حالياً، أي أنه يوجد عنصر في هذا المكان قبل ذلك لأنك لو حددت موضع الإدخال مكان لعنصر غير موجود فسيحدث خطأ.

فمثلاً توجد مصفوفة بها العناصر الآتية :

Hello	100	R	2.71	true

حجم المصفوفة 5، فلا يجوز أن تُدخل موضع لعنصر أكبر من 4 لأن الترقيم في المصفوفة يبدأ من صفر.
هذا في حال تواجد عناصر في المصفوفة، لو لم يوجد عناصر في المصفوفة فلا يمكنك أن تضع مكان للعنصر في الدالة Insert أكبر من صفر.

عندما قولنا أن :

```
arr.Insert( 3 , " Hi " ) ;
```

يبدأ البحث عن الموضع المحدد من المكان صفر إلى أن يجده ويزيد حجم المصفوفة بمقدار عدد العناصر الجديدة فإذا كان يوجد عنصر في هذا المكان ستنتقل كل العناصر التي على يمينه بمقدار خانة واحدة لكل عنصر في جهة اليمين في الذاكرة.

فتكون المصفوفة بهذا الشكل :

Hello	100	R	Hi	2.71	true

- لحذف عنصر معين من المصفوفة نستخدم الدالة ( )RemoveAt وتأخذ هذه الدالة مكان العنصر الذي تريد حذفه فقط، ويقل حجم المصفوفة بمقدار 1. وتنتقل العناصر التي على يمين العنصر المحذوف بمقدار خانة واحدة في جهة اليسار في المصفوفة.

```
arr.RemoveAt( 3 ) ;
```

ستُحذَف Hi

- لحذف عنصر معين عن طريق قيمة العنصر وليس موضعه في المصفوفة سنستخدم الدالة ( ) Remove وتأخذ قيمة العنصر فقط.

```
Remove( R ) ;
```

سيُحذَف الـ R

- لإضافة مجموعة من العناصر في وقت واحد نستخدم الدالة ( ) AddRange، يمكنها أن تأخذ مصفوفة أخرى وتضيفها على المصفوفة القديمة، تُضاف في آخر المصفوفة، وتأخذ اسم المصفوفة الجديدة فقط.

مثال :

```
object [ ] test = { 1 , 50 , ' K ' , " XYZ " , 3.14 } ;

arr.AddRange( test ) ;
```

- لحذف مجموعة من العناصر في وقت واحد نستخدم الدالة ( )RemoveRange وهي تأخذ موضع العنصر الذي تريد أن تبدأ بالحذف منه وموضع العنصر الذي تريد أن تنتهي عنده.

```
RemoveRange( 2 , 5 ) ;
```

ستحذف العناصر من ٢ إلى ٥ بالترتيب.

إذا كان عندك مصفوفة وتريد نسخ بعض العناصر من مصفوفة أخرى ... تستخدم ( )GetRange وهي تأخذ موضع أول عنصر تبدأ بنسخه وتأخذ آخر موضع تنتهي عنده.

```
ArrayList Test = new ArrayList( ) ;
```

```
Test=arr.GetRange( 1 , 4 ) ;
```

ستُنسَخ العناصر من الموضع ١ إلى الموضع ٤ في المصفوفة القديمة " arr " إلى المصفوفة الجديدة التي تسمى " Test ".

- لحذف كل عناصر المصفوفة نستخدم الدالة ( ) Clear.

```
arr.Clear( ) ;
```

Stack

الـ stack هو عبارة عن class داخل الـ Collections مثل ArrayList لكنه يختلف قليلاً عنه، الـ stack هو مجرد مصفوفة لكنه يتعامل بطرق مختلفة لإضافة عناصر في المصفوفة أو إخراجها، حيث تتم إضافة العناصر من آخر المصفوفة وتتم عملية الإخراج أيضاً من آخر المصفوفة، أي أن آخر عنصر يُضاف هو أول عنصر يخرج،

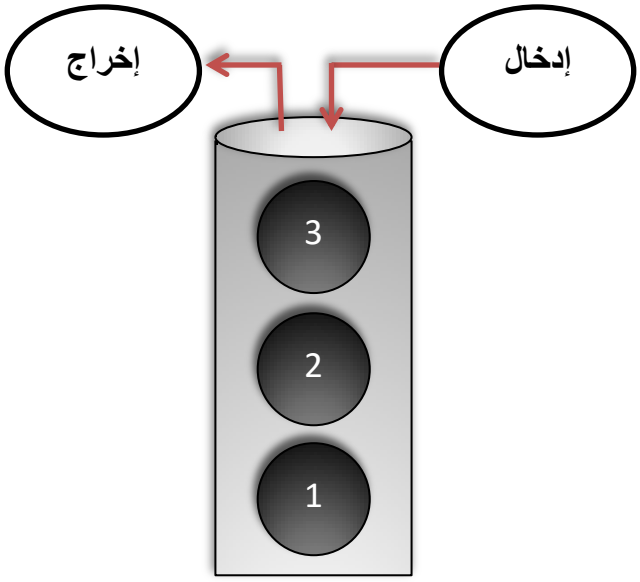
وأول عنصر يضاف هو آخر عنصر يخرج فيتم عرض البيانات التي بداخله بطريقة عكسية.

ويسمى الـ stack بـ LIFO (Last in First Out)
أو FILO (First In Last Out)

أي أن من يُضاف أولاً يخرج آخراً أو من يُضاف آخراً يخرج أولاً.

تعمل معظم التطبيقات الضخمة بنظام الـ stack كالبحث في جوجل حيث تظهر لك أولاً آخر عملية بحث قمت بها، كما في تطبيق الفيس بوك تظهر لك في مقدمة البحث آخر عملية بحث قمت بها مسبقاً.
كما تعمل به الذاكرة المؤقتة أيضاً.

يمكنك أن تفهم كيفية عمل الـ stack بوضوح من خلال الشكل الآتي :
إذا أردت إخراج الكرة (١) يجب أن تُخرج كرة (٣) أولاً ثم كرة (٢) إلى أن تصل إلى الكرة (١) .



يتم إنشاء الـ stack بالطريقة العادية مثل أي class آخر .

```
Stack Test = new Stack ( ) ;
```

ويمكنك أن تحدد حجمه أثناء الإنشاء كما الـ ArrayList.

```
Stack Test = new Stack ( 10 ) ;
```

- لإضافة عنصر بداخل الـ stack نستخم الدالة ( )Push وهي تأخذ قيمة العنصر فقط.

```
Test.Push( 12 ) ;

Test.Push(" H ") ;

Test.Push( 2.71 ) ;

Test.Push( False ) ;
```

لطباعة ما بداخل الـ stack نستخدم foreach

```
foreach( object i in Test )

Console.WriteLine( i );
```

ناتج الطباعة :

سيبدأ بآخر قيمة دخلت في الـ stack فهو يعرض بياناته بطريقة عكسية.

```
False -

2.71 -

H -

12 -
```

عند إخراج قيمة من الـ stack ثم حذفها منه سنستخدم الدالة ( )Pop ولا تأخذ الدالة أي مُدخلات، تُخرج آخر عنصر دخل إلى الـ stack وتحذفه.

Test.Pop( ) ;

إذا أردنا إظهار آخر قيمة دخلت في الـ stack بدون حذفها نستخدم الدالة ( )Peek.

Console.WriteLine(Test.Peek( )) ;

- أما لحذف كل عناصر الـ stack نستخدم الدالة ( )Clear.

Test.Clear( ) ;

- إذا أردنا التأكد من وجود عنصر معين في الـ stack نستخدم الدالة ( )Cotains وترجع إما true أو false أي العنصر موجود أو غير موجود في الـ stack.

Console.WriteLine(Test.Contains(12)) ;

سترجع true لأن الـ ١٢ موجودة في الـ stack

Console.WriteLine(Test.Contains(3)) ;

سترجع false لأن الـ ٣ غير موجودة في الـ stack.

- لمعرفة عدد عناصر الـ stack نستخدم الخاصية Count.

Console.WriteLine(Test.Count) ;

```
using System ;
using System.Collections ;
namespace ConsoleApplication1
{
    class Program
      {
```

```
        static void Main( )
            {
                Stack st = new Stack ( );
                st.Push(" Ali ");
                st.Push(" Ahmed ");
                st.Push(" Mahmoud ");
                st.Push(" Khaled ");
                st.Push(" Amr ");
                Console.WriteLine( st.Count );
                foreach( string i in st )
                    Console.WriteLine( i );
                st.Pop( );
                Console.WriteLine( st.Count );
                foreach( string i in st )
                    Console.WriteLine( i );
            }
        }
}
```

في هذا الكود أنشأنا stack وأدخلنا به 5 عناصر ثم طبعنا عدد العناصر الموجودة بالـ stack، بعدها طبعنا كل العناصر بدون حذفها، ثم حذفنا آخر عنصر دخل إلى الـ stack وطبعنا عدد العناصر مرة أخرى ثم طبعنا كل العناصر.
حاول أن تُخرج ناتج هذا الكود بدون أن تنفذه على البرنامج.

Queue

الـ queue هو عبارة عن class داخل الـ Collections وهو يشبه الـ stack حيث أنه مصفوفة أيضاً، غير أن الـ queue يتعامل مع البيانات بطريقة مختلفة قليلاً عن الـ stack.

تتم إضافة العناصر من آخر المصفوفة وتتم عملية الحذف أو الإخراج من أول المصفوفة، أي أن آخر عنصر يُضاف هو آخر عنصر يُحذف "يخرج"، وأول عنصر يضاف هو أول عنصر يُحذف "يخرج" ويتم عرض البيانات التي بداخله بطريقة عادية وليست عكسية مثل الـ stack، ويسمى الـ queue بـ FIFO (First in First Out) أي أن من يُضاف أولاً يخرج أولاً. فهو ببساطة يشبه الطابور مَن يأتي أولاً يذهب أولاً.

1	2	3	4	5

إذا أردت إخراج الـ 2 فيجب أن تُخرج الـ 1 أولاً لأنه خُزِنَ في الـ queue قبلها.
معظم دوال الـ queue تشبه دوال الـ stack.

- لإضافة عناصر داخل الـ queue نستخدم الدالة ( )Enqueue وهي تعادل Push في الـ stack.

```
Queue qu = new Queue( );
qu.Enqueue(" A ");
qu.Enqueue( 100 );
qu.Enqueue( 3.7 );
```

- لطباعة عناصر الـ queue نستخدم foreach، وتخرج العناصر بنفس الترتيب الذي دخلت به.

```
foreach( object i in qu )
Console.WriteLine( i );
```

- لإخراج أول عنصر دخل في الـ queue وحذفه نستخدم دالة ( )Dequeue

```
qu.Dequeue( );
```

- لإظهار أول عنصر يخرج من الـ queue بدون حذفه نستخدم دالة ( )Peek

```
qu.Peek( );
```

- لحذف كل عناصر الـ queue نستخدم دالة ( )Clear

```
qu.Clear( );
```

- للتأكد من وجود عنصر ما داخل الـ queue نستخدم دالة ( )Contains ونمرر لها القيمة التي نبحث عنها.

```
Console.WriteLine( qu.Contains( 100 ) );
```

ستطبع إما true أو false.

- لمعرفة عدد عناصر الـ queue نستخدم الخاصية Count.

```
Console.WriteLine( qu.Count );
```

29

(Generics)

الـ generic تتيح لك إستخدام الـ classes والدوال بدون الحاجة إلى تعريف نوعها مؤقتاً حتى لحظة إدخال قيم لها. وهذا يجعل الأداء في البرنامج أعلى ويمكنك أن تستبدل الـ classes التي استخدمتها في الـ collections بالـ generic فهي متماثلة تقريباً.

ولإستخدام الـ classes الموجودة بداخل هذه الـ namespace نستدعي الـ namespace كالتالي :

```
using System.Collections.Generic;
```

تعرف الـ generic من خلال هذه العلامة < >

توجد بعض الـ generic classes مثل :

List< >

هذا الـ class هو عبارة عن مصفوفة ديناميكية تشبه الـ ArrayList في الـ collections، ويتم تعريفها بهذا الشكل :

```
List <int> arr = new List <int> ( );
```

هذا الكود معناه أننا أنشأنا مصفوفة اسمها arr ونوعها int. وتستطيع إستقبال بيانات من النوع int فقط كما حددنا مسبقاً. وكلما أدخلنا عناصر أكثر يزيد حجمها تلقائياً.

لإضافة عنصر لها نكتب :

```
arr.Add( 5 );

arr.Add( 2 );
```

وبها معظم الدوال التي شرحتها مسبقاً في الـ ArrayList وبها دوال إضافية ليست موجودة في الـ ArrayList مثل ( )Sort وتستخدم هذه الدالة في ترتيب المصفوفة تصاعدياً.

```
arr.Sort( );
```

وتوجد بها أيضاً دالة ( )Reverse وتستخدم هذه الدالة لعكس عناصر المصفوفة.

```
arr.Reverse( );
```

مثال على الـ List< >

```
using System ;
using System.Collections.Generic ;
namespace ConsoleApp
{
public class Employee
{
   string name ;
   string mobile ;
   public string _Name
      {
        get
           {
                return _Name ;
           }
      }
```

```
    public string _Mobile
     {
        get
        {
           return _Mobile;
        }
     }
     public Employee( string name , string mobile )
     {
        this.name = name;
        this.mobile = mobile;
     }
     public override string ToString()
     {
        return name+" ( " + mobile + " ) " ;
     }
}
class Program
{
   static void Main( )
     {
         List<Employee> emp = new List<Employee>( ) ;
         emp.Add( new Employee(" Ahmed ", " 01111111111 " ) );
        // Constructor
         emp.Add( new Employee(" Ali ", " 01222222222 " ) );
         emp.Add( new Employee(" Mohamed ", " 01555555555 " ) );
         emp.Add( new Employee(" Khaled ", " 01000000000 " ) );
         foreach ( Employee x in emp )
                  Console.WriteLine( x );
     }
```

```
}
}
```

في هذا الكود أنشأنا class اسمه Employee وبداخله حقل الأسم ورقم التليفون، وأنشأنا constructor يأخذ الأسم ورقم التليفون ويضع القيم في الحقول، ثم عملية override للدالة ToString وهي دالة جاهزة في اللغة لطباعة النصوص، وجعلناها ترجع الأسم ورقم التليفون.
وأنشأنا List ويكون نوع الـ List هو Employee أي الـ class فهو يُعدّ كنوع بيانات جديد، ولكي نستطيع إضافة عناصر للـ List من نفس نوع الـ Employee نستخدم new ثم نكتب اسم الـ constructor ونمرر له المُدخلات، وفي عملية الطباعة نستخدم foreach ويكون نوع العدّاد إما اسم الـ class الذي يُعدّ نوع بيانات والبيانات التي ستُطبع ستكون من نفس نوعه أو نستخدم object أو var أو dynamic.

Stack< >

يعمل الـ generic stack بنفس كيفية عمل الـ collection stack أول عنصر يدخل في المصفوفة هو آخر عنصر يخرج منها وآخر عنصر يدخل في المصفوفة هو أول عنصر يخرج منها. وبه نفس الدوال التي في الـ collection stack.

```
Stack<int> s = new Stack<int> ( );

s.Push( 10 ) ;

s.Push( 20 ) ;

s.Pop( ) ;
```

```
s.Peek( ) ;

Console.WriteLine( s.Count ) ;
```

Queue< >

يعمل الـ generic queue بنفس كيفية عمل الـ collection queue حيث العنصر الذي يدخل المصفوفة أولاً يخرج أولاً ومن يدخل آخراً يخرج آخراً. وبه نفس الدوال التي في الـ collection queue.

```
Queue <int> q = new Queue <int>( );

q.Enqueue( 1 ) ;

q.Enqueue( 3 ) ;

q.Enqueue( 20 ) ;

q.Dequeue( ) ;

Console.WriteLine( q.Count ) ;
```

LinkedList< >

الـ LinkedList نوعان فردية ومزدوجة

الفردية : (غير موجودة في الـ Generics)

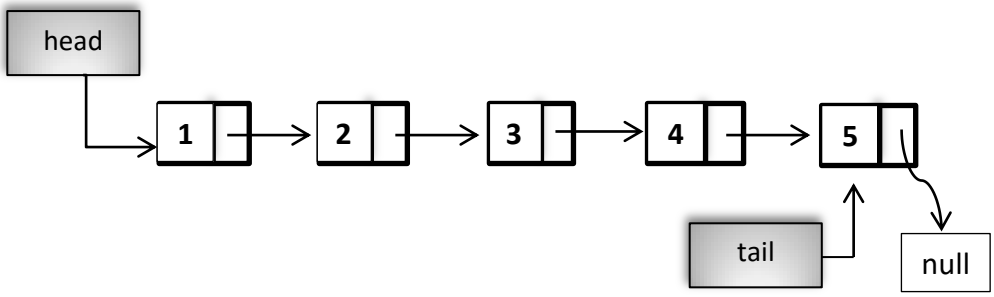
هي عبارة عن سلسلة مترابطة من العناصر، تكون هذه العناصر مرتبطة مع بعضها من خلال مؤشرات فكل عنصر يُشير إلى مكان العنصر الآخر الذي يليه، تكون هذه العناصر مُخزنة بطريقة عشوائية في الذاكرة بعكس المصفوفة التي تحجز عدد من الاماكن في الذاكرة بطريقة متتالية.

الـ linkedList تمكننا من الإضافة أو الحذف في منصف السلسلة بدون الحاجة إلى تبديل أماكن عناصر أخرى. لذا فهي أفضل من المصفوفة " Array " في عمليات الحذف أو الإضافة، فيمكننا إضافة العناصر في البداية أو المنتصف أو نهاية السلسلة، أما في عمليات البحث عن عنصر ما فإنه يأخذ وقت طويل لأنه كما قلت فعناصرها ليست متتابعة في الذاكرة بل إنها مبعثرة في الذاكرة خاصةً إذا كنت تبحث عن عنصر في منتصف السلسلة أو نهايتها.

أما المصفوفة فهي أسرع في عمليات البحث والوصول إلى العناصر المُخزَنة بها، والـ linkedList لا تمتليء أبداً، فكلما أضفت عنصر تضيفه هي في السلسلة بشكل مباشر فهي ليس لها حجم ثابت، أما المصفوفة يجب أن تحدد لها الحجم ولو لم تحدد لها الحجم فإنه كلما تُدخل عنصر بها تُنشَأ مصفوفة جديدة بحجم أكبر وتُنسَخ بها المصفوفة القديمة ثم تُحذَف القديمة من خلال الـ (Garbage collector) GC.

أما الـ LinkedList لا يكون لها حجم ثابت فكلما أضفت عنصر بها تشير إليه فقط وعندئذٍ يضاف إلى السلسلة مباشرةً، فالسلسلة الفردية يكون لها مؤشر واحد يُشير إلى العنصر الذي بعده. وهي كما قلت غير موجودة في الـ Generics لكن شرحتها حتى تستطيع أن تفهم كيفية عمل السلسلة المزدوجة.

لنفترض أنه توجد سلسلة بها هذه العناصر (1 , 2 , 3 , 4 , 5) ستكون بهذا الشكل :



null هي كلمة محجوزة في اللغة وتعني لا شيء، أي أنه لا يوجد شيء بعد العنصر الأخير في السلسلة.

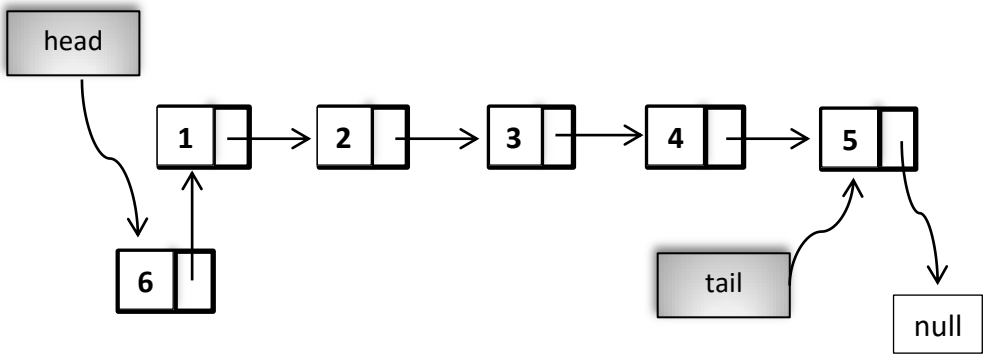
head ---> هو مؤشر يشير إلى أول عنصر في السلسلة.

tail ---> هو مؤشر يشير إلى آخر عنصر في السلسلة.

في حالة أن السلسلة لا يوجد بها عناصر فيكون head , tail يؤشران على null.

وكما نرى كل عنصر يُشير إلى العنصر الذي بعده إلى أن نصل إلى آخر عنصر في السلسلة حيث يُشير إلى null.

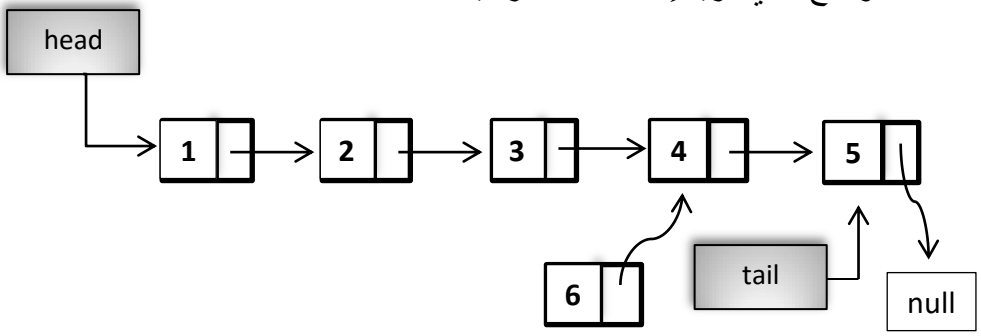
- في حالة الإضافة في بداية السلسلة إضافة عنصر (6) سيكون بهذا الشكل :



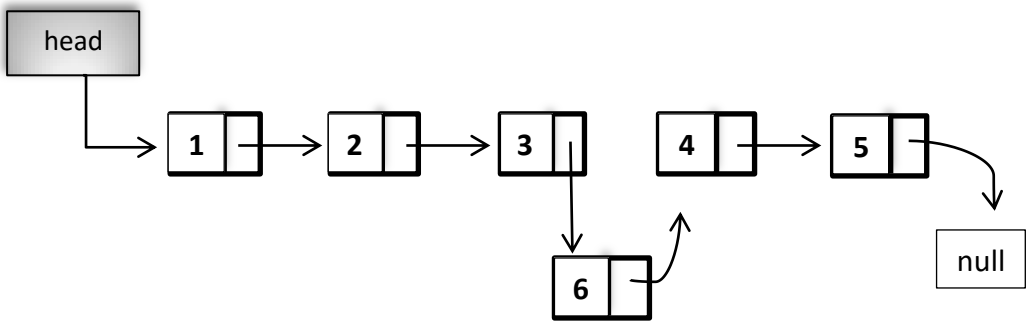
سنجعله يُشير إلى أول عنصر في السلسلة وبعد ذلك نجعل الـ head يؤشر عليه وبذلك يرتبط بالسلسلة ويكون العنصر 6 هو أول عنصر في السلسلة.

- في حالة الإضافة في منتصف السلسلة أو في أي موضع آخر بخلاف البداية والنهاية :

أولاً نحدد الموضع الذي نريد إضافة العنصر فيه.



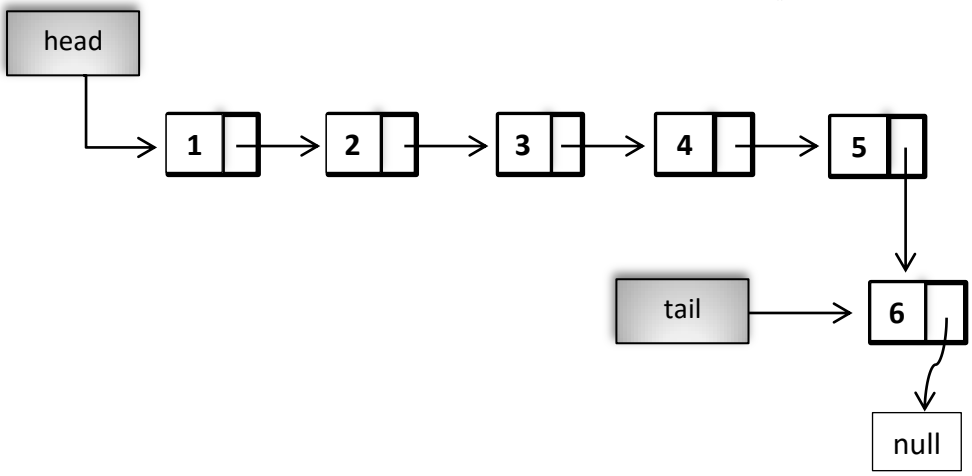
نجعل العنصر الجديد يُشير إلى العنصر التالي للموضع الذي حددناه.



وآخراً نجعل العنصر الذي يسبق الموضع المحدد أن يُشير إلى العنصر الجديد وهكذا يُضاف إلى السلسلة.

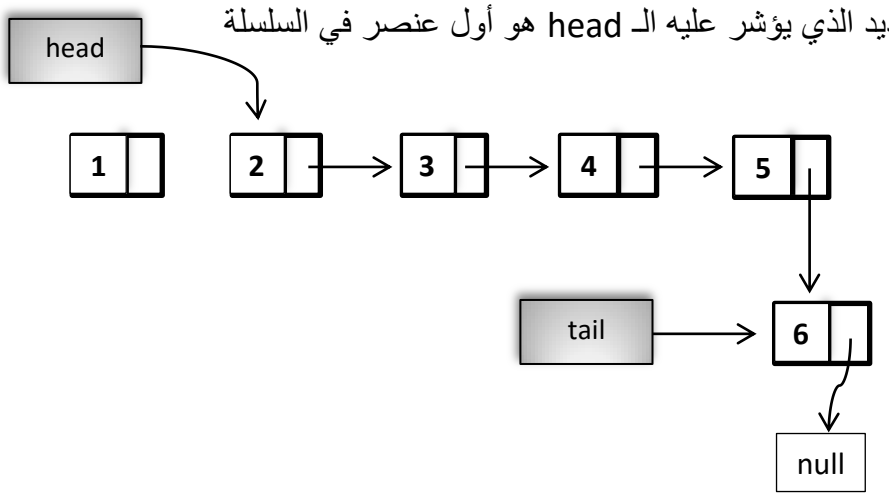
لو جعلنا العنصر الذي يسبق المكان المحدد أن يُشير أولاً على العنصر الجديد ستقع السلسلة من بعد هذا المكان ولن يبقى بها سوا العناصر التي تسبق الموضع المحدد.

- في حالة الإضافة في النهاية :

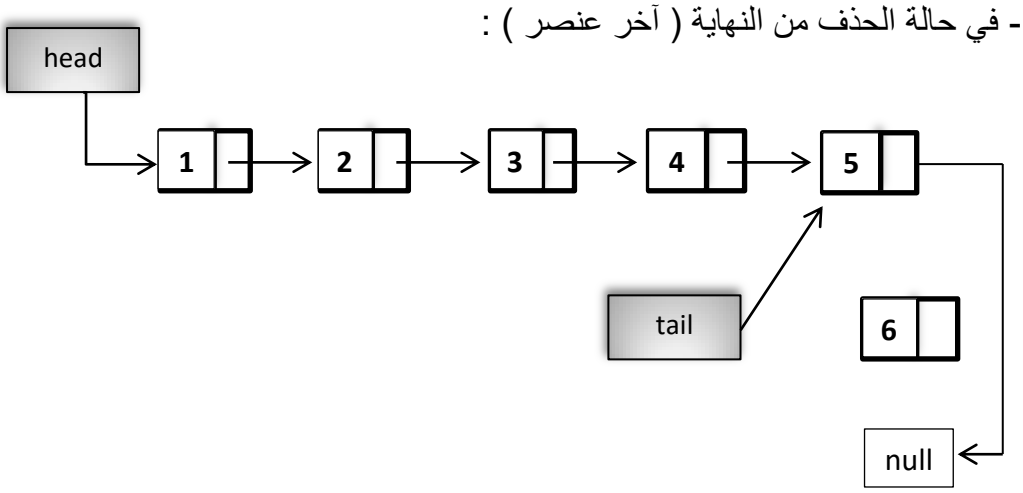


سنجعل آخر عنصر يُشير إليه ثم نؤشر عليه بـ tail.

- في حالة الحذف من البداية (أول عنصر) :
سنجعل الـ head يؤشر على العنصر الذي يلي أول عنصر وبذلك يصبح العنصر الجديد الذي يؤشر عليه الـ head هو أول عنصر في السلسلة



- في حالة الحذف من النهاية (آخر عنصر) :

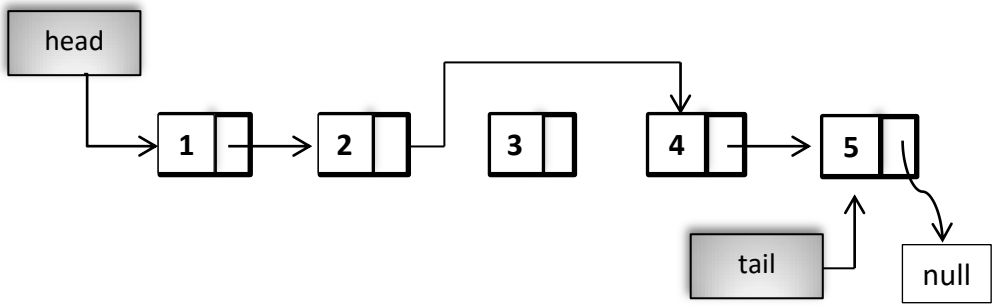


سنجعل العنصر الذي يسبق العنصر الأخير يؤشر مباشرةً على null وبذلك يصبح هو آخر عنصر في السلسلة أي الـ tail.

- في حالة الحذف من منتصف السلسلة أو أي مكان في السلسلة بخلاف البداية والنهاية:

أولاً نحدد موضع العنصر الذي سنحذفه ثم نجعل العنصر الذي يسبقه يشير إلى العنصر الذي يلي العنصر الذي سنحذفه.
لنفترض أنه توجد سلسلة بها العناصر التالية (1 , 2 , 3 , 4 , 5)

ونريد حذف العنصر 3

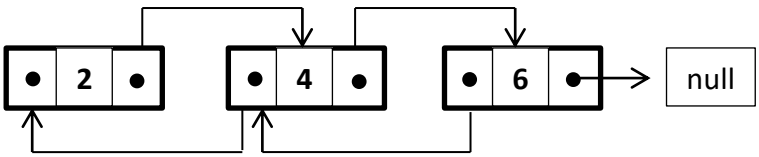


لنفترض أن هذه هي الذاكرة المؤقتة (RAM)، وبما أن الذاكرة تُخزِن القيم بها بطريقة عشوائية وغير مرتبة فيمكن ان تكون السلسلة بهذا الشكل :

1		~		~		~			~
	~		~			~	~		
				~	**3**			~	
5	~	~	~				~		
				~	~	~		~	~
	~	~					~		~
~			~	~	**6**				~
	~	**2**				~	**4**	~	

السلسلة المزدوجة (Double LinkedList):

هي نفس السلسلة الفردية لكن الإختلاف بينهما أن المزدوجة يكون لها مؤشران وليس مؤشراً واحداً، مؤشر يشير إلى العنصر الذي بعده والآخر يشير إلى الذي قبله.



يُعرفّ (generic LinkedList) class بهذه الطريقة :

```
LinkedList < DataType > Name = new LinkedList < DataType >( ) ;
```

فمثلاً لو كانت العناصر int :

```
LinkedList < int > Link = new LinkedList < int >( );
```

- في حالة الإضافة من البداية نستخدم الدالة ( )AddFirst وهي تأخذ قيمة العنصر فقط.

```
Link.AddFirst( 10 ) ;
```

- في حالة الإضافة في نهاية السلسلة نستخدم الدالة ( )AddLast وهي تأخذ قيمة العنصر فقط.

```
Link.AddLast( 20 ) ;
```

- لحساب عدد عناصر السلسلة نستخدم الخاصية Count.

```
Console.WriteLine( Link.Count ) ;
```

- للبحث عن عنصر معين نستخدم الدالة ( )Contains وتأخذ قيمة العنصر وهي تُرجع true أو false.

```
Console.WriteLine( Link.Contains ( ) ) ;
```

- لحذف عنصر من بداية السلسلة نستخدم الدالة ( )RemoveFirst ولا تأخذ مُدخلات.

```
Link.RemoveFirst( ) ;
```

- لحذف عنصر من نهاية السلسلة نستخدم الدالة ( )RemoveLast لا تأخذ مُدخلات.

```
Link.RemoveLast( ) ;
```

- لحذف عنصر معين نستخدم الدالة ( )Remove وهي تأخذ العنصر الذي تريد حذفه، ولو تكرر العنصر أكثر من مره فإنها تحذف أول عنصر تجده في السلسلة الذي يحمل هذه القيمة.

مثال :

توجد في السلسلة هذه العناصر (6 , 3 , 5 , 4 , 3 , 2 , 1) وأنت تريد حذف الـ 3، ستحذف الدالة أول 3 تقابلها فتكون السلسلة بهذا الشكل (6 , 3 , 5 , 4 , 2 , 1).

```
Link.Remove( 3 ) ;
```

تمارين :

أنشيء برنامج لإستقبال بيانات المرضى والدخول بأولوية الحجز.

```
using System;
using System.Collections.Generic;
namespace ConsoleApplication2
{
   class patient : Ifunctions
   {
                                        تعريف بيانات المريض .
      int age;
      string name;
      string mobile;
      int id;
      public int Age
      {
         set{ age = value; }
         get{ return age; }
      }
      public string Mobile
      {
         set { mobile = value; }
         get { return mobile; }
      }
```

```
public string _Name
{
   set { name = value; }
   get { return name; }
}
public int Id
{
   set { id = value; }
   get{ return id; }
}
public patient( ) { }
public patient( int iD , string N , string M , int A )
{
```

مُشيِّد (constructor) لإدخال بيانات المريض .

```
   this.name = N;
   this.mobile = M;
   this.age = A;
   this.id = iD;
}
Queue<patient> P = new Queue<patient>( );
```

أنشأنا Generics Queue لأنه كما نعرف يعمل بالأولوية فمن حجز أولاً يدخل أولاً. ويكون نوعه هو نفس نوع الـ class حيث أن بيانات الـ class هي التي ستُخَزن به.

```
public void insert( int id , string name , string mobile , int age )
{
                                    دالة لإدخال بيانات المريض .
   bool check=false;
   for ( int i = 0; i < P.Count; i++ )
   {
      if ( id == P.ElementAt( i ).id )
      {
           الدالة ( )ElementAt لإرجاع البيانات المحددة سواء كانت
                                                ( name , id , moblie )
      من خلال ترقيم ( فهرس ) المريض ( العنصر ) في المصفوفة.
      فنبحث عن الرقم التسلسلي الجديد هل يطابق رقم تسلسلي قديم؟
    إذا كان الرقم موجود من قبل فلن تتم عملية إضافة بيانات المريض
          حيث يكون لكل مريض رقم تسلسلي لا يتكرر لمريض آخر.
         check = true;
      }
      else
         check = false;
   }
   if ( check == true )
   {
  Console.WriteLine(" this id ( "+id+" ) is exist before , try again ! ");
   }
   else
```

```
    {
        P.Enqueue( new patient( id , name , mobile , age ) );
```

تُكتب new لتنفيذ الـ constructor كمُدخل للدالة (Enqueue)

فيُنفَذّ الـ constructor

وتكون البيانات التي دخلت إليه مُدخل للدالة.

```
    }
}

public bool Search( int id )
{
```

دالة للبحث عن مريض من خلال رقمه التسلسلي.

```
    bool check = false;
    for ( int i = 0; i < P.Count; i++ )
    {
        if ( P.ElementAt( i ).id == id )
        {
            check = true;
        }
    }
    return check;
}
```

```
  public void delete( )
  {
                                        دالة لحذف بيانات أول مريض.
     P.Dequeue( );
  }
  public void print( )
  {
                                        دالة لطباعة بيانات كل المرضى.
     for ( int i = 0; i < P.Count; i++ )
     {
        Console.WriteLine("id : " + P.ElementAt( i ).id);
        Console.WriteLine("name : " + P.ElementAt( i ).name);
        Console.WriteLine("mobile : "+P.ElementAt( i ).mobile );
        Console.WriteLine("age : " + P.ElementAt( i ).age);
        Console.WriteLine( );
     }
  }
}
interface Ifunctions
{
  void insert( int id , string name , string mobile , int age );
  bool Search( int id );
  void delete( );
  void print( );
```

```
}
class Program
 {
   static void Main( )
   {
      try
      {
           patient pat = new patient( );
           int id;
           string name;
           string mobile;
           int age;
           while (true)
            {
               Console.WriteLine("Enter 1 to insert patient data");
               Console.WriteLine("Enter 2 to search on patient data");
               Console.WriteLine("Enter 3 to delete patient data");
               Console.WriteLine("Enter 4 to show all patient data");
               Console.WriteLine(); Console.Write("Your choice is : ");
               int i = int.Parse(Console.ReadLine( ));
               switch ( i )
                {
                  case 1:
```

```
            Console.Write(" id : ");
            id = int.Parse(Console.ReadLine( ));
            Console.Write(" name : ");
            name = Console.ReadLine( );
            Console.Write(" mobile : ");
            mobile = Console.ReadLine( );
            Console.Write(" age : ");
            age = int.Parse(Console.ReadLine( ));
            pat.insert(id, name, mobile, age);
            break;
        case 2:
            Console.Write("\t enter patient id :");
            id = int.Parse(Console.ReadLine( ));
            Console.WriteLine(pat.Search(id));
            Console.WriteLine( );
            break;
        case 3:
            pat.delete( );
            break;
        case 4:
            pat.print( );
            break;
```

```
                default:
                        Console.WriteLine(" false choice ");
                        Console.WriteLine( );
                        break;
                }
            }
        }
        catch( Exception Ex )
        {
                Console.WriteLine( Ex.Message );
        }
    }
  }
}
```

30

هياكل البيانات

(Data structures)

هياكل البيانات (Data structure) إنما هي فقط أساليب لتخزين البيانات وإستراجعها والبحث عنها وترتيبها بطرق معينة، مثلما تحدثت سابقاً عن الـ stack والـ queue وأنهما مصفوفة عادية لكن نحن نتحكم في طريقة إدخال وإسترجاع البيانات بشكل معين.

البحث (Searching)

توجد عدة طرق للبحث عن البيانات منها :

١- البحث الخطي (Linear search)

وهذه الطريقة تُعني أنه يتم فحص كل العناصر من بداية أول عنصر في المصفوفة إلى آخر عنصر بها إلى أن يتم إيجاد العنصر الذي نبحث عنه وربما يكون غير موجود في المصفوفة.

فمثلاً إذا كنا نبحث عن القيمة 16 في مصفوفة مُكونة من 10 عناصر كما في المثال التالي فسيتم فحص كل العناصر إلى أن يصل إلى القيمة التي نبحث عنها " 16 ".

0	1	2	3	4	5	6	7	8	9
4	2	12	8	34	76	9	15	16	100

هذه الطريقة في البحث غير عملية وبطيئة في الحالات التي تكون فيها المصفوفة ذات مساحة كبيرة جداً لأنه يمر على كل عناصر المصفوفة، لذا فتكون كفاءة البرنامج أقل لأنه يستهلك وقت أطول.

مثال :

أنشيء مصفوفة مُكونة من 15 عنصر وأضف لها قيم عشوائية ثم أجعل المُستخدم يبحث عن قيمة معينة بها بإستخدام linear search.

```
using System;
namespace First_program
{
    class Program
        {
           static void Main( )
             {
                int[ ] arr = new int[ 15 ];
                Random r = new Random( );
                for ( int i = 0; i < arr.Length; i++ )
                {
                   arr[ i ] = r.Next( ) % 10;
                }
                bool check = false;
                int search = int.Parse(Console.ReadLine( ));
                for( int i = 0 ; i <arr.Length ; i++ )
                {
                   if (arr[ i ] == search)
                   {
                      Console.WriteLine(" Exist ");
                      check = true;
                      break;
                   }
                }
                if (check == false)
                   Console.WriteLine(" Not Exist ");
             }
```

```
        }
}
```

في هذا الكود أنشأنا مصفوفة بها 15 عنصر ثم أستخدمنا Random وهو عبارة class يُستخدم لتوليد قيم عشوائية وأخذنا منه كائن (object) وأستدعينا الدالة ( )Next لتأتي لنا بالقيم العشوائية وأستخدمنا " % " لنحدد المدى الذي يأتي بالقيم العشوائية منه، فمثلاً إذا كتبنا 10% فإنه يولِدّ قيم من 0 إلى 10 فقط، وإذا كانت 50% سيولِدّ قيم من 0 إلى 50 فقط وهكذا، أو يُمكنك تحديد مدى معين وتختار نقطة البداية والنهاية من خلال الدالة (Next(Start_Point , End_Point.

مثل (Next(100 , 150 سيولِدّ أرقام في المدى من 100 إلى 150 فقط.

٢- البحث الثنائي أو المزدوج (Binary search)

وهو يقوم على طريقة أنه يقسم المصفوفة إلى نصفين ويقارن العنصر الذي تبحث عنه بالعنصر الذي يكون في منتصف المصفوفة فإذا كان أكبر منه فإنه يتجاهل كل العناصر الأقل منه وينتقل إلى العناصر الأكبر من منتصف المصفوفة ثم يقسمهم إلى نصفين ويقارن منتصف المصفوفة الجديد بالعنصر الذي تبحث عنه فإذا كان أقل فإنه يتجاهل الشق الي به عناصر أكبر وينتقل إلى العناصر الأقل ويظل يقسم العناصر ويظهر منتصف جديد ويقارنه مرة أخرى مع العنصر الذي نبحث عنه إلى أن يصل إليه أو أنه يكون غير موجود في المصفوفة كلها، فهذه الطريقة أفضل في عمليات البحث عندما تكون المصفوفة كبيرة، هذا في حالة أن المصفوفة كانت مُرتبة ولو لم تكن مُرتبة فيجب ترتيبها أولاً لتعمل عليها هذه الطريقة.
يُمكنك ترتيب العناصر بأي طريقة تعرفها.
أو يمكنك إستخدام الدالة (Sort(array_name وهي دالة جاهزة داخل class الـ Array تُستخدم لترتيب المصفوفة ونمرر للدالة اسم المصفوفة المراد ترتيبها فقط.

مثال :

أبحث عن القيمة 20 بإستخدام binary search.

منتصف المصفوفة الأصلي

20 >12

1	3	6	8	12	15	16	18	20	27

منتصف المصفوفة الجديد

20 >18

15	16	18	20	27

20	27

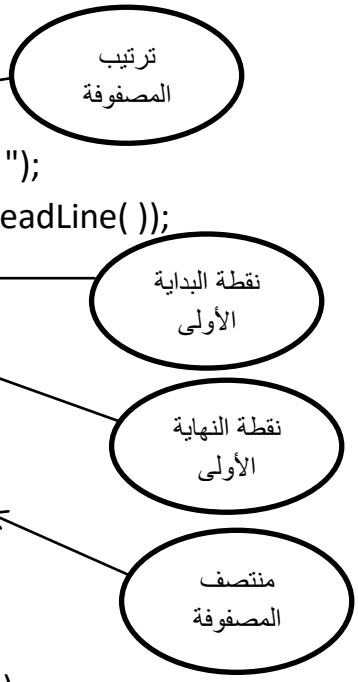
20

✓

مثال :

```
using System;
namespace ConsoleApplication3
{
   class Program
   {
      static void Main( )
      {
         int[ ] arr = new int[10];
         Random r = new Random( );
```

```
        for ( int i = 0 ; i < arr.Length ;  i++ )
        {
           arr[ i ] = r.Next( 30 , 100 );
        }
        Array.Sort( arr );
        Console.Write(" enter target : ");
        int target=int.Parse(Console.ReadLine( ));
        int start = 0;
        int end = arr.Length - 1 ;
        while(start <= end)
        {
           int mid = (start + end )/ 2;

           if( target == arr[mid] )
           {
              Console.WriteLine("exist");
              break;
           }
           else if( target < arr[mid] )
           {
              end = mid-1;
           }
           else
              start = mid+1;
        }
     }
  }
}
```



في هذا الكود عرفّنا نقطة البداية ومررنا لها قيمة إبتدائية صفر ونقطة النهاية وهي تساوي طول المصفوفة ناقص 1، لأنها تعبر عن الـ index (الترقيم) حيث يكون آخر index طول المصفوفة ناقص 1، ثم جئنا بمنتصف المصفوفة حيث يكون مجموع نقطتي البداية والنهاية مقسوماً على 2، ثم نقارن العنصر الذي نبحث عنه بأول منتصف للمصفوفة فإذا كان هو نفس العنصر فيتوقف ويطبع Exist ما عدا ذلك سيقارنه هل العنصر الذي نبحث عنه أكبر من المنتصف أو أقل، إذا كان أقل فإنه سيتجاهل كل العناصر التي أكبر من منتصف المصفوفة وتصبح نقطة النهاية الجديدة هي منتصف المصفوفة ناقص 1، ونقطة البداية تظل كما هي (صفر) مؤقتاً.

ثم يخرج منتصف جديد للمصفوفة وتحدث عليه نفس العملية وإذا كان العنصر الذي نبحث عنه أكبر من المنتصف الجديد فستكون نقطة البداية الجديدة هي المنتصف الجديد زائد 1، وتكون نقطة النهاية تحتفظ بآخر قيمة مُرِرت إليها طالما أن قيمتها لم تتغير.
وتظل تتكرر هذه العملية إلى أن يجد العنصر الذي تبحث عنه في حالة أنه موجود في المصفوفة.
وهي تأخذ وقت أقل لأنها لا تمر على كل العناصر في المصفوفة.

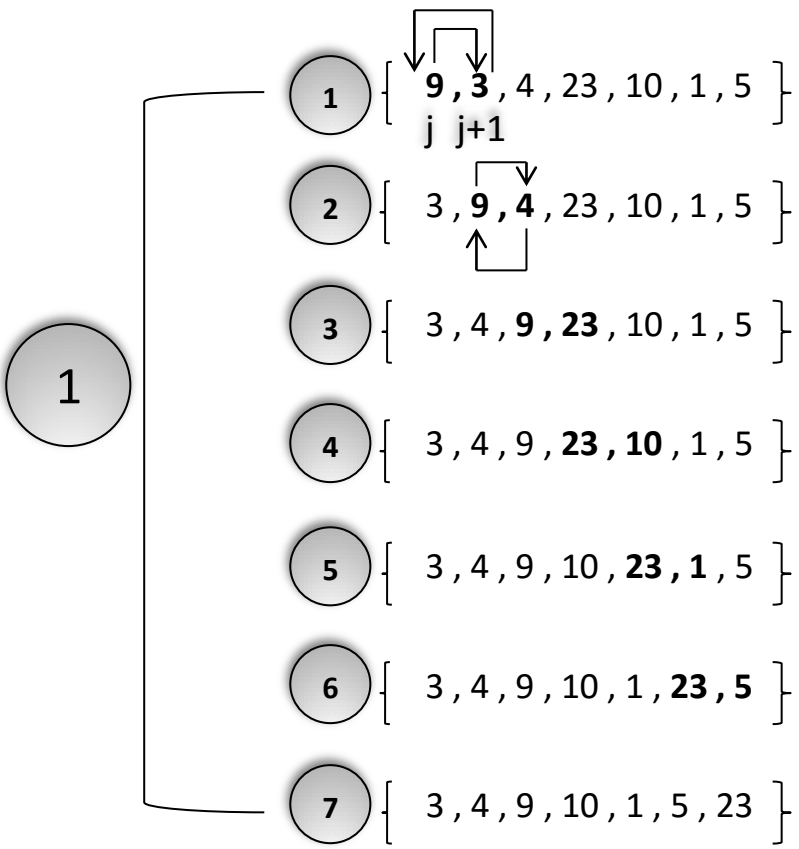
الترتيب (Sorting)

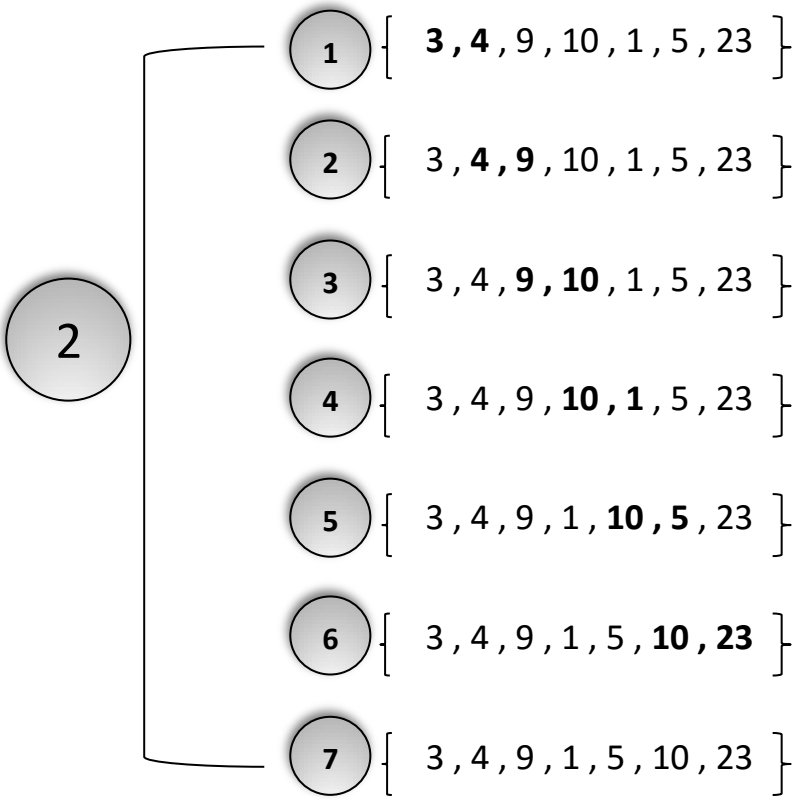
هي عبارة عن عملية ترتيب للبيانات ويوجد العديد من طرق الترتيب، منها :

١- Bubble Sort

يتم الترتيب بهذه الطريقة من خلال مقارنة كل عنصر بالذي يليه فإذا كان العنصر الذي يليه أصغر منه فإنه يبدل القيمتين ببعضهما ثم يبدأ في مقارنة العنصر الثاني بالثالث وإذا كان العنصر الثالث أصغر من الثاني فإنه سيبدل القيمتين وهكذا إلى أن يتم ترتيب المصفوفة أو السلسلة التي بها البيانات وهذه الطريقة هي من أسهل طرق الترتيب لكنها تأخذ وقت أطول ولا يُفضلّ إستخدامها في حالة البيانات الكثيرة جداً.

```
if ( arr[ j ] > arr[ j+1 ] )
    swap ( arr[ j ] , arr[ j+1 ] )
```





وتظل تتكرر هذه العملية إلى أن يتم ترتيب المصفوفة.

مثال :

أكتب برنامج لترتيب مصفوفة بطريقة Bubble.

```
using System;
namespace Buuble_Sort
{
   class Program
   {
      static void buuble( int[ ] arr )
      {
         for (int i = 0 ; i < arr.Length-1 ; i++)
```

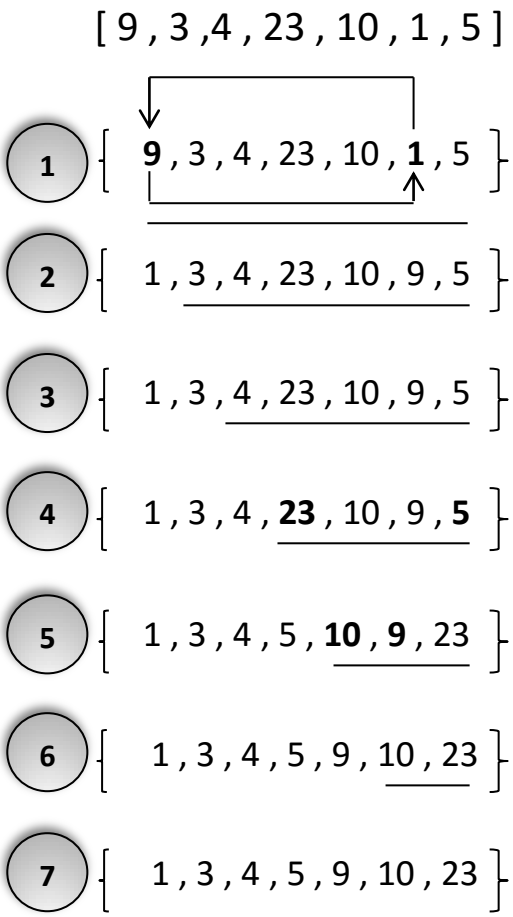
```
        {
            for (int j = 0; j < arr.Length-1; j++)
            {
                if (arr[ j ] > arr[ j+1 ])
                {
                    int temp = arr[ j ];
                    arr[ j ] = arr[ j+1 ];
                    arr[ j+1 ] = temp;
                }
            }
        }
    }
    static void Main(string[ ] args)
    {
        int[ ] x = { 5, 6, 2, 4, 1, 9, 3, 0 };
        buuble( x );
        foreach ( int i in x )
            Console.Write( i +"  ");
    }
  }
}
```

الناتج :

[0 , 1 , 2 , 3 , 4 , 5 , 6 , 9]

٢- Selection Sort

يتم الترتيب بهذه الطريقة من خلال البحث عن أصغر عنصر في المصفوفة أو السلسلة وتبديله مع أول عنصر في المصفوفة ثم نبحث عن أصغر في المصفوفة لكن نبدأ من بعد أول موضع في المصفوفة لأننا بالفعل حصلنا على أصغر عنصر في المرة الأولى ثم نبدله مع العنصر الذي يكون في الموضع الثاني في المصفوفة وهكذا... إلى أن يتم ترتيب المصفوفة.



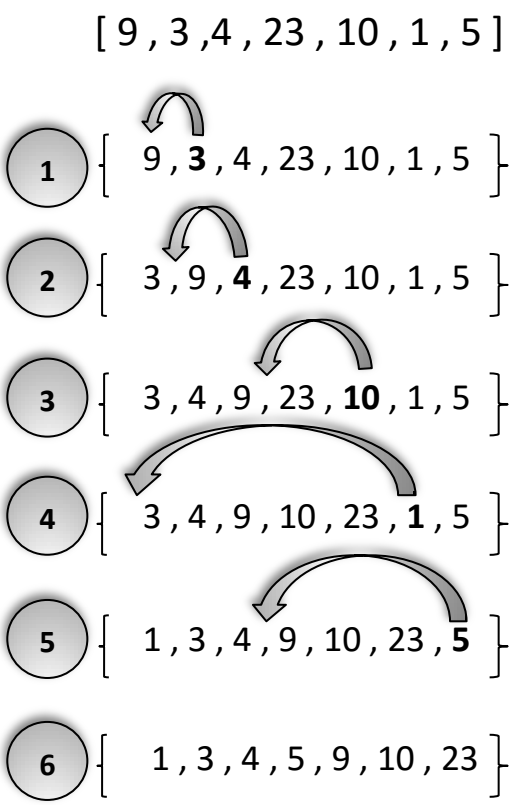
مثال :

أكتب برنامج لترتيب مصفوفة بطريقة Selection.

```
using System;
namespace Selection_Sort
{
   class Program
   {
      static void Selection(int[ ] arr)
      {
         for (int i = 0; i < arr.Length - 1; i++)
         {
            int mini_index = i;
            for (int j = i + 1; j < arr.Length; j++)
            {
               if (arr[ j ] < arr[ mini_index ])
                  mini_index = j;
            }
            int temp = arr[ mini_index ];
            arr[ mini_index ] = arr[ i ];
            arr[ i ] = temp;
         }
      }
      static void Main(string[ ] args)
      {
         int[ ] x = { 5, 6, 2, 4, 1, 9, 3, 0 };
         Selection( x );
         foreach ( int i in x )
            Console.WriteLine( i );
      }
   }
}
```

٣- insertion sort

تتم عملية الترتيب بهذه الطريقة بإفتراض أن المصفوفة عبارة عن جزئين، جزء مُرتب والجزء الآخر لا، بحيث أن أول عنصر في المصفوفة يُعدّ أول جزء مُرتب فيها ثم ننتقل إلى العنصر الذي يليه فإذا كان أصغر منه فنبدّل العنصرين ببعضهما ثم ننتقل إلى العنصر الثالث ونقارنه بباقي العناصر ونضعه في مكانه الصحيح ويتم نقل باقي العناصر الغير مُرتبّة إلى اليمين خطوة واحدة لكل عنصر، وتتكرر هذه العملية إلى أن يتم ترتيب المصفوفة.



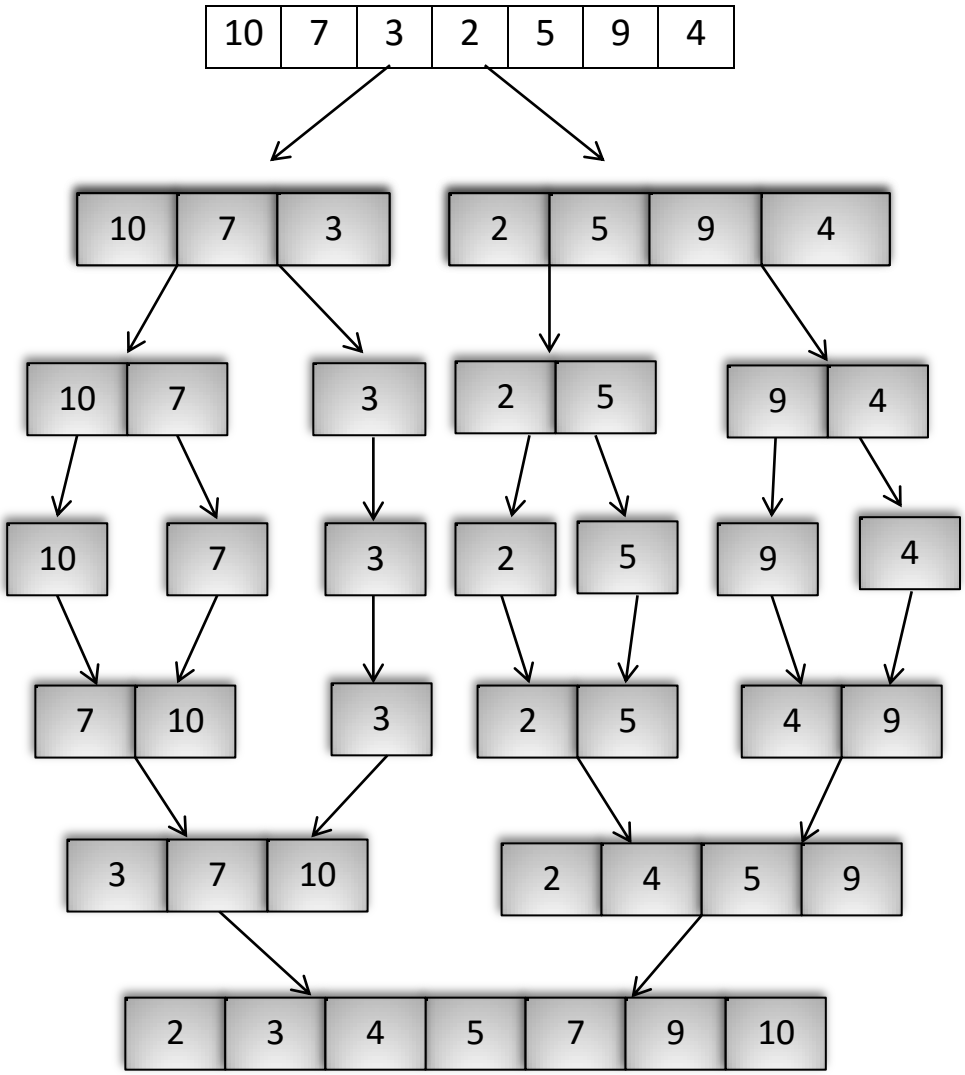
مثال :

أكتب برنامج لترتيب مصفوفة بطريقة insertion.

```
using System;
namespace insertion_sort
{
   class Program
   {
      public static void insertion(int[ ] arr)
      {
         for (int i = 1; i < arr.Length; ++i)
         {
            int key = arr[i];
            int j = i - 1;
            while ( j >= 0 && arr[ j ] > key )
            {
               arr[ j + 1 ] = arr[ j ];
               j = j - 1;
            }
            arr[ j + 1 ] = key;
         }
      }
      static void Main(string[ ] args)
      {
         int[ ] x = { 6 , 7 , 2 , 3 , 5 , 1 , 0 };
         insertion( x );
         foreach ( int i in x )
            Console.WriteLine( i );
      }
   }
}
```

٤- Merge Sort

يتم الترتيب بهذه الطريقة من خلال تقسيم المصفوفة إلى نصفين وتقسيم كل نصف منهما إلى نصفين وهكذا... حتى نحصل على كل عنصر فردي ثم بعد ذلك نعيد دمجهم مرة أخرى لكن بطريقة مرتبة، فتظل الدالة التي ترتبهم تستدعي نفسها بداخل نفسها في النصف الأيسر والأيمن حتى يتم ترتيب المصفوفة كاملة.



مثال :

أكتب برنامج لترتيب مصفوفة بطريقة Merge.

```
using System;
namespace Merge_Sort
{
   class Program
   {
      static void merge( int [ ]arr, int left, int middle, int right)
      {
      int size1 = middle - left + 1;
      int size2 = right - middle;
      int [ ]L = new int [size1];
      int [ ]R = new int [size2];
      for (int i = 0; i < size1; ++i)
      {
         L[ i ] = arr[ left + i ];
      }
      for (int j = 0; j < size2; ++j)
      {
         R[ j ] = arr[ middle + 1 + j ];
      }
      int index1 = 0, index2 = 0;
      int k = left;
      while (index1 < size1 && index2 < size2)
      {
         if (L[ index1 ] <= R[ index2 ])
         {
            arr[ k ] = L[ index1 ];
            index1++;
         }
```

```
      else
      {
         arr[ k ] = R[ index2 ];
         index2++;
      }
      k++;
   }
   while (index1 < size1)
   {
      arr[ k ] = L[ index1 ];
      index1++;
      k++;
   }
   while (index2 < size2)
   {
      arr[ k ] = R[ index2 ];
      index2++;
      k++;
   }
}
static void sort(int [ ]arr, int left, int right)
{
   if (left < right)
   {
      int middle = ( left + right ) /2;
      sort(arr, left, middle);
      sort(arr , middle+1, right);
      merge(arr, left, middle, right);
   }
}
```

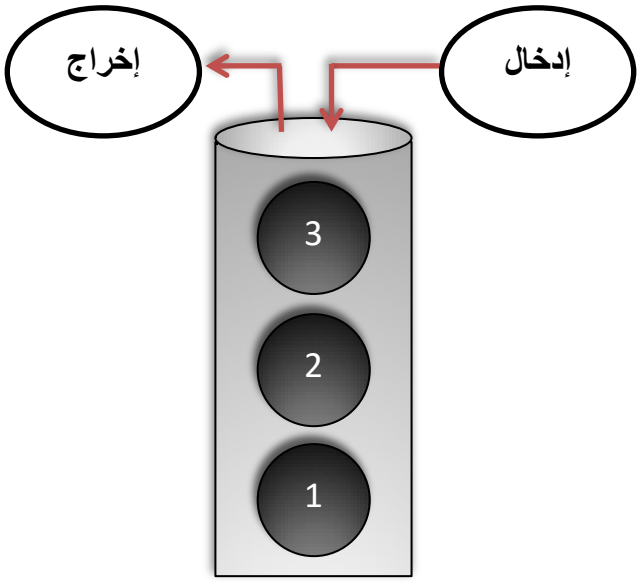
```
    static void Main(string[ ] args)
    {
       int[ ] x = { 10,7,3,2,5,9,4};
       sort(x, 0, x.Length - 1);
       foreach( int i in x )
          Console.Write( i+"  " );
    }
  }
}
```

Stack

الـ stack هو عبارة عن نظام معين أو طريقة معينة لتخزين البيانات وإسترجاعها ولا يهم ما إذا كانت هذه البيانات ستكون بـ Array أو Linked List ، فمثلاً نفرض أنها مصفوفة، فتتم إضافة العناصر من آخر المصفوفة وتتم عملية الإخراج أيضاً من آخر المصفوفة، أي أن آخر عنصر يُضاف هو أول عنصر يخرج، وأول عنصر يضاف هو آخر عنصر يخرج فيتم عرض البيانات التي بداخله بطريقة عكسية، وتعمل مواقع وتطبيقات ضخمة بنظام الـ stack مثل google , youtube , facebook حيث أن آخر عملية بحث قمت بها تكون في قمة عمليات البحث.

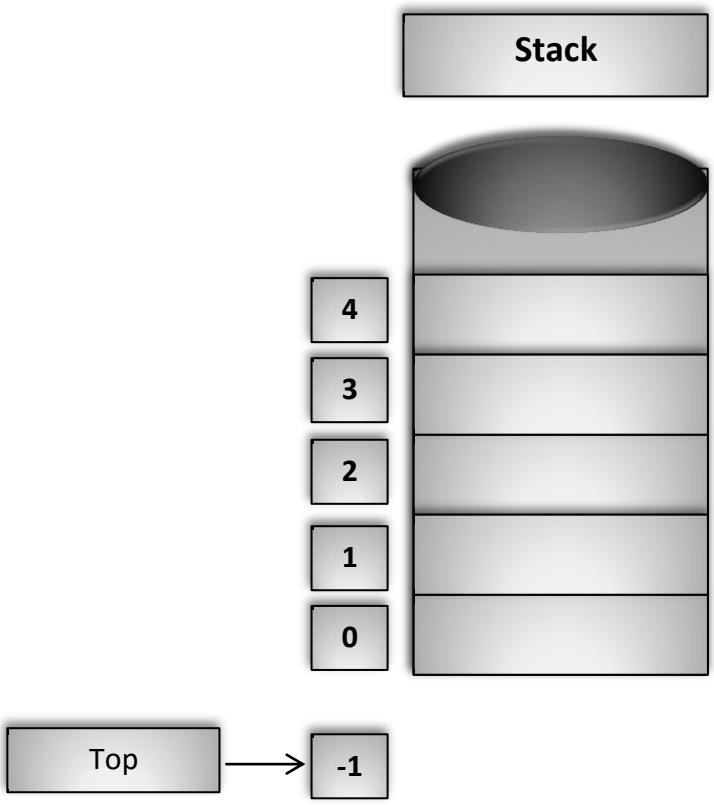
ويتم إختصار طريقة الـ stack بـ LIFO (Last in First Out)
أو FILO (First In Last Out)

أي أن من يُضاف أولاً يخرج آخراً أو من يُضاف آخراً يخرج أولاً.



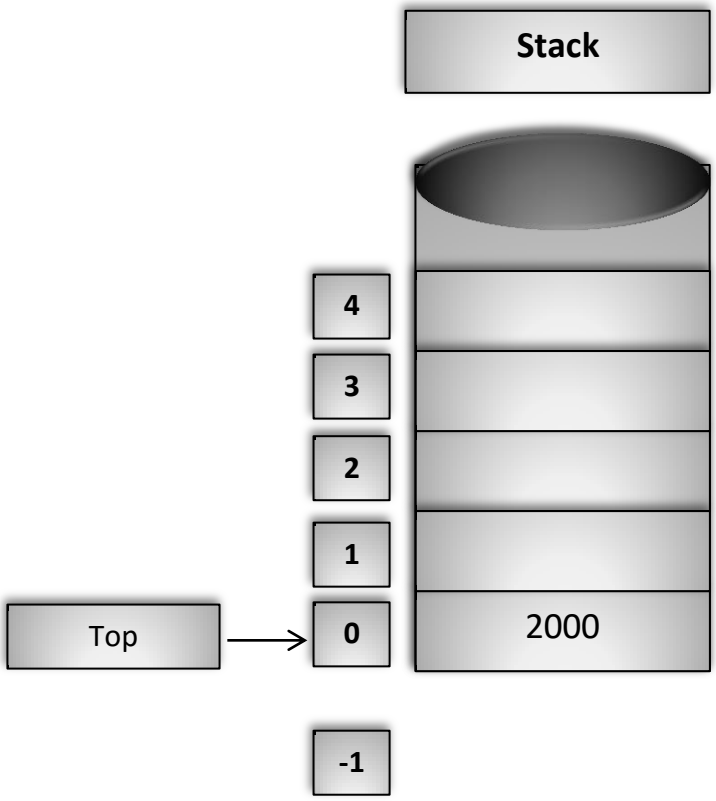
في هذا الجزء لن نستخدم classes جاهزة في اللغة بل سننفذ كل شيء عن طريق الكود ونعرف كيفية تنفيذ الـ stack

في حالة أن الـ stack فارغ



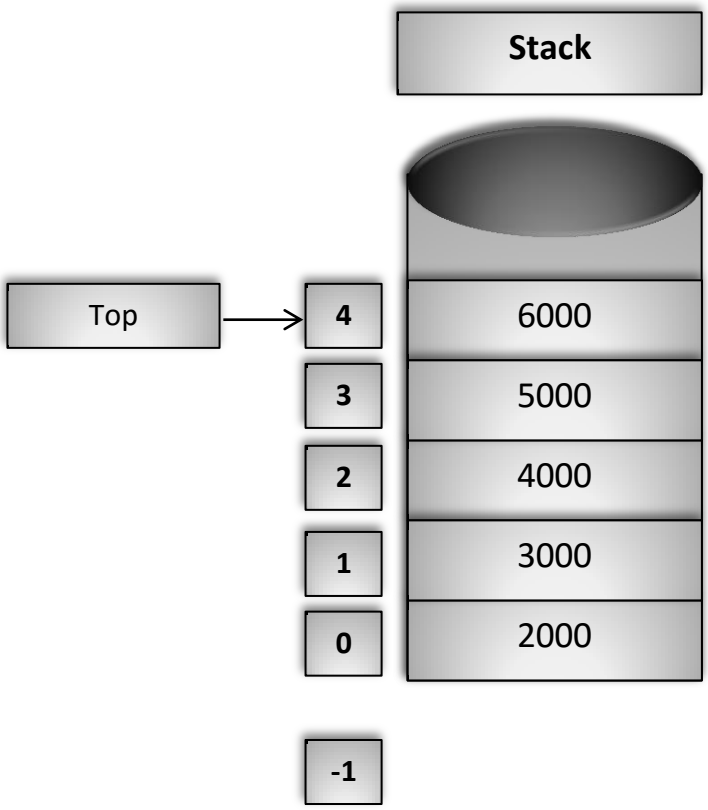
المتغير Top نستخدمه في عمليات الإدخال والإخراج في الـ stack وتكون قيمته الإبتدائية = سالب 1 ،حيث يُعبرّ عن الـ index (الترقيم) الذي يقف عنده. فإذا بدأ ترقيمه بصفر فذلك معناه أنه يقف عند أول موضع في الـ stack أي أنه توجد قيمة في الموضع 0. فيجب أن تبدأ قيمته بسالب 1.

في حالة إضافة أول عنصر للـ Stack



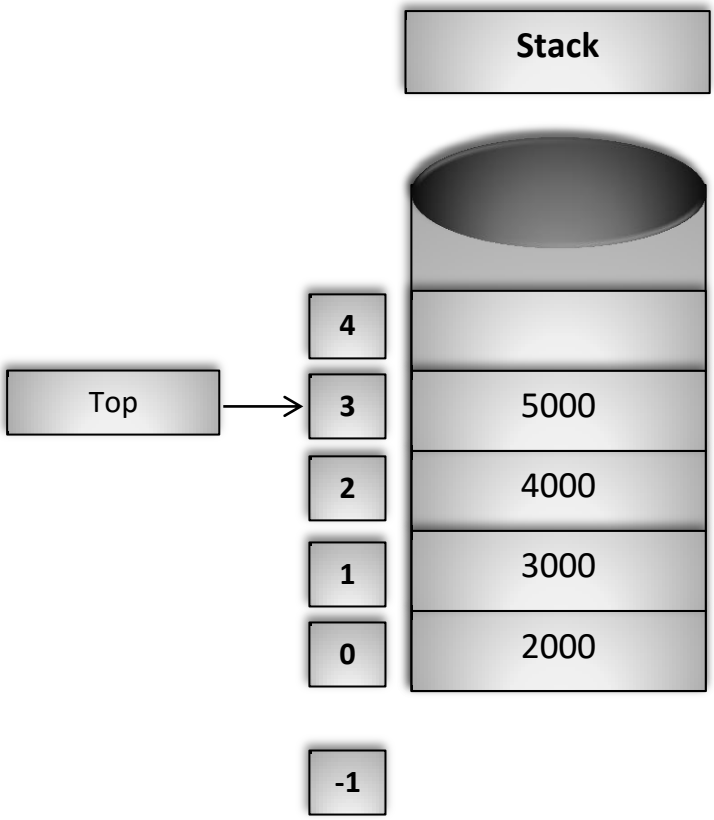
وعند إضافة عنصر للـ stack فإنه تزيد قيمة المتغير Top بمقدار 1 أولاً ثم تُضاف القيمة الجديدة، وآخر قيمة يصل إليها الـ Top هي حجم المصفوفة -1.

عند حذف عنصر من الـ stack



نُخرج القيمة أولاً ثم تنقص قيمة المتغير Top بمقدار 1
في هذا المثال ستخرج القيمة 6000 أولاً ثم ينزل الـ Top إلى العنصر الذي بعده.
تظل قيمة الـ Top تقل كلما أخرجنا عنصر من الـ stack إلى أن يصبح الـ stack فارغ مجدداً، أي تكون قيمة الـ Top سالب 1.

فتكون بهذا الشكل :

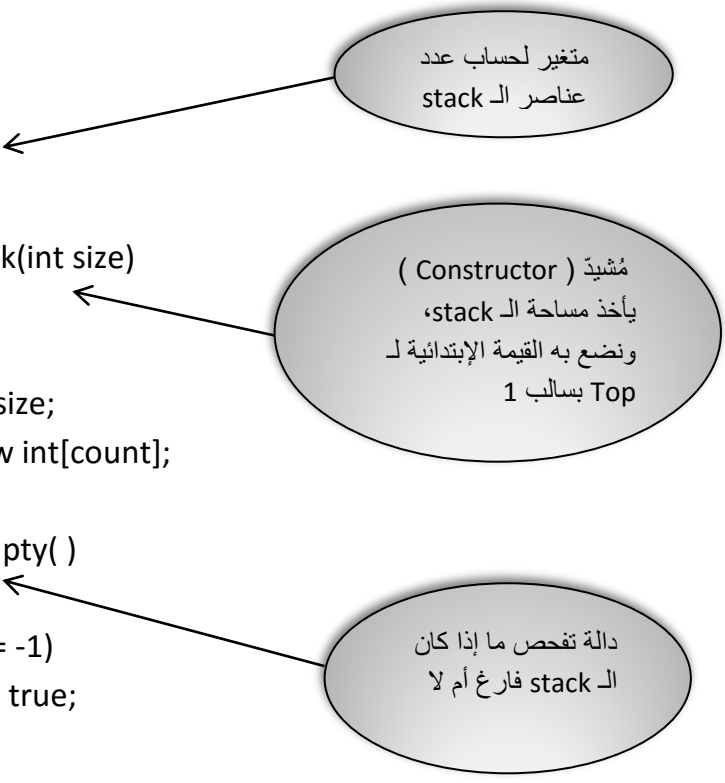


وهكذا....

مثال :

```
using System;
namespace ConsoleApplication4
{
   class Program
   {
      static void Main( )
```

```
  {
    Stack s = new Stack( 9 );
    s.Push(1);
    s.Push(7);
    s.Push(5);
    s.Push(4);
    Console.WriteLine(s.Pop( ));  // 4
    Console.WriteLine(s.Pop( )); // 5
    Console.WriteLine(s.Pop( )); // 7
    s.Push(2);
    Console.WriteLine(s.Pop( )); // 2
    Console.WriteLine(s.Pop( )); // 1
    Console.WriteLine(s.Pop( )); // stack is empty
  }
}
class Stack
{
  int top;
  int count;
  int[ ] arr;
  public Stack(int size)
  {
    top = -1;
    count = size;
    arr = new int[count];
  }
  bool is_Empty( )
  {
    if (top == -1)
      return true;
```



```
        else
            return false;
    }
    bool is_Full( )
    {
        if ( top == count – 1 )
            return true;
        else
            return false;
    }
    public void Push( int value )
    {
        if (is_Full( ) == true)
            Console.WriteLine(" Stack is full ! ");
        else
            arr[++top] = value;
    }
    public object Pop( )
    {
        if (is_Empty( ) == true)
            return " Stack is empty ";
        else
            return arr[ top - - ];
    }
  }
}
```

دالة تفحص ما إذا كان الـ stack ممتليء أم لا

دالة تضيف عناصر في الـ stack فتتأكد أولاً ما إذا كان ممتليء أم لا، لأنه لا يجوز إضافة عناصر به وهو ممتليء

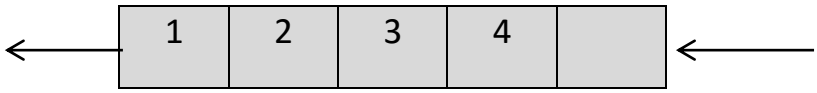
دالة لإخراج عناصر من الـ stack فتتأكد أولاً ما إذا كان فارغ أم لا، لأنه لا يجوز إخراج عناصر منه وهو فارغ

هذا مجرد شكل أولي للـ stack ليُبيَن لك كيفية تنفيذه فقط، يُمكنك أن تضيف به الدوال التي تريدها وتُعدلَّ به كيفما تشاء.

Queue

الـ queue يشبه الـ stack غير أن الـ queue يتعامل مع البيانات بطريقة مختلفة قليلاً عن الـ stack.

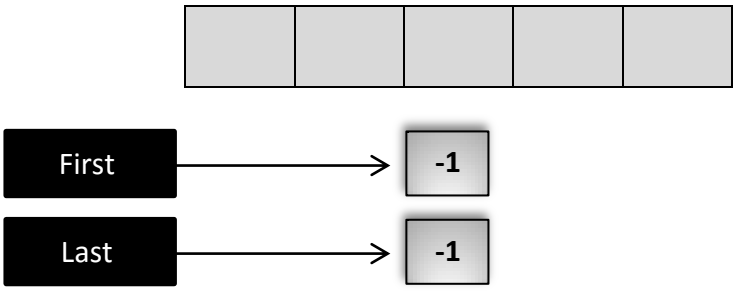
تتم إضافة العناصر من آخر المصفوفة وتتم عملية الحذف أو الإخراج من أول المصفوفة، أي أن آخر عنصر يُضَاف هو آخر عنصر يُحذَف "يخرج"، وأول عنصر يُضَاف هو أول عنصر يُحذَف "يخرج" ويتم عرض البيانات التي بداخله بطريقة عادية وليست عكسية مثل الـ stack، ويسمى الـ queue بـ FIFO (First in First Out) أي أن من يُضاف أولاً يخرج أولاً. فهو ببساطة يشبه الطابور مَن يأتي أولاً يذهب أولاً.



إذا أردت إخراج الـ 2 فيجب أن تُخرج الـ 1 أولاً لأنه خُزِنَ في الـ queue قبلها.

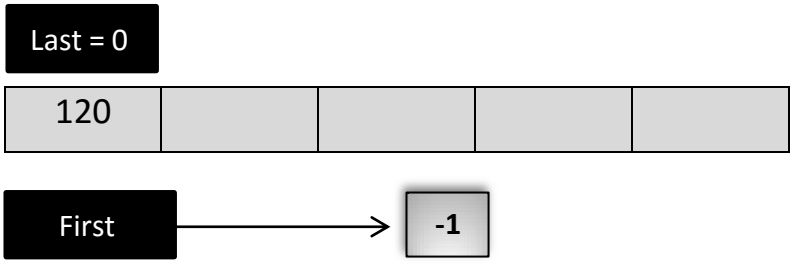
مثلما كان يوجد متغير في الـ stack يُستخدم في عمليات الإدخال والإخراج اسمه Top فيوجد أيضاً في الـ queue متغيرات تُستخدم لعمليات الإدخال والإخراج لكن في الـ stack الإدخال والإخراج كان يحدث من نفس المكان فكان له مدخل واحد وأيضاً المدخل كان نفسه هو المخرج أما في الـ queue الأمر مختلف قليلاً حيث تدخل البيانات من مكان وتخرج من مكان آخر، يُمكنك أن تشبهه بالطابور، أول شخص يخرج من الطابور هو أول شخص في الطابور ثم الذي يليه وهكذا، وإذا أراد شخص ما الدخول في الطابور فإنه يقف في آخر الطابور، إذاً يوجد مكان للدخول ومكان مختلف للخروج من الطابور، في هذه الحالة سنحتاج إلى متغيرين وليس متغير واحد، متغير مسئول عن إدخال العناصر ومتغير مسئول عن إخراجها.
يُمكنك تسمية المتغيرات بأي اسم فهي مجرد متغيرات.

في حالة إذا كان الـ queue فارغ



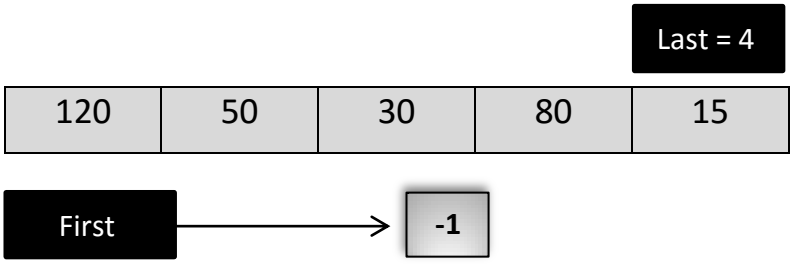
ونعطي لهم قيم إبتدائية أيضاً بسالب 1 فذلك يُعبرّ عن أن الـ queue فارغ.

- عند إضافة أول عنصر في الـ queue يتحرك المتغير Last إلى أول خانة في الـ queue أي تزيد قيمته بمقدار 1 فتصبح 0 وذلك هو أول ترقيم في المصفوفة.



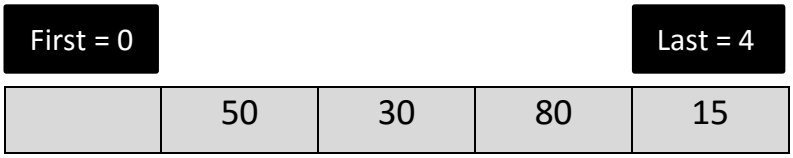
تظل قيمة المتغير First كما هي سالب 1 لأننا لم نخرج أي قيم فيظل واقف خارج الـ queue.

- عند إضافة عنصر آخر للـ queue

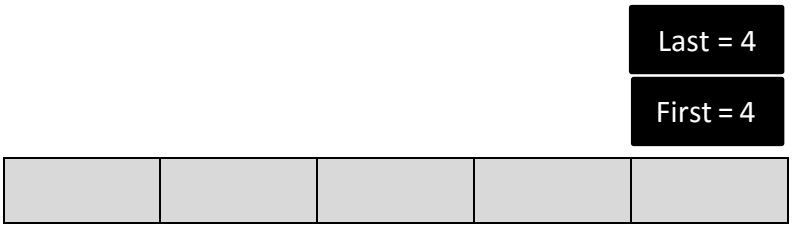


وهكذا...
يمتليء الـ queue في حالة أن الـ Last يساوي طول الـ queue ناقص 1 والـ First يساوي سالب 1.

- عند إخراج عنصر من الـ queue تزيد قيمة الـ First بمقدار 1 أي تصبح قيمته 0 وعندها يخرج أول عنصر في الـ queue.



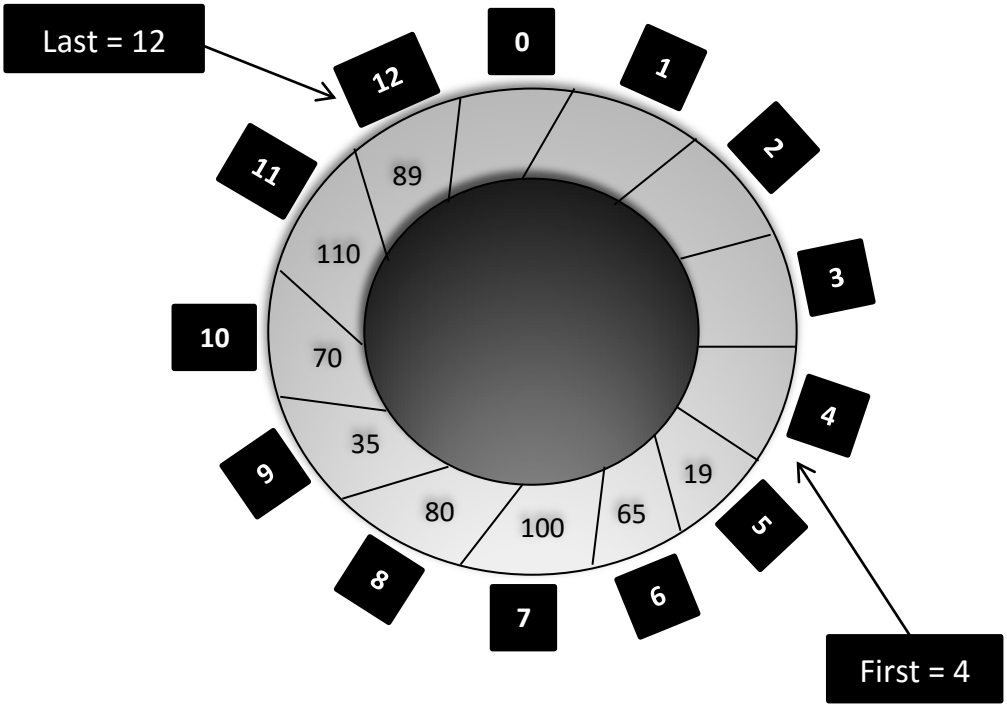
وعند إخراج عنصر آخر تحدث نفس العملية فتزيد قيمة الـ First بمقدار 1 في كل مرة ويُخرج العنصر التالي.



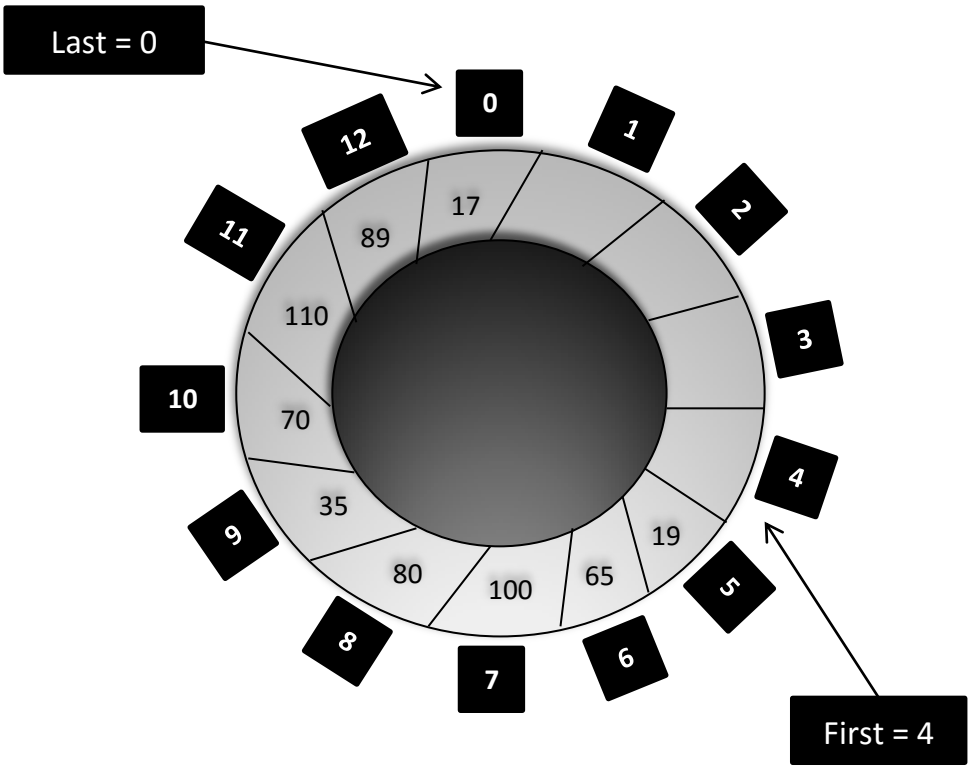
في الحالة العادية لا يُمكنك أن تضيف عنصر آخر طالما أن الـ Last يقف عند آخر خانة في الـ queue حتى إذا كان الـ queue فارغ.

يوجد نوع آخر من الـ queue يُسمى Circular queue وفي هذا النوع تستطيع إضافة عناصر أخرى حتى إذا كان الـ Last يقف عند آخر خانة في الـ queue فهو سيعمل بطريقة دائرية فإذا كان أول عنصر في الـ queue قد خرج فإن الـ Last يعود إلى أول خانة ويُضيف فيها.

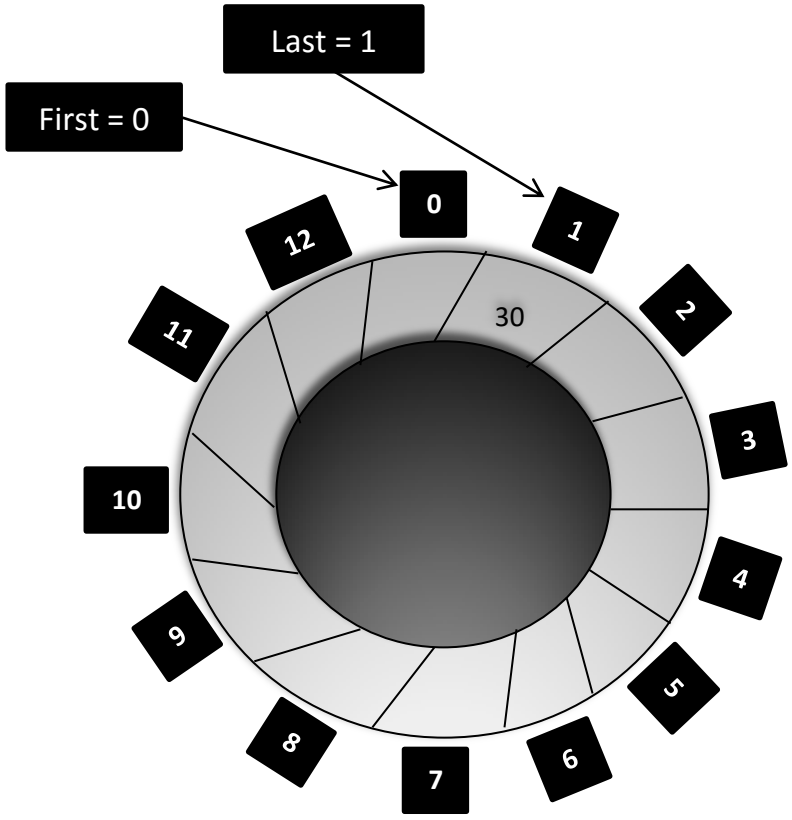
يُمكنك أن تشبهه بهذا الشكل :



في هذه الحالة إذا أردت إضافة عنصر جديد فيجب أن ترجع قيمة الـ Last إلى 0 مرة أخرى


Last = 0
0
1
2
3
4
5
6
7
8
9
10
11
12
17
89
110
70
35
80
100
65
19
First = 4

وأيضاً في حالة أن الـ First ظل يُخرِج عناصر من الـ queue إلى أن وصل إلى آخر خانة في الـ queue فإن أيضاً قيمته ترجع إلى صفر في حالة أنك أردت إخراج عناصر أخرى.



نستخدم هذه المعادلة لجعل الـ queue دائري مثلما ذكرت :

في حالة الـ Last

```
last = (last + 1) % queue_Length;
```

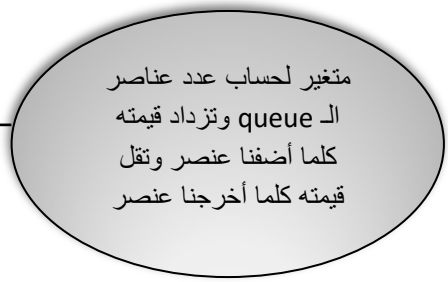
في حالة الـ First

```
First = (First + 1) % queue_Length;
```

الـ queue_Length أي طول الـ queue أو المصفوفة بشكل عام.

مثال :

```
using System;
namespace ConsoleApplication4
{
   class Program
   {
      static void Main( )
      {
         Queue q = new Queue(5);
         q.Enqueue(12);
         q.Enqueue(5);
         q.Enqueue(8);
         Console.WriteLine( q.Dequeue());
         Console.WriteLine(q.Dequeue());
         Console.WriteLine(q.Dequeue());
         q.Enqueue(12);
         q.Enqueue(10);
         Console.WriteLine(q.Dequeue());
      }
   }
   class Queue
   {
      int first;
      int last;
      int count = 0;
      int length;
      int[ ] arr;
```



```
public Queue(int size)
{
   first = -1;
   last = -1;
   length = size;
   arr = new int[ length ];
}
bool is_Empty( )
{
   if ( count == 0 )
      return true;
   else
      return false;
}
bool is_Full( )
{
   if (count == length)
      return true;
   else
      return false;
}
public void Enqueue( int value )
{
   if (is_Full( ) == true)
      Console.WriteLine(" Queue is full ! ");
   else
   {
      last = ( last + 1 ) % length;
      count++;
```

مُشيدّ (Constructor)
يأخذ مساحة الـ queue،
ونضع به القيمة الإبتدائية لـ
first , last بسالب 1

دالة تفحص ما إذا كان
الـ queue فارغ أم لا

دالة تفحص ما إذا كان
الـ queue ممتليء أم
لا

دالة لإدخال عناصر في
الـ queue

```
            arr[ last ] = value;
        }
    }
    public object Dequeue( )
    {
        if (is_Empty( )==true)
            return " Queue is empty";
        else
        {
            first = ( first + 1 ) % length;
            count- - ;
            return arr[ first ];
        }
    }
  }
}
```

LinkedList

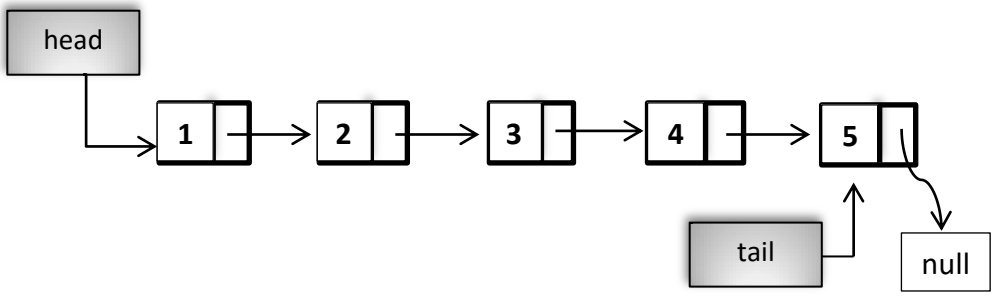
هي عبارة عن سلسلة مترابطة من العناصر، تكون هذه العناصر مرتبطة مع بعضها من خلال مؤشرات فكل عنصر يُشير إلى مكان العنصر الآخر الذي يليه، تكون هذه العناصر مُخزنة بطريقة عشوائية في الذاكرة بعكس المصفوفة التي تحجز عدد من الاماكن في الذاكرة بطريقة متتالية.

الـ linkedList تمكننا من الإضافة أو الحذف في منصف السلسلة بدون الحاجة إلى تبديل أماكن عناصر أخرى. لذا فهي أفضل من المصفوفة " Array " في عمليات الحذف أو الإضافة، فيمكننا إضافة العناصر في البداية أو المنتصف أو نهاية السلسلة، أما في عمليات البحث عن عنصر ما فإنه يأخذ وقت طويل لأنه كما قلت فعناصرها ليست متتابعة في الذاكرة بل إنها مبعثرة في الذاكرة خاصةً إذا كنت تبحث عن عنصر في منتصف السلسلة أو نهايتها.

أما المصفوفة فهي أسرع في عمليات البحث والوصول إلى العناصر المُخزَنة بها، والـ linkedList لا تمتليء أبداً، فكلما أضفت عنصر تضيفه هي في السلسلة بشكل مباشر فهي ليس لها حجم ثابت، أما المصفوفة يجب أن تحدد لها الحجم ولو لم تحدد لها الحجم فإنه كلما تُدخل عنصر بها تُنشَأ مصفوفة جديدة بحجم أكبر وتُنسَخ بها المصفوفة القديمة ثم تُحذَف القديمة من خلال الـ (Garbage Collector) GC.

أما الـ LinkedList لا يكون لها حجم ثابت فكلما أضفت عنصر بها تشير إليه فقط وعندئذٍ يضاف إلى السلسلة مباشرةً.

لنفترض أنه توجد سلسلة بها هذه العناصر (5 , 4 , 3 , 2 , 1) ستكون بهذا الشكل :



null هي كلمة محجوزة في اللغة وتعني لا شيء، أي أنه لا يوجد شيء بعد العنصر الأخير في السلسلة.

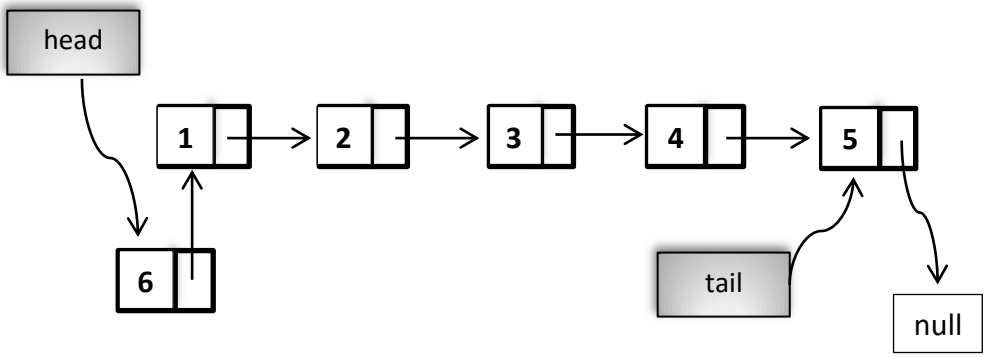
head ---> هو مؤشر يشير إلى أول عنصر في السلسلة.

tail ---> هو مؤشر يشير إلى آخر عنصر في السلسلة.

في حالة أن السلسلة لا يوجد بها عناصر فيكون head , tail يؤشران على null.

وكما نرى كل عنصر يُشير إلى العنصر الذي بعده إلى أن نصل إلى آخر عنصر في السلسلة حيث يُشير إلى null.

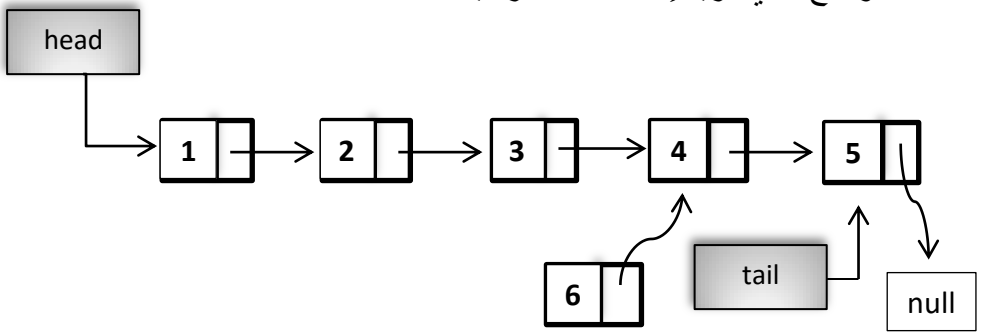
- في حالة الإضافة في بداية السلسلة إضافة عنصر (6) سيكون بهذا الشكل :



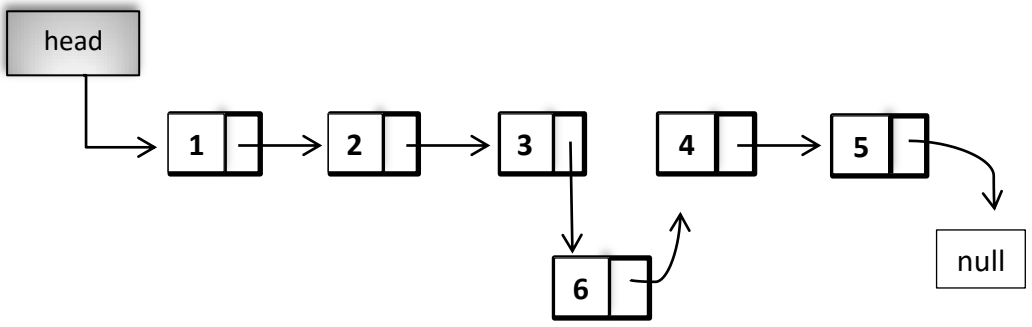
سنجعله يُشير إلى أول عنصر في السلسلة وبعد ذلك نجعل الـ head يؤشر عليه وبذلك يرتبط بالسلسلة ويكون العنصر 6 هو أول عنصر في السلسلة.

- في حالة الإضافة في منتصف السلسلة أو في أي موضع آخر بخلاف البداية والنهاية :

أولاً نحدد الموضع الذي نريد إضافة العنصر فيه.



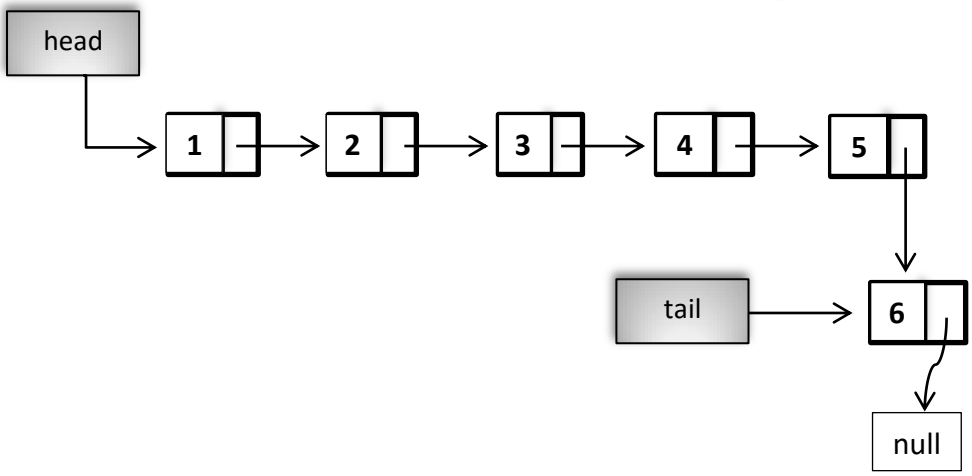
نجعل العنصر الجديد يُشير إلى العنصر التالي للموضع الذي حددناه.



وآخراً نجعل العنصر الذي يسبق الموضع المحدد أن يُشير إلى العنصر الجديد وهكذا يُضاف إلى السلسلة.

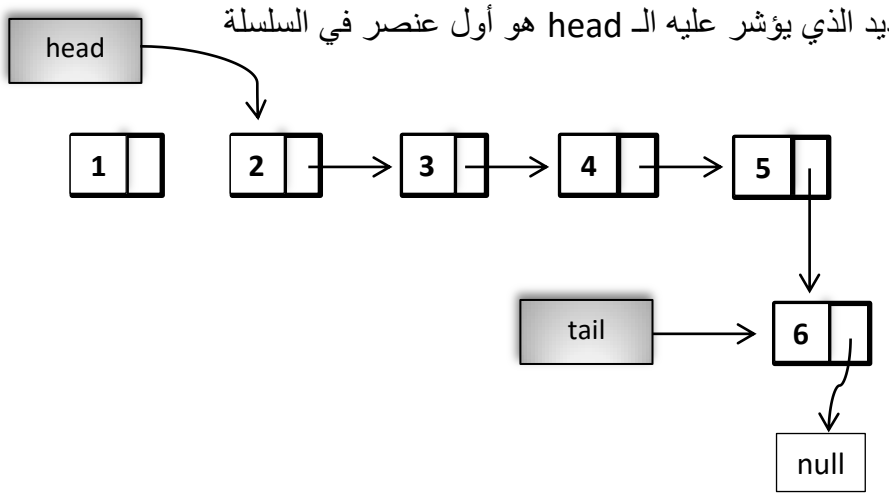
لو جعلنا العنصر الذي يسبق المكان المحدد أن يُشير أولاً على العنصر الجديد ستقع السلسلة من بعد هذا المكان ولن يبقى بها سوى العناصر التي تسبق الموضع المحدد.

- في حالة الإضافة في النهاية :

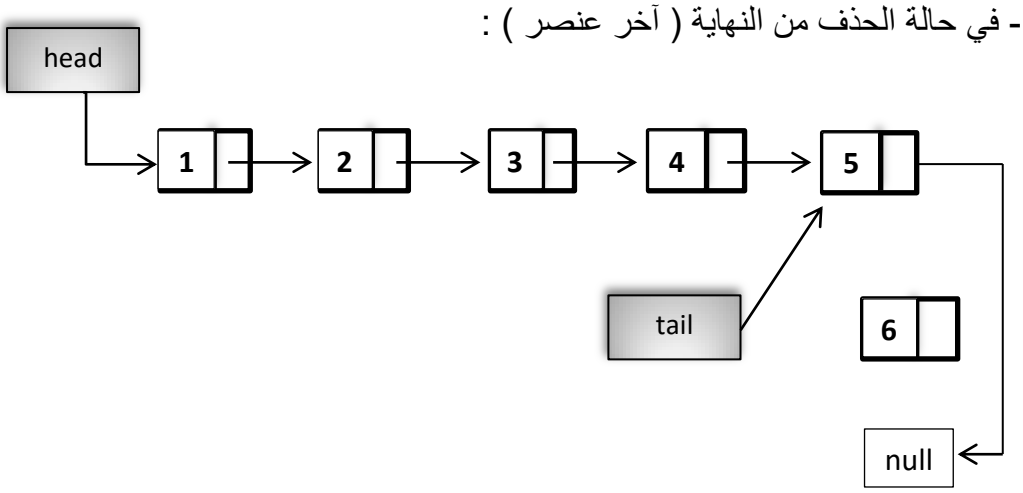


سنجعل آخر عنصر يُشير إليه ثم نؤشر عليه بـ tail.

- في حالة الحذف من البداية (أول عنصر) :
سنجعل الـ head يؤشر على العنصر الذي يلي أول عنصر وبذلك يصبح العنصر الجديد الذي يؤشر عليه الـ head هو أول عنصر في السلسلة



- في حالة الحذف من النهاية (آخر عنصر) :

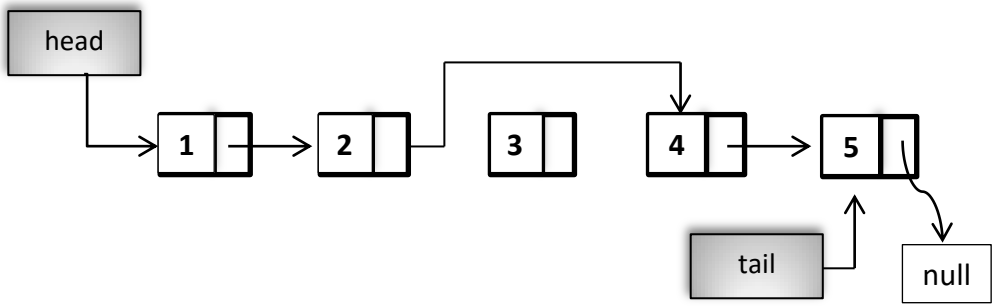


سنجعل العنصر الذي يسبق العنصر الأخير يؤشر مباشرةً على null وبذلك يصبح هو آخر عنصر في السلسلة أي الـ tail.

- في حالة الحذف من منتصف السلسلة أو أي مكان في السلسلة بخلاف البداية والنهاية:

أولاً نحدد موضع العنصر الذي سنحذفه ثم نجعل العنصر الذي يسبقه يشير إلى العنصر الذي يلي العنصر الذي سنحذفه.
لنفترض أنه توجد سلسلة بها العناصر التالية (1 , 2 , 3 , 4 , 5)

ونريد حذف العنصر 3



مثال :

```
using System;
namespace ConsoleApplication5
{
   class node
   {
      public int data;
      public node next;
      public node(int data)
      {
         this.data = data;
      }
   }
   class list
   {
      node head, last;
      public void insert_at_last(int data)
      {
         node n = new node (data);
         if (head == null)
         {
           head = n ;
           last = n ;
         }
        else
         {
               n.next = null ;
              last.next = n ;
              last = n ;
          }
      }
```

أنشأنا class يعبرّ عن العنصر الذي سيُضاف إلى السلسلة وأنشأنا به constructor يأخذ البيانات كمُدخَل له ويضيفها في العنصر وعرفّنا مؤشر ويكون من نفس النوع الذي سيؤشر عليه وبما أنه سيؤشر على node (عنصر) فيجب أن يكون من نفسه نوعه لذا سيكون نوعه class (node)

أنشأنا class يعبرّ عن السلسلة وسيكون به الدوال الخاصة بالسلسلة وعرفّنا المؤشر head, last ويكون من نفس النوع الذي سيؤشر عليه وبما أنه سيؤشر على node (عنصر) فيجب أن يكون من نفسه نوعه لذا سيكون نوعه node فالعنصر الواحد لا يوجد به بيانات فقط بل يوجد به أيضاً مؤشر يؤشر على العنصر الي يليه.

أنشأنا دالة الإضافة من النهاية وتأخذ البيانات، في حالة أن السلسلة لا يوجد بها عناصر أي أن الـ head يساوي null نستدعي الـ constructor الذي أنشأناه في node class ونمرر له البيانات ونجعل الـ head يؤشر عليه، أما إذا كان يوجد بها عناصر بالسلسلة نجعل مؤشر الـعنصر الجديد يؤشر على null ثم نجعل مؤشر الـ last يؤشر على العنصر الجديد ثم نجعل العنصر الجديد last أي آخر عنصر في السلسلة

```
public void insert_at_position(int data, int pos)
{
   node n = new node(data);
   if (pos < 0 || pos > length( ))
      Console.WriteLine("\t this position is out of range");
   else
   {
      if (pos == 0)
      {
         n.next = head;
         head = n;
      }
      else
      {
         node temp = head;
         for (int i = 0; i < pos - 1; i++)
         {
            temp = temp.next;
         }
         n.next = temp.next;
         temp.next = n;
      }
   }
}
public void insert_at_start(int data)
{
   node newhead = new node(data);
   newhead.next = head;
   head = newhead;
}
```

هذه الدالة للإضافة في أي موضع في السلسلة

هذه الدالة للإضافة في بداية السلسلة

هذه الدالة لإظهار عناصر السلسلة

```
public void print( )
{
   node temp = head;
   if (is_Empty( ))
      Console.WriteLine("\t list is empty");
   else
   {
      Console.Write(" head --> ");
      while (temp != null)
      {
         Console.Write(temp.data + " --> ");
         temp = temp.next;
      }
      Console.Write("null");
   }
}
```

هذه الدالة لحساب طول السلسلة أو عدد عناصرها

```
public int length( )
{
   node temp = head;
   int len = 0;
   if (is_Empty( ))
      Console.WriteLine("\t list is empty");
   else
   {
      while (temp != null)
      {
         len++;
         temp = temp.next;
      }
   }
   return len;
}
```

```
public bool is_Empty( )
{
   if (head == null)
      return true;
   else
      return false;
}
```

هذه الدالة للتأكد من أن السلسلة فارغة أم لا

```
public void remove_by_value(int data)
{
   if (head == null)
      return;
   if (head.data == data)
   {
      head = head.next;
      return;
   }
   node current = head;
   while (current.next != null)
   {
      if (current.next.data == data)
      {
         current.next = current.next.next;
         return;
      }
      current = current.next;
   }
}
```

هذه الدالة لحذف عنصر من خلال (تحديد قيمة العنصر)

```
public void remove_by_index(int index)
{
   if (index < 0 || index > length( ))
      Console.WriteLine("\t this index is out of range");
```

هذه الدالة لحذف عنصر من خلال (ترقيمه في السلسلة)

```
    else
    {
        if (index == 0)
            head = head.next;
        else
        {
            node temp = head;
            for (int i = 0; i < index - 1; i++)
            {
                temp = temp.next;
            }
            temp.next = temp.next.next;
        }
    }
}
public void remove_from_last( )
{
    if (is_Empty( ))
        Console.WriteLine(" \t list is empty");
    else if (head == last)
    {
        head = null;
        last = null;
    }
    else
    {
        node current;
        current = head;
        node prev = head;
        while (current.next != null)
        {
            prev = current;
            current = current.next;
```

هذه الدالة لحذف عنصر من نهاية السلسلة

```
        }
        last = prev;
        prev.next = null;
    }
}
public void remove_from_first( )
{
    if (is_Empty( ))
        Console.WriteLine("\t list is empty");
    else if (head == last)
    {
        head = null;
        last = null;
    }
    else
    {
        head = head.next;
    }
}
public void Swap_every_two_node_at_all_node()
{
    node curr = head;
    int temp;
    while (curr != null && curr.next != null)
    {
        temp = curr.data;
        curr.data = curr.next.data;
        curr.next.data = temp;
        curr = curr.next.next;
    }
}
```

هذه الدالة لحذف عنصر من بداية السلسلة

هذه الدالة لتبديل كل عنصرين مع بعضهما في السلسلة

هذه الدالة لعكس عناصر السلسلة

```
public void reverse( )
{
   node var1, var2, var3;
   var3 = null;
   var1 = head;
   while (var1 != null)
   {
      var2 = var3;
      var3 = var1;
      var1 = var1.next;
      var3.next = var2;
   }
   head = var3;
}
```

هذه الدالة لترتيب عناصر السلسلة

```
public void sort_list( )
{
   node curr = head;
   int temp;
   for (int i = 0; i < length( ) - 1; i++)
   {
      for (int j = i; j < length( ) - 1; j++)
      {
         if (curr.data > curr.next.data)
         {
            temp = curr.data;
            curr.data = curr.next.data;
            curr.next.data = temp;
         }
         curr = curr.next;
      }
      curr = head;
   }
}
```

```
public void get_element_by_position(int pos)
{
   node temp;
   temp = head;
   if (pos < 0 || pos > length( ))
      Console.WriteLine("\t this position is out of range");
   else
   {
      for (int i = 0; i < pos; i++)
         temp = temp.next;
      Console.WriteLine("\t the element of this position is " + temp.data);
   }
}
```

هذه الدالة لإظهار بيانات العنصر من خلال موضعه في السلسلة

```
public void searching(int value)
{
   node temp = head;
   if (is_Empty( ))
      Console.WriteLine("\t list is empty");
   else
   {
      while (temp.data != value && temp.next != null)
         temp = temp.next;
      if (temp.data == value)
         Console.WriteLine("\t" + value + " is exist in the list");
      else
         Console.WriteLine("\t" + value + " opps! this element is not exist
                                                in the list ");
   }
}
```

هذه الدالة للبحث عن عنصر في السلسلة من خلال قيمته

```
public void swap_two_nodes(int First_Index, int Second_Index)
{
   if (is_Empty( ))
      Console.WriteLine("\t list is empty");
   if ((First_Index < 0 || First_Index > length()) && (Second_Index < 0
       || Second_Index > length()))
      Console.WriteLine("\t  One of indexes is out of range or both ");
   else
   {
      node temp1;
      node temp2;
      temp1 = temp2 = head;
      for (int i = 0; i < First_Index - 1; i++)
         temp1 = temp1.next;
      for (int i = 0; i < Second_Index - 1; i++)
         temp2 = temp2.next;
      int temp = temp1.data;
      temp1.data = temp2.data;
      temp2.data = temp;
   }
}
public void clear( )
{
   head = null;
   last = null;
}

public int max( )
{
   node temp = head;
   int max = 0;
   for (int i = 0; i < length( ); i++)
   {
```

هذه الدالة لتبديل عنصرين فقط في السلسلة مع بعضهما

هذه الدالة لحذف كل عناصر السلسلة

هذه الدالة لترجع أكبر قيمة في السلسلة

```
        if (temp.data > max)
            max = temp.data;
        temp = temp.next;
      }
      return max;
   }
   public int mini( )
   {
      node temp = head;
      int mini;
      if (is_Empty())
      {
         Console.WriteLine(" the list is empty");
         mini = 0;
      }
      else
      {
         mini = temp.data;
         for (int i = 0; i < length(); i++)
         {
            if (temp.data < mini)
               mini = temp.data;
            temp = temp.next;
         }
      }
      return mini;
   }
}
class Program
{
   static void Main( )
   {
```

هذه الدالة لتُعيد أقل قيمة في السلسلة

```
Console.Title = " Linked List";
Console.ForegroundColor = ConsoleColor.DarkGreen;
list li = new list( );
int choice;
int value;
int position;
Console.WriteLine("\t \t \t\t\t\t\t< Welcome > ");
Console.WriteLine("\t \t \t\t\t\t\t  -------");
try
{
   do
   {
      Console.WriteLine("Select the operation that you want to do it :");
      Console.WriteLine("[ 1 ] insert element at last \t\t\t
                              [ 6 ] Remove element by position ");
      Console.WriteLine("[ 2 ] insert element at first \t\t\t
      [ 7 ] Remove element by value");
      Console.WriteLine("[ 3 ] insert element at any position\t\t
      [ 8 ] Swap every two elements with each other");
      Console.WriteLine("[ 4 ] Remove element from first \t\t
      [ 9 ] Reverse the list");
      Console.WriteLine("[ 5 ] Remove element from last \t\t\t
      [ 10 ] Print the list");
      Console.WriteLine("[ 11 ] Get element by position  \t\t
      [ 12 ] Get length of the list");
      Console.WriteLine("[ 13 ] search in the list by element\t\t
      [ 14 ] Swap two nodes by index ");
      Console.WriteLine(" \t\t[ 15 ] Clear List \t\t\t\t
      [ 16 ] Maximum value in the list ");
      Console.WriteLine("  \t\t[ 0 ] Press 0 to exit \t\t\t\t
      [ 17 ] Minimum value in the list");
      Console.WriteLine(" \t\t[ 18 ] Sorting list \t \t \t\t    ");
```

```
Console.Write(" \t\t\t\t\t Your choice is : ");
choice = int.Parse(Console.ReadLine( ));
switch (choice)
{
   case 1:
      Console.Write("\t enter data of node : ");
      value = int.Parse(Console.ReadLine( ));
      li.insert_at_last(value);
      break;
   case 2:
      Console.Write("\t enter data of node : ");
      value = int.Parse(Console.ReadLine( ));
      li.insert_at_start(value);
      break;
   case 3:
      Console.Write("\t enter data of node : ");
      value = int.Parse(Console.ReadLine( ));
      Console.Write("\t enter position of node : ");
      position = int.Parse(Console.ReadLine( ));
      li.insert_at_position(value, position);
      break;
   case 4:
      li.remove_from_first( );
      break;
   case 5:
      li.remove_from_last( );
      break;
   case 6:
      Console.Write("\t enter position of node : ");
      position = int.Parse(Console.ReadLine( ));
      li.remove_by_index(position);
      break;
   case 7:
```

```
    Console.Write("\t enter value that you want to delete it : ");
    value = int.Parse(Console.ReadLine( ));
    li.remove_by_value(value);
    break;
case 8:
    li.Swap_every_two_node_at_all_node( );
    break;
case 9:
    li.reverse( );
    break;
case 10:
    li.print( );
    Console.WriteLine( );
    break;
case 11:
    Console.Write("\t enter position of element : ");
    position = int.Parse(Console.ReadLine( )); Console.WriteLine();
    li.get_element_by_position(position);
    break;
case 12:
    Console.WriteLine("\t Length of this list = " + li.length( ));
    break;
case 13:
    Console.Write("\t enter element that you search about it : ");
    value = int.Parse(Console.ReadLine( )); Console.WriteLine( );
    li.searching(value);
    break;
case 14:
    int index1, index2;
    Console.Write("\t enter first index to swap with another : ");
    index1 = int.Parse(Console.ReadLine( ));
    Console.Write("\t enter second index to swap with the first:");
    index2 = int.Parse(Console.ReadLine( )); Console.WriteLine( );
```

```
                li.swap_two_nodes(index1, index2);
                break;
            case 15:
                li.clear( );
                break;
            case 16:
                Console.WriteLine(" the maximum value in the list = " +
                                        li.max( ));
                break;
            case 17:
                Console.WriteLine(" the Minimum value in the list = " +
                                        li.mini( ));
                break;
            case 18:
                li.sort_list( );
                break;
            case 0:
                Console.WriteLine("\t \t\t Thanks for your Time ");
                break;
            default:
                Console.WriteLine(" False choice ");
                break;
          }
        }
        while ( choice != 0 );
      }
      catch ( Exception ex )
      {
        Console.WriteLine( ex.Message );
      }
    }
  }
}
```

وآخراً...
الحمد لله الذي قدرّ لي التوفيق في كتابة هذا الكتاب وأتمنى أن يكون قد نال إعجابكم وخرج بشكل يُرضي القاريء، وقد كان الكتاب بمثابة رحلة علمية ممتعة للإرتقاء بموضوع الكتاب لذلك بذلت الكثير من الجهد في إخراجه على المستوى المطلوب، ولكني لا أستطيع أن اقول أنه كتاب شامل ويتصف بالكمال، لأن كل شيء ناقص ويحتاج إلى المزيد ليصل إلى مستوى مرتفع من العلم والمعرفة.

وإن كان الله قد وفقني في كتابة هذا الكتاب فإني أعتبر ذلك مكافأة من الله تعويضاً منه عما بذلته فيه من جهد، وقد كان ذلك هدفي منذ البداية وأتشرف أنني وصلت إليه.