الإبحار في لغة C#.net

Visual Stuio 2013

إعداد: المهندس حسام الدين الرز

الجزء الأول

الإبحار في لغة

VISUAL STUDIO 2013

المستوى:

- ✓ مبتدئ.
- ✓ متوسط.

اعداد:المهندس حسام الدين الرز

بسم الله الرحمن الرحيم

سُبْحَانَكَ لَا عِلْمَ لَنَا إِلَّا مَا عَلَّمْتَنَا إِنَّكَ أَنْتَ الْعَلِيمُ الْحَكِيمُ

الإهداء:

أقدم هذا الكتاب معطراً بعطر ياسمين دمشقي..

الى الحبيبة الغالية سوريا..

وأدعوا الله لها و للأبناء شعبي بالخلاص العاجل والفرج القريب..

الى أمي التي أعجز عن شكرها..

الى والدي الذي حملني مسؤلية الحياة مبكراً..

الى إخوتي وأخواتي..

الى أصدقائي الغالين على قلبي و أخص منهم بالذكر صديقي السيف الدمشقي..

الى من تشاطرني هموم الحياة و متاعبها زوجتي الغالية..

حسام الدين الرز

في حال وجود أي استفسار يرجى التواصل على الإيميل:

Freesyria.syria123@gmail.com

الفهرس

مقدمة:

Introduction:

تعتبر لغة السي شارب الموضوع الساخن الذي يطرحه المبرمجون اليوم أثناء حديثهم عن تطوير التطبيقات وتعد هذه اللغة ثورة نوعية في الأوساط البرمجية شبيه بتلك الثورة التي حدثت في التسعينات عندما صدرت لغة برمجة الجافا. صدرت هذه اللغة في حزيران عام 2000 فهي مازالت حديثة العهد. تم أنشاءها من قبل شركة Microsoft بواسطة فريق Microsoft بقيادة أندرس هيجلزبرج وهو مهندس متميز في Microsoft قام بإنتاج منتجات ولغات برمجة أخرى بها في ذلك "بورلاند توربو C++ وبورلاند دلفي" وركز المهندس في السي شارب على أخذ نقاط القوة التي تتصف فيها اللغات الأخرى مع أضافة التحسينات لجعل هذه اللغة أفضل.

لقد صممت لغة C# من قبل شركة Microsoft لتعمل على منصة خاصة بها تسمى تلك المنصة بإطار عمل .NET – دون الاعتماد المباشر على نظام التشغيل فالشيفرة المكتوبة بلغة C# لا تتخاطب مع نظام التشغيل مباشرة وإنما مع إطار عمل .NET.

لقد صممت شركة Microsoft مجموعة من العمليات والاجراءات ضمن مكتبة ضخمة جدا توفر هذه المكتبة على المبرمجين عناء كتابة الكثير من الشيفرات البرمجية التي يمكن أن توجد بصورة مجردة أو بشكل قياسي للاستخدام العام. تسمى هذه المكتبة بإطار عمل .NET. وهذا واضح من خلال تصريحات شركة Microsoft والتي تشير إلى أن لغة C# هي اللغة الام لكتابة تطبيقات تعتمد على منصة .NET..

إن لغة برمجة C# هي لغة كائنية التوجه (Object-oriented programming -OOP) تجمع بين القوة البرمجية للغة C++ وبين سهولة وبساطة البرمجة بلغة Visual Basic ولن أبالغ إذا قلت أن هذه اللغة قامت بجمع مزايا لغات البرمجة السابقة مثل Delphi و Java وابتعدت عن مساوئ هذه اللغات و أخطائها.

تضع شركة Microsoft جملة من الأهداف من أنشاء لغة برمجة. ومن أهداف لغة C#:

1- لغة بسيطة: جاءت C# لتقضي على التعقيدات والمشاكل الخاصة باللغات مثل Java و C++ فقامت بالغاء الماكرو والقوالب والتوارث المتعدد فهذه تسبب الالتباس لدى مطوري C++ وكذلك ظهور المشاكل. إذا كنت ممن يدرسون C# أول مرة فلا داعي لدراسة هذه الموضوعات.
2- لغة حديثة: أن معالجة الاستثناء وأنواع البيانات القابلة للتوسع وكذلك أمن الأوامر هي سمات تتصف بها اللغات الحديثة pointer مكون أساسي في لغتي C و C++ وهذا المكون من أكثر الأجزاء التي تسبب الالتباس لدى المبرمجين. وقد تم ألغاء العديد من التعقيدات والمشاكل التي يحدثها هذا المكون في C#. لا تقلق بشأن المكونات سوف يتم شرحها في الدروس القادمة.

3- لغة برمجة كائنية التوجه: لكي تكون لغة البرمجة كائنية لابد لها من مفاهيم أساسية تتصف بها وهي الكبسلة capsulationوالتوارث Inheritance وتعدد الأوجهPolymorphism تدعم لغة السي شارب كل هذه المفاهيم وسنتعرف على كل هذه المفاهيم في الدروس المتقدمة.

4- لغة قوية ومرنة: قلنا سابقا لا حدود لهذه اللغة فقط أطلق العنان لخيالك فيمكننا استخدام لغة السي شارب في المشاريع الكبيرة ذات الأشكال المتعددة كالبرامج الرسومية وجداول البيانات وبرامج compilers للغات أخرى.

5- لغة ذات كلمات قليلة: تستخدم لغة C# كلمات دليليه أو أساسية قليلة وهي الأساس التي تبنى عليه إجراءات اللغة. قد تعتقد أن اللغة ذات العديد من الكلمات الأساسية هي لغة قوية ولكن هذا غير صحيح فعندما تقوم بالبرمجة باستخدام لغة C# ستجد أنها لغة يمكن استخدامها في أداء أي مهمة. سنتعرف على الكلمات الأساسية لاحقا.

6- لغة نمطية: الأوامر في C# تكتب على شكل Classes أي أصناف وتحتوي على أساليب العضو وهذه الأصناف يمكن إعادة استخدامها في برامج أخرى.

ويكفي أن نقول إنك بواسطة لغة C# ستتمكن من تصميم أعقد التطبيقات وبمجهود أقل بكثير من الذي يمكن أن تبذله باستخدام لغات برمجة أخرى.

لمن هذا الكتاب؟

يستهدف هذا الكتاب الأشخاص الذين يودون تعلم البرمجة ويودون البدء بلغة برمجية جديدة ويستهدف أيضا المبرمجين المبتدئين الذين يرغبون تطوير التطبيقات بواسطة لغة C# أما بالنسبة للأشخاص الذين سبق وأن تعلموا لغة برمجية سهلة مثل Visual Basic فإن هذا الكتاب هو وسيلتهم لاحتراف لغة C#.

وكخلاصة: فإن هذا الكتاب موجه إلى كل شخص سأم من الكتب التي تتناول لغة C# سواء العربية أو حتى الأجنبية والتي تفترض منه معرفة مسبقة بلغة برمجية أخرى.

إن هذا الكتاب مثالي لنوعين من المبتدئين:

- إذا كنت مبتدئا في عالم البرمجة واخترت لغة C# لتبدأ معها مشوارك سيساعدك هذا الكتاب على تعلم مفاهيم برمجية قوية جدا ستعتبر ركيزة أساسية تعتمد عليها أثناء تصميم تطبيقاتك.
- إذا كانت لديك خبرة مسبقة بإحدى لغات البرمجة ولو كانت خبرة ضئيلة ولكنك تود تعلم كيفية البرمجة باستخدام إطار عمل NET. فإطار عمل NET. يمثل ثورة برمجية بحد ذاته ويدخل عددا من المفاهيم الجيدة على عالم البرمجة هذا ويقدم الكتاب المفاهيم البرمجية كائنية التوجه المتعلقة بإطار عمل NET. وإذا كنت مبرمجا بإحدى لغات البرمجة التي لا تدعم البرمجة كائنية التوجه فإننا قد أفردنا جزءا كاملا في هذا الكتاب لن يعلمك مفاهيم البرمجة كائنية التوجه في NET. وحسب وإنما إتقان هذه المفاهيم أيضا.

ما الذي تحتاج إليه لاستخدام هذا الكتاب؟

إن أهم ما تحتاج إليه لاستخدام هذا الكتاب هو مترجم C# يمثل المترجم Compiler الأداة التي ستحول شيفرتك المكتوبة بلغة C# إلى برنامج تنفيذي ويأتي هذا المترجم كجزء من مجموعة تطوير إطار عمل .NET (.NET Framework SDK) والتي يمكنك جلبها من موقع شركة Microsoft

ولكن كي تتمكن من الاستفادة القصوى من هذا الكتاب فأنت بحاجة إلى بيئة التطوير المتكاملة Visual Studio.NET 2013 والتي تبسط عليك كتابة شيفرة C# من عدة جوانب كما أنها مفيدة جدا وربما أساسية لتطوير تطبيقات Windows وذلك لأنها تحوي على مصمم مرئي للنماذج يحول تصميم واجهات المستخدم إلى متعة حقيقة.

تحميل نسختك من Visual Studio.NET 2013:

أصدرت شركة Microsoft ثلاث نسخ من Visual Studio.NET 2013 وهي:

1- Visual Studio Express 2013 وهي نسخة مجانية.

2- Visual Studio Professional 2013 وهي نسخة مدفوعة الثمن.

3- Visual Studio Ultimate 2013 وهي نسخة مدفوعة الثمن.

الفصل الأول

مدخل إلى لغة C#

أهلا بك في الفصل الأول من هذا الكتاب سوف نلقي نظرة في هذا الفصل على بعض المعلومات الأساسية التي نحن بحاجة إليها للانطلاق مع لغة C#. سوف نتناول لغة C# في هذا الفصل بإيجاز، بالإضافة إلى إطار عمل .NET (NET Framework) وكيفية ارتباط هاتين التكنولوجيتين ببعضهما البعض.

سنبدأ أولا مع وصف عام لإطار عمل .NET. حيث أن إطار عمل .NET يمثل تكنولوجيا جديدة تتضمن العديد من المفاهيم الجديدة والتي تغير المنحى الذي اعتاد المبرمجون سلوكه في التكنولوجيا البرمجية السابقة.

بعد أن نتناول إطار عمل .NET. سننتقل لشرح بسيط عن لغة C# حيث سنقدم الجذور التي نشأت منها تلك اللغة وأوجه التشابه بينها وبين لغة C++.

وأخيرا سنلقي نظرة سريعة على أداة التطوير التي سنستخدمها خلال هذا الكتاب وهي Visual Studio.NET.

ما هو إطار عمل .NET؟

What is the .NET Framework?

يمثل إطار عمل .NET. منصة جديدة ونقلة نوعية تم تطويرها من قبل شركة Microsoft بهدف تطوير التطبيقات.

يمكننا أن نستشف من تعريف إطار عمل .NET. السابق النقاط التالية:

1- ان إطار عمل .NET.موجه لتطوير التطبيقات لجميع أنظمة التشغيل وليس حكرا على تطوير تطبيقات أنظمة التشغيل Windows.
2- إن إطار عمل .NET. مخصص لتطوير جميع أنواع التطبيقات من تطبيقات أنظمة التشغيل إلى تطبيقات الانترنت وخدمات ويب (Web Services) وتطبيقات أخرى يمكننا تصميمها بواسطتها.
3- إن إطار عمل .NET. صمم بطريقة تسمح باستخدامه من خلال اية لغة برمجة فيمكننا استخدام إطار عمل .NET. مع لغة C# ولغة Visual Basic ولغة Jscript ولغة البرمجة C++ ولغة البرمجة F# وغيرها من لغات البرمجة وبالإضافة إلى تواصل هذه اللغات البرمجية مع إطار عمل .NET.فإنها يمكن أن تتواصل مع بعضها البعض أيضا مما يمكن تكوين فريق برمجي

تقني أفراده كل واحد منهم متمكن من لغة برمجة مختلفة عن الاخر وبالتالي يجعل الفريق البرمجي أكثر ديناميكية.

ماذا يوجد ضمن إطار عمل .NET؟

What is in the .NET Framework?

يتضمن إطار عمل .NET. مكتبة ضخمة من الشيفرة التي يمكننا استخدامها من خلال لغات البرمجة المتوافقة مع هذا الإطار مثل لغة C# وذلك بواسطة تقنيات البرمجة كائنية التوجه OOP لقد صنفت هذه المكتبة ضمن وحدات برمجية مختلفة يعتمد تصنيفها على نوعية النتائج التي نود الحصول عليها منها.

على سبيل المثال هناك وحدة برمجية خاصة ببناء الكتل في تطبيقات ويندوز وهناك وحدة برمجية أخرى للتعامل مع الشبكات وأخرى لتطوير تطبيقات الويب وغيرها من الوحدات البرمجية وتنقسم بعض هذه الوحدات إلى وحدات برمجية فرعية أيضا مثل الوحدة البرمجية المستخدمة لتطوير خدمات الويب (Web Services) والتي تمثل جزء من الوحدة البرمجية المستخدمة لتطوير تطبيقات الانترنت.

هناك جزء خاص في إطار عمل .NET. يعرف بعض الأنواع الأساسية يمثل النوع type وصفا للبيانات وللنوع دور كبير في تسيير التشارك بين لغات البرمجة عند استخدام إطار عمل.NET تسمى هذه الآلية بنظام النوع المشترك Common Type System (CTS).

توضيح:

تستخدم لغات البرمجة المختلفة أنواع معطيات مختلفة فعلى سبيل المثال تستخدم لغة C++ نوع المعطيات int لمثيل القيم الصحيحة بينما تستخدم لغة Visual Basic.NET نوع المعطيات integer لتمثيل القيم الصحيحة أيضا إن ما يقوم به نظام النوع المشترك CTS هو توفير قاعدة مشتركة يمكن للغات البرمجة المختلفة استخدام أنواع العناصر البرمجية المختلفة كالمتحولات والكائنات مع بعضها البعض والتواصل بين تلك اللغات بواسطة تلك القاعدة.

وبالإضافة إلى هذا الجزء الهام من مكتبة إطار عمل .NET. فإن .NET يتضمن أيضا محرك زمن تنفيذ اللغة المشترك Common Language Runtime (CLR) والذي يمثل الجزء المسؤول عن تنفيذ جميع التطبيقات المصممة بواسطة مكتبة .NET.

كيف يمكن كتابة التطبيقات باستخدام إطار عمل .NET؟

How do I Write Application using the .NET Framework?

إن كتابة شيفرة برمجية باستخدام إطار العمل NET. يعني كتابة شيفرة برمجية باستخدام إي لغة برمجية تدعم إطار العمل NET. في هذا الكتاب سوف نستخدم Visual Studio .NET 2013 كأداة لتطوير التطبيقات.

إن Visual Studio .NET هي أداة تطوير متكاملة وقوية تدعم لغة البرمجة C# بالإضافة إلى لغات برمجة أخرى مثل F# ولغة البرمجة C++ ولغة البرمجة Visual Basic .NET و غيرها من لغات البرمجة.

لكي نستطيع تنفيذ شيفرة مكتوبة بلغة البرمجة C# يجب أن نحول هذه اللغة إلى لغة يستطيع نظام التشغيل فهمها وتسمى هذه اللغة باللغة المحلية (native code) أو لغة الالة (machine code) تسمى عملية التحويل تلك بالترجمة (compiling) وهي وظيفة المترجم (compiler).

تمر عملية ترجمة شيفرة برمجية مكتوبة بإحدى اللغات التي تدعم إطار العمل NET. عبر مرحلة وسيطة حيث تحول فيها الشيفرة البرمجية المكتوبة بإحدى لغات البرمجة مثل Visual Basic أو C# عبر شيفرة وسيطة تسمى باللغة المتوسطة (Microsoft Intermediate Language) أو تسمى اختصارا بلغة (MISL) إن ترجمة الشيفرة البرمجية إلى لغة (MSIL) هي مهمة بيئة التطوير المتكاملة (Visual Studio .NET) ولكن إلى الان لا يستطيع التطبيق أن ينفذ فما زالت هناك خطوة ضرورية أخرى لتنفيذ التطبيق. إن هذه المرحلة هي مهمة مترجم (Just-In-Time) أو (JIT) والذي يقوم بترجمة لغة (MSIL) إلى لغة محلية مفهومة لنظام التشغيل والتي تسمى لغة الالة.

والشكل (1-1) يبين خطوات عملية الترجمة.

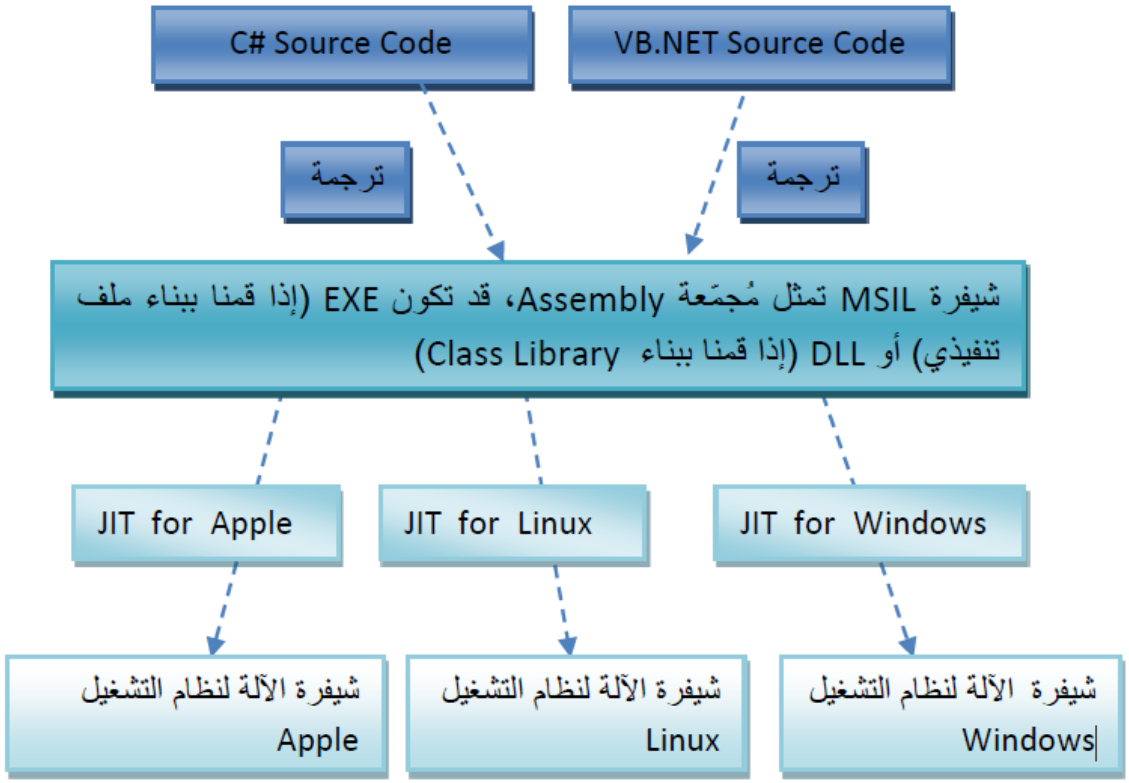
بعد ترجمة الشيفرة البرمجية إلى اللغة الوسيطة (MSIL) سوف يتم حفظ شيفرة (MSIL) ضمن ما يعرف بالمجمعة assembly تمثل هذه المجمعات ملفات التطبيق التنفيذية والتي يمكن أن تنفذ مباشرة دون الحاجة إلى تطبيقات أو أدوات أخرى (لهذه الملفات امتداد exe.) وهناك نوع أخر من المجمعات وهي عبارة عن مكتبات تستخدم مع التطبيقات وتأخذ الامتداد (dll.)

تتضمن المجمعات معلومات وصفية metadata بالإضافة إلى شيفرة MSIL ويمكن أن تحوي مصادر resources اختبارية تمثل بيانات إضافية مثل ملفات صوت أو أيقونات إن مهمة المعلومات الوصفية هي جعل المجمعة ذاتية الوصف بصورة تامة أي أننا لسنا بحاجة إلى اية معلومات أخرى لاستخدام المجمعة وهذا يعني أننا لم نعد بحاجة إلى تزويد مسجل النظام بمعلومات محددة عن المكونات لنتمكن من استخدامها أو الإخفاق في تشغيل التطبيق بسبب حدوث تضارب في هذه المكونات كما كانت المشكلة مع المكونات قبل عهد NET.

هذا يعني أن تطوير التطبيقات أصبح بسهولة نسخ الملفات إلى مجلد على جهاز الكمبيوتر وبما انه لم تعد هناك حاجة لمعلومات إضافية عن الأنظمة التي يستهدفها التطبيق فإن بإمكاننا تشغيل الملف التنفيذي من هذا المجلد مباشرة بافتراض أن محرك زمن تنفيذ اللغة المشتركة NET CLR. مثبت على ذلك النظام.

بالطبع ليس بالضرورة وضع كل شيء نحتاجه لعمل تطبيق في مكان واحد قد تكون لدينا شيفرة تقوم بأداء وظائف عدة مشتركة مع تطبيقات متعددة وفي حالات كهذه من الأفضل وضع المكونات التي يشترك في استخدامها عدة تطبيقات في مكان بحيث يمكن لجميع التطبيقات الوصول إليها يسمى هذا المكان في إطار

عمل .NET. بذاكرة المجمعة العامة Global Assembly Cache (GAC) إن وضع الشيفرة البرمجية في هذه الذاكرة سهل للغاية ويمكننا بذلك بمجرد وضع المجمعة المشتركة ضمن المجلد الذي يمثل هذه الذاكرة.



الشكل (1-1)

المكونات:

Components:

المكون هو عبارة عن برنامج فرعي أو جزء من برنامج يحوي شيفرة تنفيذية وليس شيفرة مصدرية مما يعني أن بإمكان البرامج الأخرى أن تستخدمه دون الحاجة لإعادة ترجمة الشيفرة المصدرية ودون الحاجة لمعرفة الشيفرة المصدرية الخاصة بالمجمعة مما يوفر نوعاً من الأمن.

مزايا المكونات:

من أهم مزايا المكونات:

1- إعادة استخدام البرامج الفرعية في برامج عديدة مثلا: في حال قمت ببناء مكتبة خاصة بك وأردت أن تعطيها لشخص آخر دون أن يعرف ما هي الخوارزميات المتبعة في كتابة الكود الخاص بك، يمكنك أن تعطيه ملف DLL بدلا من الشيفرة المصدرية للمكتبة.

2- المجمعة التي قمت ببنائها يمكنك أن تقوم ببيعها، ويمكنك أن توقعها باسم فريد يُسمى Strong Name لتكون وحيدة على مستوى العالم ولحفظ حقوقك من السرقة أو الاستخدام غير المشروع.

الشيفرة البرمجية المدارة:

Managed code:

تُعتبر إدارة الذاكرة من أهم المواضيع التي على المبرمج المحترف أن يتقنها مع أ ن .NET. تؤمن لنا ما يسمى بالشيفرة المدارة (managed cod) و هذه الشيفرة تحظر علينا التعامل بشكل مباشر مع الذاكرة من خلال ما يُعرف بالمرجع (Reference) إلا أننا وفي بعض الأحيان قد نضطر إلى التعامل بشكل مباشر مع الذاكرة وبالتالي ينبغي علينا فهم المبادئ الأساسية المُستخدمة لكيفية حجز المساحات ضمن المنطقة Heap (تسمى الكومة) وضمن المكدس (Stack).

أي أن الشيفرة المكتوبة باستخدام .NET. تكون مدارة عند تنفيذها أي أنها تخضع للإدارة من قبل محرك CLR الذي لا يهتم بالتطبيق أثناء التنفيذ وحسب وإنما يعالج مواضيع عدة مثل إدارة الذاكرة ومعالجة المسائل المتعلقة بأمن التطبيق والسماح بالتنقيح متداخل اللغات وغير ذلك وبالمقابل تسمى شيفرة التطبيقات التي لا تخضع للسيطرة من قبل محرك CLR بالشيفرة غير المدارة Unmanaged code وتُعتبر لغتي C/C++ هي من أهم اللغات التي تبني تطبيقات بشيفرة غير مدارة وذلك بهدف الوصول إلى الوظائف منخفضة المستوى لنظام التشغيل أما في لغة C# فإنه لا يمكننا كتابة إلا شيفرة برمجية مدارة فقط وبالتالي فإنها تستفيد من مزايا الإدارة الموجودة في CLR والسماح لاطار عمل .NET. بمعالجة أي تفاعل مع نظام التشغيل ولكن هذا لا يعني أننا لن نتمكن من استخدام الوظائف منخفضة المستوى في لغة C# كالوصول المباشر إلى الذاكرة وإنما يمكننا ذلك من خلال ما يسمى بالشيفرة غير الآمنة unsafe code.

مجمع النفايات:

Garbage Collection:

أحد أهم مزايا اللغات التي تعمل تحت منصة .NET. مثل C# أنها لا تُتعب المبرمج في تفاصيل إدارة الذاكرة وكيفية حجز الأغراض وتحريرها وعلى وجه الخصوص جامع النفايات Collector Garbage والذي يوفر على المبرمج عناء تحرير المناطق المحجوزة من قبل البرنامج في الذاكرة وبالنتيجة الحصول على لغة تملك من الفعالية ما يؤهلها لمنافسة لغة C++ من دون الخوض في تفاصيل إدارة الذاكرة .لكن إذا أردنا ان نكتب شيفرة فعالة وسريعة فإنه يجب علينا أن يكون لدينا نظرة عامة عن طريقة حجز الذاكرة ضمن الحاسب.

أي أن مجمع نفايات .NET. يعمل على مراقبة الذاكرة التي يحتلها التطبيق ويقوم بمتابعة وإزالة أية بيانات من الذاكرة لم يعد التطبيق بحاجة لها.

ما هي الخطوات اللازمة لإنشاء تطبيق .NET؟

What are the steps required to create .NET Application?

تتلخص خطوات إنشاء تطبيق في .NET بالخطوات التالية:

1- كتابة شيفرة التطبيق بإحدى لغات البرمجة التي تدعم إطار العمل .NET. مثل C#.
2- ترجمة الشيفرة إلى لغة MSIL والتي سيتم حفظها ضمن المجمعة.
3- عند تنفيذ الشيفرة يجب أن تترجم أولا إلى لغة محلية وذلك بواسطة مترجم JIT.
4- ستنفذ الشيفرة المحلية بتحكم من CLR مع التطبيقات الأخرى المنفذة.

الربط:

Linking:

إن شيفرة C# التي تمت ترجمتها إلى لغة MSIL ليست بحاجة لأن تكون موجودة ضمن ملف وحيد فمن الممكن توزيع شيفرة التطبيق الواحد على ملفات متعددة حيث يتم ترجمة هذه الملفات مع بعضها البعض ضمن مجمعة وحيدة.

تسمى هذه العملية بالربط (Linking) وهي مفيدة للغاية وذلك لعدة أسباب أهمها:

1- سهولة العمل مع ملفات صغيرة بدلا من العمل مع ملف واحد كبير.
2- توزيع المهام البرمجية على عدة مطورين بحيث يمكن لكل مطور العمل بصورة منفصلة عن الاخر مما يوفر مرونة أكبر في تصميم التطبيقات من قبل فريق من المطورين الذين توزع عليهم مهام البرمجة.

ما هي لغة C#؟

What is C#?

وهي إحدى لغات البرمجة التي تستخدم لتطوير التطبيقات التي تعمل ضمن بيئة .NET. فهذه اللغة تمثل نقلة متطورة ومدروسة للغتي C و++C ولغة Visual Basic وقد صممت من قبل شركة Microsoft للعمل خصيصا على منصة .NET.

إن تصميم التطبيقات بواسطة لغة C# أسهل من تصميمها بواسطة لغة ++C وذلك باعتبار ان الصيغ المستخدمة فيها أبسط إن لغة C# هي لغة برمجة قوية وهناك القليل من الأشياء التي يمكنك القيام بها في ++C ولا يمكنك ذلك في C# في الحقيقة يمكننا القيام بالمزايا المتقدمة التي توفرها لغة ++C مثل الوصول والتعامل المباشرين مع الذاكرة ضمن C# وذلك بواسطة الشيفرة غير الآمنة unsafe code

إن هذه التقنية البرمجية المتقدمة خطيرة باعتبار أنه من المحتمل الكتابة فوق كتل من الذاكرة يستخدمها النظام أو تستخدمها تطبيقات أخرى مما يؤدي إلى نتائج وخيمة كانهيار نظام التشغيل ولهذا السبب ولأسباب عدة أيضا فإننا لن نتناول موضوع شيفرة C# غير الآمنة في هذا الكتاب لأنه موضوع متقدم.

ما نوع التطبيقات التي يمكن تطويرها باستخدام لغة C#؟

What Kind of Application Can I Write with C#?

ليست هناك قيود على انواع التطبيقات التي يمكننا إنشائها بواسطة لغة C# حيث تستخدم هذه اللغة إطار عمل .NET. ومن أهم التطبيقات التي يمكن تطويرها باستخدام لغة C#:

تطبيقات Windows: وهي التطبيقات الموجهة لنظام التشغيل ويندوز مثل برنامج Microsoft Office يمكننا إنشاء هذه التطبيقات بواسطة الوحدة البرمجية Windows Forms في إطار عمل .NET. والتي تمثل مكتبة ضخمة من عناصر التحكم Controls مثل أزرار الأوامر وأشرطة الأدوات والقوائم يمكن أن نستخدمها لبناء واجهة المستخدم user interface.

تطبيقات ويب: مثل تصميم صفحات الانترنت وتصميم تطبيقات الانترنت كبرامج المتصفحات وغيرها والتي تسمى بصفحات المخدم النشط (Active Server Pages.NET) وتدعى اختصارا بـ (ASP.NET) ويتم ذلك بواسطة الوحدة البرمجية Web Forms.

خدمات ويب: وهي التطبيقات التي تمكننا من تبادل أي نوع من البيانات ظاهريا عبر شبكة الانترنت باستخدام صيغة بسيطة وذلك بغض النظر عن اللغة البرمجية المستخدمة لإنشاء خدمة ويب أو النظام الذي تعمل الخدمة من خلاله.

ويمكن تحقيق هذه التطبيقات من خلال تقنية (Active Data Objects.NET) أو (ADO.NET).

هناك أشكال أخرى من التطبيقات التي يمكن تحقيقها أيضا بواسطة إطار عمل .NET. مثل إنشاء مكونات الشبكات أو إخراج الرسوميات وتنفيذ المهام الرياضية والتحليلية المعقدة وغيرها من المهام والمتطلبات التطبيقية الاخرى.

برنامج Visual Studio .NET:

Visual Studio.NET Program:

سوف نستخدم في هذا الكتاب بيئة التطوير المتكاملة Visual Studio .NET 2013 التي طورتها شركة Microsoft وذلك لإنشاء أنواع التطبيقات السابقة بدء من تطبيقات سطر الاوامر ذات الواجهة البسيطة ووصولا إلى مشاريع أكثر تطورا.

لم يصمم Visual Studio .NET 2013 لتطوير تطبيقات C# بشكل حصري وإن تطوير التطبيقات بواسطة بيئة التطوير المتكاملة التي يوفرها يجعل من البرمجة أمرا سهلا جدا بالمقارنة مع كتابة الشيفرة المصدرية للتطبيق ككل بواسطة محرر نصوص عادي (مثل برنامج المفكرة Notepad).

سوف نسرد هنا بعضا من النقاط الرئيسية والمزايا التي تجعل من Visual Studio .NET 2013 الاختيار الصائب لتطوير تطبيقات .NET.

1- يستطيع Visual Studio .NET 2013 أتمته الخطوات المطلوبة لترجمة الشيفرة المصدرية وإعطاء المطور تحكما كاملا بأية خيارات مستخدمة لذلك في نفس الوقت.
2- يتضمن محرر نصوص ذكي لكتابة الشيفرة المصدرية فهو يستشعر أخطاء كتابة الشيفرة بذكاء ويقترح الحلول الصائبة وكل ذلك أثناء كتابتنا للشيفرة.
3- يتضمن مصممات لتطبيقات Windows Forms وWeb Forms بمزايا سحب واسقاط عناصر التحكم على نموذج واجهة المستخدم.
4- هناك العديد من أنواع المشاريع التي يمكن تطويرها بلغة C# والتي تحتاج إلى شيفرة تحضيرية ثابتة وبدلا من كتابة هذه الشيفرة في كل مرة نقوم فيها بإنشاء مشاريع جديدة فإن Visual Studio .NET 2013 يقوم بكتابة هذه الشيفرة وتوفيرها لنا بصورة مباشرة مما يقلص الزمن المستغرق للبدء بالمشروع.
5- يتضمن Visual Studio .NET 2013 العديد من المعالجات المساعدة Wizards التي تقوم بأتمتة المهام شائعة الاستخدام تقوم هذه المعالجات المساعدة في الغالب بإضافة شيفرة لملفات موجودة مسبقا بعد أن تطلب منك إتباع عدد من الخطوات الواضحة لتحقيق المهمة المطلوبة منها.
6- يتضمن Visual Studio .NET 2013 العديد من الادوات القوية لاستعراض عناصر المشروع والانتقال فيما بينها كانت هذه العناصر عبارة عن ملفات تحوي شيفرة برمجية أو مصادر أخرى كملفات رسومية أو ملفات صوتية.
7- يتضمن Visual Studio .NET 2013 أداة لنشر المشروع على شكل تطبيق جاهز للاستخدام النهائي.
8- استخدام تقنيات متقدمة لتنقيح الأخطاء (debugging) عند تطوير المشاريع كإمكانية الانتقال في تنفيذ التطبيق خطوة بخطوة، بينما نركز في أثناء ذلك على حالة التطبيق.

الخلاصة:

Summary

لقد تحدثنا عن إطار عمل NET. في هذا الفصل بصورة عامة وناقشنا كيف يجعل NET. من تطوير التطبيقات أمرا أسهل بكثير وأكثر مرونة من السائق لقد تعرقنا أيضا على أنواع التطبيقات التي يمكننا تطويرها بواسطة إطار عمل NET. ولغة C# وما هي الفوائد التي نجنيها من استخدام الشيفرة المدارة (managed code).

لقد تعرقنا على لغة C# ومدى ارتباطها بإطار عمل NET. وشرحنا الأداة التي ستستخدمها لتطوير تطبيقات C# ألا وهي Visual Studio NET.

الفصل الثاني

كتابة برنامج بلغة C#

بعد أن ألقينا نظرة سريعة على لغة C# ومدى ارتباطها بإطار عمل .NET. لقد حان الوقت لكتابة بعض الشيفرات البرمجية.

سوف نستخدم برنامج Visual Studio .NET 2013 أو اختصارا VS لكتابة تطبيقات C# خلال هذا الكتاب.

وبالتالي فإن أول ما سنقوم به هو القاء الضوء على بعض أساسيات بيئة التطوير تلك.

إن Visual Studio .NET 2013 هو برنامج معقد وضخم ويمكن أن يرهب المستخدمين المبتدئين للوهلة الاولى إلا أن استخدامه لتطوير التطبيقات أسهل بكثير من استخدام أية برامج أو أدوات أخرى.

بعد أن نبدأ باستخدام VS في هذا الفصل سنجد أنه من غير الضروري الاحاطة بكل تفاصيل هذا البرنامج كي نتمكن من كتابة شيفرة برمجية بلغة C#.

بعد أن نلقي نظرة سريعة على VS سنقوم بإنشاء تطبيقين بسيطين بواسطته لن نركز في هذا الفصل على الشيفرة البرمجية للتطبيقين وإنما سنحاول إيضاح كيف ستظهر مشاريعك البرمجية بلغة C# في VS.

التطبيق الأول الذي سنقوم بإنشائه هنا يمثل تطبيق Console بسيط أن تطبيقات Console هي تلك التطبيقات التي لا تستخدم واجهات نظام Windows الرسومية وبالتالي لن يكون هناك أزرار أوامر أو قوائم أو أي اهتمام بمؤشر الفأرة (mouse pointer) في هذه التطبيقات وبدلا من ذلك فإن تطبيقنا سيعمل ضمن نافذة سطر الأوامر فقط (مثل برامج نظام التشغيل DOS).

أما التطبيق الثاني في هذا الفصل فسيمثل تطبيق Windows Forms إن هذا النوع من التطبيقات يظهر مشابها من حيث واجهة استخدامه لمعظم تطبيقات Windows الأخرى وعلى الرغم من أن برمجة هذا النوع من التطبيقات يتطلب صيغا أكثر تعقيدا من تلك المستخدمة لتطوير تطبيقات Console إلا أن انشاء هذا النوع من التطبيقات لن يستغرق منك الكثير من العناء.

سوف نستخدم نوعي التطبيقات (Console و Windows Forms) خلال الجزء الثاني والثالث من هذا الكتاب.

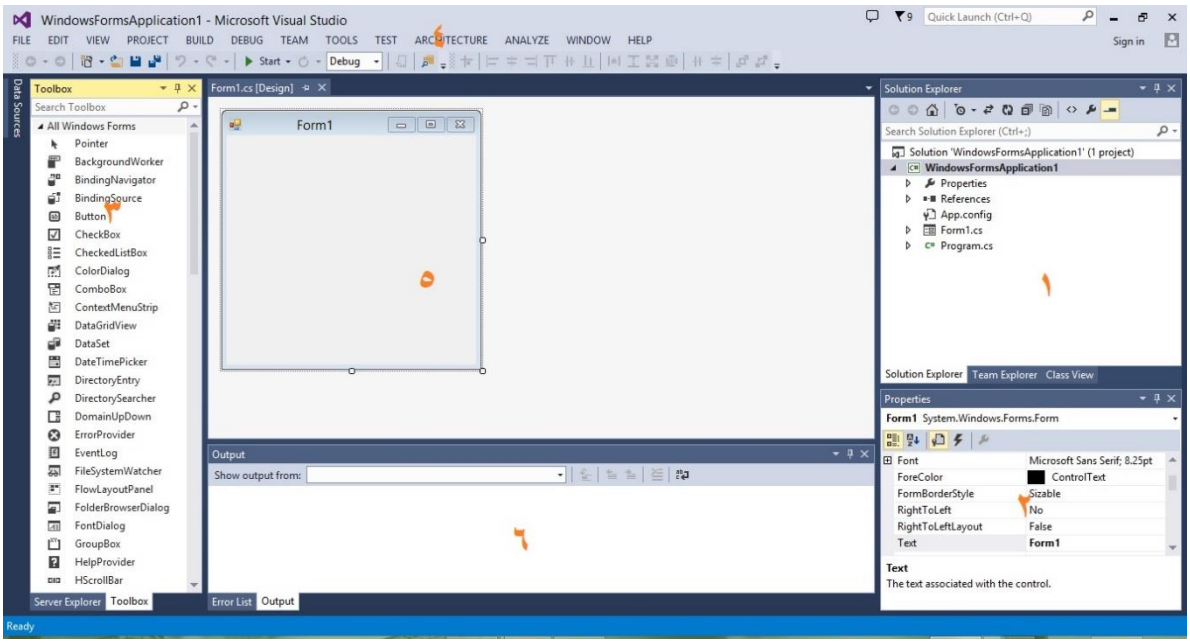
إن التفاصيل الاضافية لتطبيقات Windows ليست ضرورية لتعلم لغة C# فتطبيقات Console تجعلنا نركز على تعلم الصيغ البرمجية بدلا من الاهتمام بمظهر وواجهة التطبيق.

بيئة التطوير المتكاملة Visual Studio .NET:

The Visual Studio .NET Development Environment:

إن بيئة التطوير المتكاملة التي سنبني عليها مشاريعنا في هذا الكتاب هي Microsoft Visual Studio Ultimate 2013 عند تشغيل هذا البرنامج سوف تظهر لدينا نافذة البرنامج كما في الشكل (2-1)

فكما يظهر في الشكل فإن هذا البرنامج يحوي على مجموعة من القوائم المنسدلة كما في أغلب تطبيقات ويندوز وسوف نتحدث عن باقي العناصر في المستقبل وما المهمة الموكلة لكل منها.



الشكل (2-1)

1- **مستعرض الحلول أو العناصر:** يظهر في هذا القسم جميع مكونات الحل Solution والذي يحوي على مشروع Project أو أكثر وكل مشروع يتكون من عدة عناصر كالنماذج Forms والمكتبات Classes وغيرها.

2- **نافذة الخصائص:** عند تحديد أي عنصر في الصفحة الرئيسية أو في نافذة مستكشف الحلول فإن خصائص هذا الكائن تظهر هنا ويمكن تعديلها بسهولة.

3- **صندوق الأدوات:** هذا الصندوق يحوي كل الأدوات التي تحتاجها في برنامجك كصناديق النصوص والأزرار والقوائم وغيرها.

4- **القوائم وشريط الأدوات:** كل الخصائص والامكانيات الموجودة في VS يمكنك التحكم بها بثلاث طرق إما عن طريق القوائم أو الأدوات العلوية الأزرار أو الاختصارات.

5- **الشاشة الرئيسية:** وهي أهم منطقة في VS لأنها منطقة العمل الفعلية حيث تظهر فيها العناصر المكونة لمشروعك والكود المرافق لها.

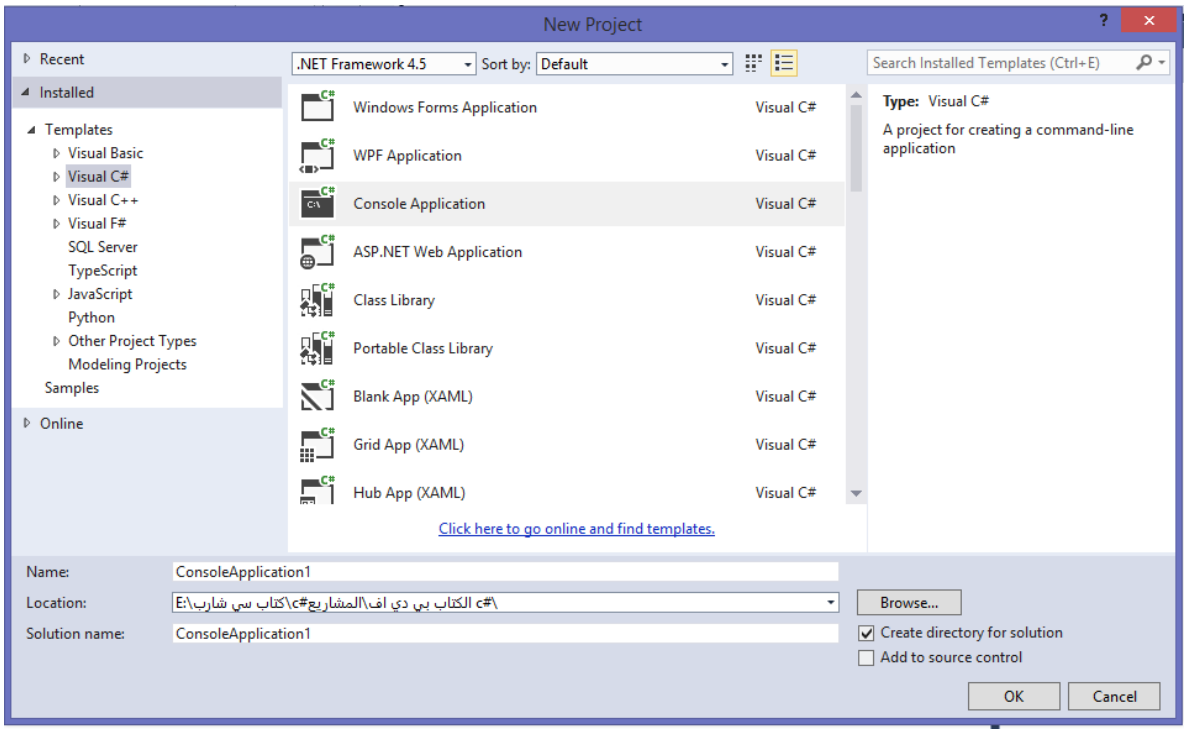
6- **نافذة الخرج:** سوف نتحدث عنها لاحقا.

تطبيقات Console:

Console Applications:

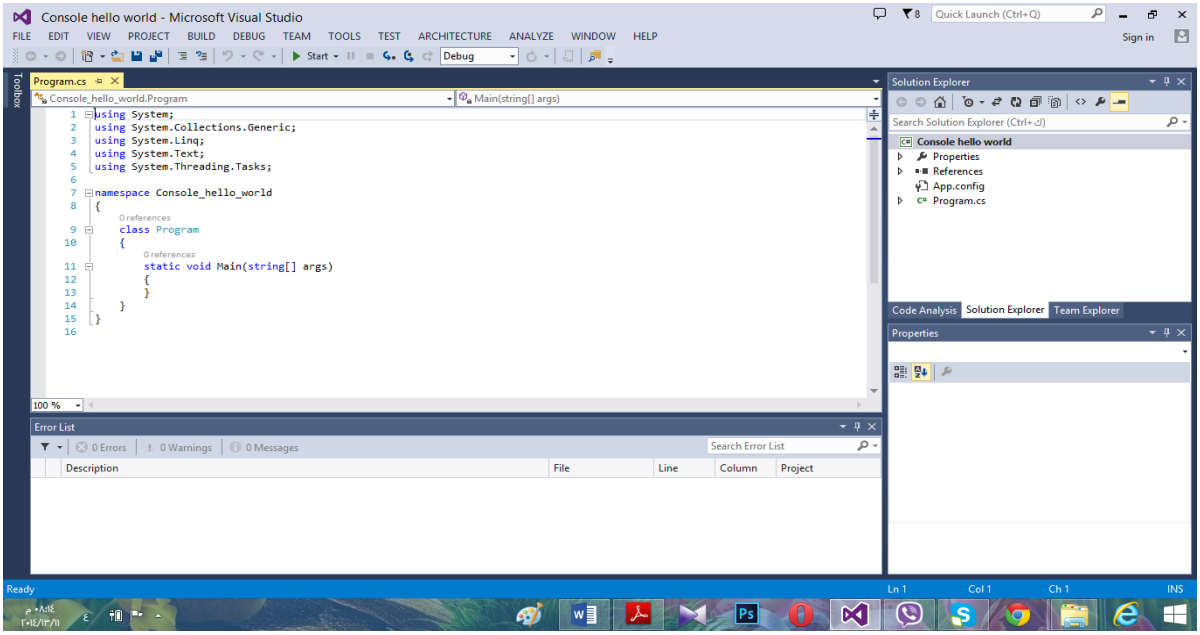
سوف نستخدم تطبيقات كونسول باستمرار في هذا الكتاب وكبداية سنقوم بإنشاء تطبيق Console بسيط.

من نافذة بيئة VS كما في الشكل (2-1) نضغط على الرابط New project من النافذة Start Page أو من القائمة المنسدلة FILE في شريط القوائم المنسدلة نختار الامر New project فتظهر لدينا النافذة كما في الشكل (2-2).



الشكل (2-2)

نختار من القائمة المنسدلة installed على يسار النافذة الخيار Visual C# ومن النافذة المتوسطة نختار Console Application ونختار من القاتمة المنسدلة في الوسط اصدار .NET Framework. الذي نريد بناء التطبيق تحت منصته ويفضل اختيار أحدث اصدار و هو مختار بشكل افتراضي من قبل البرنامج ثم نقوم بتغيير اسم التطبيق ام الخانة Name إلى الاسم Console hello world ثم نختار مكان حفظ المشروع بالضغط على الزر Browser ثم نضغط على زر ok فيظهر لنا الشكل (2-3):



الشكل (2-3)

الان نأتي لمرحلة كتابة الشيفرة البرمجية لكن اين نكتب الشيفرة وماذا سنكتب لا تقلق سنتعلم كل ذلك فلا تستعجل.

ان الشكل (2-3) يظهر اربعة نوافذ وشريط القوائم المنسدلة اين سيتم كتابة البرنامج:

ان النافذة المسماة Solution Explorer تظهر قائمة بالملفات المكونة للبرنامج أما النافذة Properties فهي تظهر خصائص الادوات وسوف نستخدمها في تطبيقات Windows Forms بشكل كبير أما النافذة المسماة Error List فهي تظهر قائمة الاخطاء البرمجية التي نرتكبها أثناء كتابة البرنامج فكما قلنا سابقا محرر نصوص ذكي يكتشف الاخطاء.

يتبقى لدينا النافذة الرابعة المسماة Program.cs وهذه النافذة هي مبتغانا لكتابة الشيفرة البرمجية كما يظهر في هذه النافذة هناك الكثير من التعليمات البرمجية الموجودة سابق بشكل افتراضي فاين سنكتب شيفرتنا انظر للشكل (2-4) لتعرف أين سنكتب شيفرتنا البرمجية:

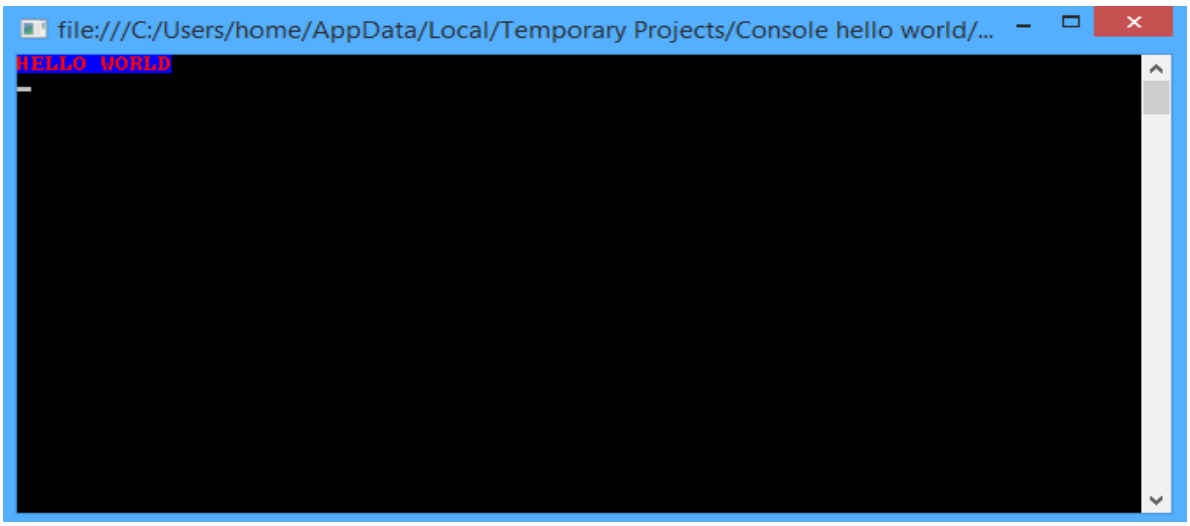
الشكل(2-4)

والان قم بكتابة الشيفرة البرمجية التالية:

```
// لجعل الخلفية زرقاء
Console.BackgroundColor = ConsoleColor.Blue;
// لغيير لون النص للأحمر
Console.ForegroundColor = ConsoleColor.Red;
// لإظهار نص معين على شاشة الكونسول
Console.WriteLine("HELLO WORLD");
// لتمكين المستخدم من قراءة النص الظاهر على شاشة الكونسول
Console.Read();
```

انتبه أن النص المسبوق بشرطتين // هو ليس نص برمجي انما هو تعليق يوضح ما تقوم به التعليمات البرمجية واضافة الشرطتين // ضروري جدا كي لا يحدث خطأ في البرنامج الان نأتي للمرحلة الاخيرة من بناء التطبيق وهي مرحلة الاختبار.

هناك عدة طرق لاختبار البرنامج اما بالضغط على زر F5 من لوحة المفاتيح او بالضغط على زر Start الظاهر أسفل شريط القوائم المنسدلة أو من القائمة المنسدلة DEBUG نختار الامر Start Debugging فيظهر لدينا الشكل (2-5).



الشكل (2-5)

نأتي الان إلى مرحلة حفظ المشروع نختار من القائمة المنسدلة FILE الامر Save all.

نعود الأن إلى شيفرة البرنامج ونقوم بحذف الفاصلة المنقوطة من نهاية أي سطر نريد وليكن من السطر الذي يحوي الشيفرة التالية:

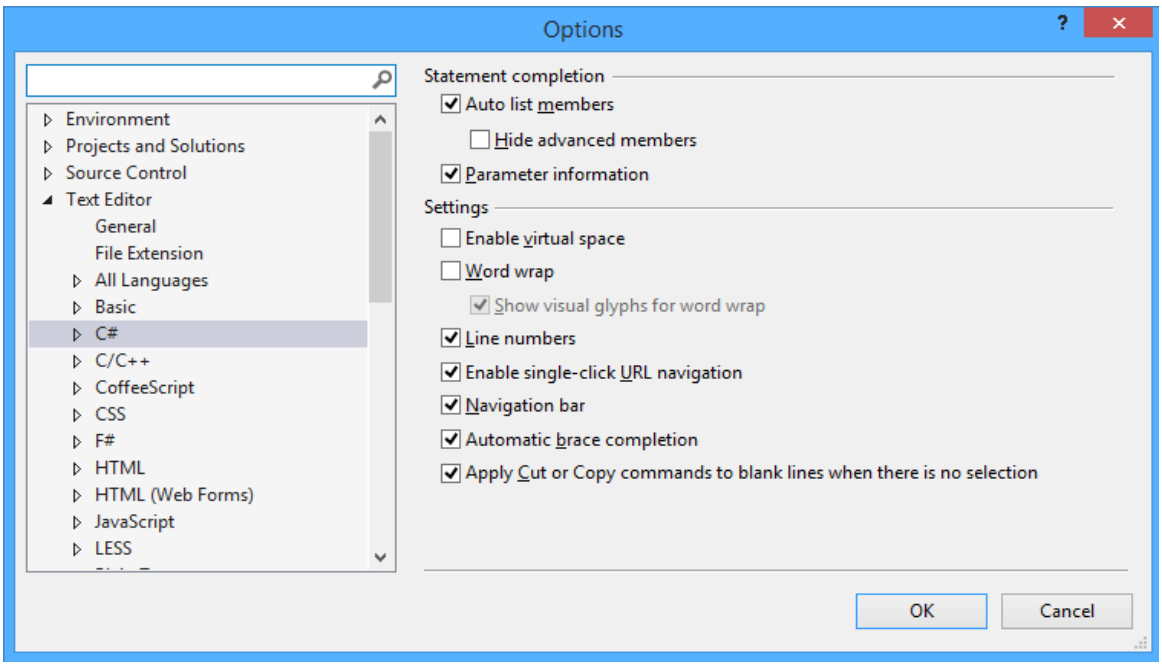
```
Console.BackgroundColor = ConsoleColor.Blue
```

نلاحظ أن نافذة Error List قد اظهرت رسالة تشير إلى وجود خطأ ما في السطر ذي الرقم 14 كما في الشكل (2-6):

الشكل(2-6)

بالنفر المزدوج على رسالة الخطأ نلاحظ ان المؤشر يذهب إلى مكان الخطأ كما أن الرسالة تبين لنا نوع الخطأ المرتكب و هو نقصان الفاصلة المنقوطة لاحظ أنه تم تحيد الخطأ بناء على رقم السطر كما أن أرقام الاسطر لا تعرض بشكل افتراضي في محرر نصوص VS للقيام بذلك اتبع الخطوات التالية:

1- اختر الامر Option من القائمة المنسدلة TOOL سيظهر عندئذ صندوق الحوار Option.
2- انقر على البند Text Editor في شجرة العناصر الموجودة على يسار صندوق الحوار ثم اختر C# سيظهر لك شكل مشابه للشكل (2-7).



الشكل (2-7)

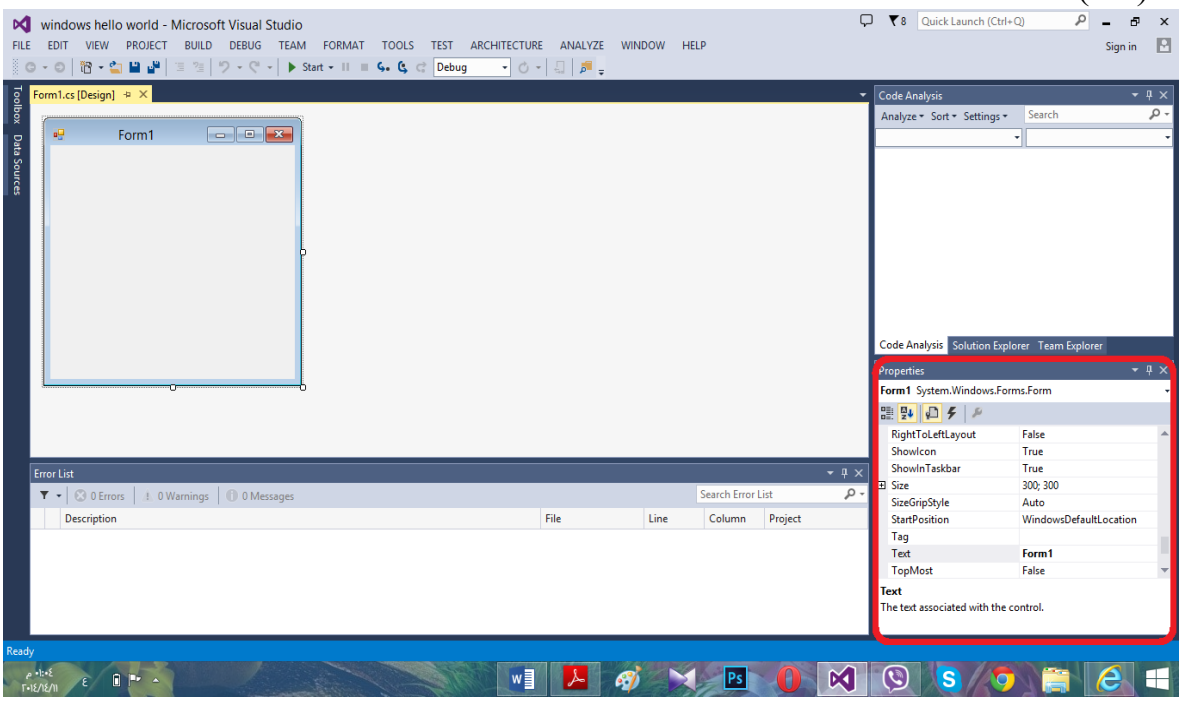
3- فعل الخانة الملاصقة لـ Line numbers ثم انقر على زر ok ستلاحظ عندئذ ترقيم أسطر الشيفرة بصورة تلقائية.

تطبيقات Windows Forms:

Windows Forms Applications:

لن نتعلم في هذا المثال سوى أساسيات بناء تطبيقات Windows Forms ويتم ذلك وفقا للخطوات التالية:

1- اضغط على New Project
2- قم باختيار الخيار Windows Forms Application
3- غير الاسم Name إلى windows hello world ثم اضغط على الزر ok فيظهر لدينا الشكل (2-8).

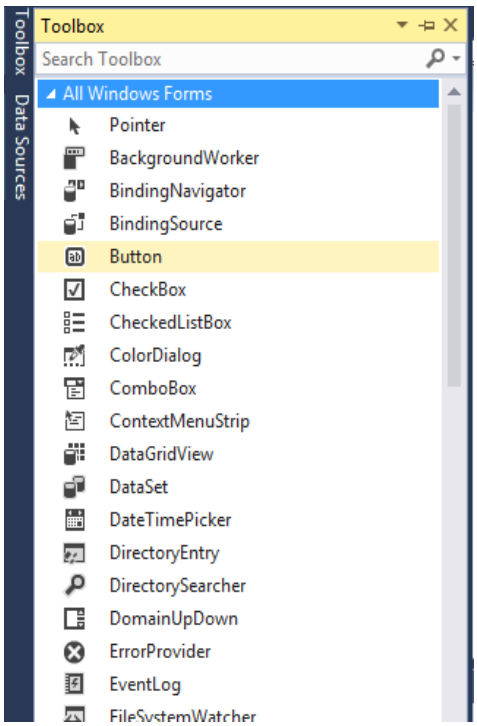


الشكل (2-8)

4- نقوم بضبط خصائص Form1 من نافذة Properties المحددة بإطار أحمر في الصورة وفقا للجدول التالي:

Object	Property	Setting
Form	Name	frmhello
	Icon	shield_green
	RightToLeft	yes
	RightToLeftLayout	True
	Size	400 ;400
	startposition	CenterScreen
	Text	ترحيب

5- ننتقل بمؤشر الماوس إلى يسار نافذه البرنامج فيظهر لدينا صندوق أدوات البرنامج Toolbox نقوم باختيار البند Windows Forms ثم انتقل إلى العنصر Button وانقر عليه نقرا مزدوجا الشكل (2-9) أو قم بسحب وافلاته فوق Form ليظهر زر على Form باسم Button1.



الشكل (2-9)

6- نقوم بالضغط على Button1 مرة واحة فتظهر خصائص Button نقوم بضبطها كما في الجدول التالي:

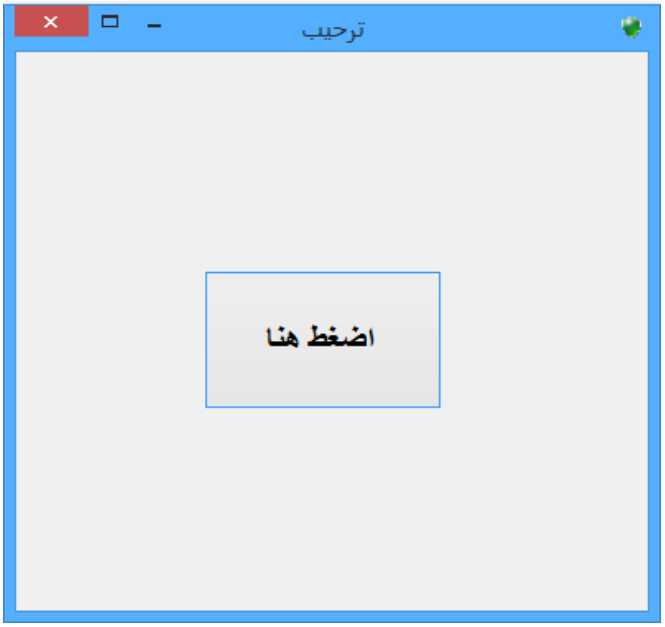
Object	Property	Setting
Button	Name	buthello
	Size	145 ;90
	Font	Times New Roman; 15.75pt;
	Text	اضغط هنا

7- انقر نقرا مزدوجا على الزر الذي تمت إضافته إلى النموذج عندئذ سيتم عرض محرر الشيفرة ويتضمن شيفرة C# لهذا النموذج (أي للملف Form1.cs).

8- اكتب الشيفرة التالية كما كتبناها سابقا في تطبيق Console (لم نذكر هنا إلا جزء من الشيفرة الموجودة في الملف فقط وذلك للاختصار لا أكثر):

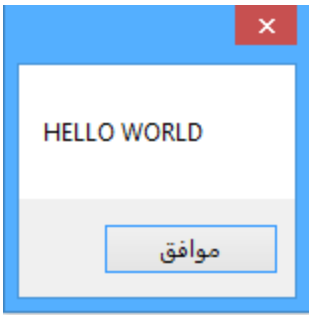
```
private void buthello_Click(object sender, EventArgs e)
{
    MessageBox.Show("HELLO WORLD");
}
```

9- نفذ التطبيق (وذلك بالضغط على مفتاح F5) عندئذ سيظهر الشكل (2-10).



الشكل (2-10)

10- اضغط على الزر اضغط هنا عندئذ ستظهر الرسالة كما في الشكل (2-11).



الشكل (2-11)

كيفية العمل:

How it Works:

ومجددا نقول أن VS قام بجل العمل عوضا عنا و قد سهل علينا لدرجة كبيرة إنشاء تطبيقات Windows وظيفية بقليل من الجهد لاحظ أن التطبيق الذي قمنا بإنشائه للتو يتصرف تماما كأي تطبيق Windows أخر لاحظ أننا لم نكتب أية شيفرة لتحقيق ذلك (كرسم النموذج على الشاشة و التحكم بأزرار التصغير والتكبير الموجودة ضمن سطر عنوان النموذج).

إن الأمر مشابه أيضا للزر الذي أضفناه إلى ذلك النموذج فبمجرد النقر المزدوج على الزر سيعلم VS تلقائيا من أننا نود كتابة شيفرة تنفذ عند النقر على ذلك الزر أثناء تنفيذ التطبيق.

وطبعا فإن تطبيقات Windows ليست محصورة بالأزرار والنماذج فقط فإذا ألقيت نظرة على صندقToolbox ورأيت ما يتضمنه من بنود وعناصر ستلحظ عدد عناصر التحكم التي يمكننا أن نستخدمها في تطبيقاتنا سوف نتعلم استخدام معظم هذه العناصر في موضع ما من هذا الكتاب لاحقاوسوف تدهش لمدى سهولة القيام بذلك.

ربما تكون الشيفرة البرمجية الموجودة في الملف Form1.cs معقدة بعض الشيء بالمقارنة مع الشيفرة في تطبيق Console السابق إلا أن السبب الفعلي لكونها معقدة هو عدد الأسطر البرمجية المكتوبة ضمن الملف. إن معظم هذه الشيفرة تركز على عناصر التحكم (controls) الموجودة على النموذج وهو ما يعكس إمكانية عرض الشيفرة في نمط التصميم (design view ضمن الإطار الرئيسي والذي يمثل ترجمة مرئية لهذه الشيفرة التحتية (سميت بالتحتية لأنها تمثل الاساس الذي سنتمكن بواسطته التعامل مع عناصر التحكم في النموذج). إن الزر الذي أضفناه في هذا المثال على النموذج يمثل واحدا من عناصر التحكم العديدة التي يمكننا استخدامها إن جميع عناصر التحكم التي يمكننا استخدامها في تطبيقانا هذا موجودة ضمن البند All Windows Forms في صندوق عناصر التحكم Toolbox

الخلاصة:

Summary:

لقد قدمنا في هذا الفصل بعضا من الادوات التي ستستخدمها خلال هذا الكتاب لقد أخذنا درسا سريعا عن بيئة تطوير Visual Studio.NET 2013 واستخدمناها في بناء نوعين مختلفين من التطبيقات إن ابسط هذين التطبيقين هو تطبيق Console وهو يلبي متطلباتنا الحالية ويمكننا من التركيز على اساسيات البرمجة بلغة C# أما تطبيقات Windows أكثر جاذبية وقوة باعتبارها تعتمد على بيئة Windows الرسومية.

والان نحن نعلم كيف نقوم بإنشاء تطبيقات بسيطة ويمكننا الانتقال إلى المهمة الحقيقية في تعلم لغة C# وهو موضوع الجزء القادم من هذا الكتاب.

الفصل الثالث

الصيغة الأساسية للبرمجة بلغة C#

تتشابه لغة C# من حيث الصيغة البرمجية المستخدمة ضمنها مع تلك الصيغة المستخدمة في لغتي C++ وJava في البداية قد تجد هذه الصيغة مركبة نوعا ما وهي أقل قابلية للقراءة من بعض اللغات البرمجية الأخرى (كلغة Visual Basic مثلا) ولكن بعد أن تقحم نفسك في عالم البرمجة بلغة C# ستجد أن الاسلوب الذي تستخدمه هو أسلوب محسوس ومن الممكن أن تكتب شيفرة مفهومة دون عناء شديد.

و القاعدة الاولى هي أن مترجم لغة C# لا يتأثر برموز المساحات البيضاء الزائدة في الشيفرة بعكس مترجمات اللغات البرمجية الاخرى سواء كانت هذه المساحات عبارة عن فراغات (Spaces) أو أسطر فارغة (carriage returns) أو حتى علامات جدولة (Tape) هذا يعني أن لدينا الكثير من الحرية في الطريقة التي يمكن أن ننسق فيها شيفرتنا المصدرية على الرغم من أن اتباع اسلوب محدد سيساعد كثيرا على تسهيل قراءة الشيفرة تتألف شيفرة C# من سلسلة من التعليمات بحيث يفصل بين كل تعليمةوأخرى برمز الفاصلة المنقوطة ";" وبما أنه سيتم تجاهل المساحة البيضاء فإنه يمكننا من أن نضع تعليمات عديدة ضمن سطر واحد لكن وبسبب صعوبة قراءة الشيفرة عندئذ فإننا سنكتب كل تعليمة في سطر منفرد بل ويمكننا أن نضع التعليمة الواحدة على عدة أسطر.

للغة C# بنية مؤلفة من عدة كتل (Block-Structured) وهذا يعني أن جميع التعليمات تمثل جزء من كتلة شيفرة تتوضع هذه الكتل ضمن الأقواس {} ويمكن أن تتضمن أية كمية من التعليمات ويمكن أن لا تحوي أي تعليمة بالمرة.

إذا يمكننا تمثيل كتلة بسيطة من شيفرة C# بالشكل التالي:

```
{
    MessageBox.Show
        ("HELLO WORLD");
}
```

لاحظ أن السطر الأول والثاني يمثلان تعليمة واحدة وذلك لأنه لا توجد هناك فاصلة منقوطة في نهاية السطر الأول.

التعليقات:

Comments:

بالإضافة لما سبق سنجد ضمن شيفرة C# ما يعرف بالتعليقات ولا تمثل هذه التعليقات شيفرة برمجية حقيقية وإنما وصفا توضيحيا لأسطر الشيفرة التي تكتبها.

إن كتابة التعليقات ضمن الشيفرة البرمجية له الكثير من الفوائد والتي سنكتشفها مع الوقت حيث يمكننا أن نكتب أي نص ضمن التعليقات وبأي لغة كانت وسيتجاهل مترجم C# هذه التعليقات عند ترجمة الشيفرة حيث أن هذه التعليقات ستساعدنا على تذكر ما نقوم به ضمن الشيفرة كان تكتب مثلا "يقوم هذا السطر بسؤال المستخدم عن اسمه" وللغة C# اساليب مختلفة في كتابة التعليقات يسمى الاسلوب الأول بالتعليقات السطرية أما الاسلوب الثاني فيسمى بالتعليقات الحرة.

إن الصيغة العامة للتعليقات السطرية هي كما يلي:

```
// يقوم هذا السطر بإظهار صندوق رسائل مكتوب عليه رسالة ترحيب
```

أي أن التعليقات تبدأ بالرموز "//" وبعد هذه الرموز يمكننا كتابة التعليق الذي نريد طالما أن هذا التعليق يقع ضمن سطر واحد وإلا فإن المترجم سيعطينا رسالة خطأ ولتصحيح الخطأ يجب وضع "//" قبل السطر الثاني وهكذا لكن أحيانا نحن بحاجة لكتابة عدة أسطر كتعليق فكيف ذلك إن C# توفر ذلك من خلال التعليقات الحرة وذلك بجعل التعليق بين رمزين من الشكل "*/" كما في الشكل التالي:

```
/* يقوم هذا السطر بإظهار صندوق رسائل مكتوب عليه رسالة ترحيب */
```

هناك نوع ثالث من التعليقات يسمى بتعليق التوثيق وهو يسمح بتوثيق شيفرتنا المصدرية وهو تعليق سطري يكون مسبوقا بالرمز "///" وتساعدنا تعليقات التوثيق في إنشاء ملف نصي ذو تنسيق خاص أثناء ترجمة المشروع.

هناك نقطة أخرى يجب أن نذكرها هنا وهي أن C# متحسسة لحالة الأحرف بعكس لغات البرمجة الأخرى أي اننا إذا كتبنا تعليمة ما بأحرف كبيرة وهي ممثلة في الاصل بأحرف صغيرة فإن ذلك سيتسبب بخطأ في ترجمة المشروع.

البنية الاساسية لتطبيق Console مكتوب بلغة C#:

Basic C# Console Application Structure:

لنلق نظرة على تطبيق Console الذي تناولناه في الفصل السابق ولنحاول تجزئة شيفرة هذا التطبيق إلى أجزاء محددة:

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

namespace Console_hello_world
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
            // لجعل الخلفية زرقاء
            Console.BackgroundColor = ConsoleColor.Blue;
            // لجعل لون النص أحمر
            Console.ForegroundColor = ConsoleColor.Red;
            // لإظهار نص معين على الشاشة السوداء
            Console.WriteLine("HELLO WORLD");
            // تمكين المستخدم من القراءة بعد تشغيل البرنامج
            Console.Read();
            Console.WriteLine("testing! 1,2,3");
            Console.ReadLine();
        }
    }
}
```

يمكنك أن تلاحظ أن جميع العناصر التي ناقشناها سابقا موجودة في هذه الشيفرة يمكننا أن نجد الفواصل المنقوطة بالإضافة إلى الاقواس {} والتعليقات لاحظ أيضا طريقة تنسيق الشيفرة وكيفية استخدام المسافة البادئة وفقا للكتل البرمجية.

المتغيرات (المتحولات) والثوابت:

Variables and Constants:

المتغيرات والثوابت هي عبارة عن أماكن لتخزين البيانات حيث لا يختلف مبرمجان اثنان على اهمية موضوع المتغيرات في أي لغة برمجة وإذا كان أساس إتقان اللغات الطبيعية هو تعلم حروف ومفردات تلك اللغة فان أساس إتقان لغات البرمجة هو تعلم المتغيرات والثوابت التي تبني بها إجراءات برامجك . نظريا لا تختلف فكرة المتغيرات في C# عن لغات البرمجة القديمة ولكنها تختلف اختلافاً جذرياً في بنيتها التحتية عما كانت عليه في السابق كما سترى لاحقا.

التصريح عن المتغيرات:

Variable Declaration:

إن صيغة التصريح عن المتغير في C# بسيطة للغاية حيث نحدد نوع المتغير أولا ثم نتبعه باسم المتغير كما في الشكل التالي:

```
< Type > < name > ;
```

هناك بعض الامور التي يجب أن نتبعها كي نتمكن من استخدام المتغيرات بصورة سليمة ودون حدوث أخطاء:

1- لا يمكن استخدام المتغير قبل التصريح عنه وإذا قمت بذلك فسوف يؤدي إلى حدوث خطأ في الترجمة.
2- لا يمكن استخدام المتغير قبل أن نسند له قيمة محددة وسيؤدي قيامك بذلك إلى حدوث خطأ في الترجمة أيضا.
3- أسماء المتغيرات متحسسة لحالة الاحرف وهذا يعني أن المتغير ذو الاسم hussam ليس هو نفسه المتغير HUSSAM.

والجدول التالي يبين أنواع البيانات التي يمكن تعريف المتغيرات بها في C#:

الاسم القصير	النوع او الفئة Class	الحجم	الوصف
	أرقام صحيحة		
Byte	System. Byte	8	قيمة عددية صحيحة 0 to 255
SByte	System.SByte	8	قيمة عددية صحيحة -128 to 127
int	System.Int32	32	قيمة عددية صحيحة -2,147,483,648 to 2,147,483,647
uint	System.UInt32	32	قيمة عددية صحيحة 0 to 4294967295
short	System.Int16	16	قيمة عددية صحيحة -32,768 to 32,767
ushort	System.UInt16	16	قيمة عددية صحيحة 0 to 65535
long	System.Int64	64	قيمة عددية صحيحة -9223372036854775808 to 9223372036854775807
ulong	System.UInt64	64	قيمة عددية صحيحة 0 to 18446744073709551615
	أنواع مختلفة		
Data	System.Data	64	التاريخ من 0001/1/1 إلى 9999/12/31
Char	System.Char	16	رمز (محرف) بنظام Unicode
bool	System.Boolean	8	قيمة منطقية (True or False)
Object	System.Object		جميع الانواع
String	System.String		سلسلة من المحارف بنظام Unicode
	أرقام الفاصلة العائمة		
float	System.Single	32	قيمة عددية تقبل فاصلة عائمة -3.402823e38 to 3.402823e38
Double	System.Double	64	قيمة عددية تقبل فاصلة عائمة - 1.79769313486232e308 to 1.79769313486232e308
Decimal	System.Decimal	128	قيمة عددية صحيحة أو تقبل فاصلة ±1.0 × 10e−28 to ±7.9 × 10e28

ملاحظة: يمكن كتابة الاسم القصير أو اسم النوع في شيفرة C# والنتيجة واحدة.

تسمية المتغيرات:

Variable Naming:

لا يمكن استخدام الكلمات المحجوزة (keywords) كأسماء للمتغيرات و هناك قاعدتين اساسيتين لتسمية المتغيرات و هما:

1- يجب أن يكون الرمز الاول من اسم المتحول حرفا هجائيا أو الرمز "_" أو الرمز "@" فقط ولا يمكن غير ذلك.
2- يمكن للرموز الاخرى اللاحقة بعد الرمز الاول أن تكون اي حرف هجائي أو رقم أو الرمز "_" فقط.

بالإضافة إلى ذلك فإن هناك كلمات محجوزة لها معان خاصة بمترجم C# مثل الكلمة using والكلمة namespace والتي لا يجب أن نستخدمها كأسماء للمتغيرات.

وعلينا أن نتذكر أن C# متحسسة لحالة الاحرف وبالتالي يجب ألا ننسى حالة الحروف التي استخدمناها عند التصريح عن المتغيرات و هذا يعني أن اسم المتحول الواحد المكتوب بحالات مختلفة من حالات الحرف اللاتيني تمثل أسماء متحولات مختلفة عن بعضها البعض.

قواعد التسمية:

Naming Conventions:

إن أسماء المتغيرات هي شيء سنستخدمه بكثرة عند برمجة التطبيقات وبسبب ذلك فإنه من المنطقي أن نتحدث عن المتغيرات وأساليب تسميتها بتفصيل أكبر.

إن أكثر أساليب تسمية المتغيرات المتبعة اليوم هي قواعد تسمية ليزنسكي (Leszynski Naming Conventions) وتسمى أيضا بالتدوين الهنغاري تعتمد هذه الطريقة على وضع بادئة (Prefix) لكل اسم متغير يشير إلى نوع بيانات المتغير على سبيل المثال إذا كان المتغير من نوع integer فإننا سنضع int في بداية اسم المتحول أو ربما وضع الحرف i فقط ووضع البادئة str قبل اسم المتغير من النوع string إن استخدام هذا الاسلوب في تسمية المتغيرات مفيد جدا ولكن ليس للغة برمجة حديثة مثل لغة C# والسبب في ذلك يعود إلى إمكانية تعريف عدد غير محدد من الانواع في C# والذي سيضطرنا إلى استخدام بادئات عديدة منها ما قد يكون عونا لنا ومنها ما قد يتسبب لنا ارباكا حقيقيا.

لقد أدرك المطورون أخيرا أن تسمية المتغيرات وفقا لمعنى البيانات المحفوظة ضمنها ولغرض هذه المتغيرات هو أفضل أسلوب لتسمية المتغيرات أما بالنسبة لمعرفة نوع هذا المتغير فإن Visual Studio يوفر لنا طريقة ذكية لذلك فبمجرد أن نضع مؤشر الفارة فوق اسم المتغير سيظهر تلميح على الشاشة يبين نوع المتغير.

في الحقيقة هناك طريقتان مستخدمتان لتسمية المتغيرات في فضاءات أسماء إطار عمل .NET. وتعرف هاتان القاعدتين بقاعدة PascalCase وقاعدة camelCase تستخدم هاتين القاعدتين بصورة خاصة لوصف محتوى المتحول من خلال اسمه حتى وإن تخلل هذا الوصف أكثر من كلمة واحدة إن الفارق الرئيسي بين هاتين القاعدتين هو أن القاعدة PascalCase يكون الحرف الاول منها كبيرا أما قاعدة camelCase فإن الحرف الأول منها يكون صغيرا.

إليك بعض أسماء المتغيرات وفقا لقاعدة camelCase:

age

firstName

وأسماء المتحولات التالية وفقا لقاعدة PascalCase:

Age

FirstName

سوف نستخدم قاعدة التسمية PascalCase لأغراض التسمية الأكثر تقدما وقاعدة التسمية camelCase للمتغيرات البسيطة وهذا ما تنصح به شركة Microsoft.

تطبيق حول التصريح عن المتغيرات والتعامل معها:

1- قم بإنشاء تطبيق Console جديد باسم Console Variables.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

```
{
    //التصريح عن المتغيرات
    int myInteger;
    string myString;
    // اسناد قيم للمتغيرات
    myInteger = 17;
    myString ="\"myinteger\" is";
    //  إظهار القيم
    Console.WriteLine("{0}....{1}.", myString, myInteger);
}
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 فيظهر الشكل (3-1).

الشكل (3-1)

كيفية العمل:

How it Works:

أن الشيفرة التي أضفناها تقوم بثلاثة أمور:

1- التصريح عن متغيرين.
2- اسناد القيم إلى هذين المتغيرين.
3- إخراج قيم هذين المتغيرين على نافذه Console.

حيث يصرح السطر الاول عن متغير باسم myInteger من نوع int والسطر الثاني يصرح عن متغير باسم myString من نوع String.

```
int myInteger;
string myString;
```

أما السطرين التاليين فمهمتهما اسناد القيم إلى المتغيرين المصرح عنهما.

```
myInteger = 17;
myString ="\"myinteger\" is";
```

لقد قمنا هنا بإسناد قيمتين حرفيتين (literal values) للمتغيرين باستخدام عامل الاسناد "=" لقد أسندنا القيمة 17 للمتغير myInteger وأسندنا النص "myinteger" is مع رمزي الاقتباس " " للمتحول myString ووفقا للشيفرة السابقة نلاحظ أنه كي نتمكن من إسناد أو استخدام نص أو سلسلة بشكل حرفي في شيفرتنا البرمجية يجب أن نضع السلسة النصية ضمن علامتي اقتباس " " هذا بالإضافة إلى أن علينا تجنب استخدام رموز محددة مثل علامة الاقتباس نفسها ضمن النص ولكي نتمكن من استخدام هذه الرموز يجب أن نستخدم التركيب البديل لها أو ما يسمى بتتابع الفرار(escape sequence) وذلك باستخدام التركيب \" كي نتمكن من كتابة رمز علامة الاقتباس " ضمن السلسلة النصية :

```
myString ="\"myinteger\" is";
```

والان إذا كتبنا السطر السابق دون استخدام التركيب البديل كما يلي:

```
myString =""myinteger" is";
```

سيؤدي ذلك إلى حدوث خطأ في الترجمة.

هناك قواعد أخرى تتعلق باستخدام السلاسل النصية بصورة حرفية عند كتابة الشيفرة من هذه القواعد أنه لا يمكن للسلسلة النصية أن تتجاوز سطر واحد وإذا أردنا كتابة سلسلة نصية على عدة أسطر علينا أن نستخدم التركيب البديل لرمز فاصل السطر (line break) وهو \n على سبيل المثال:

```
myString ="This string has no
line break";
```

سيؤدي هذا السطر إلى حدوث خطأ في الترجمة وبدلا من ذلك يجب كتابة:

```
myString ="This string has no \n line break";
```

والنتيجة ستكون كما في الشكل (2-3):

الشكل (2-3)

بالعودة إلى شيفرتنا السابقة فإن هناك سطر أخير لم نتحدث عنه حتى الآن و هو:

```
Console.WriteLine("{0}....{1}.", myString, myInteger);
```

هذا السطر هو لإخراج نص ما على نافذة Console إلا أننا نجد هنا استخداما للمتغيرات ضمن هذه التعليمة لإخراج النصوص على نافذة Console نلاحظ أن ما لدينا ضمن الأقواس هو ما يلي:

1- سلسلة نصية.
2- لائحة من المتغيرات التي نود أن نضيف قيمتها لسلسلة الخرج حيث يفصل بين كل متغير وأخر برمز الفاصلة "," إن جزء السلسلة النصية التي نود إخراجها هي "{0}....{1}." ونلاحظ أنها لا تحوي أي نص ذو معنى وكما رأينا أثناء التنفيذ فإن هذه السلسلة النصية ليست بالنص الذي ظهر على نافذة Console إن السبب في ذلك يعود إلى أن السلسلة النصية تلك تملك قالبا يمكننا من إضافة محتوى كل متغير إليها فكل تركيب من الشكل {i} حيث i تمثل عددا صحيحا ضمن القالب يقابل قيمة لمتغير ضمن لائحة المتغيرات التي تلي القالب والعدد الصحيح الموجود ضمن القوسين يمثل موقع المتغير من لائحة المتغيرات وعند إخراج النص على نافذة Console سيتم استبدال كل تركيب من الشكل {i} بقيمة المتغير الموجود في الموقع i على لائحة المتغيرات ترقم سلسلة المتغيرات بدأ من الصفر وبناء على ذلك فإن المتغير myString يمثل المتغير رقم صفر من لائحة المتغيرات والمتغير myInteger يمثل المتغير رقم 1 من لائحة المتغيرات أي أن {0} تشير إلى المتحول myString و {1} يشير إلى محتوى المتغير myInteger .

القيم الحرفية:

Literal Values:

لقد رأينا في المثال السابق مثالين لقيم حرفية رقم صحيح وسلسلة نصية ولأنواع البيانات الاخرى قيم حرفية موافقة لها أيضا وهي موضحة بالجدول التالي تستخدم معظم القيم الحرفية لهذه الأنواع لاحقات (suffixes) حيث نظيف سلسلة من الرموز لنهاية القيمة الحرفية وذلك لتحديد النوع المطلوب.

لبعض القيم الحرفية أنواع متعددة يتم تحديد نوعها بواسطة المترجم وبالاعتماد على السياق الذي وردت ضمنه هذه القيم:

نوع البيانات	الفئة	اللاحقة	مثال
Bool	قيم منطقية	يدون لاحقة	Trueأو false
int , uint , long , ulong	قيم صحيحة	بدون لاحقة	100
Uint , ulong	قيم صحيحة	Uأو u	100U
Long , ulong	قيم صحيحة	Lأو l	100L
Ulong	قيم صحيحة	ULأو Ul أو uL أو ul أو lu أو LU أو Lu أو lu	100UL
Float	قيم حقيقية	F أو f	1.5F
Double	قيم حقيقية	D أو d أو بدون لاحقة	1.5
Decimel	قيم حقيقية	M أو m	1.5M
Char	رمز(محرف)	بدون لاحقة	"a" أو تركيب بديل
string	سلسلة رمزية	بدون لاحقة	"Ujhbujnfjb" و يمكن أن تتضمن تراكيب بديلة

القيم الحرفية النصية:

String Literals:

لقد رأينا سابقا بعض التراكيب البديلة (escape sequences) التي يمكننا استخدامها في القيم الحرفية للسلاسل النصية والجدول التالي يجمع التراكيب البديلة:

التركيب البديل	الرمز الناتج (الأصلي)	شيفرة الرمز في نظام Unicode
\'	رمز اقتباس وحيد(')	0x0027
\"	رمز اقتباس مزدوج (")	0x0022
\\	الشرطة المائلة (\)	0x005c
\0	قيمة معدومة (Null)	0x0000
\a	رمز تحذير (Beep)	0x0007
\b	إلغاء للخلف (Back space)	0x0008
\f	رمز تلقيم النموذج (Form feed)	0x000C
\n	سطر جديد (Line Break)	0x000A

\r	رمز الإرجاع المركب (Carriage Return)	0x000D
\t	رمز الجدولة الأفقية (Horizontal Tab)	0x0009
\v	رمز الجدولة العمودية (Vertical Tab)	0x000B

أن عمود شيفرة الرمز في نظام Unicode يشير إلى القيم الست عشرية (Hexadecimal) للرمز كما هي موجودة في جدول رموز نظام تشفير Unicode ووفقا للجدول السابق يمكننا أن نستخدم شيفرة Unicode الست عشرية لتمثيل التراكيب البديلة للرموز الاصلية ولكي نستخدم هذه الطريقة يجب أن نضع التركيب "\u" متبوعا بقيمة معينة تحدد القيمة الست عشرية لشيفرة الرمز وفقا لنظام Unicode وهي الخانات الأربعة التي تلي الحرف x في الجدول السابق.

إن هذا يعني أن السلستين التاليتين متساويتان تماما:

```
"hussam\'s string"
"hussam\u0027s string"
```

إن استخدام التراكيب البديلة أمر لا بأس به حتى الآن ولكن ماذا لو صادفتنا سلسلة نصية تتضمن العديد من الرموز الموجودة في الجدول السابق لنأخذ على سبيل المثال السلسلة النصية التالية:

```
"C:\Temp\MyDir\MyFileName.doc"
```

تتضمن هذه السلسلة النصية مسار واسم الملف MyFileName.doc لكن تتضمن هذه السلسلة النصية عدد من الرموز لا يمكننا استخدامها بشكل مباشر هكذا وإنما علينا تحويل تلك الرموز إلى تراكيب مكافئة لها نلاحظ هناك الرمز "\" في ثلاثة مواضع ضمن السلسلة النصية وبالتالي فإن السلسلة النصية السابقة تكتب كما يلي (كي تصبح نظامية في C#):

```
"C:\\Temp\\MyDir\\MyFileName.doc"
```

إن هذا مربك للغاية ولحسن الحظ فإن هناك ما يعرف بالسلسلة النصية الحرة (Verbatim string) هذا يعني أن جميع الرموز الموجودة ضمن علامتي الاقتباس ستظهر كما في السلسلة النصية، حتى ولو تضمن على أي من الرموز المسرودة في الجدول السابق كما هي بصورتها الأصلية (الحرفية) ولإنشاء سلسلة نصية حرة يجب أن نضع الرمز @ قبل السلسلة النصية كما يلي:

```
@"C:\Temp\MyDir\MyFileName.doc"
```

إن الرمز الوحيد الذي لا يمكننا وضعه بحرية أي بصورته الحرفية هوة علامة الاقتباس المزدوجة "

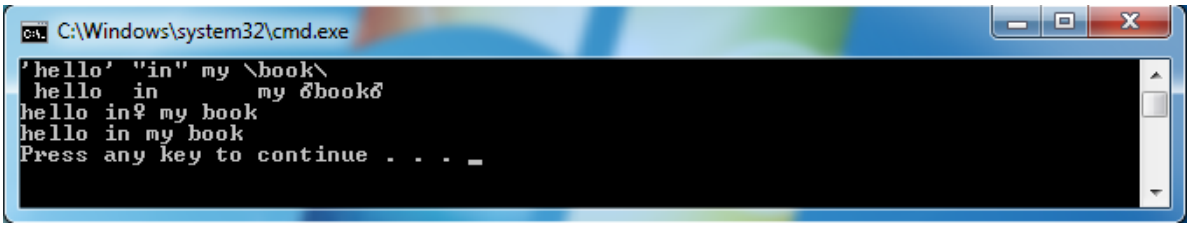
تطبيق حول استخدام القيم الحرفية النصية:

1- قم بإنشاء تطبيق Console جديد باسم Console String Literals.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

```
static void Main(string[] args)
{
    Console.WriteLine("\'hello\' \"in\" my \\book\\");
```

```
        Console.WriteLine("\0hello\0\tin\t my \vbook\v");
        Console.WriteLine("\ahello in\f my bb\book\a\nhello in my book");
    }
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 فيظهر الشكل (3-3).



الشكل (3-3)

كيفية العمل:

How it Works:

الأمور واضحة تماما سأترك لك التعليق.

التصريح عن المتغير وإسناد قيم له:

Variable Declaration and Assignment:

كما علمنا سابقا بأنه يتم التصريح عن المتغير بذكر نوع بياناته متبوعا باسم المتغير كما يلي:

```
Int age;
```

ومن ثم نسند قيمة لهذا المتغير باستخدام عامل الاسناد "=" كما يلي:

```
age=25;
```

هناك بعض الامور علينا أن نعتاد عليها أثناء البرمجة بلغة C# أولا التصريح عن عدة متغيرات من نفس النوع و الذي يمكننا القيام به ضمن سطر تصريح واحد بدلا من التصريح عن متغيرين أو أكثر في عدة أسطر حيث سيفصل بين كل متغير وأخر الرمز "," كما في المثال التالي:

```
Int x,y,z;
```

التقنية التالية التي سنعتاد عليها هي إسناد القيم للمتغيرات (ما تسمى بعملية التهيئة) في نفس سطر التصريح عنها وتتم هذه العملية كما يلي:

```
Int x=20,y=33,z=0;
```

التعابير:

Expressions:

تتضمن لغة C# عددا من العوامل (operators) التي تستخدم بغرض معالجة المتغيرات ومن بين هذه العوامل عامل الاسناد "=" الذي استخدمناه مسبقا نطلق على كل تشكيلة من هذه العوامل والمتغيرات والقيم الحرفية بالتعبير (expression) والذي يمثل الكتلة الاساسية لعمليات الحساب في البرمجة.

تتدرج العوامل المستخدمة في C# من العوامل البسيطة إلى العوامل المعقدة جدا والتي يمكن أن تتعرض لها في التطبيقات الرياضية تتضمن العوامل البسيطة جميع العمليات الرياضية متل الجمع والضرب والطرح والقسمة وصولا إلى العوامل المعقدة التي تتضمن معالجة محتوى المتغيرات عبر التمثيل البياني لقيمها هناك أيضا عوامل منطقية مخصصة للتعامل مع القيم المنطقية (البولينية) بالإضافة إلى عوامل الاسناد مثل العامل "=" سوف نركز في هذا الفصل على العوامل الرياضية وعوامل الاسناد وسنترك العوامل المنطقية للفصل التالي حيث سنتناول المنطق البولياني في سياق حديثنا عن التحكم في تدفق البرنامج.

يمكننا تصنيف العوامل ضمن ثلاث فئات:

1- العوامل الاحادية (unary) والتي تعالج حدا واحدا فقط.
2- العوامل الثنائية (binary) والتي تعالج حدين.
3- العوامل الثلاثية (ternary) والتي تعالج ثلاثة حدود.

تتدرج معظم العوامل تحت فئة العوامل ثنائية الحد وهناك القليل من العوامل أحادية الحد وليس في C# سوى عامل ثلاثي الحد وحيد يسمى بالعامل الشرطي.

العوامل الرياضية:

Mathematical Operators:

هناك خمس عوامل رياضية بسيطة اثنان منها يمكن أن يستخدم كعوامل ثنائية أو أحادية الجدول التالي يسرد العوامل الثنائية مع عرض مثال سريع لاستخدامها ونتائج هذه العوامل عند تطبيقها على متحولات من أنواع رقمية بسيطة.

العامل	الفئة	مثال	الناتج
+	ثنائي	var1=var2+var3	اسناد حاصل جمع قيمة متغيرين (var2 ,var3) إلى متغير var1
-	ثنائي	var1=var2-var3	اسناد حاصل طرح قيمة متغيرين (var2 ,var3) إلى متغير var1
*	ثنائي	var1=var2*var3	اسناد حاصل ضرب قيمة متغيرين (var2 ,var3) إلى متغير var1
/	ثنائي	var1=var2/var3	اسناد حاصل قسمة المتغير var2 على var3 إلى متغير var1
%	ثنائي	var1=var2%var3	اسناد حاصل باقي قسمة المتغير var2 على var3 إلى متغير var1

وكما قلنا فإن لبعض هذه العوامل هيئة أحادية و هي مسرودة بالجدول التالي:

العامل	الفئة	مثال	الناتج
+	أحادي	var1=+var2	جمع قيمة المتغير var2 إلى المتغير var1 واسناد القيمة إلى var1
-	أحادي	var1=-var2	طرح قيمة المتغير var2 من المتغير var1 واسناد القيمة إلى var1

من الواضح أن استخدام هذه العوامل مع المتغيرات ذات البيانات الرقمية منطقي جدا فجميع هذه العوامل تمثل عمليات رياضية أساسية يمكن تطبيقها على أي نوع من أنواع البيانات العددية لكن استخدام هذه العوامل مع متغيرات من أنواع مختلفة كالمتغيرات المنطقية أو النصية مبهم والنتائج المتوقعة غير واضحة

فما الذي تتوقعه عندما نقوم بجمع قيمتين منطقيتين مثلا:

في الحقيقة لا يمكننا استخدام العوامل الرياضية لجمع المتغيرات المنطقية واستخدامها يؤدي إلى حصول خطأ في الترجمة.

إن الأمر مربك أيضا بالنسبة للمتغيرات الرمزية من نوع (char) فهذه المتحولات تتضمن رمزا واحدا فقط لكن يتم الاحتفاظ بهذا الرمز على هيئة رقم صحيح يمثل شيفرة هذا الرقم في نظام تشفير Unicode وبالتالي إذا حاولنا جمع متحولين رمزيين فإن النتيجة هي رمز تمثل شيفرته حاصل جمع شيفرة المتحول الاول مع المتحول الثاني إن هذا مثال للتحول المطلق (implicit conversion) وسوف نتحدث عن هذا الموضوع بتفصيل أكبر لاحقا كما أن هناك التحويل الصريح (explicit conversion) و هو ما يحصل في التعابير التي تتضمن متحولات(متغيرات) من أنواع مختلفة.

لكن بالنسبة للمتغيرات النصية من نوع string فإن استخدام عامل الجمع (+) منطقي جدا:

العامل	الفئة	مثال	النتائج
+	ثنائي	var1=var2+var3	سيحوي المتحول var1 على السلسلة النصية للمتحول var2 و السلسلة النصية للمتحول var3

وليست هناك أية عوامل رياضية أخرى (عدا العامل +) يمكن استخدامها مع المتغيرات النصية.

هناك عاملان أخران مهمان جدا و هما عامل الزيادة والنقصان هذان العاملان أحاديان ويمكن استخدامهما بطريقتين إما بوضع العامل بعد الحد مباشرة أو قبله مباشرة والجدول التالي يوضح كيفية استخدام هذا العامل:

العامل	الفئة	مثال	النتائج
++	أحادي	var1=++var2	اسناد القيمة var2+1 إلى المتحول var1
--	أحادي	var1=--var2	اسناد القيمة var2-1 إلى المتحول var1
++	أحادي	var1=var2++	اسناد القيمة var2 إلى المتحول var1 ثم زيادة قيمة المتحول var2 بمقدار 1

--	أحادي	var1=var2--	اسناد القيمة var2 إلى المتحول var1 ثم أنقاص قيمة المتحول var2 بمقدار 1

إن الفكرة من هذين العاملين هي أن هناك دائما تغير في قيمة الحد فالعامل ++ سيؤدي دائما إلى زيادة الحد بمقدار 1 والعامل – سيؤدي دائما إلى إنقاص الحد بمقدار 1 إن قيمة المتغير var1 هنا متعلقة بأمرين اثنين:

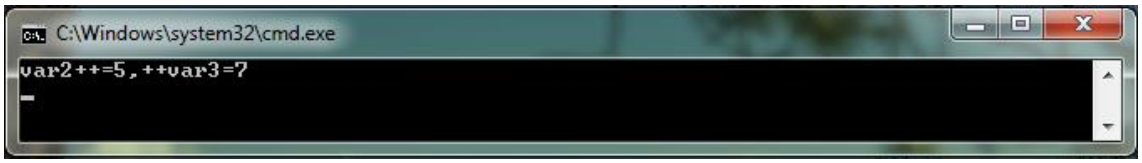
1- قيمة المتحول var2
2- موقع العامل بالنسبة للمتغير var2 فوضع العامل قبل الحد مباشرة يعني أن الحد سيتأثر قبل أي عمليات حسابية متضمنة في التعبير ووضعه بعد الحد مباشرة يعني أن الحد سيتأثر بعد اتمام جميع العمليات الحسابية في التعبير و هذا مهم جدا.

تطبيق حول العوامل الرياضية:

1- قم بإنشاء تطبيق Console جديد باسم Console Mathematical Operators.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

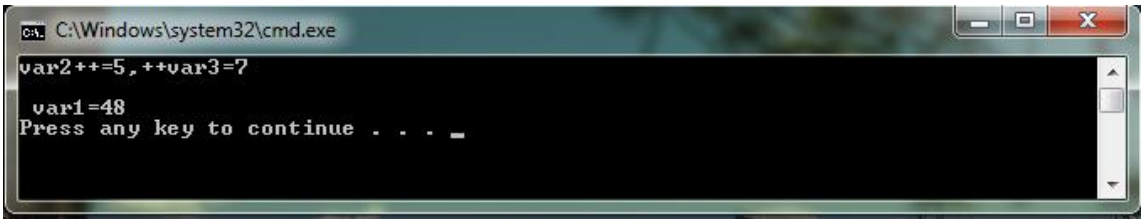
```
static void Main(string[] args)
{
    int var1, var2=5, var3=6;
    Console.WriteLine("var2++={0},++var3={1}", var2++, ++var3);
    Console.ReadLine();
    var1 = var2++ * ++var3;
    Console.WriteLine(" var1={0}", var1 );
}
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 فيظهر الشكل (3-4).



الشكل (3-4)

4- قم بالضغط على زر Enter سيظهر لك الشكل (3-5).



الشكل (3-5)

كيفية العمل:

How it Works:

إن الشيفرة التي أضفناها قامت بما يلي:

1- في السطر الاول التصريح عن ثلاث متغيرات من النوع int الأول لم نقم بإسناد له قيمة مباشرة أما الاخرين قمنا بإسناد قيم مباشرة لهما.
2- في السطر الثاني قمنا بإظهار قيم التحولات على نافذة Console بعد أن طبقنا عليها بعض العوامل الرياضية (var2++, ++var3)
3- في السطر الثالث تم اسناد أمر تمكين القراءة والانتقال للأمر التالي بعد الضغط على زر enter.
4- في السطر الرابع تم اسناد قيمة للمتحول (var1).
5- في السطر الخامس تم إظهار قيمة (var1) على شاشة Console

لكن كيف حدث ذلك أن الحد (var2++) يعمل عمل عداد تصاعدي ((var2--) هو عداد تنازلي) قيمته الاولى هي الاسناد الذي حددناه وهو الرقم خمسة فقيمته في الاظهار الأول كانت تساوي (5) أما في الاظهار الثاني تساوي(6).

أما الحد (var3++) فهو يعمل عمل عداد تصاعدي ((var2--) هو عداد تنازلي) أيضا إلا أن الفرق عن العداد السابق هو أن قيمته الأولى تزيد بمقدار (1) فكانت قيمة إظهاره الاول تساوي (7) وفي الاظهار الثاني تساوي (8)

ملاحظة: لا يمكننا استخدام القيم الحرفية مع العاملين ++ و – وهذا يعني أن التعبير التالي خاطئ:

Var1=++6

وسيؤدي إلى حدوث خطأ في الترجمة ولكي نحقق عملية كهذه يجب أن نضع القيمة الحرفية ضمن متغير كما فعلنا سابقا.

تطبيق أخر حول معالجة المتغيرات بالعوامل الرياضية:

1- قم بإنشاء تطبيق Console جديد باسم Console process variables.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

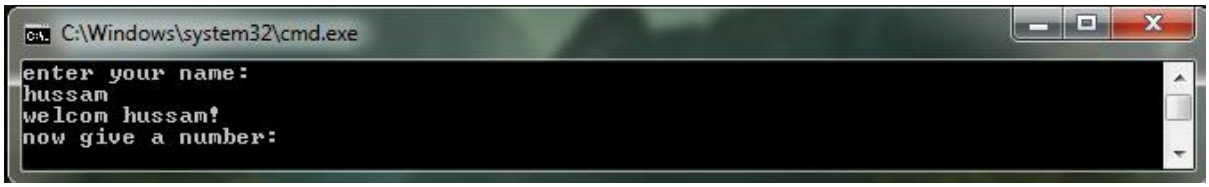
```
static void Main(string[] args)

        {
            Double firstNumber, secondNumber;
            string userName;
            Console.WriteLine("enter your name:");
            userName = Console.ReadLine();
            Console.WriteLine("welcom {0}!", userName);
            Console.WriteLine("now give a number:");
            firstNumber = Convert.ToDouble(Console.ReadLine());
            Console.WriteLine("now give me anather number:");
            secondNumber = Convert.ToDouble(Console.ReadLine());
            Console.WriteLine("the sum {0}and{1}is{2}.", firstNumber,
                secondNumber, firstNumber + secondNumber);
            Console.WriteLine("the result subtracting{0}from{1}is{2}.",
```

```
            firstNumber, secondNumber, firstNumber - secondNumber);
        Console.WriteLine("the product of {0} and {1} is {2}.",
            firstNumber, secondNumber, firstNumber * secondNumber);
        Console.WriteLine("the result dividing  {0} by {1} is {2}.",
            firstNumber, secondNumber, firstNumber / secondNumber);
        Console.WriteLine("the rrmainder of dividing {0} by {1} is {2}.",
            firstNumber, secondNumber, firstNumber % secondNumber);
    {
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 فيظهر الشكل (3-6).

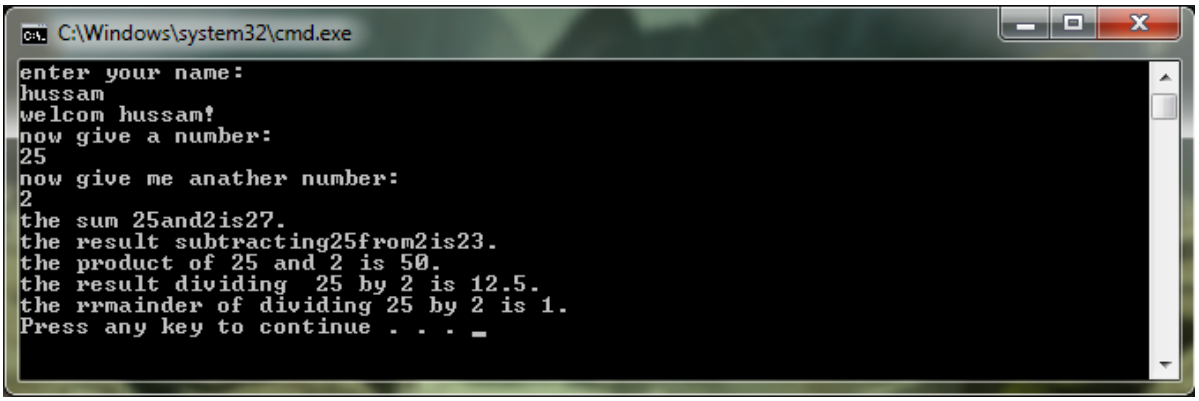
الشكل (3-6)

4- أدخل اسمك ثم اضغط Enter الشكل (3-7).



الشكل (3-7)

5- أدخل رقما ثم أضغط Enter ثم أدخل رقما أخر ثم اضغط Enter الشكل (3-8)



الشكل (3-8)

كيفية العمل:

How it Works:

بالإضافة للعوامل الرياضية فإن هذه الشيفرة تستعرض مفهومين مهمين سنستعرض لهما في العديد من الأمثلة اللاحقة:

1- دخل المستخدم.
2- تحويل النوع.

يظهر أمر الحصول على دخل المستخدم (وهو Console.Readline) مشابه لصيغة الأمر Console.WriteLine() المستخدم لإخراج النصوص على نافذة Console يقوم هذا الأمر بحث المستخدم على دخل ما و هو ما سيخزن ضمن متحول من نوع string :

```
string userName;
Console.WriteLine("enter your name:");
userName = Console.ReadLine();
Console.WriteLine("welcom {0}!", userName);
```

تقوم هذه الشيفرة بكتابة محتوى المتحول userName على نافذة الخرج.

لقد قمنا بقراءة عددين في هذا المثال حيث استخدمنا الامر Console.ReadLine() مرتين للحصول على دخل من المستخدم والذي يمثل هذين العددين لقد ذكرنا أن القيمة الناتجة من الامر Console.ReadLine() هي سلسلة نصية من نوع String تمثل الدخل الذي تم إدخاله على نافذة Console أي أن أي نوع من البيانات نقوم بإدخاله لا يمثل إلا نصا وحتى إن كان الدخل عبارة عن أرقام فقط ولهذا السبب توجب علينا استخدام مفهوم تحويل نوع البيانات (Type Conversion) وسوف نتناول موضوع تحويل الانواع لاحقا

لنعد إلى شيفرة التطبيق من البداية أولا قمن بالتصريح عن متحولين رقميين من نوع Double :

```
Double firstNumber, secondNumber;
```

بعد ذلك طلبنا من المستخدم إدخال العدد الأول ثم أستخدمنا الأمر Convert.ToDouble على سلسلة الدخل الناتجة من الأمر Console.ReadLine() وذلك لتحويل السلسلة النصية ذات النوع String إلى بيانات رقمية من نوع Double بعد ذلك قمنا بإسناد القيمة الناتجة إلى المتحول firstNumber:

```
Console.WriteLine("now give a number:");
firstNumber = Convert.ToDouble(Console.ReadLine());
```

إن عملية التحويل تلك بسيطة للغاية وستجد لاحقا أن العديد من عمليات التحويل الأخرى تؤدي بنفس الطريقة والسهولة والامر مشابه أيضا للحصول على العدد التالي:

```
Console.WriteLine("now give me anather number:");
secondNumber = Convert.ToDouble(Console.ReadLine());
```

بعد ذلك سنطبع ناتج جمع وطرح وضرب وقسمة وباقي قسمة هذين العددين على نافذة الخرج:

```
Console.WriteLine("the sum {0}and{1}is{2}.", firstNumber,
    secondNumber, firstNumber + secondNumber);
Console.WriteLine("the result subtracting{0}from{1}is{2}.",
    firstNumber, secondNumber, firstNumber - secondNumber);
Console.WriteLine("the product of {0} and {1} is {2}.",
    firstNumber, secondNumber, firstNumber * secondNumber);
Console.WriteLine("the result dividing  {0} by {1} is {2}.",
```

```
        firstNumber, secondNumber, firstNumber / secondNumber);
    Console.WriteLine("the rrmainder of dividing {0} by {1} is {2}.",
        firstNumber, secondNumber, firstNumber % secondNumber);
```

لاحظ أننا استخدمنا التعبير firstNumber + secondNumber كبارامتر للأمر Console.WriteLine() أي أننا لم نخزن ناتج الجمع ضمن متحول واستخدمنا هذا المتحول لإخراج قيمة الجمع:

```
Console.WriteLine("the sum {0}and{1}is{2}.", firstNumber,
    secondNumber, firstNumber + secondNumber);
```

إن هذا النوع من الصيغ يجعل شيفرتنا البرمجية مقروءة ويزيل كمية الأسطر البرمجية الزائدة التي قد تحتاج لكتابتها.

عوامل الاسناد:

Assignment Operators:

لم نستخدم حتى الان سوى عامل اسناد بسيط وهو (=) إلا أن هناك العديد من عوامل الاسناد الاخرى المفيدة جدا في الكثير من الحالات.

إن جميع عوامل الاسناد عدا العامل "=" تعمل بصورة مشابهة كما أن لجميع عوامل الاسناد مبدأ واحدا وهو أن قيمة المتحول على يمين العامل ستسند إلى المتحول على يساره.

الجدول التالي يبين عوامل الاسناد بالإضافة إلى طريقة استخدامها:

العامل	الفئة	مثال	الناتج
=	ثنائي	Var1=var2	اسناد قيمة var2 إلى var1
+=	ثنائي	Var1+=var2	اسناد حاصل جمع var1 مع var2 إلى المتغير var1
-=	ثنائي	Var1-=var2	اسناد حاصل طرح var2 من var1 إلى المتغير var1
*=	ثنائي	Var1*=var2	اسناد حاصل ضرب var1 مع var2 إلى المتغير var1
/=	ثنائي	Var1/=var2	اسناد حاصل قسمة var1 على var2 إلى المتغير var1
%=	ثنائي	Var1%=var2	اسناد حاصل باقي قسمة var1 على var2 إلى المتغير var1

وكما ترى فإن هذه العوامل لا تمثل إلا اختصارا للتعابير التي تستخدم عامل الاسناد الطبيعي "=" مع أحد العوامل الرياضية التي تعرفنا عليها منذ قليل فالتعبير التالي:

var1=var1+var2

مكافئ تماما للتعبير:

var1+=var2

وقس على ذلك على باقي عوامل الاسناد الاخرى الموجودة في الجدول.

ملاحظ إن العامل (=+) هو الوحيد الذي يمكن استخدامه مع المتغيرات النصية من نوع String أما باقي العوامل فلا يمكن استخدامها إلا مع المتغيرات الرقمية فقط والمتحولات الرمزية من نوع char.

أسبقية العوامل:

Operator Precedence:

عند تنفيذ تعبير ما فإن كل عامل سيطبق على حدوده وفق تتابع محدد وهذا لا يعني بالضرورة معالجة العوامل من اليسار إلى اليمين.

بالعودة إلى مثال سابق لنأخذ الشيفرة التالية:

```
var1=var2+var3;
```

هنا سيعالج العامل (+) قبل العامل (=).

هناك أوضاع أخرى تتجلى فيها أسبقية بعض العوامل على غيرها كما في المثال التالي:

```
var1=var2+var3*var4;
```

هنا سيعالج العامل (*) أولا ثم العامل (+) ومن ثم العامل (=) لاحظ أن هذا هو الترتيب الرياضي الطبيعي للحساب والنتيجة التي سنحصل عليها مطابقة تماما لما يمكن أن نحصل عليه عند احتساب تعبير كهذا بالورقة والقلم.

وكما في الرياضيات فإنه يمكننا التحكم في أسبقية العوامل باستخدام الأقواس على سبيل المثال:

```
var1= (var2+var3)*var4;
```

هنا ستتم أولا تنفيذ العامل (+) ومن ثم سيعالج العامل (*) ومن ثم العامل (=).

الجدول التالي يبين أسبقية العوامل التي تناولناها حتى الان حيث العوامل الموجودة في سطر واحد مثل (*و/) ستنفذ وفقا اليسار إلى اليمين:

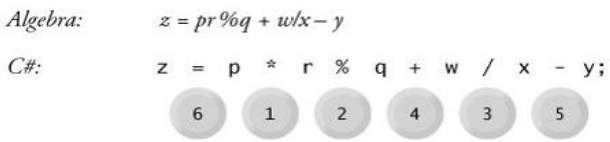
الأسبقية الأعلى	**1- ++ , -- كبادئات , +, - (الأحادية).** **2- * , / , %.** **3- + , -.** **4- = , *= , /= , %= , += , -= .**
الأسبقية الادنى	++ , -- كلواحق.

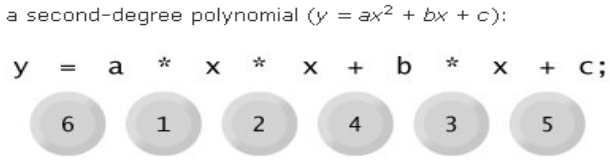
تذكر ان استخدام الاقواس سيلغي هذا الترتيب.

وبناء على ذلك نخلص للخلاصة التالية بما يخص أسبقية العوامل الرياضية حيث سيتعامل معها منا يلي:

1- ما بداخل الأقواس أولاً وإذا كانت الأقواس متداخلة فتبدأ العملية من أقصى قوس في الداخل بمعنى لو أراد المبرمج إتمام عملية حسابية الأول (حتى لو كان الطرح مع أنه أخر عملية تتم) يضعها داخل الأقواس يتم تنفيذها أول شيء.
2- (الضرب – القسمة – باقي القسمة) لهم نفس الترتيب بمعنى لو وجدت الأقواس في العملية الرياضية يتم حساب ما بالأقواس الأول وإذا وجدت إحدى العمليات الثلاثة السابقة (واحدة منهم فقط) يتم تنفيذها بعد الأقواس إن وجدوا الثلاثة أو أثنين منهم تتم العملية الحسابية من اليسار إلى اليمين بمعنى اللي موجود الأول من ناحية اليسار يتم حسابه الأول كما بالشكل التالي.
3- الجمع.
4- الطرح.

والأشكال توضح أسبقية العوامل الرياضية:





فضاء الأسماء:

Namespaces:

تمثل فضاءات الاسماء (namespaces) الاسلوب الذي يتبعه إطار عمل .NET. لتوفير حاويات لشيفرة التطبيق وتستخدم فضاءات الأسماء أيضا لأهداف تصنيف العناصر في إطار عمل .NET. إن معظم هذه العناصر عبارة عن تعاريف لأنواع البيانات مثل الانواع البسيطة المشروحة في هذا الفصل.

إن شيفرة C# موجودة بصورة افتراضية ضمن فضاء الاسماء الشامل (Global namespaces) هذا يعني أن العناصر الموجودة ضمن هذه الشيفرة يمكن الوصول إليها من خلال شيفرة أخرى موجودة في فضاء الاسماء الشامل بمجرد الاشارة إليها بالاسم فقط ومع ذلك يمكننا استخدام الكلمة المحجوزة namespace لتعريف فضاء أسماء بصورة صريحة وذلك لكتلة معينة من الشيفرة حيث ستتوضع هذه الشيفرة ضمن قوسي كتلة "{}" و بناء على ذلك يجب ذكر اسم فضاء الاسماء الذي تنتمي له كتلة الشيفرة كي نتمكن من استخدام العناصر الموجودة ضمن هذه الكتلة خارج فضاء الأسماء هذا.

إن اسم فضاء الاسماء يمثل اسما وصفيا محددا يتضمن تحته كافة المعلومات الهرمية لكتلة الشيفرة التي يحتويها هذا يعني أنه إذا كان لدينا شيفرة في فضاء أسماء تحتاج لاستخدام اسم معرف في فضاء أسماء أخر فإن علينا تضمين مرجع لفضاء الاسماء الأول تستخدم الاسماء المحددة (Qualified Names) النقطة "." بين مستويات الاسماء الهرمية.

لنأخذ الشيفرة التالية على سبيل المثال:

```
namespace LevelOne
{
//code in LevelOne namespace
//name "NameOne" defined
}
//code in Global namespace
```

تعرف هذه الشيفرة فضاء أسماء باسم LevelOne وكذلك تعريف اسم ضمن فضاء الاسماء هذا و هو NameOne لاحظ أننا لم نقم فعليا بتعريف الاسم لكي يكون حديثنا موجها بصورة عامة وسنفترض هنا أن هناك شيفرة لتعريف الاسم NameOne.

عندما نود استخدام الاسم NameOne ضمن فضاء الاسماء LevelOne فإننا سنستخدمه كما هو (أي NameOne) دون الحاجة لذكر فضاء الاسماء الذي ينتمي له.

أما بالنسبة للشيفرة الموجودة ضمن فضاء الاسماء الشامل فإن علينا هنا أن نشير إلى فضاء الاسماء LevelOne ثم الاسم NameOne بالصورة التالية (و هو يمثل الاسم المحدد للاسم NameOne).

LevelOne. NameOne

يمكننا تعشيش فضاءات الاسماء ضمن بعضها البعض وذلك باستخدام الكلمة المحجوزة namespace أيضا ولكي نتمكن من الوصول إلى عناصر كل فضاء أسماء علينا ذكر تسمية فضاءات وفقا لهرمية التعشيش وذلك بوضع رمز النقطة "." بين كل مستوى من مستويات فضاءات الاسماء لنأخذ فضاءات الاسماء التالية:

```
namespace LevelOne
{
//code in LevelOne namespace
  namespace LevelTow
    {
      //code in LevelOne. LevelTow namespace
      // name "NameTow" defined
    }
 }
//code in Global namespace
```

و هنا بفرض أننا نود الوصول إلى الاسم NameTow من ضمن فضاء الاسماء الشامل عندئذ يجب أن نشير إلى الاسم وفقا لما يلي:

LevelOne.LevelTow. NameTow

وإذا كنا ضمن فضاء الاسماء LevelOne عندئذ يجب أن نشير إلى الاسم وفقا لما يلي:

LevelTow. NameTow

وإذا كنا ضمن فضاء الاسماء LevelTow عندئذ يكفي أن نشير إلى الاسم فقط كما يلي:

NameTow

النقطة الهامة التي يجب أن نشير إليها هنا هو أن الاسماء يجب أن تكون فريدة ضمن فضاء الاسماء التابعة له فلا يمكننا تعريف الاسم NameThree مرتين ضمن فضاء الاسماء الواحد ولكن يمكننا تعريف الاسم NameThree ضمن فضاء الأسماء LevelTow وتعريف اسم NameThree ضمن فضاء الأسماء LevelOne أيضا:

```
namespace LevelOne
{

// name "NameThree" defined
  namespace LevelTow
    {
      // name "NameThree" defined
    }
 }
//code in Global namespace
```

لاحظ أن الاسمين LevelOne.NameThree و LevelTow.NameThree .LevelOne يمثلان اسمين منفصلين وسيستخدمان بصورة مستقلة عن بعضهما البعض.

بالعودة إلى الشيفرة السابقة نلاحظ أننا كي نتمكن من الوصول إلى الاسماء الموجودة ضمن فضاء الاسماء LevelTow من ضمن فضاء الاسماء الشامل علينا أن نشير إلى اسم فضاء الاسماء LevelOne ثم اسم فضاء الاسماء LevelTow ومن ثم الاسم المعرف ضمن فضاء الاسماء LevelTow على الرغم من أن هذا منطقي جدا إلا انه مربك في كثير من الاحيان (تصور أن يكون لدينا أربعة أو خمسة مستويات من تعشيش فضاءات الاسماء تخيل كيف ستشير إلى اسم معرف ضمن فضاء الاسماء في المستوى الخامس!) لهذا السبب وجدت في C# الكلمة المفتاحية using بواسطة هذه الكلمة لن يطلب منك المترجم ذكر الاسم المحدد كاملا على سبيل المثال لقد ذكرنا في الشيفرة السابقة أن الشيفرة الموجودة ضمن فضاء الاسماء LevelOne يجب أن تتمكن من الوصول إلى الاسماء الموجودة ضمن فضاء الاسماء LevelOne. LevelTow دون ذكر الاسم المحدد كاملا:

```
namespace LevelOne
{

Using LevelTow;

  namespace LevelTow
    {
      // name "NameTow" defined
    }
 }
//code in Global namespace
```

والان يمكن للشيفرة ضمن فضاء الاسماء LevelOne الوصول إلى الاسم LevelTow. NameTow بمجرد ذكر NameTow فقط.

في بعض الحالات قد يسبب ذلك بعض الارباك خصوصا إذا كان لدينا أسماء متشابهة في فضائي أسماء مختلفين (كالاسم NameThree في المثال السابق) إن استخدام الكلمة المحجوزة using في حالة كهذه

سيقود إلى مشاكل تضارب بين الاسمين عندئذ يمكننا توفير اسم مستعار (alias) لفضاء الأسماء كجزء من التعليمة using:

```
namespace LevelOne
{

Using LT=LevelTow;

// name "NameThree" defined

  namespace LevelTow
    {
      // name "NameThree" defined
    }
 }
//code in Global namespace
```

عندئذ يمكننا الوصول إلى الاسم NameThree (الذي ينتمي إلى فضاء الأسماء LevelTow) في فضاء الأسماء الشامل أو فضاء أسماء LevelOne بالاسم المحدد LT.NameThree.

من هنا نلاحظ أن الكلمة المحجوزة using تساعدنا في اختصار كتابة الاسماء التي تنتمي إلى فضاءات أسماء أخرى غير فضاء الاسماء الحالي.

بالعودة إلى شيفرة أحد المشاريع السابقة وليكن ما قبل أخر مشروع نلاحظ أن هناك استخدام لفضاءات الاسماء:

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

namespace Console_process_variables
{
    class Program
    {
        static void Main(string[] args)
        {
         ..........
        }
    }
}
```

يستخدم السطر الاول الكلمة المحجوزة using للتصريح عن فضاء الاسماء System أي أنه سيستخدم في الشيفرة هنا وبالتالي يمكن لأي فضاء أسماء في هذا الملف الوصول إلى فضاء الاسماء System (بما يتضمنه من أسماء) دون الحاجة للإشارة إلى فضاء الأسماء System بشكل صريح.

في الحقيقة يمثل فضاء الاسماء System فضاء الأسماء الجذر لتطبيق NET. ويتضمن جميع الوظائف الأساسية التي نحتاجها لبرمجة تطبيقات Console.

وبالمثل أيضا بالنسبة لباقي الاسطر التي تستخدم الكلمة المحجوزة using.

الخلاصة:

Summary:

لقد تناولنا في هذا الفصل كما لا باس به من المعلومات التي ستعتمد عليها بصورة كلية في تصميم تطبيقاتك الاساسية بلغة C# لقد تناولنا الصيغة الاساسية لكتابة شيفرة C# وقمنا بتحليل شيفرة تطبيقات Console البسيطة.

كما ركزنا في هذا الفصل على شرح المتغيرات وتعرفنا على أنواع البيانات ومن ثم انتقلنا للتعرف على كيفية معالجة المتغيرات وتعرفنا على انواع العوامل الرياضية وكيفية التعامل مع القيم الحرفية النصية كما تعرفنا على كيفية التصريح عن فضاءات الأسماء وكيف يتم استدعاء اسم من فضاء أسماء معشش ضمن فضاء أسماء أخر...

الفصل الرابع

التحكم في سير البرنامج

هناك شيء مشترك في جميع الأمثلة البرمجية التي تناولناها حتى الان لا بد وأنك قد لاحظته وأن تنفيذ البرنامج ينتقل من سطر إلى السطر الذي يليه مباشرة من بداية الشيفرة إلى نهايتها دون تجنب أي سطر إن اقتصرت التطبيقات على هذا النهج من التنفيذ فعندئذ سنكون محدودين جدا بما يمكننا فعله أثناء تصميم التطبيقات.

سوف نتناول في هذا الفصل طريقتين للتحكم في سير البرنامج (أي التحكم في ترتيب تنفيذ الاسطر البرمجية) هاتين الطريقتين هما:

1- التفرع (branching) حيث ستنفذ الشيفرة التي تحقق شرطا معينا بالاعتماد على نتيجة تعبير ما.
2- الحلقات(looping) أي تكرار تنفيذ الشيفرة نفسها (لعدد محدد من المرات أو إلى أن يتحقق شرط ما).

إن كلا الطريقتين تتطلبان استخدام للمنطق البولياني لقد تعرفنا في الفصل السابق على نوع المعطيات البولياني إلا أننا لم نخض فيه كثيرا لكن سوف نستخدم المتحولات المنطقية (Bool) بشكل مكثف في هذا الفصل وسوف نبدأ هذا الفصل بمناقشة المقصود بالمنطق البولياني بحيث يفتح لنا ذلك الباب للحديث عن سيناريوهات التحكم في سير البرنامج.

المنطق البولياني:

Boolean Logic:

لا يمكن للمتحولات من نوع Bool أن تحتوي إلا على واحدة من إحدى القيمتين (True،False) يستخدم هذا النوع لتسجيل نتيجة عملية ما وبصورة عامة يستخدم نوع البيانات Bool لحفظ نتيجة مقارنة (comparison) ما.

تتطلب المقارنات المنطقية استخدام عوامل المقارنة البوليانية وتسمى أيضا بالعوامل العلائقية (relational operators) الجدول التالي يسرد هذه العوامل حيث أن المتحول var1 من نوع Bool أما المتحولين vre2 وvar3 فهما من أي نوع كان:

العامل	الفئة	مثال	الناتج
==	ثنائي	var1=var2==var3	سيأخذ المتحول var1 القيمة True إذا كانت قيمة المتحول var2 تساوي قيمة المتحول var3 وإلا فسيأخذ القيمة False
!=	ثنائي	var1=var2!=var3	سيأخذ المتحول var1 القيمة True إذا كانت قيمة المتحول var2 لا تساوي قيمة المتحول var3 وإلا فسيأخذ القيمة False
<	ثنائي	var1=var2<var3	سيأخذ المتحول var1 القيمة True إذا كانت قيمة المتحول var2 أصغر من قيمة المتحول var3 وإلا فسيأخذ القيمة False
>	ثنائي	var1=var2>var3	سيأخذ المتحول var1 القيمة True إذا كانت قيمة المتحول var2 أكبر من قيمة المتحول var3 وإلا فسيأخذ القيمة False
<=	ثنائي	var1=var2<=var3	سيأخذ المتحول var1 القيمة True إذا كانت قيمة المتحول var2 أصغر أو تساوي قيمة المتحول var3 وإلا فسيأخذ القيمة False
>=	ثنائي	var1=var2>=var3	سيأخذ المتحول var1 القيمة True إذا كانت قيمة المتحول var2 أكبر أو تساوي قيمة المتحول var3 وإلا فسيأخذ القيمة False

ويمكننا أن نطبق العوامل المنطقية على المتغيرات أو على قيم حرفية مباشرة كما يلي:

```
bool isLessThan10;
isLessThan10 = myVal < 10;
```

سيأخذ المتحول isLessThan10 القيمة True إذا كانت قيمة المتحول myVal أقل من 10 وإذا كانت قيمة المتحول myVal مساوية لـ 10 أو أكبر منها فإن المتحول isLessThan10 سيأخذ القيمة False.

كما يمكننا استخدام عوامل المقارنة على أنواع أخرى غير الأنواع الرقمية كمقارنة السلاسل النصية كما يلي:

```
bool isHussam;
isHussam = myString =="Hussam";
```

هنا سيأخذ المتحول isHussam القيمة True إذا كانت السلسلة النصية في المتغير myString هي "Hussam" دون علامتي الاقتباس طبعا.

ويمكننا أن نقارن القيم المنطقية أيضا لتعطينا قيم منطقية أيضا:

```
bool isTrue;
isTrue = myBool ==true;
```

هنا سيأخذ المتحول isTrue القيمة true إذا كانت قيمة المتحول myBool هي true أيضا وإلا فسيأخذ القيمة false.

هناك عوامل منطقية أخرى خاصة بالمقارنات بين القيم والمتحولات المنطقية فقط وهي مسرودة في الجدول التالي:

العامل	الفئة	مثال	الناتج
!	أحادي	var1=! var2	سيأخذ المتحول var1 القيمة true إذا كانت قيمة المتحول var2 هي false وسيأخذ القيمة false إذا كانت قيمة var2 هي true تسمى هذه العملية عملية النفي المنطقية NOT
& أو &&	ثنائي	var1=var2 & var3 أو var1=var2 && var3	سيأخذ المتحول var1 القيمة true إذا كانت قيمة كلا المتحولين var2 وvar3 هي true وسيأخذ القيمة false إذا كانت قيمة أحد المتحولين هي false تسمى هذه العملية عملية AND المنطقية
| أو ||	ثنائي	var1=var2 | var3 أو var1=var2 || var3	سيأخذ المتحول var1 القيمة true إذا كانت قيمة أحد المتحولين var2 أوvar3 على الاقل هي true وسيأخذ القيمة false إذا كانت قيمة كلا المتحولين هي false تسمى هذه العملية عملية OR المنطقية
^	ثنائي	var1=var2 ^ var3	سيأخذ المتحول var1 القيمة true إذا كانت قيمة كلا المتحولين var2 وvar3 غير متساوية أي أحدهما true و الاخر false وسيأخذ القيمة false إذا كانت قيمة المتحولين متساوية تسمى هذه العملية عملية XOR المنطقية

وبناء على ذلك فإن الشيفرة التالية مطابقة للشيفرة السابقة:

```
bool isTrue;
isTrue = myBool & true;
```

لاحظ أن للعامل & شكل أخر هو && إن نتيجة هذين العاملين (أي & و &&) هي نفسها إلا أن هناك اختلاف بين طريقة معالجة العامل الذي يقود على تحسين في الأداء يقوم && بقراءة قيمة الحد الأول أولا (وهو المتحول var2 في المثال في الجدول) وحسب قيمة الحد الأول سيقرر ما إذا كان من الضروري قراءة الحد الثاني أم لا.

فإذا كانت قيمة الحد الاول للعامل && هي false فإنه من غير الضروري أن نقرأ قيمة الحد الثاني وذلك لأنه مهما تكن قيمة الحد الثاني فإن نتيجة العامل هي false. أن الامر مشابه للعامل || أيضا فإذا كانت قيمة أحد الحدين هي true فمن غير الضروري قراءة الحد الثاني لأنه مهما تكن قيمته فإن نتيجة العامل هي true عندئذ.

إن الامر مختلف بالنسبة للعاملين & و | لأن هذين العاملين سيقرآن قيمة الحد الأول والثاني مهما كانت قيمة أحدهما.

سنجد مع المقارنات الكثيرة أن هناك تحسنا بسيطا في الاداء عند استخدام العاملين && و || بدلا من & و | تباعا وبصورة عامة فإنني انصح باستخدام العاملين && و || متى كان ذلك ممكنا.

أمثلة:

مثال على && (And):

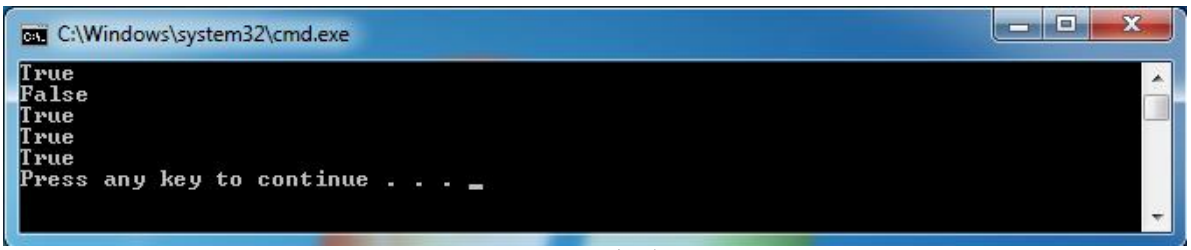
```
Console.WriteLine(5 < 6 && 8 < 9); // true
Console.WriteLine(false && false); // false
Console.WriteLine(false && true);   // false
Console.WriteLine(true && false);  //false
Console.WriteLine(true && true);    // true
```

الشكل (4-1)

الحالة الوحيدة يكون الناتج فيه true أن يكون كلا الشرطين true

مثال على || (Or):

```
Console.WriteLine(5 < 6 || 8 < 9); // true
Console.WriteLine(false || false); // false
Console.WriteLine(false || true);   // true
Console.WriteLine(true || false);  // true
Console.WriteLine(true || true);    // true
```



الشكل (4-2)

الحالة الوحيدة يكون فيها الناتج false أن يكون كلا الشرطين false

مثال على! (Not):

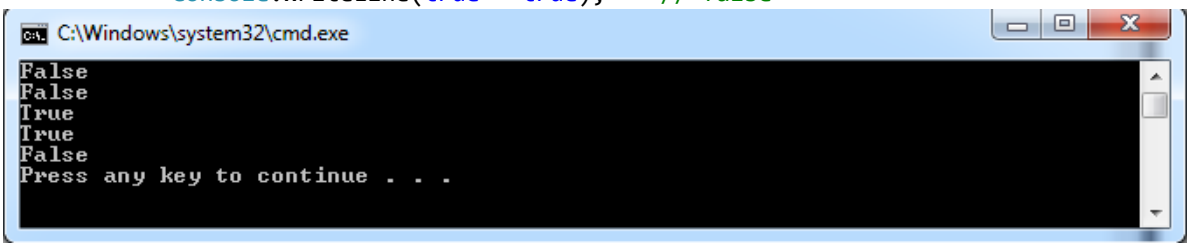
```
Console.WriteLine(!true);  // false
Console.WriteLine(!false); // true
```

الشكل (4-3)

يستخدم هذا المعامل لنفى الحقيقة.

مثال على ^ (XOR):

```
Console.WriteLine(5 < 6 ^ 8 < 9); // false
Console.WriteLine(false ^ false); // false
Console.WriteLine(false ^ true);   // true
Console.WriteLine(true ^ false);  // true
Console.WriteLine(true ^ true);    // false
```



الشكل(4-4)

ملاحظة: الكتابة باللون الأخضر هي الخرج كتبتها هكذا كي أذكرك بأننا يمكن أن نستخدم التعليقات ضمن سطر التعليمات أيضا.

العوامل الخاصة بالبتات:

Bitwise operators:

لقد لاحظنا أن استخدام العوامل المنطقية مع المتحولات المنطقية (البولیانية) أمر منطقي جدا في الحقيقة إن وجود عوامل مثل & و | لم يقتصر على المتحولات البولیانية وإنما على المتحولات والقيم العددية أيضا.

فالأرقام تخزن في الحاسوب على هيئة سلسلة من الاصفار والوحدات فعلى سبيل المثال يمكننا تمثيل الرقم 5 بثلاث خانات ثنائية :101 وذلك وفقا لقاعدة محددة تعرف بنظام العد الثنائي حيث تسمى كل خانة من خانات العدد الثنائي بالبت (Bit) وبالتالي فإننا عندما نستخدم هذه العوامل مع القيم الرقمية فإن هذا يعني أننا نقوم بتطبيق هذه العوامل على بتات القيمة بتمثيلها الثنائي.

لنأخذ العامل & سيتم مقارنة كل بت في الحد الأول مع البت المقابل له في الحد الثاني ونتيجة العامل هي مجموعة من البتات التي تمثل قيمة الخرج.

الجدول التالي يبين نتيجة تنفيذ العامل & على البتات:

البت في الحد الاول	البت في الحد الثاني	البت الناتج من &
1	1	1
1	0	0
0	1	0
0	0	0

والجدول التالي يبين نتيجة تنفيذ العامل | على البتات:

البت في الحد الاول	البت في الحد الثاني	البت الناتج من عملية |
1	1	1
1	0	1
0	1	1
0	0	0

والجدول التالي يبين نتيجة تنفيذ العامل ^ على البتات:

البت في الحد الاول	البت في الحد الثاني	البت الناتج من عملية ^
1	1	0
1	0	1
0	1	1
0	0	0

هناك عامل احادي أخر(~) يسمى بعامل NOT يستخدم على البتات يماثل ما يقوم به العامل الاحادي! فما هو إلا مقلوب العدد بنظام العد الثنائي كما يبين ذلك الجدول التالي:

بت الحد	البت الناتج عن العملية المنطقية ~
1	0
0	1

لا حظ أنه يمكننا اعتبار القيمة true تمثل البت ذو القيمة 1 والقيمة false تمثل البت ذو القيمة 0.

على سبيل المثال لنأخذ الشيفرة التالية:

```
int result ,op1 ,op2;
op1=4;
op2=5;
result = op1 & op2;
```

علينا هنا أن نأخذ التمثيل الثنائي للمتحولين op1 وop2 بعين الاعتبار فقيمة المتحول op1 بالتمثيل الثنائي هي 100 وقيمة المتحول op2 وفقا لنظام العد الثنائي هي 101 أن نتيجة تطبيق العامل & على المتحولين op1 و op2 تتمثل بتطبيق العامل & على بتات التمثيل الثنائي للقيمتين وفقا لما يلي:

1- البت الموجود في أقصى يسار المتحول result يمثل نتيجة تطبيق العامل & على البت الموجود في أقصى يسار المتحول op1 مع البت الموجود في أقصى يسار المتحول op2
2- البت التالي في المتحول result يمثل نتيجة تطبيق العامل & على البت التالي الموجود في المتحول op1 مع البت التالي الموجود في المتحول op2
3- البت الموجود في أقصى يمين المتحول result يمثل نتيجة تطبيق العامل & على البت الموجود في أقصى يمين المتحول op1 مع البت الموجود في أقصى يمين المتحول op2.

و بالعودة إلى المتحولين op1 و op2 نلاحظ أن البت الموجود في أقصى يسار المتحول op1 و المتحول op2 هو 1 في كلا المتحولين و بالتالي فان قيمة البت الموجود في أقصى يسار المتحول result هي 1 لان (1&1=1) و البت التالي نلاحظ أن قيمته 0 في المتحول op1 و op2 و بالتالي فإن قيمته البت التالي للمتحول result هي 0 لأن (0&0=0) أما البت الأخير فإن قيمته في المتحول op1 هي 0 و قيمته في المتحول op2 هي 1 و بالتالي فإن قيمة البت الأخير في المتحول result 0 لأن (0&1=0) و بناء على ذلك فإن التشفير الثنائي للمتحول result هو 100 و هو ما يساوي القيمة 4 بنظام العد الطبيعي.

الشكل التالي يستعرض ذلك:

العامل	العدد بالعشري	الترميز الثنائي			
&	4	1	0	0	**1**
	5	1	0	1	**0**
النتيجة	4	1	0	0	**0**

والأمر مشابه أيضا بالنسبة للعامل | فيما عدا أن النتيجة ستختلف كما يوضح ذلك الجدول التالي:

العامل	العدد بالعشري	الترميز الثنائي			
|	4	1	0	0	**1**
	5	1	0	1	**0**
النتيجة	5	1	0	1	**1**

يمكننا أن نستخدم العامل ^ بنفس الطريقة وفقا للجدول التالي:

العامل	العدد بالعشري	الترميز الثنائي			
^	4	1	0	0	**1**
	5	1	0	1	**0**
النتيجة	1	0	0	0	**1**

يمكننا أن نستخدم العامل الأحادي ~ بنفس الطريقة على أحد العددين وفق الجدول التالي:

العامل	العدد بالعشري	الترميز الثنائي			
~	5	1	0	1	**1**
النتيجة	2	0	1	0	**0**

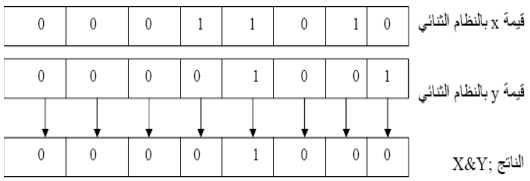
في الحقيقة أن معكوس القيمة 5 في نظام العد الثنائي هو -6 وليس 2 إلا أن المثال السابق لتوضيح حيث تمثل القيمة -6 في نظام العد الثنائي ب 16 خانة أو 32 خانة أو 64 خانة و هذا تمثيل العدد -6 وفق 16 خانة: 1111 1111 1111 1010

أمثلة أخرى:

المعامل & (And):

يكون الناتج 1 في حالة واحدة إذا كان كلا الشرطين 1.

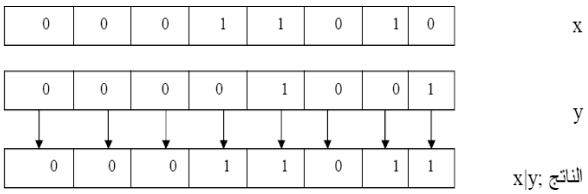
```
int x = 26;                 //  26   11010
int y = 9;                  //  9    01001
Console.WriteLine(x & y); //  8    1000
Console.ReadKey();
```



المعامل | (Or):

يكون الناتج 1 إذا كان أحد الشرطين 1.

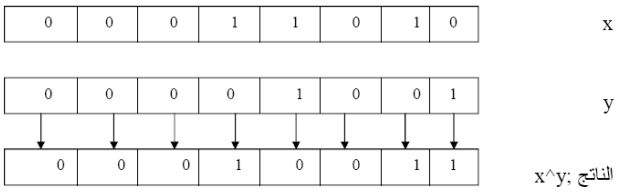
```
int x = 26;                 //   26    11010
int y = 9;                  //   9     01001
Console.WriteLine(x | y); //  27    11011
Console.ReadKey();
```



المعامل ^ (XOR):

يكون الناتج واحد في حالة واحدة إذا كان أحد الطرفين فقط 1.

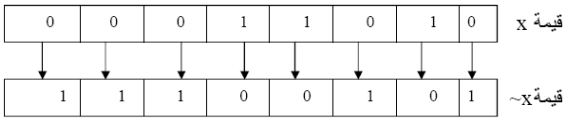
```
int x = 26;                 //  26    11010
int y = 9;                  //  9     01001
Console.WriteLine(x ^ y); //  19    10011
Console.ReadKey();
```



المعامل ~ (Complement Notation):

هذا المعامل يسمى أيضا بالمتمم أي يجعل 1 صفر ويجعل 0 واحد.

```
int x = 13;               //  13    00000000 00000000 00000000 00001101
Console.WriteLine(~x);    // -14    11111111 11111111 11111111 11110010
Console.ReadKey();
```



هذا المثال على اعتبار أن النظام 32 بت يعنى 32 خانة تحتوي أما على صفر أو واحد هو تمثيل الرقم 13 في النظام 32 يكون بهذا الشكل باستخدام المعامل ~ وعكس الواحد صفر والصفر واحد أصبح الناتج بالسالب -14 طبعاً ولو كتب هذا المعامل مرتين يعود بنفس الرقم ~~x = 13

تطبيق حول العوامل المنطقية والعوامل الخاصة بالبتات:

1- قم بإنشاء تطبيق Console جديد باسم Console Bitwise operators.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

```
int myNumber1 = 4, myNumber2 = 5;
int result1,result2,result3, result4;
result1 =myNumber1 & myNumber2 ;
result2 =myNumber1 | myNumber2 ;
result3 = myNumber1 ^ myNumber2;
result4 = ~ myNumber2;
Console.WriteLine("result1={0}", result1);
Console.WriteLine("result2={0}", result2);
Console.WriteLine("result3={0}", result3);
Console.WriteLine("result4={0}", result4);
Console.WriteLine(!true);
Console.WriteLine(!false);
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 فيظهر الشكل (5-4).

الشكل (5-4)

كيفية العمل:

How it Works:

في الشيفرة التي أضفناها قمنا بما يلي:

1- في السطر الاول من الشيفرة قمنا بالتصريح عن متحولين من النوع int واسناد قيم لهما.
2- في السطر الثاني قمنا بالتصريح عن أربعة متحولات من النوع int.
3- في الأسطر الأربعة التالية قمنا بتطبيق العمليات المنطقية على البتات واسناد كل عملية من العمليات لمتحول من المتحولات الاربعة التي صرحنا عنها في السطر الثاني.
4- في الاسطر الاربعة التالية قمنا بإظهار الخرج على نافذة Console كما تعلمنا سابقا.

5- في السطرين التاليين قمنا بإظهار نتيجة تطبيق العامل الاحادي (!) على القيمتين البوليانيتان (true وfalse)

6- السطر الاخير يمكننا من مشاهدة الخرج على شاشة Console.

بالإضافة للعوامل السابقة فإن هناك عاملان لم نتحدث عنهما حتى الان الجدول التالي يوضح هذين العاملين:

العامل	الفئة	مثال	الناتج
>>	ثنائي	var1=var2>>var3	سيأخذ المتحول var1 قيمة المتحول var2 بعد إزاحة بتات قيمته إلى اليمين بقيمة المتحول var3
<<	ثنائي	var1=var2<<var3	سيأخذ المتحول var1 قيمة المتحول var2 بعد إزاحة بتات قيمته إلى اليسار بقيمة المتحول var3

تسمى هذه العملية عادة بعملية إزاحة البتات (bitwise shift operators) لابد وأنك لم تفهم تماما ما يقوم به هذان العاملان لذا سوف نستعرضهما بمثال بسيط:

```
int var1, var2 = 10,var3=2;
var1=var2<<var3;
```

هنا سيأخذ المتحول var1 القيمة 40 ولكن كيف ذلك؟

إن التمثيل الثنائي لقيمة المتحول var2 هي (1010) وبإزاحة بتات هذه القيمة إلى اليسار بمقدار بتين اثنين سيصبح التمثيل الثاني بالشكل (101000) والذي يساوي القيمة 40 بالنظام العشري في الحقيقة ان عملية الازاحة إلى اليسار أي العامل >> تمثل ضرب قيمة المتحول var2 بالقيمة 2 لعدد var3من المرات أي أن إزاحة القيمة 10 بمقدار 2 يعني ضرب 10 بالقيمة 4 أي 40.

وبصورة مشابهة فإن عملية الازاحة إلى اليمين أي العامل << تعني قسمة المتحول var2 على القيمة 2 لعدد var3 من المرات:

```
int var1, var2 = 10;
var1=var2 >> 1;
```

هنا سيتضمن المتحول var1 القيمة 5 وذلك لأن إزاحة العدد الثنائي "1010" إلى اليمين مرة واحدة يساوي العدد الثنائي "101" أي 5 بالنظام العشري.

أما في المثال التالي فإن قيمة المتحول var1 هي 2:

```
int var1, var2 = 10;
var1=var2>>2;
```

لن نستخدم هذا النوع من العوامل في شيفرتنا البرمجية إلا نادرا إلا أنه من الواجب عليك أن تتعرف عليها.

عوامل الإلحاق البوليانية:

Boolean Assignment Operators:

إن أخر مجموعة من العوامل سوف نتعرض لها في هذا القسم هي تلك العوامل نفسها التي تناولناها سابقا متضمنة عامل الإلحاق و هي مشابهة لعوامل الالحاق الرياضية التي تحدثنا عنها في الفصل السابق (مثل =+ و =*...) الجدول التالي يسرد هذه العوامل:

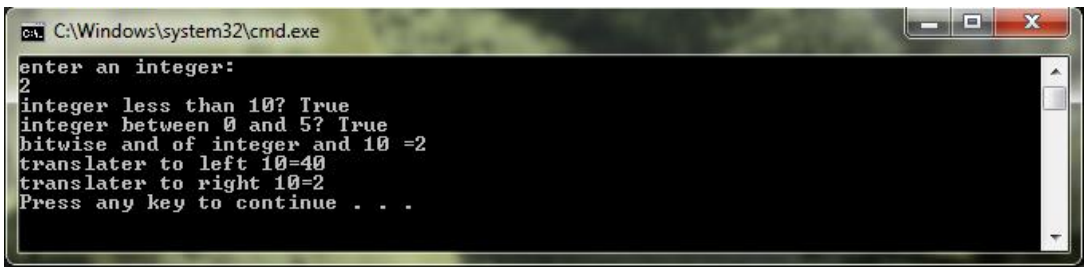
العامل	الفئة	مثال	الناتج
&=	ثنائي	var1&=var2	اسناد التعبير var1&var2 للمتحول var1
|=	ثنائي	var1|=var2	اسناد التعبير var1|var2 للمتحول var1
^=	ثنائي	var1^=var2	اسناد التعبير var1^var2 للمتحول var1
<<=	ثنائي	var1<<=var2	اسناد التعبير var1<<var2 للمتحول var1
>>=	ثنائي	var1>>=var2	اسناد التعبير var1>>var2 للمتحول var1

تطبيق حول استخدام العوامل المنطقية والعوامل الخاصة بالبتات:

1- قم بإنشاء تطبيق Console جديد باسم Console Bitwise operators2.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

```
static void Main(string[] args)
{
    Console.WriteLine("enter an integer:");
    int myInt = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("integer less than 10? {0}", myInt < 10);
    Console.WriteLine("integer between 0 and 5? {0}",
        0 <= myInt && myInt <= 5);
    Console.WriteLine("bitwise and of integer and 10 ={0}", myInt & 10);
    Console.WriteLine("translater to left 10={0}", 10 << myInt);
    Console.WriteLine("translater to right 10={0}", 10 >> myInt);
}
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 وأدخل عددا صحيحا عند طلب ذلك كما في الشكل (4-6).



الشكل (4-6)

كيفية العمل:

How it Works:

في الشيفرة التي أضفناها قمنا بما يلي:

1- في السطر الاول من الشيفرة تظهر رسالة على شاشة Console تطلب من المستخدم ادخال عدد صحيح "enter an integer:".
2- في السطر الثاني قمنا بالتصريح عن متحول باسم myInt من النوع int قيمته تساوي قيمة دخل المستخدم بعد تحويل الدخل إلى النوع int32 عبر التعليمة Convert.ToInt32(Console.ReadLine()).
3- في السطر الثالث من الشيفرة قمنا بإظهار قيمة منطقية (true أو false) على شاشة Console بناء على المقارنة المنطقية myInt < 10 أي هل العدد المدخل أصغر من 10.
4- في السطر الرابع من الشيفرة قمنا بإظهار قيمة منطقية (true أو false) على شاشة Console بناء على المقارنة المنطقية 0 <= myInt && myInt <= 5 أي هل العدد المدخل يقع ضمن المجال المغلق [0-5].
5- في السطر الخامس قمنا بإظهار ناتج عملية منطقية خاصة بالبتات على شاشة Console وذلك بمقارنة بتات الرقم المدخل مع العدد 10 عبر التعليمة myInt & 10.
6- في السطر السادس قمنا بإظهار ناتج عملية منطقية خاصة بالبتات على شاشة Console وذلك بإزاحة بتات العدد 10 لليسار بقيمة العدد المدخل عبر التعليمة myInt << 10.
7- في السطر السابع قمنا بإظهار ناتج عملية منطقية خاصة بالبتات على شاشة Console وذلك بإزاحة بتات العدد 10 لليمين بقيمة العدد المدخل عبر التعليمة myInt >> 10.

أسبقية العوامل:

Operator Precedence:

لقد أصبح لدينا الآن عددا لا بأس به من العوامل وكما للعوامل الرياضية موقع من جدول أسبقية العوامل فإن للعوامل المنطقية موقع أيضا في هذا الجدول وبالتالي فإن جدول أولوية العوامل يصبح بالصورة التالية:

الاسبقية الأعلى	(1) ++ , -- (كبادئات) , +, - (الأحادية) , ~ ,!
	(2) * , / , %
	(3) + , -
	(4) << , >>
	(5) < , > , <= , >=
	(6) == , !=
	(7) &
	(8) ^
	(9) |
	(10) &&

(11) || (12) = , *= , /= , %= , += , -= , <<= , >>= , &= , ^= , |=	
(13) ++ , -- (كلواحق)	الأسبقية الأدنى

لقد احتوى الجدول الآن على مستويات أكثر من قبل إلا أنه يصف أسبقية هذه العوامل على بعضها البعض بشكل صريح.

تعليمة goto:

The goto Statement:

تسمح لنا لغة C# بعنونة أسطر الشيفرة البرمجية وذلك بغرض القفز إلى تلك الأسطر باستخدام تعليمة goto إن لذلك فوائد ومشاكل فالفائدة الرئيسية هي أن هذه التعليمة تمثل طريقة سهلة جدا للتحكم في الشيفرة المنفذة أما المشكلة الرئيسية فهي أن استخدام هذه التعليمة بشكل مكثف سيؤدي لتشكيل شيفرة برمجية صعبة القراءة والفهم.

والمثال التالي سيوضح ألية عمل هذه التعليمة:

```
    int myInteger = 5;
    goto myLabel1;
    myInteger += 10;
myLabel1:
    Console.WriteLine("myInteger ={0}", myInteger);
```

سيتم تنفذ هذه الشيفرة وفقا لما يلي:

1- التصريح عن المتحول myInteger بالنوع int واسناد قيمة له تساوي 5.
2- ستقاطع التعليمة goto التنفيذ المتسلسل للشيفرة وستنقل التحكم بالبرنامج إلى السطر ذو العنوان myLabel1.
3- كتابة قيمة المتحول myInteger على نافذة Console.

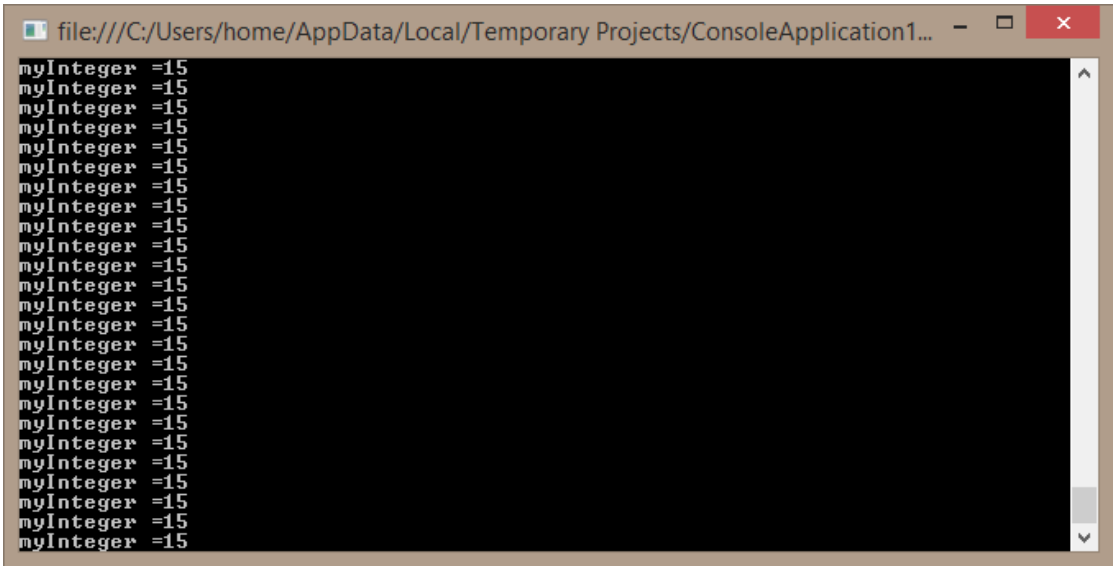
لتعليمة goto استخداماتها إلا أنها يمكن أن تجعل الأمور أكثر إرباكا مما هي عليه خصوصا عند كتابة مئات الاسطر البرمجية التي تتضمن تعليمتين أو ثلاثة للقفز goto.

إليك الشيفرة التالية التي تستعرض بعضا من الامور المربكة لاستخدام تعليمة goto:

```
Start:
    int myInteger = 5;
    goto addVal;
writeResult:
    Console.WriteLine("myInteger ={0}", myInteger);
goto Start;
```

```
addVal:
myInteger += 10;
goto writeResult;
```

إن هذه الشيفرة سليمة تماما إلا أن قراءتها صعبة للغاية ويمكنك أن تتحقق من ذلك إذا حاولت تتبع سير البرنامج والكيفية التي ينفذ وفقها الشكل (7-4) يبن طريقة تنفيذ البرنامج.



الشكل (7-4)

سوف نأتي على هذه التعليمة لاحقا حيث أن لها استخدامات في بنى أخرى من هذا الفصل.

التفرع:

Branching:

يمثل التفرع حدث التحكم بالأسطر البرمجية التي سيتم تنفيذها إن السطر الذي سيتم الانتقال إليه لتنفيذه يعتمد على تحقيق شرط معين وذلك وفقا لتعليمة شرطية (conditional statement) تعتمد نتيجة هذه التعليمة الشرطية على المقارنة بين قيمة اختيارية وبين قيمة أو أكثر باستخدام المنطق البولياني.

سوف نتناول في هذا القسم ثلاثة تقنيات متوفرة في لغة C# للتفرع.

1- العامل ثلاثي الحدود.
2- تعليمة if.
3- تعليمة switch.

العامل ثلاثي الحدود:

The Ternary Operator:

إن أبسط أسلوب للقيام بعملية مقارنة تقتضي باستخدام العامل ثلاثي الحدود (temary operator)ويسمى بالعامل الشرطي (conditional operator) الذي نوهنا عنه في الفصل السابق لقد تعرفنا على طريقة عمل العوامل الأحادية التي تطبق على حد واحد وكذلك العوامل الثنائية التي تطبق على حدين ولذا لن يكون العامل ذو الحدود الثلاثة بشيء غريب.

للعامل ثلاثي الحدود الصيغة التالية:

```
<test> ? < resultIfTrue > : < resultIfFalse >
```

هنا سيتم تنفيذ التعبير الممثل بـ <test> والذي سيمثل تعبيرا منطقيا يعطي قيمة بوليانية (إما true أو false) و بناء على ذلك فإن نتيجة العامل إما ستكون < resultIfTrue > أو < resultIfFalse > فيمكننا أن نستخدم العامل بالصورة التالية:

```
string resultString = (myInt < 10) ? "Less than 10":
    "Greater than or equal to 10";
```

إن نتيجة العامل الثلاثي هنا هي إحدى السلسلتين النصيتين فإحداهما يمكن أن تمثل قيمة المتحول النصي resultString وذلك بناء على نتيجة المقارنة (myInt < 10) فإذا كانت قيمة myInt أقل تماما من 10 فستسند السلسلة النصية الأولى إلى المتحول resultString إذا كانت قيمة myInt أكبر أو تساوي 10 فستسند السلسلة النصية الثانية إلى المتحول resultString أي إذا كانت نتيجة الشرط هي true فإن الحد الثاني سيمثل نتيجة العامل وإن كانت نتيجة الشرط هي false فإن الحد الثالث سيمثل نتيجة العامل.

إن هذا العامل جيد جدا بالنسبة لعمليات الاسناد المشروطة كما وجدنا هذا في المثال السابق إلا أنه غير مناسب لتنفيذ أسطر برمجية عديدة اعتمادا على عملية مقارنة وللقيام بذلك فإن علينا استخدام تعليمة if.

تطبيق حول العامل ثلاثي الحدود:

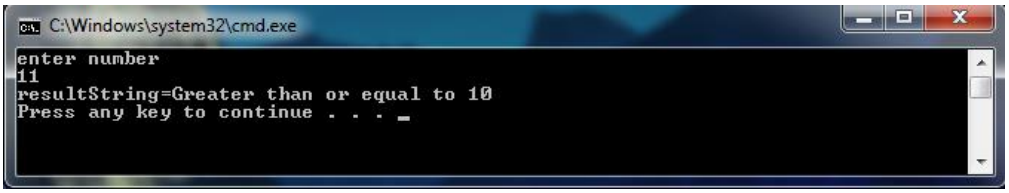
1- قم بإنشاء تطبيق Console جديد باسم Console Ternary Operator.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

```
static void Main(string[] args)
{
    int myInt;
    Console.WriteLine("enter number");
    myInt = Convert.ToInt32(Console.ReadLine());
    string resultString = (myInt < 10) ? "Less than 10" :
           "Greater than or equal to 10";
    Console.WriteLine("resultString={0}", resultString);
}
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 وأدخل عددا صحيحاً أصغر من 10 عند طلب ذلك ثم اضغط enter فتظهر النتيجة كما في الشكل (4-8).

الشكل (4-8)

4- أعد تنفيذ التطبيق بالضغط على مفتاح Ctrl+F5 وأدخل عددا صحيحاً أكبر من 10 عند طلب ذلك ثم اضغط enter فتظهر النتيجة كما في الشكل (4-9).



الشكل (4-9)

كيفية العمل:

How it Works:

في الشيفرة التي أضفناها قمنا بما يلي:

1- في السطر الاول قمنا بالتصريح عن متحول myInt من النوع int.
2- في السطر الثاني من الشيفرة تظهر رسالة على شاشة Console تطلب من المستخدم ادخال عدد صحيح enter number.
3- في السطر الثالث قمنا بإسناد قيمة الدخل إلى المتحول myInt بعد تحويل الدخل إلى النوع int32 عبر التعليمة Convert.ToInt32(Console.ReadLine().
4- في السطر الرابع قمنا بالتصريح عن متحول باسم resultString من نوع string ومن ثم اسندنا له عامل ثلاثي الحدود باستخدام التعبير

```
resultString = (myInt < 10) ? "Less than 10":
    "Greater than or equal to 10";
```

حيث يقوم هذا التعبير بالمقارنة بين قيمة الدخل والعدد 10 إذا كان أصغر من 10 فأن المتحول resultString يقوم بتخزين النص "Less than 10" وإذا كان أكبر أو يساوي 10 فإن هذا المتحول يخزن القيمة "Greater than or equal to 10"

5- في السطر الخامس يتم إظهار النص المسند للمتحول resultString على شاشة Console.

تعليمة if:

The if Statement:

لتعليمة if جوانب استعمال عديدة ومفيدة لصنع القرارات فبعكس العامل :? لا تعيد تعليمة if نتيجة محددة (وبالتالي لا يمكننا استخدامها للإسناد) وبدلا من ذلك فإننا سنستخدم التعليمة لتنفيذ تعليمة أو تعليمات أخرى عند تحقق شرط معين.

الصيغة البسيطة لتعليمة if لها الشكل التالي:

```
If (< test >)
< Code executed if < test > is true >;
```

سيتم تنفيذ السطر البرمجي لتعليمة if إذا أخذ التعبير البولياني < test >القيمة true فقط وإن كانت نتيجة التعبير < test > هي false فلن ينفذ سطر تعليمة if عندئذ.

تقوم صيغة التعليمة تلك بتنفيذ شيفرة عند تحقق شرط محدد ولكن إذا لم يتحقق هذا الشرط عندئذ سينتقل التنفيذ إلى السطر التالي وهناك صيغة موسعة لتعليمة if تمكننا من تنفيذ شيفرة معينة أخرى عند عدم تحقق الشرط وتلك الصيغة لها الشكل التالي:

```
If (< test >)
< Code executed if < test > is true >;
Else
< Code executed if < test > is false >;
```

يمكننا ان ننفذ كتله برمجيه كامله (اي عدة اسطر من الشيفرة) ضمن تعليمه if وذلك وفقا للصيغة:

```
If (< test >)
{
< Code executed if < test > is true >;
}
Else
{
< Code executed if < test > is false >;
}
```

وكمثال بسيط لنعيد كتابة المثال الذي كتبناه باستخدام العامل الشرطي الثلاثي :? في القسم التالي باستخدام تعليمة if:

```
int myInt;
string resultString;
Console.WriteLine("enter number:");
myInt = Convert.ToInt32(Console.ReadLine());
if (myInt < 10)
    resultString ="Less than 10" ;
else
    resultString ="Greater than or equal to 10";
Console.WriteLine("resultString ={0}", resultString);
Console.Read();
```

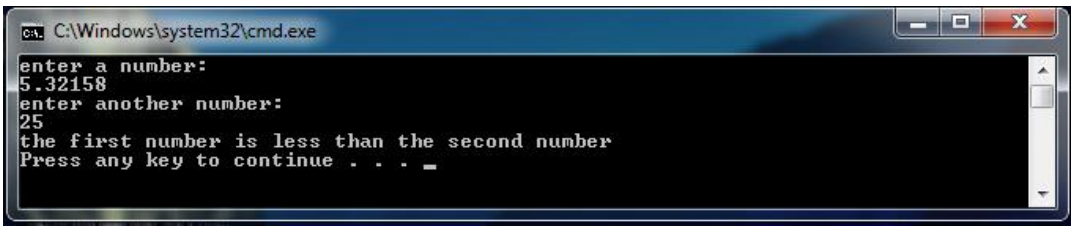
على الرغم من أن هذه الشيفرة البرمجية تحوي أسطر برمجية إضافية أي أن الشيفرة أكبر إلا اننا نلاحظ أنه يمكن قراءة هذه الشيفرة بسهولة أكثر من تعليمة العامل الثلاثي يمكنك تطبيق هذا المثال بمفردك وتشاهد نتائج التنفيذ.

تطبيق حول استخدام تعليمة if:

1- قم بإنشاء تطبيق Console جديد باسم Console if Statement.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

```
static void Main(string[] args)
{
    String comparison;
    Console.WriteLine("enter a number:");
    double var1 = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("enter another number:");
    double var2 = Convert.ToDouble(Console.ReadLine());
    if (var1 < var2)
        comparison = "less than";
    else
    {
        if (var1 == var2)
            comparison = "equal to";
        else
            comparison = "greater than";
    }
    Console.WriteLine("the first number is {0} the second number"
        , comparison);
}
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 ثم قم بإدخال العدد الأول عندما يطلب منك البرنامج ثم اضغط Enter ثم قم بإدخال العدد الثاني ومن ثم اضغط Enter فيظهر الشكل (10-4).



الشكل (10-4)

كيفية العمل:

How it Works:

في الشيفرة التي أضفناها قمنا بما يلي:

1- في السطر الاول قمنا بالتصريح عن متحول comparison من النوع String.
2- في السطر الثاني من الشيفرة تظهر رسالة على شاشة Console تطلب من المستخدم ادخال عدد enter a number.
3- في السطر الثالث قمنا بالتصريح عن متحول جديد باسم var1 وأسندنا له قيمة الدخل بعد تحويل الدخل إلى النوع Double (عشري أي يقبل فاصلة عائمة) عبر التعليمة Convert.ToDouble(Console.ReadLine()).
4- في السطر الرابع تظهر رسالة على شاشة Console تطلب من المستخدم ادخال عدد enter another number:
5- في السطر الخامس قمنا بالتصريح عن متحول جديد باسم var2 وأسندنا له قيمة الدخل بعد تحويل الدخل إلى النوع Double عبر التعليمة Convert.ToDouble(Console.ReadLine()).
6- تعمل تعليمة if على المقارنة بين قيمة المتحولين var1 وvar2 فإذا كانت قيمة var1 أصغر من var2 عندئذ يسند للمتحول comparison السلسلة النصية "less than" وإلا إذا كانت قيمة المتحولين متساوية فإنه يسند للمتحول comparison السلسلة النصية "equal to" وإلا فإنه يسند للمتحول comparison السلسلة النصية "greater than".
7- يعمل السطر الأخير على إظهار نتيجة المقارنة على شاشة Console

لقد استخدمنا هنا تعشيش تعليمة if ويمكننا أن نعيد كتابة قسم تعليمة if السابقة كما يلي:

```
if (var1 < var2)
comparison = "less than";
if (var1 == var2)
comparison = "equal to";
if (var1 >var2 )
```

إلا أنني لا أفضل استخدام هذه الطريقة وذلك لأننا هنا نقوم بثلاث عمليات مقارنة بغض النظر عن قيمة المتحولين var1 وvar2أما في الطريقة الأولى فإن هناك عمليتا مقارنة كحد اقصى وتلك مسألة تؤثر في الاداء بالنسبة للتطبيقات التي تكون فيها سرعة التنفيذ أمرا حاسما.

تفحص شروط إضافية باستخدام تعليمات if:

Checking More Conditions using if Statements:

لقد تفحصنا في المثال السابق ثلاثة شروط استخدمت المتحول var1 لقد غطت هذه الشروط كافة القيم الممكنة لهذا المتحول قد تحتاج في العديد من الأحيان إلى تفحص قيم محددة للمتحول var1 كأن نقول

مثلا إذا كانت قيمة var1 مساوية لـــ 1 قم بكذا وإذا كانت مساوية لـــ 2 قم بكذا. إن تنفيذ شيفرة كهذه باستخدام مبدأ تعشيش تعليمات if سيؤدي إلى شيفرة مشابهة لما يلي:

```
If (var1==1)
{
// do something
}
     Else
    {
    If (var1==2)
      {
      // do something else
       }
        Else
  {
       If (var1==3 || var1==4)
     {
      // do something else
        }
         Else
       {
       // do something else
        }
     }
  }
```

ملاحظة:

من الاخطاء البرمجية الشائعة كتابة الشرط الثالث في الشيفرة السابقة بالشكل if (var1==3||4) وهنا بالعودة إلى أسبقية العوامل على بعضها فإن العامل == سينفذ قبل العامل || والذي سيؤدي إلى تنفيذ العامل || على حد منطقي وأخر رقمي إن هذا ما سيسبب في حدوث خطأ باعتبار أن نتيجة الشرط في تعليمة if يجب أن تعيد دائما قيمة منطقية.

في حالة كهذه من الأفضل أن نستخدم طريقة مختلفة قليلا لتحقيق هذا النوع من الشروط عندئذ من الأفضل وضع شيفرة قسم else ضمن كتلة شيفرة وفي حالة كهذه سوف ننتهي ببنية من تعليمات else if المركبة كما في المثال التالي:

```
If (var1==1)
{
// do something
}
    Else  If (var1==2)
{
  // do something else
}
       Else   If (var1==3 || var1==4)
 {
  // do something else
}
Else
{
// do something else
}
```

قد تظن من خلال البنية السابقة أن if…else تمثل شكلا حديثا من تعليمة if إلا أن if وelse هنا هما تعليمتان منفصلتان تماما والشيفرة التي كتبناها هنا هي نفسها الشيفرة السابقة إلا أن الأخيرة تم تنسيقها لتصبح أكثر قابلية للقراءة من الأولى (يمكنك أن تلحظ ذلك بنفسك إذا تأملت الشيفرتين).

وعلى الرغم من ذلك إلا أن تعليمة if…. Else, ما تزال غير عملية عندما نواجه شروطا عديدة (كثلاثة أو أربعة شروط أو أكثر) و لهذا فإن هناك تعليمة بديلة هي switch تمثل بنية تفرع بديلة.

تعليمة switch:

The switch Statement:

إن تعليمة switch مشابهة لتعليمة if من حيث مبدأ العمل (تنفيذها لشيفرة معينة مشروط بقيمة اختبارية ما) إلا أن تعليمة switch تسمح لنا بقيم عديدة لمتحول الاختبار ضمن بنية واحدة بدلا من شرط واحد فقط (كما في تعليمة if) إن هذا الشرط محدد بقيمة معينة بدلا من استخدام عبارات كـ "أكبر من X" وبالتالي فإن بنية تعليمة switch مختلفة عن بنية تعليمة if قليلا إلا أنها تمثل تقنية قوية جدا للتفرع.

البنية الاساسية لتعليمة switch لها الصيغة التالية:

```
Switch (< testvar >)
{
Case < comparisonval1 >:
< Code to execute if < testvar >==< comparisonval1 >>
Break;
Case < comparisonval2 >:
< Code to execute if < testvar >==< comparisonval2 >>
Break;
.....
Case < comparisonvalN >:
< Code to execute if < testvar > == < comparisonvalN >>
Break;
Default:
< Code to execute if < testvar >!=< comparisonvals >>
Break;
}
```

سيتم مقارنة القيمة في < testvar > مع جميع قيم < comparisonvalN > (المحددة في تعليمات case) وإن كان هناك تطابق سيتم تنفيذ الشيفرة الموجودة في القسم الذي حدث عنده التطابق وإن لم يكن هناك تطابق مع أية قيمة سيتم عندئذ تنفيذ الشيفرة الموجودة في القسم default إن كان هذا القسم موجودا. هناك أمر واحد يجب أن نذكره هنا وهو أنه من غير المسموح متابعة تنفيذ الشيفرة في أقسام case التالية بعد تنفيذ الشيفرة الموجودة في أحد أقسام case ولذا فإن التعليمة break هنا تقوم بإنهاء تعليمة switch ونقل التحكم للتعليمة التي تلي بنية switch مباشرة.

هناك طرق بديلة لتجنب وصول التنفيذ من تعليمة case إلى تعليمة case التالية في لغة C# ويمكننا استخدام التعليمة return والتي تؤدي إلى إنهاء تنفيذ التابع الحالي بدلا من مجرد الخروج من بنية switch وهناك تعليمة goto التي شرحناها مسبقا والتي يمكننا بواسطتها نقل التحكم إلى عنوان سطر خارج بنية switch لاحظ أن أقسام case في بنية switch تأخذ عناوين الأسطر وبناء على ذلك يمكننا نقل التحكم إلى تعليمة case أخرى باستخدام تعليمة goto كما في المثال التالي:

```
Switch (< testvar >)
{
Case < comparisonval1 >:
< Code to execute if < testvar >== < comparisonval1 >>
Goto case < comparisonval2 >;
Case < comparisonval2 >:
< Code to execute if < testvar >== < comparisonval2 >>
Break;
…….
```

هناك استثناء وحيد للقاعدة التي تقول إنه لا يمكن الانتقال من تنفيذ تعليمة case إلى تنفيذ تعليمة case التالية بحرية ويقتضي ذلك بوضع تعليمات case مع بعضها (بتكديسها) قبل وضع كتلة من الشيفرة ونحن بذلك نقوم بفحص عدة شروط في وقت واحد فإن تحقق واحد من هذه الشروط ستنفذ الشيفرة الموجودة ضمن الكتلة على سبيل المثال:

```
Switch (< testvar >)
{
Case < comparisonval1 >:
Case < comparisonval2 >:
< Code to execute if < testvar >== < comparisonval1 > or
                    < testvar >== < comparisonval2 >>
Break;
```

لاحظ أن هذه الحالات مطبقة على تعليمة default أيضا فليست هناك قاعدة تقول إن تعليمة default يجب أن تأتي في نهاية لائحة المقارنات ويمكننا تكديسها مع تعليمات case إذا أردنا ذلك.

إن وضع نقطة توقف بواسطة أحد التعليمات break أو goto أو return يضمن وجود مسار تنفيذ نظامي خلال بنية switch في كافة الحالات.

يجب أن تمثل كل مقارنة من مقارنات < comparisonvalN > قيمة ثابتة وأحد أبسط الطرق للقيام بذلك استخدام القيم الحرفية كما في المثال:

```
Switch (muInteger)
{
Case 1:
<Code to execute if myInteger == 1>
Break;
Case -1:
<Code to execute if myInteger == -1>
Case default:
< Code to execute if myInteger != comparisons >
Break;
}
```

هناك طريقة أخرى لذلك تقتضي باستخدام المتحولات الثابتة (constant variables) إن المتحولات الثابتة هي كأي متحولات أخرى فيما عدا أن لها ميزة واحدة فقط وهي أن القيمة التي تسند إليها عند التصريح عنها لا يمكن أن تتغير مطلقا للمتحولات الثابتة فوائد عديدة في بعض المواضع كما ستجد ذلك خلال هذا الكتاب.

يمكننا التصريح عن المتحولات الثابتة باستخدام الكلمة المحجوزة const متبوعة بنوع المتحول ثم اسمه ومن ثم يجب إسناد قيمة له ضمن سطر التصريح مباشرة كما في المثال:

```
Const int intTwo=2;
```

أما الشيفرة التالية فهي غير قانونية وستؤدي على حدوث خطا في الترجمة:

```
Const int intTwo;
intTwo=2;
```

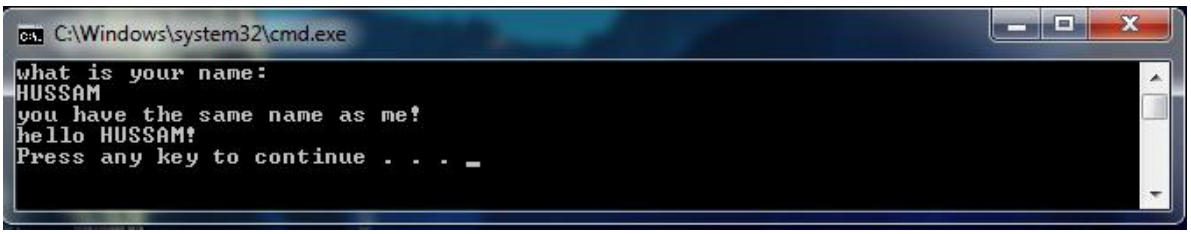
كذلك الأمر إذا حاولنا تغيير قيمة المتحول الثابت في أي مكان من البرنامج بعد الإسناد الأولي.

تطبيق حول استخدام تعليمة switch:

1- قم بإنشاء تطبيق Console جديد باسم Console switch Statement.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

```
static void Main(string[] args)
{

    //حجز متغير يملك قيمة ابتدائية
    const string myName = "hussam";
    const string sexyName = "hussom";
    const string sillyName = "abo omar";
    string name;
    Console.WriteLine("what is your name:");
    name = Console.ReadLine();
    //تحويل الاحرف الى أحرف صغيرة و مقارنتها مع القيم المخزنة
    switch (name.ToLower())
    {
        case myName:
            Console.WriteLine("you have the same name as me!");
            break;
        case sexyName:
            Console.WriteLine("my,what a sexy name you have!");
            break;
        case sillyName:
            Console.WriteLine("that's a very silly name.");
            break;
    }
    Console.WriteLine("hello {0}!", name);
}
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 ثم قم بإدخال اسم عندما يطلب منك البرنامج ثم اضغط Enter فيظهر الشكل (11-4).



الشكل (11-4)

كيفية العمل:

How it Works:

في الشيفرة التي أضفناها قمنا بما يلي:

1- في الأسطر الثلاثة الاولى قمنا بالتصريح عن ثلاث ثوابت هي (myName،sexyName ، sillyName) من النوع String وأسندنا لكل متغير قيمة معينة.
2- في السطر الرابع قمن بالتصريح عن متحول باسم name من النوع string.

3- في السطر الخامس تظهر رسالة على شاشة Console تسأل المستخدم عن اسمه "what is your name:".
4- بعد إدخال المستخدم لاسم وضغطه على زر Enter يتم اسناد القيمة المدخلة للمتحول name.
5- أما باقي الاسطر البرمجية فإن تعليمة switch تعمل على مقارنة قيمة دخل المستخدم مع إحدى الحلات الثلاث للثوابت وذلك بعد تحويل النص الذي أدخله المستخدم إلى أحرف صغيرة لان مترجم C# حساس لحالة الاحرف كما نوهنا سابقا ويتم هذا التحويل عبر العبارة name.ToLower() في حالة تطابق الدخل مع إحدى القيم الثلاث للثوابت السابقة تظهر رسالة تشير إلى تشابه الاسم المدخل مع إحدى الحالات الثلاث السابقة كما تظهر رسالة ترحيب بهذا الاسم وفي حالة عدم التشابه مع إحدى القيم السابقة سينتقل التحكم إلى خارج نطاق تعليمة switch و تظهر رسالة ترحيب متبوعة بالاسم المدخل ("hello {0}!", name).

ملاحظة:

يعمل الأمر ToLower على تحويل حالة الحرف المدخلة إلى حالة الأحرف الصغيرة وبالمثل هناك الأمر ToUpper يعمل على تحويل الاحرف المدخلة إلى حالة الأحرف الكبيرة.

تطبيق أخر حول استخدام تعليمة switch:

1- قم بإنشاء تطبيق Console جديد باسم Console switch statement2.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

```
static void Main(string[] args)
{

    string jop;
    Console.WriteLine("enter your jop");
    jop = Console.ReadLine();
    switch (jop.ToUpper ())
    {
        case "DOCTOR":
            Console.WriteLine("you are a doctor!");
            break;
        case "ENGINEER":
            Console.WriteLine("you are an engineer");
            break;
        default :
            Console .WriteLine ("jop unknow!!");
            break;
    }
}
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 ثم قم بإدخال اسم عندما يطلب منك البرنامج ثم اضغط enter فيظهر الشكل (12-4).

الشكل (12-4)

سأترك لك شرح هذا البرنامج باعتقادي أن الأمور واضحة.

الحلقات:

Loops:

تمثل الحلقات المكان الذي يمكننا فيه تنفيذ الشيفرة البرمجية بصورة تكرارية إن هذه البنية مفيدة جدا وكما يشير اسمها فإنه يمكننا تكرار تنفيذ سطر أو كتابة كتلة برمجية لأي عدد من المرات التي نريد وبالتالي سنتجنب كتابة الشيفرة نفسها أكثر من مرة عندما نود تنفيذها بصورة متكررة.

وكمثال بسيط لنأخذ الشيفرة التالية والتي تقوم باحتساب قيمة حساب في البنك (بقيمة ابتدائية قدرها 1000 ليرة) بعد عشر سنوات وبافتراض أن معدل الفائدة هو 1.05 أي (5%) وهو يدفع كل سنة وليست هناك أية أموال تدخل أو تخرج من هذا الحساب خلال هذه المدة:

تطبيق الحساب البنكي:

1- قم بإنشاء تطبيق Console جديد باسم Console bank.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

```
static void Main(string[] args)
{
    Double yourCount, yearRate;
    Console.WriteLine("enter yourCount:");
    yourCount = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("enter rateYear:");
    yearRate = Convert.ToDouble(Console.ReadLine());
    yourCount *= yearRate;
    yourCount *= yearRate;
    yourCount *= yearRate;
    yourCount *= yearRate;
    yourCount *= yearRate;
    yourCount *= yearRate;
    yourCount *= yearRate;
    yourCount *= yearRate;
    yourCount *= yearRate;
    yourCount *= yearRate;
    Console.WriteLine("yourCount after 10 years" + yourCount);
}
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 ثم قم بإدخال قيمة الحساب yourCount عندما يطلب منك البرنامج وليكن (1000ليرة) ثم اضغط Enter من ثم أدخل قيمة معدل الزيادة السنوي yearRate وليكن (1.05) من ثم قم بالضغط على Enter فيظهر الشكل (13-4).

الشكل (13-4)

كيفية العمل:

How it Works:

في الشيفرة التي أضفناها قمنا بما يلي:

1- في السطر الأول قمنا بالتصريح عن متحولين (yourCount،yearRate) من النوع Double.
2- الاسطر الأربعة التالية تقوم بالطلب من المستخدم إدخال قيمة الحساب "enter yourCount:" ومعدل الزيادة السنوي "enter rateYear:" وتسندها للمتحولات السابقة بعد تحويل قيم الإدخال إلى النوع ToDouble.
3- تعمل الأسطر العشرة التالية على حساب قيمة الحساب بعد عشر سنوات وفقا لقيمة معدل الزيادة المدخل.
4- يعمل السطر الاخير على إظهار نتيجة الحساب بعد عشر سنوات.

ملاحظة:

يعمل العامل (+) على الجمع بين النصوص وهو بذلك يشبه عمل {} التي كنا نستخدمها سابقا.

نلاحظ في الشيفرة السابقة اننا أعدنا كتابة الشيفرة yourCount *= yearRate; عشر مرات إن هذا الامر غير منطقي وسيسبب الكثير من التعب في حال أردنا حساب قيمة الحساب بعد 100 سنة مثلا فما الحل؟

لحسن الحظ فإذنا لسنا بحاجة للقيام بذلك وإنما يمكننا وضع الشيفرة تلك ضمن حلقة تنفذ الامر السابق لعشر مرات أو أي عدد تكرار نريده وسوف نتعرف على بعض حلقات التكرار ضمن لغة البرمجة C#.

حلقات DO:

DO Loops:

يتم تكرار الشيفرة ضمن حلقة DO بالصورة الآتية: ستنفذ الشيفرة الموجودة ضمن الحلقة أولا ومن ثم سيتم تفحص شرط منطقي فإن كانت قيمته True ستنفذ شيفرة الحلقة مرة أخرة وإذا كانت قيمته False فسيتم الخروج من الحلقة وسينتقل التحكم لخارج الحلقة إلى السطر البرمجي التالي.

الصيغة العامة لحلقة DO هي بالشكل التالي:

```
Do
{
<Code to be looped>
}
While (<test>);
```

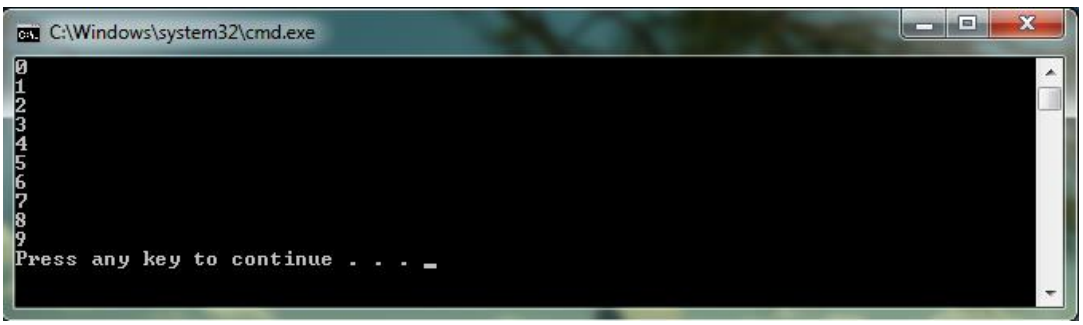
حيث يعيد التعبير <test> قيمة منطقية

تطبيق حول حلقة do:

1- قم بإنشاء تطبيق Console جديد باسم Console do.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

```
int i = 0;
do
{
    Console.WriteLine("{0}", i++);
}
while (i < 10);
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 فيظهر الشكل (4-14).



الشكل (4-14)

كيفية العمل:

How it Works:

في الشيفرة التي أضفناها قمنا بما يلي:

1- في السطر الأول قمنا بالتصريح عن متحول i من النوع int وأسندنا له قيمة أولية تساوي الصفر.
2- في الاسطر التالية أدخلنا أمر الطباعة على شاشة Console ضمن حلقة do ليطبع لنا ناتج تغير العداد ++i ذو القيمة الابتدائية 0 إلا أن أمر الطباعة حددناه بشرط وهو شرط الخروج من الحلقة وذلك باستخدام تعليمة while

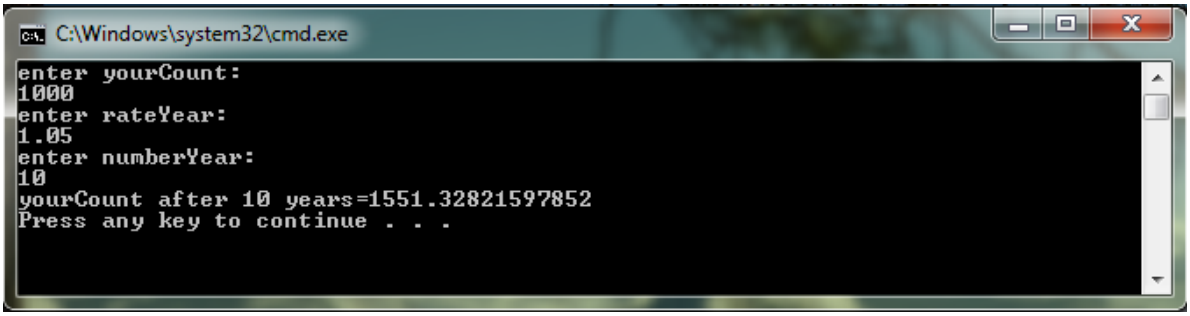
لنعد إلى مثال الحساب البنكي لقد كنا نحسب قيمة الحساب بعد عشر سنوات سوف نستخدم هنا حلقة لاحتساب قيمة الحساب بعد مرور عدد من السنوات يحدده المستخدم باستخدام حلقة do.

تطبيق الحساب البنكي باستخدام حلقة do:

1- قم بإنشاء تطبيق Console جديد باسم Console bank do.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

```
static void Main(string[] args)
{
    Double yourCount, yearRate;
    int numberYear, i = 1;
    Console.WriteLine("enter yourCount:");
    yourCount = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("enter rateYear:");
    yearRate = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("enter numberYear:");
    numberYear = Convert.ToInt32(Console.ReadLine());
    do
    {
        yourCount *= yearRate;
    }
    while (i++ < numberYear);
    Console.WriteLine("yourCount after {0} year{1}={2}",
        numberYear, numberYear == 1 ? "" : "s", yourCount);
}
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 ومن ثم أدخل قيمة الحساب مساوي لـ 1000 ومعدل الزيادة السنوي مساوي لـ 1.05 وعدد السنوات مساوي لـ 10 فتظهر النتيجة كما في الشكل (4-15).



الشكل (4-15)

كيفية العمل:

How it Works:

في الشيفرة التي أضفناها قمنا بما يلي:

1- في السطر الأول قمنا بالتصريح عن متحولين (yourCount, yearRate) من النوع Double.
2- في السطر الثاني قنا بالتصريح عن متحولين (numberYear,i=1) من النوع int و اسندنا قيمة للمتحول i تساوي 1.
3- في الشيفرة التالية:

```
Console.WriteLine("enter yourCount:");
yourCount = Convert.ToDouble(Console.ReadLine());
Console.WriteLine ("enter rateYear:");
yearRate = Convert.ToDouble(Console.ReadLine());
Console.WriteLine("enter numberYear:");
numberYear = Convert.ToInt32 (Console.ReadLine());
```

قمنا بالطلب من المستخدم ادخال قيم المتحولات السابقة وأسندنا قيم هذه الإدخالات للمتوحل المقابل لكل منها.

4- إن شيفرة حلقة do تعمل على تكرار حساب قيمة الحساب وفقا لمعدل الزيادة السنوي المدخل طالما أن قيمة العداد ++i أصغر من قيمة عدد السنوات المدخل لاحظ أننا هنا أدخلنا قيمة مسبقة للعداد تساوي 1 أي أن العداد سوف يكرر عملية حساب الشيفرة (yourCount *= yearRate;) تسع مرات عوضا عن عشر مرات وذلك لأننا أثناء عملية طباعة الحساب على شاشة Console بعد الخروج من الحلقة سنكرر حساب الشيفرة للمرة العاشر.

ملاحظة:

لاحظ في الشيفرة التالية:

```
Console.WriteLine("yourCount after {0} year{1}={2}" ,
    numberYear,numberYear ==1?"":"s",yourCount  );
```

أننا استخدمنا العامل الثلاثي (:?) لوضع الحرف "s" بعد الكلمة year إذا كان عدد السنوات لا يساوي الواحد.

حاول أن تدخل عدد السنوات مساوي للصفر ولاحظ ما الذي سيحدث الشكل (4-16) لا تستعجل في السؤال عن السبب فالجواب سيأتيك في الفقرة التالية.

الشكل (4-16)

حلقات while:

While Loops:

حلقات while مشابهة لحلقات do إلا أن هناك فرق هام بينها وهو أن التحقق من شرط الاستمرار في حلقة do يتم بعد تنفيذ الشيفرة في جسم الحلقة بينما يتم التحقق من شرط الاستمرار في حلقة while قبل أن ينفذ جسم الحلقة وهذا يعني أنه إذا كانت قيمة الشرط هي false في حلقة while فإن شيفرة جسم الحلقة لن تنفذ أبدا بعكس حلقة do والتي سيتم تنفيذ الشيفرة ضمنها مرة واحدة على الأقل حتى إن كانت قيمة الشرط هي false.

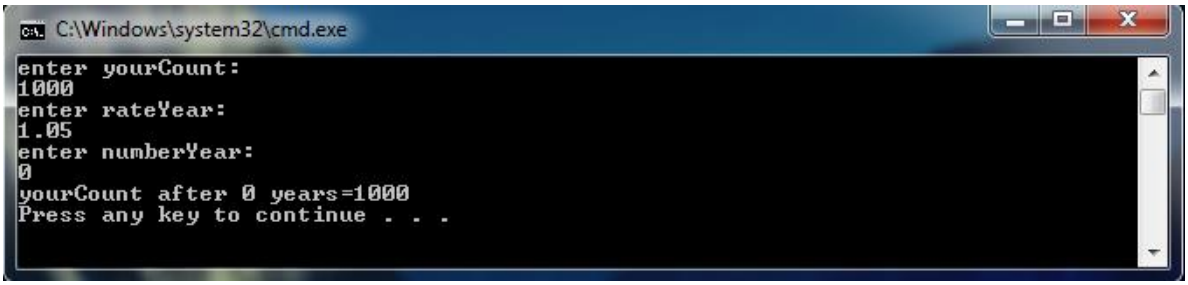
للحلقة while الصيغة التالية:

```
While (<test>)
{
< Code to be looped >
}
```

```
int i = 0;
while (i < 10)
{
    Console.WriteLine("{0}", i++);
}
Console.ReadLine();
```

في مثال الحساب البنكي قم باستبدال حلقة do بحلقة while كما تبين ذلك الشيفرة التالية ومن ثم أدخل عدد السنوات مساوي للصفر ولاحظ اختلاف النتيجة كما يظهر في الشكل (4-17):

```
static void Main(string[] args)
{
    Double yourCount, yearRate;
    int numberYear, i = 1;
    Console.WriteLine("enter yourCount:");
    yourCount = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("enter rateYear:");
    yearRate = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("enter numberYear:");
    numberYear = Convert.ToInt32(Console.ReadLine());
    while (i++ < numberYear)
    {
        yourCount *= yearRate;
    }
    Console.WriteLine("yourCount after {0} year{1}={2}",
        numberYear, numberYear == 1 ? "" : "s", yourCount);
}
```



الشكل (4-17)

موضوع التحقق من إدخال المستخدم موضوع مهم جدا عند تصميم التطبيقات وسوف نجد العديد من الامثلة التي تتطلب تحققا من إدخال المستخدم في هذا الكتاب.

ملاحظة:

بالطبع هناك طرق بديلة لحل هذا النوع من المشاكل باستخدام حلقة do وذلك باستخدام تعليمة if للتحقق من أن قيمة عدد السنوات المدخل أكبر من قيمة i الاولية وذلك كما يلي:

```
do
{
    if (numberYear >i)
    yourCount *= yearRate;
}
while (i++ < numberYear);
Console.WriteLine("yourCount after {0} year{1}={2}" ,
    numberYear,numberYear ==1?"":"s",yourCount  );
```

حلقات for:

For Loops:

إن النوع الأخير والأقوى من الحلقات في لغة C# هي حلقة for لاحظنا أن تنفيذ حلقتي do وwhile اعتمد على تحقيق شرط الحلقة فقط هذا يعني أنه إذا كانت لدينا حلقة do طبيعة فإنها ستنفذ إلى الأبد طالما أن الشرط محقق أما في حلقة for فستنفذ لعدد محدد من المرات فلهذه الحلقة عداد (يعرف بمتحول الحلقة) يتتبع عدد مرات تنفيذ لتعريف حلقة for فإننا نحتاج إلى ما يلي:

1- القيمة الابتدائية لمتحول الحلقة.
2- شرط الاستمرار في تنفيذ الحلقة (وهو يعتمد على متحول الحلقة).
3- العملية التي ستؤدى على متحول الحلقة في نهاية كل دورة.

على سبيل المثال إذا أردنا أن نستخدم حلقة بعداد من 1 إلى 10 بحيث يخطو العداد خطوة واحدة في كل دورة عندئذ يجب أن يكون الشرط هو أن متحول الحلقة يجب ان يكون أقل أو يساوي 10 والعملية التي ستؤدى على متحول الحلقة هي زيادة قيمته الحالية بمقدار واحد أما قيمة متحول الحلقة الابتدائية فهي 1.

يجب أن تتوضع هذه المعلومات ضمن بنية الحلقة كما في الصيغة التالية:

```
For (< initialization > ; < condition > ; < operation >)
{
< Code of loop >
}
```

مثال طباعة الاعداد:

```
int i;
for (i = 0; i < 10; i++)
{
    Console.WriteLine("{0}", i);
}
Console.ReadLine();
```

متحول الحلقة هنا هو متحول عددي من نوع int باسم i وله قيمة ابتدائية قدرها 0 أي أنه يبدأ بالعد من القيمة 0 وتتم زيادة قيمته مع كل دورة بمقدار 1 إلى ان تصل قيمته إلى 9 يقوم البرنامج السابق بطباعة قيمة متحول الحلقة i على نافذة الخرج.

لاحظ أنه عند نهاية تنفيذ الحلقة ككل والانتقال إلى الاسطر البرمجية التالية من الشيفرة (أي عند الخروج من الحلقة) فإن قيمة المتحول i هي 10 إن هذا يعود إلى أن الزيادة في قيمة متحول الحلقة يحدث عند نهاية كل دورة وقبل أن نتحقق من الشرط.

وكما في حلقة while فإن حلقة for تنفذ إذا كانت قيمة الشرط هي true قبل أول دورة وبالتالي ليس بالضرورة أن تنفذ الشيفرة ضمن جسم الحلقة دائما.

ملاحظة:

يمكننا التصريح عن متحول الحلقة كجزء من تعليمة for سنعيد كتابة الشيفرة السابقة كما يلي:

```
for (int i = 1; i <= 10; i++)
{
    Console.WriteLine("{0}", i);
}
Console.ReadLine();
```

ملاحظة:

لاحظ أننا لسنا مضطرين في هذا المثال إلى وضع تعليمة الطباعة ضمن كتلة برمجية وذلك لأن جسم الحلقة لا يتضمن سوى تعليمة واحدة فقط أي أنه يمكننا كتابة الشيفرة السابقة كما يلي:

```
for (int i = 1; i <= 10; i++)
    Console.WriteLine("{0}", i);
Console.ReadLine();
```

ملاحظة:

لاحظ أيضا أن أية أسطر برمجية بعد تعليمة الطباعة لا تمثل جزءا من جسم الحلقة إذا لم نضعها ضمن كتلة برمجية (أي ضمن الأقواس "{}") أي أن السطر الجديد في الشيفرة التالية لن ينفذ ضمن الحلقة:

```
for (int i = 1; i <= 10; i++)
    Console.WriteLine("{0}", i);
Console.WriteLine("HELOO WORLD");
Console.ReadLine();
```

ملاحظة:

وإذا أردنا أن ننفذه ضمن الحلقة يجب أن نكتب كما يلي:

```
for (int i = 1; i <= 10; i++)
{
    Console.WriteLine("{0}", i);
    Console.WriteLine("HELOO WORLD");
}
Console.ReadLine();
```

تطبيق حول استخدام حلقةFOR :

1- قم بإنشاء تطبيق Console جديد باسم Console for.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

```
static void Main(string[] args)
{
    UInt64   number;
    UInt64  i;
    UInt64  factoriel=1;
    Console.WriteLine("enter an integer number:");
    number =Convert .ToUInt64  (Console .ReadLine ());
    for (i = number; i > 0; i--)
    {
        factoriel = factoriel * i;
    }
    Console.WriteLine("factoriel for {0}!={1}", number, factoriel);
  }
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 ومن ثم أدخل عدد صحيح من أجل حساب قيمة العلملي له فتظهر النتيجة كما في الشكل (4-18).

الشكل (4-18)

كيفية العمل:

How it Works:

1- في السطر الأول والثاني قمنا بالتصريح عن متحولين (number،i) من النوع UInt64.

2- في السطر الثالث قنا بالتصريح عن متحول باسم factoriel من النوع UInt64 وأسندنا قيمة لهذا المتحول قيمة ابتدائية تساوي 1.
3- تعمل الشيفرة التالية

```
Console.WriteLine("enter an integer number:");
number =Convert .ToUInt64 (Console .ReadLine ());
```

على الطلب من المستخدم إدخال عدد صحيح "enter an integer number:"ومن ثم يتم اسناد قيمة الادخال للمتحول number بعد تحويل نوع بيانات الادخال إلى النوع UInt64.

4- في حلقة for قيمة لعداد الحلقة i و هي قيمة دخل المستخدم وحددنا شرط للخروج من الحلقة و هو أن تكون قيمة i أكبر من الصفر وبعملية رياضية تناقصية أي عداد متناقص ومن ثم قمن بوضع الشيفرة البرمجية التي تقوم باحتساب قيمة العاملي factoriel كما في الشيفرة:

```
for (i = number; i > 0; i--)
{
    factoriel = factoriel * i;
}
```

مقاطعة الحلقات:

Interrupting Loops:

هناك أوضاع نحتاج فيها إلى التوقف عند تنفيذ جسم الحلقة قبل أن نصل إلى الحالة التي لا يتحقق عندها شرط الحلقة توفر لغة C# أربعة أوامر لذلك تعرفنا على ثلاثة منها مسبقا:

1- الأمر break والذي يؤدي إلى إنهاء الحلقة مباشرة والانتقال للأسطر البرمجية التي تلي الحلقة:

```
int i;
for (i = 1; i <= 10; i++)
{
    if (i == 6)
        break;
    Console.WriteLine("{0}", i);
}
Console.ReadLine();
```

ستقوم هذه الشيفرة بطباعة الأرقام من 1 إلى 5 حيث ستنفذ تعليمة break عندما تصل قيمة i=6.

2- الأمر continue والذي يؤدي إلى مقاطعة دورة الحلقة الحالية والانتقال على الدورة التالية على سبيل المثال.

```
int i;
for (i = 1; i <= 10; i++)
```

```
{
    if (i == 6)
        continue ;
    Console.WriteLine("{0}", i);
}
Console.ReadLine();
```

حيث سوف يتم طباعة الاعداد من 1 إلى عشرة باستثناء الرقم 6 لأنه تم مقاطعة التنفيذ عنده.

3- الأمر goto والذي يسمح بالقفز إلى سطر معنون من الشيفرة وقد سبق وتناولنا هذا الأمر.

ملاحظة: إن الأمر goto يسمح لنا بالقفز من داخل الحلقة إلى خارجها أما العكس فهو غير صحيح.

4- الأمر return والذي يقوم بالقفز من الحلقة ومن التابع الذي يحتويها قم بتنفيذ الشيفرة التالية ولاحظ ماذا سيحدث.

```
int i;
for (i = 1; i <= 10; i++)
{
    if (i == 6)
        return  ;
    Console.WriteLine("{0}", i);
}
Console.ReadLine();
```

الحلقات اللانهائية:

Infinite Loops:

من الممكن أن نعرف حلقات تستمر بالتنفيذ للأبد وتسمى هذه الحلقات بالحلقات اللانهائية وأبسط حلقة لانهائية يمكن أن نكتبها كما يلي:

```
While (true)
{
// code in loop
}
```

إن هذا الوضع هام في بعض الاحيان ويمكننا في حالة كهذه استخدام أمر مثل break للخروج من الحلقة.

على كل حال عندما يحدث ذلك بسبب خطأ برمجي فإن هذا سيسبب إزعاجا بالنسبة إلينا لنأخذ الحلقة التالية على سبيل المثال وهي مشابهة لحلقة for في القسم السابق:

```
int i;
Console.WriteLine("inter a number");
i = Convert.ToInt32(Console.ReadLine());
while (i <= 10)
{
    if ((i % 2) == 0)
        continue;
    Console.WriteLine("{0}", i++);
}
```

لاحظ أن قيمة i لن تزيد إلا عند السطر الاخير من جسم الحلقة و هو ما سينفذ بعد تعليمة continue لكن عند الوصول على الأمر continue سيتم تجاهل جميع الاسطر البرمجية التالية في حسم الحلقة والانتقال إلى دورة جديدة من نفس قيمة i السابق وباعتبار أن باقي قسمة 1 على 2 تساوي 0 فإن الأمر continue سينفذ باستمرار دون تغيير في قيمة i حيث (i=1) إن هذه الحلقة ستتسبب في تجميد النظام لاحظ أنك تستطيع إيقاف التطبيق المجمد بطريقة عادية ولست بحاجة إلى إعادة تشغيل الجهاز من جديد إذا حدث ذلك.

الخلاصة:

Summary:

لقد طورنا معلوماتنا البرمجية في هذا الفصل وذلك بمناقشة بنى عديدة يمكننا استخدامها في شيفرتنا إن استخدام هذه البنى أمر جوهري ولا بد منه عند تصميم التطبيقات في اية لغة برمجية وسوف نجد هذه البنى في معظم الشيفرة البرمجية خلال هذا الكتاب.

لقد تعرفنا في البداية على المنطق البولياني وتعرفنا على العوامل المنطقية والعوامل الخاصة بالبتات والمنطق البولياني هو الأساس الذي ستعتمد عليه البنى التي ناقشناها بعد ذلك.

يقيدنا التفرع في تنفيذ الشيفرة البرمجية بناء على تحقيق شرط معين وبوجود الحلقات سنتمكن من تنفيذ شيفرة برمجية لعدد من المرات حسب حاجتنا ورغبتنا في ذلك عندما تجد تعليمات if ضمن حلقات وحلقات ضمن حلقات ستدرك فائدة تنسيق الشيفرة بالمسافة البادئة التي تحدد لنا بداية الكتلة البرمجية ونهايتها فإن كتبت شيفرتك البرمجية دون جدولة الكتل ستظهر الشيفرة مربكة للغاية وان تتمكن من قراءتها بسهولة (على الرغم من ان تنسيق الشيفرة لا يؤثر ابدا على تنفيذها لا من قريب ولا من بعيد).

الفصل الخامس

المزيد عن المتحولات

لقد تعلمنا الكثير عن لغة C# حتى الآن وقد حان الوقت للعودة إلى موضوع مهم تناولناه في الفصل الثالث ألا وهو المتحولات.

الموضوع الأول الذي سنتناوله هنا هو تحويل النوع (Type Conversion) وسنتحدث هنا حول تحويل القيم من نوع بيانات إلى اخر لقد تعرفنا في تطبيقات سابقة على بعض أشكال التحويل إلا أننا سنتناول تحويل النوع بتفصيل أكبر في هذا الفصل إن فهمك العميق لهذا الموضوع سيساعدك على معرفة ما يحدث عند خلط الأنواع مع بعضها البعض ضمن التعابير والتحكم بطريقة معالجة البيانات وبالتالي تجنب الأخطاء والمفاجئات غير السارة في الشيفرة البرمجية.

بعد أن نغطي هذا الموضوع سنتناول أنواعا جديدة من المتحولات:

1- التعدادات (enumerations) وهي أنواع المتحولات التي لا تقبل إلا مجموعة محددة من القيم فقط.
2- البنى (structs) وهي أنواع المتحولات المركبة والمكونة من مجموعة من أنواع المتحولات الأخرى.
3- المصفوفات (arrays) وهي أنواع المتحولات التي تحتوي على قيم عديدة من نفس النوع بحيث يمكن الوصول إلى هذه القيم عبر موقعها (دليلها) ضمن المصفوفة.
4- اللوائح (Lists) وتشبه المصفوفة إلى حد ما إلا أنها ديناميكية وليست ثابتة.

إن أنواع البيانات هذه أكثر تعقيدا من الأنواع البسيطة التي تناولناها في السابق إلا أنها مفيدة جدا وتسهل البرمجة وتبسطها.

تحويل النوع:

Type Conversion:

لقد ذكرنا مع بداية هذا الكتاب حقيقة أن جميع البيانات بغض النظر عن نوعها تمثل بتتابع من البتات وهذه البتات ليست أكثر من مجرد تتابع من الأصفار والواحدات إن مفهوم المتحول يأتي من كيفية تفسير البيانات بهذا التمثيل لتوضيح ذلك أكثر لنأخذ نوع البيانات البسيط char مثلا يمثل هذا النوع رمزا بنظام تشفير Unicode وذلك باستخدام رقم موافق لهذا الرمز في جدول نظام التشفير وعلى الرغم من أننا

نتعامل مع قيم من هذا النوع على شكل رمز ما مثل "a" إلا أن تخزين هذا الرمز يكون على هيئة قيمة رقمية من نوع ushort.

هذا يعني أنه لا توجد هناك أية مشكلة من تحويل المتحولات من نوع ushort إلى نوع char وبالعكس إن ذلك صحيح ولكن الأمر ليس بهذه البساطة مع كل الأنواع فتتابع الأصفار والواحدات لنوع متحول ما ليس بالضرورة أن يشير إلى ذاته للتابع نفسه من الأصفار والواحدات لنوع آخر من المتحولات.

على كل حال لتحويل الانواع شكلان:

1- التحويل المطلق (Implicit Conversion) حيث أن التحويل من النوع A إلى النوع B ممكن في جميع الأحوال والقواعد الموافقة لهذا التحويل بسيطة لدرجة تمكننا من الوثوق بمترجم C# للقيام بعملية التحويل.
2- التحويل الصريح (Explicit Conversion) حيث أن التحويل من نوع A إلى النوع B ممكن في حالات معينة وأن قواعد التحويل معقدة لدرجة نحتاج فيها إلى تدخل برمجي من قبلنا قبل أن يتم المترجم مهمته في ذلك.

التحويلات المطلقة(الضمنية):

Implicit Conversions:

لا تتطلب التحويلات المطلقة أي عمل من طرفنا أو أية شيفرة إضافية لنأخذ الشيفرة التالية مثلا:

var1=var2;

يمكن لهذا الإسناد أن يشتمل على تحويل مطلق وذلك إذا كان نوع المتحول var2 قابلا للتحويل المطلق إلى نوع المتحول var1 ويمكن أن يكون ذلك إسناد طبيعيا دون أي تحويل لنوع المعطيات.

لنأخذ مثالا يصف لنا التحويل المطلق:

لقد ذكرنا في البداية أن النوعين ushort وchar هما نوعان مرتبطان ببعضهما لدرجة كبيرة فكلاهما يخزن القيم من 0 إلى 65535 يمكننا تحويل القيم بين هذين النوعين بشكل مطلق كما في الشيفرة التالية:

```
ushort destinationVar;
char sourceVar = 'a';
destinationVar =sourceVar;
Console.WriteLine("sourceVar Value is:{0}", sourceVar);
Console.WriteLine("destinationVar Value is;{0}",destinationVar );
```

هنا سيتم اسناد قيمة المتحول sourceVar إلى المتحول destinationVar بعد ذلك سيتم طباعة قيمة المتحولين سيظهر الخرج كما يلي:

الشكل (1-5)

على الرغم من أن كلا المتحولين يحتفظان بالمعلومات نفسها إلا أن لكل متحول تفسيرا مختلفا للمعلومات المخزنة ضمنه بحسب نوعه.

هناك الكثير من التحويلات المطلقة للأنواع البسيطة ولكن ليس للنوعين bool وstring أية تحويلات مطلقة يبين الجدول التالي إمكانيات التحويلات المطلقة للأنواع الرقمية (أي التي يمكن للمترجم التكفل بعمليات التحويل تلك):

النوع	الأنواع
Byte	Short , ushort , int , uint , long , ulong , float , double , decimal
Sbyte	Short , int , long , float , double , decimal
Short	int, long, float , double , decimal
Ushort	int , uint , long , ulong , float , double , decimal
Int	long, float , double , decimal
Uint	long , ulong , float , double , decimal
Long	float , double , decimal
Ulong	float , double , decimal
Float	double
char	ushort , int , uint , long , ulong , float , double , decimal

لا تقلق فأنت لست بحاجة إلى حفظ هذا الجدول لكي تتمكن من معرفة الأنواع التي يمكنك استخدام التحويل المطلق بين قيم متحولاتها لكن هناك أمر يجدر بك أن تلاحظه من خلال هذا الجدول وهو أن مجال قيم الأنواع التي يمكنك التحويل إليها يجب أن يلائم مجال قيم النوع الذي ستحوله ويمكننا أن نستخلص قاعدة للتحويل المطلق كما يلي: **لأي نوع A فإنه يمكننا تحويله تحويلا مطلقا إلى نوع أخر B وذلك إذا أمكننا تمثيل جميع القيم الممكن تمثيلها في النوع A ضمن النوع B.**

إن هذا منطقي جدا فإذا حاولنا أن نسند قيمة إلى متحول يقع خارج مجال نوع ذلك المتحول فإن هذا سيتسبب في مشكلة على سبيل المثال يستطيع النوع short أن يحفظ القيم من 0 إلى 32767 أما النوع byte فإن أقصى قيمة يمكنه حفظها هي 255 وبالتالي سوف نواجه مشكلة حقيقية عند تحويل قيمة من نوع short إلى نوع byte فإن كانت تلك القيمة واقعة بين 256 و32767 فسوف لن يتمكن النوع byte من استيعاب هذه القيمة لأنها تقع خارج مجاله.

على كل حال قد تقول إن المشكلة التي نتحدث عنها هنا ليست بالضرورة أن تحدث فإذا كنت تعلم أن قيمة المتحول من نوع short لن تزيد عن 255 فإنك ستتمكن من تحويل المتحول على النوع byte صحيح؟

إن ابسط جواب لهذا السؤال هو بالتأكيد يمكنك ذلك إلا أن الجواب الاكثر تحديدا هو أنك تستطيع ذلك ولكن بواسطة التحويل الصريح (Explicit Conversion) إن التحويل الصريح للأنواع مشابه لقولنا:

"حسنا أنا أعلم أنك حذرتني من القيام بذلك إلا أنني سأتحمل مسؤولية ما سيحدث".

التحويلات الصريحة:

Explicit Conversions:

وكما يشير الاسم فإن هذه التحويلات لا تحدث إلا عندما نطلب من المترجم تحويل القيمة من نوع إلى آخر بصراحة وبسبب ذلك فإن هذا النوع من التحويلات يتطلب شيفرة إضافية كما وأن هذه الشيفرة تختلف من حالة لأخرى وذلك بسبب منهجية التحويل ولكن قبل أن نتعرض لأية شيفرة تحويل صريح لنلق نظرة على ما سيحدث عندما لا نستخدم التحويل الصريح.

على سبيل المثال سنقوم بتعديل الشيفرة السابقة إلى الشيفرة التالية بهدف إجراء تحويل من النوع short إلى byte:

```
byte  destinationVar;
short sourceVar = 8;
destinationVar =sourceVar;
Console.WriteLine("sourceVar Value is:{0}", sourceVar);
Console.WriteLine("destinationVar Value is;{0}",destinationVar );
```

إذا حاولنا ترجمة هذه الشيفرة فسوف نستقبل الخطأ التالي:

Cannot implicitly convert type 'short' to 'byte'. An explicit conversion exists

لحسن الحظ فإن مترجم C# ذكي لدرجة يستطيع فيها كشف التحويلات الصريحة المفقودة.

لكي نستطيع ترجمة هذه الشيفرة يجب أن نظيف عبارة برمجية للقيام بالتحويل الصريح والطريقة الأبسط للقيام بذلك في هذا السياق تقضي بتشكيل (cast) متحول short ضمن متحول byte إن ما نقصده بالتشكيل casting هو تحويل البيانات من نوع لأخر بصورة قسرية ويأخذ هذا التحويل الصيغة التالية:

(destinationType) sourceVar

حيث سيتم تحويل المتحول sourceVar إلى النوع destinationType.

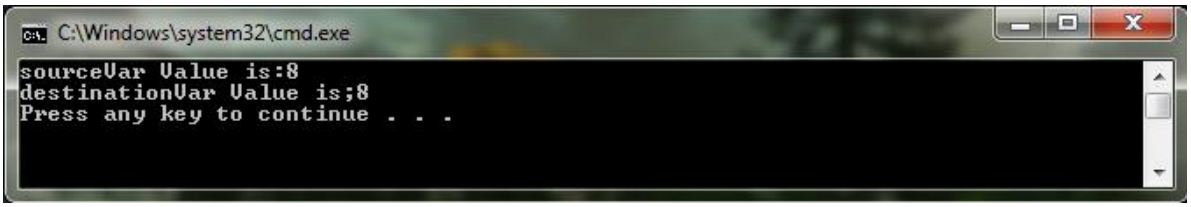
ملاحظة:

لاحظ أن تطبيق هذه الصيغة محصور في حالات محددة فالأنواع التي ليست بينها أية علاقات لا يمكننا استخدام التشكيل (casting conversion) معها

وبناء على ذلك يمكننا أن نعدل المثال السابق باستخدام صيغة التشكيل التي تجبر على التحويل من short إلى byte كما يلي:

```
byte  destinationVar;
short sourceVar = 8;
destinationVar =(byte)sourceVar;
Console.WriteLine("sourceVar Value is:{0}", sourceVar);
Console.WriteLine("destinationVar Value is;{0}",destinationVar );
```

والذي سينتج عنه الخرج كما في الشكل (5-2):

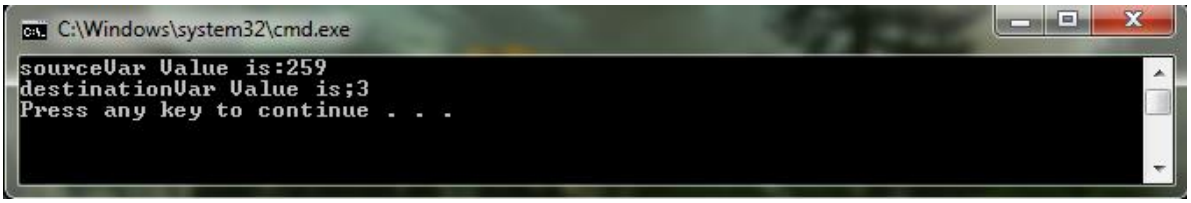


الشكل (5-2)

حسنا ماذا سيحدث عندما نحاول اسناد التحويل الصريح مع قيمة لا تتسع ضمن مجال النوع المراد التحويل إليه؟! الشيفرة التالية توضح ذلك:

```
byte  destinationVar;
short sourceVar = 259;
destinationVar =(byte)sourceVar;
Console.WriteLine("sourceVar Value is:{0}", sourceVar);
Console.WriteLine("destinationVar Value is;{0}",destinationVar );
```

والنتيجة هي كما في الشكل (5-3):



الشكل (5-3)

ماذا حدث؟ إن ما قمنا به هنا هو محاولة إسناد القيمة 259 إلى متحول من نوع byte وهو نوع لا يمكن أن يتقبل قيما أكبر من 255 لكن نتيجة التحويل الصريح ظهرت غريبة نوعا ما حسنا لنلق نظرة على التمثيل الثنائي لهذين الرقمين بالإضافة إلى أقصى قيمة لنوع المعطيات byte:

259=100000011

3=000000011

يمكننا أن نرى ان ما حدث هنا هو فقدان البتات في الطرف الايسر من القيمة 259 إن ما حدث هنا هو محاولة لتخزين القيمة 259 ضمن حيز النوع byte وحسب تعريف النوع byte فإن أكبر قيمة يمكنه استيعابها هي 255 والتي تمثل بثماني بتات بالمقارنة مع القيمة 259 والتي تمثل بتسع بتات وبناء على ذلك سيتم إهمال جميع البتات في أقصى اليسار الواقعة خارج مجال النوع byte وهي بت واحد فقط في

حالتنا هنا إن حدوث تحويل كهذا يمكن أن يؤدي إلى أخطاء جسيمة جدا (تخيل حدوث مثل هذه الهفوات في برنامج محاسبة).

لكي نتجنب الوقوع في هفوات كهذه علينا تفحص قيمة المصدر ومقارنتها مع حدود المتحول الهدف المعلومة سوف نتعلم كيفية القيام بذلك في الفصل القادم عندما نتناول منطق التطبيق والتحكم في سير التنفيذ على كل حال هناك تقنيات أخرى يمكننا الاستفادة منها هنا حيث سنستخدمها لتنبيه النظام حول إمكانية التحويل الصريح القسري في زمن التنفيذ لأشكال كهذه من التحويلات وتعتمد هذه التقنية على تفحص ما يعرف بحالة الطفحان (overflow) حيث أن أية محاولة لوضع قيمة ضمن متحول لا يمكنه استيعابها ستؤدي إلى حدوث حالة الطفحان.

هناك كلمتان مفتاحيتان لإعداد ما نسميه بسياق تفحص الطفحان (overflow checking context) لتعبير ما وهما checked وunchecked يمكننا استخدام هاتين الكلمتين بالصورة التالية:

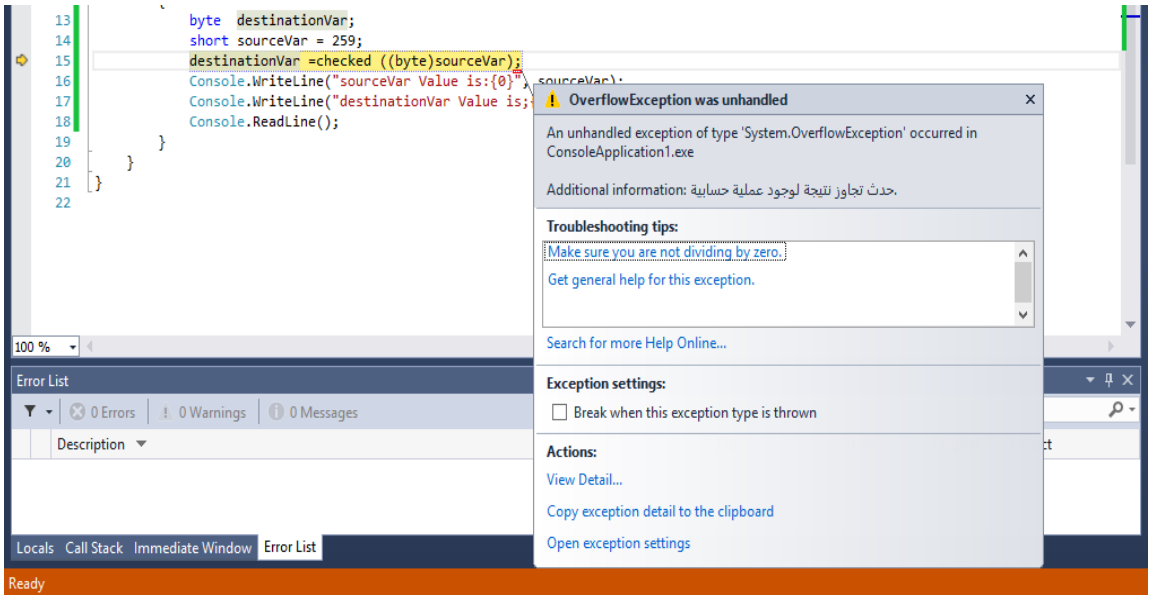
Checked (expression)

Unchecked (expression)

وبالتالي يمكننا تفحص الطفحان في المثال السابق كما يلي:

```
byte  destinationVar;
short sourceVar = 259;
destinationVar =checked ((byte)sourceVar);
Console.WriteLine("sourceVar Value is:{0}", sourceVar);
Console.WriteLine("destinationVar Value is;{0}",destinationVar );
```

عند تنفيذ الشيفرة ستظهر رسالة خطأ كما في الشكل (4-5) والتي تخبرنا بوجود حالة طفحان.



الشكل (4-5)

والآن إذا استبدلنا الكلمة checked بالكلمة unchecked في هذه الشيفرة فسوف نحصل على النتيجة التي حصلنا عليها في السابق وهو مطابق للسلوك الافتراضي عند عدم وضع أي كلمة.

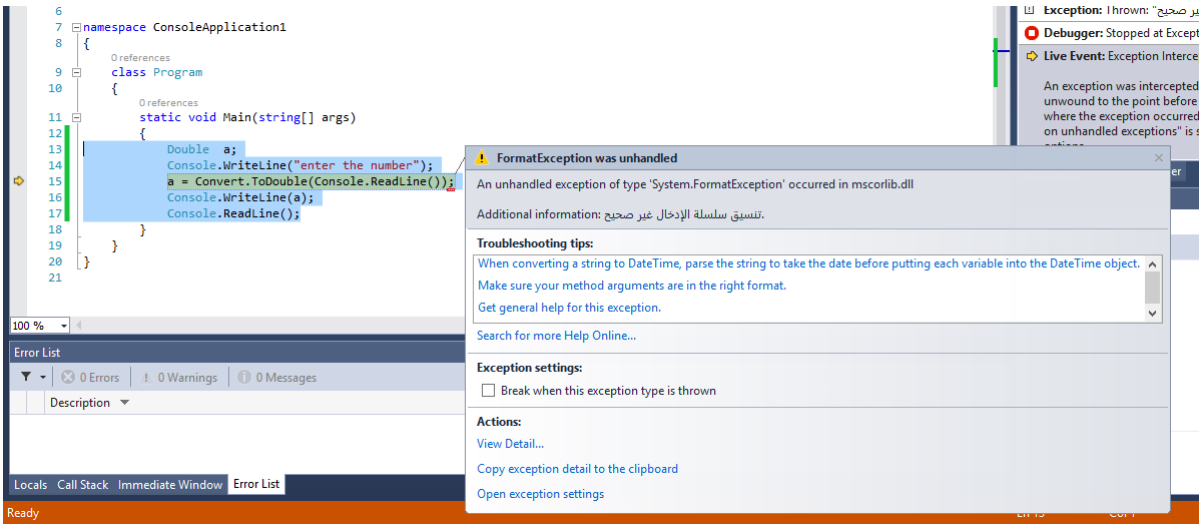
التحويلات الصريحة بواسطة أوامر التحويل:

Explicit Conversions Using the Convert Commands:

إن نمط التحويل الصريح الذي استخدمناه في تطبيقاتنا السابقة في هذا الكتاب مختلف قليلا عما تعلمناه عن التحويل الصريح حتى الآن في هذا الفصل لقد قمنا في السابق بتحويل السلاسل النصية إلى أرقام باستخدام أوامر مثل Convert.ToDouble() وهو أمر لا يمكن تطبيقه على أي سلسلة نصية.

على سبيل المثال: إذا حاولت تحويل سلسلة نصية مثل "Number" إلى قيمة من نوع double باستخدام الأمر Convert.ToDouble() كما في الشيفرة التالية فإن ذلك سوف يؤدي أثناء التنفيذ لظهور رسالة الخطأ كما في الشكل (5-5).

```
Double  a;
Console.WriteLine("enter the number");
a = Convert.ToDouble(Console.ReadLine());
Console.WriteLine(a);
```



الشكل (5-5)

وكما ترى هنا فإن العملية ستخفق وهذا شيء طبيعي ولكي تنجح عملية تحويل كهذه يجب أن تمثل السلسلة النصية تمثيلا شرعيا رقميا ويجب ألا يتسبب هذا الرقم بحالة طفحان.

ملاحظة:

نقصد بالتمثيل الشرعي الرقمي هو أن تتضمن السلسلة النصية علامة اختيارية (سواء كانت – أو +) متبوعة بعدد من الأرقام ومن ثم متبوعة بفاصلة اختيارية متبوعة بعدد من الأرقام متبوعة برمز "e" أو "E" اختياري ثم متبوعة بعلامة (+ أو -) اختياريةمتبوعة بعدد من الأرقام ولا شيء سوى ذلك عدا وجود الفراغات (قبل وبعد هذا التتابع وليس ضمنه) باستخدامنا لجميع هذه الأمور الاختيارية يمكننا أن نكتب سلسلة نصية بالشكل

-5.39594846161 ومعاملتها كقيمة رقمية.

هناك العديد من التحويلات الصريحة التي يمكننا تطبيقها بهذه الطريقة على سبيل المثال:

التحويل	النتيجة
Convert.ToBoolean(val)	تحويل قيمة val إلى النوع bool
Convert.ToByte (val)	تحويل قيمة val إلى النوع byte
Convert.ToChar (val)	تحويل قيمة val إلى النوع char
Convert.ToDecimal(val)	تحويل قيمة val إلى النوع decimal
Convert.ToDouble(val)	تحويل قيمة val إلى النوع double
Convert.ToInt16(val)	تحويل قيمة val إلى النوع Short
Convert.ToInt32(val)	تحويل قيمة val إلى النوع Int
Convert.ToInt64(val)	تحويل قيمة val إلى النوع Long
Convert.ToSByte(val)	تحويل قيمة val إلى النوع SByte
Convert.ToSingle(val)	تحويل قيمة val إلى النوع Float
Convert.ToString (val)	تحويل قيمة val إلى النوع String
Convert.ToUInt16(val)	تحويل قيمة val إلى النوع UShort
Convert. ToUInt32 (val)	تحويل قيمة val إلى النوع ULong

ملاحظة:

لاحظ أن أسماء بعض أوامر التحويل مختلفة عن أسماء الأنواع في C# فعلى سبيل المثال لتحويل قيمة ما إلى النوع int سنستخدم الأمر Convert.ToInt32 يعود ذلك إلى أن هذه الأوامر تمثل جزءا من إطار عمل .NET وليست جزءا من لغة C#.

إن ما يجب الإشارة إليه هنا هو أن هذه الأوامر تتفحص حالة الطفحان دائما وبالتالي فإن الكلمتين checked وunchecked وإعدادات المشروع لا تؤثر على هذه الأوامر.

لنلق نظرة على مثال يغطي العديد من أنواع التحويلات التي تحدثنا عنها في هذا القسم.

تطبيق حول تحويلات الأنواع:

4- قم بإنشاء تطبيق Console جديد باسم Console Convert Commands.
5- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

```
static void Main(string[] args)
{
    short shortresult, shortval = 4;
    int integerval = 67;
    long longresult;
    float floatval = 10.5F;
    double doubleresult, doubleval = 99.999;
    string stringresult, stringval = "17";
    bool boolval = true;
    Console.WriteLine("variabel conversion exampel\n");
    doubleresult = floatval * shortval;
    Console.WriteLine("implicet,->short:{0}*{1}->{2}"
        , floatval, shortval, doubleresult);
    shortresult = (short)floatval;
    Console.WriteLine("explicit,->short:{0}->{1}", floatval, shortresult);
    stringresult = Convert.ToString(boolval) + Convert.ToString(doubleval);
    Console.WriteLine("explicit,->string:\"{0}\"+\"{1}\"->{2}"
        , boolval, doubleval, stringresult);
    longresult = integerval + Convert.ToInt64(stringval);
    Console.WriteLine("mixed,->long:{0}+{1}->{2}"
        , integerval, stringval, longresult);
}
```

6- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 فيظهر الشكل (5-6).

الشكل (5-6)

كيفية العمل:

How it Works:

يتضمن هذا المثال جميع أنواع التحويلات كما ترى ضمن الشيفرة وذلك ضمن عمليات إسناد وتعابير بسيطة شرحناها مسبقا لنبدأ بتناول هذه الشيفرة خطوة بخطوة:

```
doubleResult = floatval * shortval;
```

لقد قمنا هنا بضرب قيمة من نوع short بقيمة من نوع float وفي حالة كهذه حيث لم نحدد فيها تحويلا صريحا فإنه سيتم استخدام التحويل المطلق متى أمكن ذلك والتحويل المطلق المعقول في السطر السابق هو تحويل القيمة من نوع short إلى float.

ويمكننا أن نتجاوز هذا السلوك بحيث نكتب:

```
doubleResult = (short)floatval * shortval;
```

يتم تطبيق التحويلات الصريحة للأنواع بواسطة صيغ التشكيل كما في السطر السابق وتأخذ صيغة التشكيل الأولوية نفسها التي تأخذها العوامل الأحادية مثل ++ (عند استخدامها كبادئة) أي أن لها الأسبقية الأعلى دونا عن جميع العوامل الأخرى.

ملاحظة:

إن هذا لا يعني بالضرورة أن تكون النتيجة من نوع short وبما أن حاصل عملية ضرب قيمتين من نوع short يمكن أن تتجاوز مجال النوع short فإن هذا النوع من العمليات سيعيد قيمة من نوع int بصورة افتراضية.

وعندما نواجه تعابيرا تتضمن أنواع مختلفة فإن التحويلات تحدث مع كل عامل يعالج أولا وذلك بحسب أولوية تلك العوامل أي أن:

```
doubleResult = floatval + (floatval * shortval);
```

إن العمل الذي سينفذ هنا أولا هو * وهذا سيؤدي إلى تحويل القيمة shortval إلى النوع float بعد ذلك ستتم معالجة العامل + والذي لا يتطلب أية عملية تحويل باعتبار أن حدي العامل هما هنا من نوع float (المتحول floatval وقيمة float الناتجة من التعبير floatval * shortval) وأخيرا فإن حاصل التعبير الناتج من نوع float سيحول إلى النوع double وذلك عند اسناد قيمة float إلى متحول من نوع double (وهو المتحول doubleResult).

يظهر هذا النوع من التحويلات معقدا من الوهلة الأولى إلا أنه عندما نقوم بتجزئة التعبير إلى عدة أجزاء سنتمكن من تحديد عمليات التحويل الحاصلة بناء على أولوية العوامل.

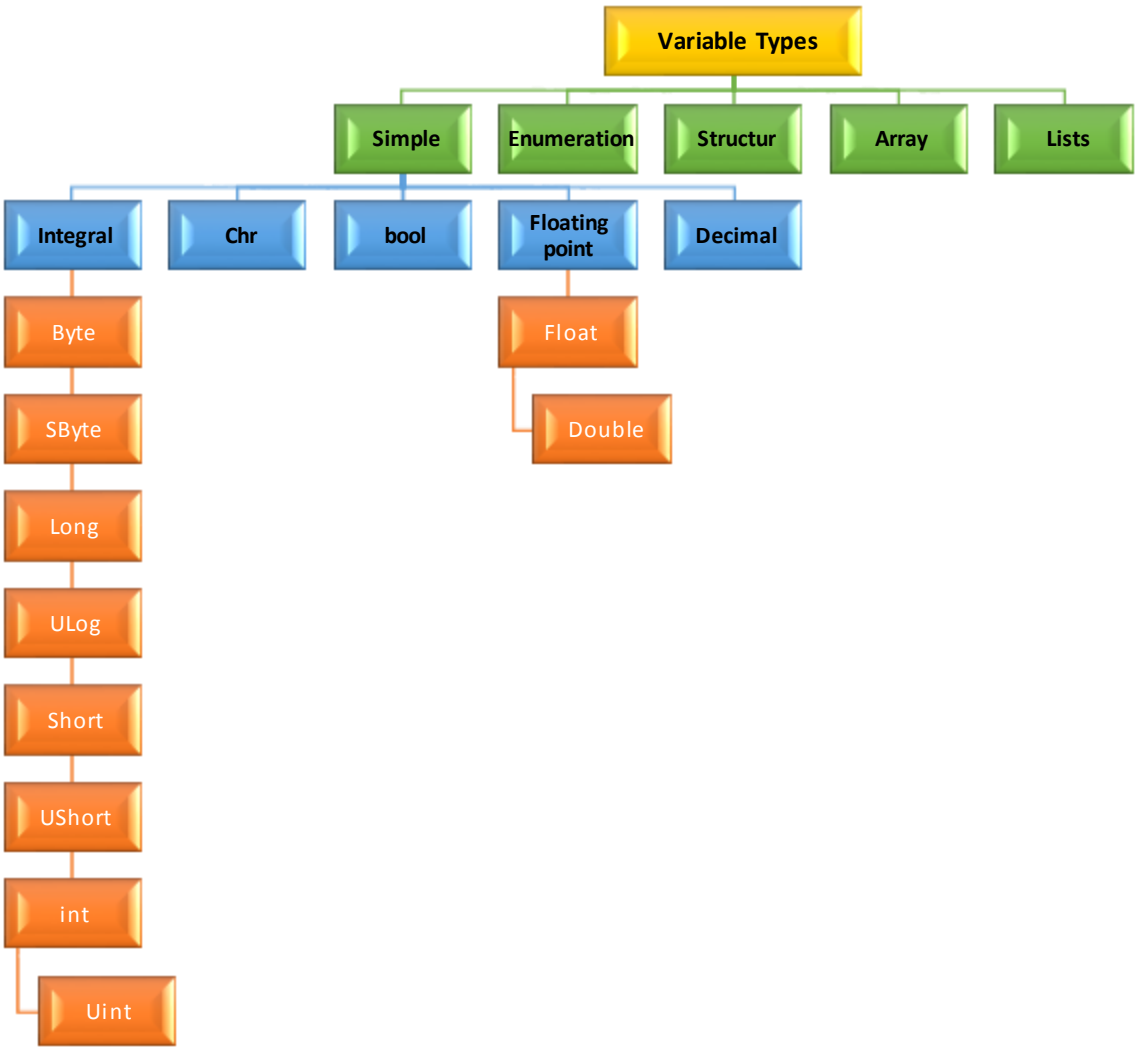
أنواع المتحولات المعقدة:

Complex Variable Types:

لقد تناولنا حتى الآن جميع الأنواع البسيطة المتوفرة في C# تتضمن لغة C# ثلاثة أنواع معقدةللمتحولات لكنها مفيدة جدا و هي:

1- التعدادات أو مجموعات القيم.
2- البنى.
3- المصفوفات.
4- اللوائح

الشكل التالي يبين أنواع البيانات المستخدمة في C#.



التعدادات:

Enumerations:

إن جميع الأنواع التي تناولناها إلى الآن (عدا النوع string) لها مجموعة محددة من القيم المسموح بها وهو ما سميناه بمجال قيم النوع إن هذا المجال كبير جدا ومتصل في أنواع مثل double نقصد بالمجال المتصل أنه من أجل قيمتين قريبتين جدا من بعضهما في هذا المجال فإن هناك قيمة تقع بين هاتين القيمتين إن أبسط هذه الأنواع هو bool والذي لا يتقبل سوى قيمتين فقط: true أو false...

هناك الكثير من الحالات التي نحتاج فيها إلى تعريف متحول لا يتقبل إلا مجموعة ثابتة من القيم على سبيل المثال يمكن أن يكون لدينا النوع orientation مثلا والذي يستطيع تخزين القيم north، south، east، west.

في حالات كهذه يكون استخدام التعدادات هو الحل المثالي تمكننا التعدادات من صنع أنواع جديدة مثل النوع orientation مثلا فهي تسمح لنا بتعريف نوع جديد يتقبل عددا منهيا من القيم التي يمكن أن تأخذها المتحولات التي تنتمي لهذا النوع.

ما نحتاجه للقيام بذلك هو إنشاء نوع جديد باسم orientation يتضمن القيم الأربعة الممكنة السابقة أي أن علينا تعريف النوع الجديد أولا ومن ثم نستطيع استخدامه مع متحولاتنا.

تعريف التعدادات:

Defining Enumeration:

تعرف التعدادات بواسطة الكلمة enum كما في الصيغة التالية:

```
enum typeName
{
Value1,
Value2,
……
ValueN,
}
```

يمكننا بعد ذلك التصريح عن المتحولات لهذا النوع الجديد بالطريقة الاعتيادية:

```
typeName varName;
```

وإسناد القيم إلى هذا المتحول كما يلي:

```
varName=typeName.value;
```

وللتعدادات نوع تحتي (underlying type) وهو نوع القيم في التعداد فكل قيمة في التعداد يمكن أن تخزن على شكل قيمة لهذا النوع التحتي إن النوع الافتراضي هو int ويمكننا أن نحدد نوعا آخر باستخدام التصريح التالي:

```
enum typeName : underlyingType
{
Value1,
Value2,
Value3,
.....
ValueN
}
```

يمكن أن تأخذ التعدادات الأنواع التحتية التالية:

Short , ushort , int , uint , long , ulong , Byte , Sbyte

إن كل قيمة موجودة ضمن التعداد تقابل قيمة من النوع التحتي ويتم ذلك وفقا لترتيب توضع القيم ضمن التعداد بدا من الصفر هذا يعني أن للقيمة value1 القيمة 0 وفقا للنوع التحتي والقيمة value2 لها القيمة 1 وفقا للنوع التحتي وهكذا يمكننا أن نتجاوز ذلك وأن نضع قيما من عندنا وذلك باستخدام عامل الاسناد كما يلي:

```
enum typeName : underlyingType
{
Value1=actualVal1,
Value2=actualVal2,
.....
ValueN=actualValN
}
```

وبالإضافة لذلك يمكن أن تأخذ Value1, Value2..... قيما متشابهة كما يلي:

```
enum typeName : underlyingType
{
Value1=actualVal1,
Value2= Value1,
Value3,
.....
ValueN=actualValN
}
```

لاحظ أننا هنا لم نحدد قيمة صريحة لـ Value3 و بالتالي ستأخذ قيمة تلقائية من النوع التحتي وتلك القيمة هي آخر قيمة تم تحديدها بصورة صريحة مضافا إليها واحد على سبيل المثال ستأخذ القيمة Value4 نفس قيمة Value2:

```
enum typeName : underlyingType
{
Value1=actualVal1,
Value2,
Value3= Value1,
Value4
.....
ValueN=actualValN
}
```

تحذير:

لا يمكننا إسناد القيم بصورة تعاوديه دائرية كما في الشيفرة:

```
enum typeName : underlyingType
{
Value1= Value2,
Value2= Value1,
}
```

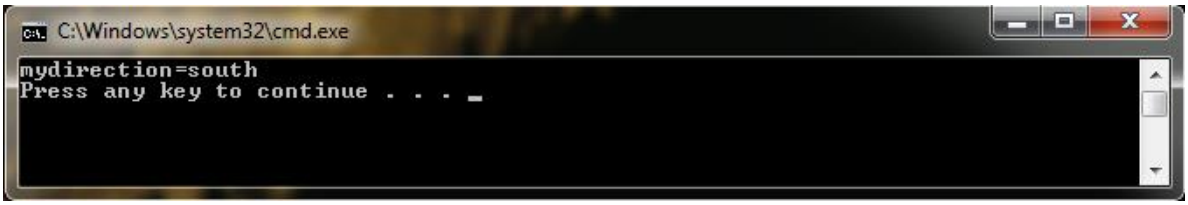
حيث سيتسبب ذلك في حدوث خطأ.

تطبيق حول التعدادات:

1- قم بإنشاء تطبيق Console جديد باسم Console Enumeration.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

```
namespace Console_Enumeration
{
    enum orientation : byte
    {
        north = 1,
        south = 2,
        east = 3,
        west = 4
    }
    class Program
    {
        static void Main(string[] args)
        {
            orientation mydirection = orientation.south;
                Console .WriteLine ("mydirection="+mydirection);
        }
    }
}
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 فيظهر الشكل (7-5).



الشكل (7-5)

4- اخرج من التطبيق وقم بتعديل الشيفرة كما يلي:

```
namespace Console_Enumeration
{
    enum orientation : byte
    {
        north = 1,
        south = 2,
        east = 3,
        west = 4
    }
    class Program
    {
        static void Main(string[] args)
        {
            byte directionByte;
            string directionString;
            orientation myDirection = orientation.south;
```

```
            Console .WriteLine ("mydirection="+myDirection);
            directionByte = (byte)myDirection;
            directionString = Convert.ToString(myDirection);
            Console.WriteLine("byte equivalent={0}", directionByte);
            Console.WriteLine("string equivalent={0}", directionString );
        }
    }
}
```

5- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 فيظهر الشكل (5-8).

الشكل (5-8)

كيفية العمل:

How it Works:

لقد قمنا هنا بتعريف نوع تعداد جديد باسم orientation إن أول ما نلاحظه حول هذه الشيفرة هو أن تعريف النوع الجديد موجود ضمن فضاء الأسماء مباشرة وليس ضمن باقي الشيفرة يعود ذلك إلى توفير هذا النوع لكامل فضاء الأسماء وليس لجزء معين من المشروع ويمكن للمترجم الوصول إلى تعريف النوع متى احتاج لذلك من خلال فضاء الأسماء.

يستعرض هذا المثال أساسيات تعودنا على رؤيتها في الفصول السابقة فلقد استعرض الطريقة الأساسية لإنشاء متحول لنوع البيانات الجديد وإسناد قيمة لهذا المتحول ومن ثم إخراج قيمة المتحول على الخرج.

أما في المرحلة التالية فقد عدلنا الشيفرة بهدف تحويل قيمة المتحول من نوع orientation إلى النوع byte لاحظ أن علينا استخدام التحويل الصريح هنا فعلى الرغم من أن النوع التحتي للنوع orientation هو byte إلا أنه مازلنا نحتاج لاستخدام التشكيل byte كي نتمكن من تحويل قيمة myDirection إلى النوع byte.

```
directionByte = (byte)myDirection;
```

علينا استخدام التحويل الصريح ذاته إذا عكسنا الوضع فإذا أردنا تحويل قيمة من نوع byte إلى orientation فإننا سنستخدم التشكيل orientation فعلى سبيل المثال إذا كان لدينا متحول من نوع byte باسم myByte وأردنا تحويل قيمته إلى النوع orientation وإسناد القيمة الناتجة إلى المتحول myDirection فإننا سنكتب:

```
myDirection = (orientation)myByte;
```

وبالطبع فإنه في هذه الحالة علينا الانتباه إلى أن ليست جميع قيم byte لها قيم مقابلة في النوع orientation يستطيع تخزين قيم byte أخرى غير تلك الأربعة إلا أن هذا قد يؤثر على منطق التطبيق لاحقا.

ولكي نتمكن من الحصول على القيمة النصية من قيمة تعداد سنستخدم الأمر Convert.ToSring():

```
directionString = Convert.ToString(myDirection);
```

إن استخدام التشكيل string لا يعمل هنا باعتبار أن المعالجة المطلوبة هنا أكثر تعقيدا من مجرد وضع البيانات المخزنة في متحول تعداد من نوع string.

وكبديل لذلك يمكننا استخدام الأمر ToString() على المتحول نفسه فالشيفرة التالية تعطي نفس النتيجة التي سنحصل عليها باستخدام Convert.ToSring():

```
directionString = myDirection. ToString();
```

إن تحويل قيمة نصية إلى قيمة من نوع مجموعة قيم هو أمر ممكن فيما عدا أن الصيغة اللازمة لتحقيق ذلك معقدة نوعا ما فهناك أمر خاص لهذا النوع من التحويل وهو Enum.Parse() ويمكننا استخدامه كما يلي:

(enumerationType)Enum.Parse(typeof(enumerationType),enumeratioValueString);

هنا تم استخدام عامل جديد وهو typeof والذي يعيد لنا نوع بيانات الحد يمكننا استخدام هذه الصيغة مع النوع orientation كما يلي:

```
namespace Console_Enumeration
{
    enum orientation : byte
    {
        north = 1,
        south = 2,
        east = 3,
        west = 4
    }
    class Program
    {
        static void Main(string[] args)
        {
            string myString;
            Console.WriteLine("enter name direction:");
                myString = Convert.ToString(Console.ReadLine());
            orientation myDirection1 =(orientation )Enum .
                Parse (typeof (orientation),myString);
            Console.WriteLine(myDirection1);
                Console.ReadLine();
        }
    }
}
```

الشكل(9-5)

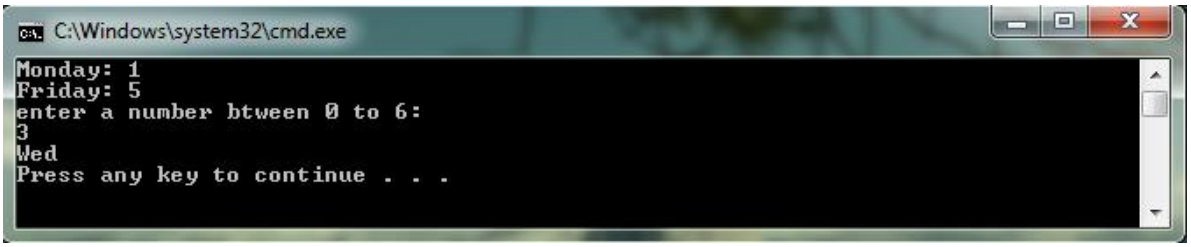
وبالطبع فإنه ليست جميع السلاسل النصية تستطيع أن تشكل قيما لـ orientation فإذا كانت قيمة المتحول النصي myString هي "left" مثلا فإنه ليست هناك أية قيمة موافقة ضمن التعداد orientation وفي حالة كهذه سنحصل على خطأ من المترجم وكما في كل شيء ضمن C# فإن هذه القيم متحسسة لحالة الحرف أيضا وبالتالي إذا كانت قيمة المتحول myString هي "North" فإننا سنحصل على خطأ أيضا لان North ليست north في C#.

تطبيق أخر حول التعدادات:

1- قم بإنشاء تطبيق Console جديد باسم Console enum days.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

```
namespace Console_enum_days
{
    enum Days
    {
        Sun,
        Mon,
        tue,
        Wed,
        thu,
        Fri,
        Sat
    }
    class Program
    {
        static void Main(string[] args)
        {
            int number;
            int WeekdayStart = (int)Days.Mon;
            int WeekdayEnd = (int)Days.Fri;
            Console.WriteLine("Monday: {0}", WeekdayStart);
            Console.WriteLine("Friday: {0}", WeekdayEnd);
            Console.WriteLine("enter a number btween 0 to 6:");
            number = Convert.ToInt32(Console.ReadLine());
            Days myDay = (Days)number;
            Console.WriteLine(myDay);
        }
    }
}
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 ومن ثم قم بإدخال رقم يقع بين 0 و 6 و ليكن 3 فيظهر الشكل (5-10).



الشكل (10-5)

كيفية العمل:

How it Works:

في الشيفرة السابقة قمنا بالتصريح عن تعداد باسم Days يحوي أيام الاسبوع ضمن فضاء الاسماء namespace ومن ثم قمنا بالتصريح عن ثلاث متحولات (number،WeekdayStart، WeekdayEnd) من النوع int حيث اسند للمتحول WeekdayStart رقم يوم الاثنين ضمن التعداد وللمتحول WeekdayEnd رقم يوم الجمعة ضمن التعداد ومن ثم قمنا بإظهار قيم الاسناد على شاشة Console باستخدام الشيفرات:

```
Console.WriteLine("Monday: {0}", WeekdayStart);
Console.WriteLine("Friday: {0}", WeekdayEnd);
```

ومن ثم طلبنا من المستخدم إدخال رقم يقع بين 0 و6 باستخدام الشيفرة:

```
Console.WriteLine("enter a number btween 0 to 6:");
```

ليقوم بعد ذلك البرنامج بإسناد قيمة الدخل إلى المتحول number بعد تحويل قيمة الدخل إلى النوع int32 باستخدام الشيفرة:

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

ومن ثم قمنا بالتصريح عن متحول جديد باسم myDay من النوع Days وأسندنا له قيمة المتحول number بعد تحويله للنوع Days قسرا باستخدام الشيفرة:

```
Days myDay = (Days)number;
```

ومن ثم قمنا بإظهار قيمة المتحول myDay على شاشة Console باستخدام الشيفرة:

```
Console.WriteLine(myDay);
```

البنى:

Structures:

الصنف الثاني من أنواع المتحولات في هذا الفصل هو البنية struct (اختصارا لكلمة structure) تمثل البنية أجزاء مختلفة من البيانات يمكن ان يكون لكل جزء نوع مختلف فهي تمكننا من تعريف أنواعنا الخاصة من المتحولات بالاعتماد على هذه البنية على سبيل المثال لنفرض اننا نود حفظ مسار لموقع ما

بدءا من نقطة محددة حيث أن هذا المسار مؤلف من اتجاه ومسافة محسوبة بالأميال ولتبسيط الأمور فإننا سنفترض أن الاتجاه هو واحد من الاتجاهات الأربعة المعروفة ضمن التعداد orientation في القسم السابق و أن المسافة بالأميال ممثلة بالنوع double.

والأن يمكننا استخدام متحولين منفصلين لتمثيل هذا المسار كما في الشيفرة التالية:

```
Orientation myDirection;
Double myDistance;
```

ليس هناك أي خطا فيما قمنا به هنا إلا أنه سيكون من الأبسط لو استخدمنا متحول وحيد لحفظ هذه المعلومات.

تعريف البنى:

Defining Structs:

تعرف البنى باستخدام الكلمة struct كما يلي:

```
Struct < typeName >
{
< memberDeclarations >
}
```

يتضمن القسم < memberDeclarations > تصريحات المتحولات (تسمى هذه المتحولات بأعضاء البيانات (date members) لهذه البنية) بنفس الأسلوب المعتاد تقريبا فكل تصريح لعضو سيأخذ الصيغة التالية:

```
< accessibility > <type> <name>;
```

إن <name> يمثل اسم العضو أي المتحول و <type> يمثل نوع العضو و أما <accessibility> فهو يمثل مدى الوصول إلى هذا الموضوع من خارج البنية.

لنأخذ مثالا حيا للبنية:

```
Struct route
{
Public orientation direction;
Public double distance;
}
```

لقد عرفنا هنا بنية باسم route وبالتالي فإن route يمثل الأن نوعا جديدا من أنواع المتحولات وأما direction وdistance فهي أعضاء تنتمي إلى البنية route وبالتالي يمكننا أن نستخدم هذه البنية بالتصريح عن متحول من نوع route كالمعتاد:

Route myRoute;

ملاحظة:

سنتحدث عن مدى الوصول scope للمتحولات في الفصل التالي فلا تكترث بها الأن.

أن myRoute بحد ذاته لا يأخذ قيمة حرفية مباشرة وإنما إسناد القيم الحرفية يتم على مستوى أعضاءه أي يمكننا استخدام المتحول myRoute بالشكل:

myRoute.direction = orientation.north;

myRoute.distance = 2.5;

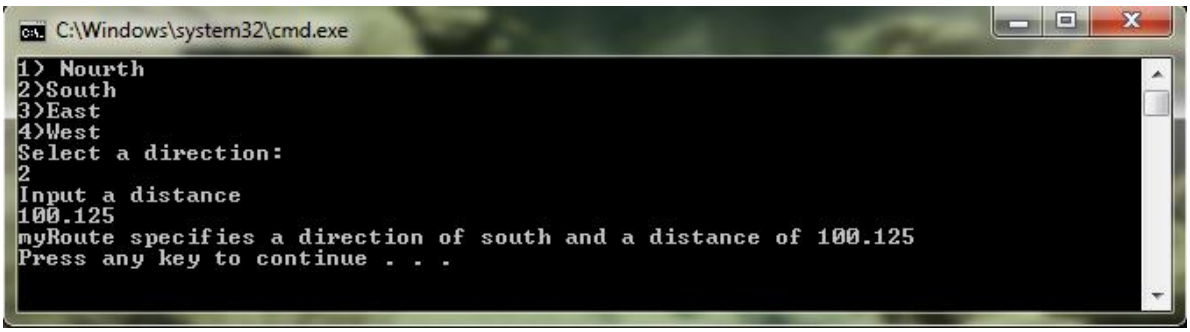
تطبيق حول استخدام البنى:

1- قم بإنشاء تطبيق Console جديد باسم Console Struct direction.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

```
namespace Console_Struct_direction
{
    enum orientation : byte
    {
        north = 1,
        south = 2,
        east = 3,
        west = 4
    }
    struct route
    {
        public orientation direction;
        public double distance;
    }
    class Program
    {
        static void Main(string[] args)
        {
            route myRoute;
            int myDirection;
            double myDistance;
            Console.WriteLine("1) Nourth\n2)South\n3)East\n4)West");
            do
            {
                Console .WriteLine ("Select a direction:");
                myDirection =Convert .ToInt32 (Console .ReadLine ());
            }
            while ((myDirection <1)||(myDirection >4));
            Console .WriteLine ("Input a distance");
            myDistance =Convert .ToDouble (Console .ReadLine ());
```

```
            myRoute .direction =(orientation )myDirection ;
            myRoute .distance =myDistance ;
            Console .WriteLine ("myRoute specifies a direction of {0} and "+
                "a distance of {1}",myRoute .direction ,myRoute .distance );
        }
    }
}
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 ومن ثم قم بإدخال رقم الجهة ومن ثم قم بإدخال المسافة فيظهر الشكل (11-5).



الشكل (11-5)

كيفية العمل:

How it Works:

يتم التصريح عن البنى كالتعدادات خارج الجسم الرئيسي للشيفرة التنفيذية لقد قمنا بالتصريح عن البنية route ضمن فضاء أسماء المشروع مع مجموعة القيم orientation:

```
enum orientation : byte
{
    north = 1,
    south = 2,
    east = 3,
    west = 4
}
struct route
{
    public orientation direction;
    public double distance;
}
```

إن الجسم الرئيسي للشيفرة التالية له بنية مشابهة لبعض الأمثلة و التطبيقات التي رأيناها مسبقا حيث أن الشيفرة التالية طلبت من المستخدم إدخالا ما ثم قامت بعرض هذا الإدخال لقد قمنا هنا بعملية تحقق من المستخدم بسيطة و ذلك بوضع قيمة الاتجاه التي سيدخلها المستخدم ضمن حلقة من نوع do….while

وبالتالي فإن هذه الحلقة ستمنع أي قيمة سيدخلها المستخدم ليست 1 أو 2 أو 3 أو 4 حصرا وفي حال أدخل المستخدم قيمة مخالفة فسيعاد التطبيق سؤال المستخدم عن الإدخال من جديد وسبب تحديد هذه القيم هو أنها القيم الوحيدة التي تمثل الاتجاهات المعرفة ضمن التعداد orientation.

إن النقطة المثيرة في هذه الشيفرة متمركزة بالطريقة التي تشير فبها إلى أعضاء البنية route في المتحول myRoute.

```
myRoute .direction =(orientation )myDirection ;
myRoute .distance =myDistance ;
```

و يمكننا ببساطة الحصول على قيمة دخل المستخدم ووضعها ضمن العضو myRoute .distance باستخدام السطر:

```
myRoute .distance = Convert .ToDouble (Console .ReadLine ());
```

إن أي وصول إلى أعضاء البنية يتم بالصورة ذاتها والتعابير التي لها الشكل structVar.memberVar تمثل متحولا من نفس نوع memberVar.

وبالتالي ووفقا للشيفرة السابقة يمكننا كتابة:

```
Double anotherDistance;

anotherDistance = myRoute.distance*2.4;
```

حيث سيتضمن المتحول anotherDistance حاصل ضرب قيمة العضو myRoute.distance في 2.4.

تطبيق أخر حول البنى:

سوف نقوم في هذا التطبيق ببناء بنية تسمح بتخزين اسم المستخدم (name) وعنوانه (adresse) وعمره (age) ومن ثم تقوم بإظهار هذه البيانات على شاشة Console وسأترك لك التعليق وقراءة هذه الشيفرة بنفسك لا تقلق فلقد أصبحت قادرا على قراءة شيفرات كهذه بشكل جيد.

1- قم بإنشاء تطبيق Console جديد باسم Console Struct Persons.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

```
namespace Console_Struct_Persons
{
    struct persons
    {
        public string name;
        public string adresse;
        public short age;
    }
    class Program
    {
        static void Main(string[] args)
        {
            persons  Person;
            Console.WriteLine("input your Name");
            Person.name = Console.ReadLine();
            Console.WriteLine("input your Adresse");
            Person.adresse = Console.ReadLine();
            Console.WriteLine("input your Age");
            Person.age = Convert.ToInt16(Console.ReadLine());
```

```
            Console.WriteLine("your Name is:{0} , your Adresse is: {1} " +
            " , your age is: {2}", Person.name,
            Person.adresse, Person.age);
        }
    }
}
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 ومن ثم قم بإدخال الاسم والعنوان والعمر فيظهر الشكل (12-5).

الشكل (12-5)

المصفوفات:

Arrays:

هناك شيء مشترك لجميع الأنواع التي تعرفنا عليها حتى الأن ألا وهو أن جميعها لا يستطيع تخزين سوى قيمة واحدة فقط (أو مجموعة واحدة من القيم وذلك بالنسبة للبنى) نحتاج في بعض الأحيان إلى حفظ الكثير من البيانات التي تنتمي إلى نفس النوع وبحيث تكون مرتبطة مع بعضها البعض بصورة أو بأخرى.

على سبيل المثال لنفترض أننا نود حفظ أسماء أصدقائنا ضمن متحولات لاستخدام هذه الأسماء لغرض ما عندئذ سنستخدم متحولات كما يلي:

```
String friendName1 = "Mohamed ALmesri";
String friendName2 = "Nader ALali";
String friendName3 = "Omar Ahmad";
```

ولكن ألا تؤيدني بإن استخدام هذه الطريقة غير عملي أبدا؟

فإذا فرضنا أن لدينا عشرة أسماء نود حفظها بنفس الصورة عندئذ فإننا بحاجة على عشر متحولات! والأسوأ من ذلك إذا كان لدينا مائة اسم وأردنا أن نعدل فيها (كأن نضع عبارة .Mr في بداية كل اسم) عندئذ فإننا بحاجة إلى كتابة سطر برمجي للقيام بذلك حيث أن الحلقات لا تخدمنا كثيرا هنا.

الحل البديل لهذا الصداع يقتضي باستخدام المصفوفة array فالمصفوفات عبارة عن قوائم مفهرسة من المتحولات موضوعة ضمن متحول وحيد على سبيل المثال لنفترض أن لدينا مصفوفة تحتفظ بالأسماء

الثلاثة السابقة ولنسميها بالاسم friendNames عندئذ يمكننا الوصول إلى عناصر المصفوفة بتحديد دليل العنصر أي موقعه من المصفوفة أو فهرسه (index) وذلك وفقا للصيغة التالية:

```
friendNames[<index>]
```

حيث سنضع دليل العنصر ضمن الأقواس "[ ]".

إن دليل العنصر عبارة عن عدد صحيح حيث يبدأ الترقيم في المصفوفة من الرقم 0 وهذا يعني أن العنصر الأول من المصفوفة له الدليل 0 والعنصر الثاني من المصفوفة له الدليل 1 وهكذا وبالتالي إذا أردنا أن نطبع جميع عناصر المصفوفة friendNames فإننا سنستخدم حلقة for كما يلي:

```
for (int i=0 ; i<3 ;i++)
{
Console.WriteLine("Name with index of {0}:{1},i", friendNames[i] );
}
```

للمصفوفات نوع أساسي (base type) واحد أي أن جميع عناصر المصفوفة تنتمي على هذا النوع وفي حالة مصفوفة أسماء أصدقائنا friendNames السابقة فإنه من الواضح أن هذه المصفوفة من نوع string.

التصريح عن المصفوفات:

Declaring Arrays:

يمكننا التصريح عن المصفوفة وفقا للصيغة التالية:

```
<baseType> [] <name>;
```

إن هذا مشابه جدا للتصريح عن المتحولات العادية والفارق هنا فقط في وجود القوسين "[ ]" بعد نوع المتحول.

يمكن لـ <baseType> أن تمثل أي نوع كان سواء كان نوع بسيطا أو تعداد أو بنية أما <name> فتمثل اسم المصفوفة (أو اسم متحول المصفوفة).

ولكي نتمكن من استخدام المصفوفة فإن علينا أن نهيئها فلا يمكننا الوصول إلى عناصر المصفوفة أو إسناد القيم إليها بالشكل:

```
int[] myIntArray;
myIntArray[10] = 5;
```

هناك طريقتان لتهيئة المصفوفة إما أن نحدد المحتوى الكامل للمصفوفة مباشرة بهيئة العناصر الحرفية أو يمكننا تحديد حجم المصفوفة ومن ثم استخدام الكلمة new لتهيئة جميع عناصر المصفوفة.

إن تحديد محتوى المصفوفة باستخدام القيم الحرفية يتطلب استخدام رمز الفاصلة "،" لفصل العناصر عن بعضها البعض بحيث تتوضع هذه القيم ضمن القوسين "{}" كما يلي:

```
int[] myIntArray = { 5, 9, 10, 2, 99 };
```

للمصفوفة myIntArray هنا خمس عناصر وهي من نوع int.

والطريقة الاخرى تتطلب صيغة كالشكل:

<arrayType> [ ] arrayName=new <arrayType> [<size>];

حيث <size> تمثل عدد عناصر المصفوفة ناقصا واحد (بسبب بدء ترقيم العناصر من الصفر).

وبالتالي فلكي نهيئ المصفوفة myIntArray فإننا سنكتب:

Int [ ] myIntArray = new int [arraySize];

ويمكننا أن ندمج الطريقتين مع بعضهما البعض بالشكل:

```
int[] myIntArray = new int[5] { 5, 9, 10, 2, 99 };
```

وفي حالة كهذه يجب أن يتطابق حجم المصفوفة مع القيم الحرفية المسندة إليها فلا يمكننا على سبيل المثال كتابة ما يلي:

```
int[] myIntArray = new int[10] { 5, 9, 10, 2, 99 };
```

هنا لقد عرفنا المصفوفة بأنها تستوعب 10 عناصر بينما قمنا بتعريف خمسة عناصر فقط لذا فإن هذا سيتسبب في حدوث خطأ هناك ملاحظة واحدة في استخدام هذا الاسلوب للتصريح عن مصفوفة وهو أن حجم المصفوفة يجب أن يمثل قيمة حرفية أو متحولا ثابتا فقط على سبيل المثال:

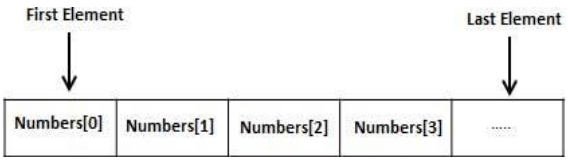
```
const int arraySize=5;
int [ ] myIntArray = new int [arraySize] {5, 9, 10, 2, 99};
```

فإن أزلنا الكلمة const من تصريح المتحول سيخفق المترجم في ترجمة الشيفرة.

وكما في أنواع المتحولات الأخرى فإننا لسنا بحاجة لتهيئة المصفوفة في نفس السطر الذي نصرح عنها فيه أي أن الشيفرة التالية صحيحة:

```
int[] myIntArray;
myIntArray = new int[5];
```

والشكل (13-5) يبين شكل المصفوفة أحادية البعد التي تحدثنا عنها فهي تتكون من سطر واحد وعدة اعمدة تمثل بعد أو حجم هذه المصفوفة.



الشكل (5-13)

تطبيق حول استخدام المصفوفات:

1- قم بإنشاء تطبيق Console جديد باسم Console Array Friend Name.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

```
namespace Console_Array_Friend_Name
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] friendNames = {"Mohamed ALmesri",
                                    "Nader ALal", "Omar Ahmad"};
            Console.WriteLine("hare are {0} of my Friend", friendNames.Length);
            for (int i = 0; i < friendNames.Length; i++)
            {
                Console.WriteLine(friendNames[i]);
            }
        }
    }
}
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 فيظهر الشكل (5-14).

الشكل (5-14)

كيفية العمل:

How it Works:

تقوم هذه الشيفرة بالتصريح عن مصفوفة باسم friendNames وتهيئتها بثلاث قيم ومن ثم سرد هذه القيم على شاشة الخرج بواسطة حلقة for لاحظ أننا استخدمنا التعبير friendNames.Length لكي يعطينا عدد العناصر ضمن المصفوفة:

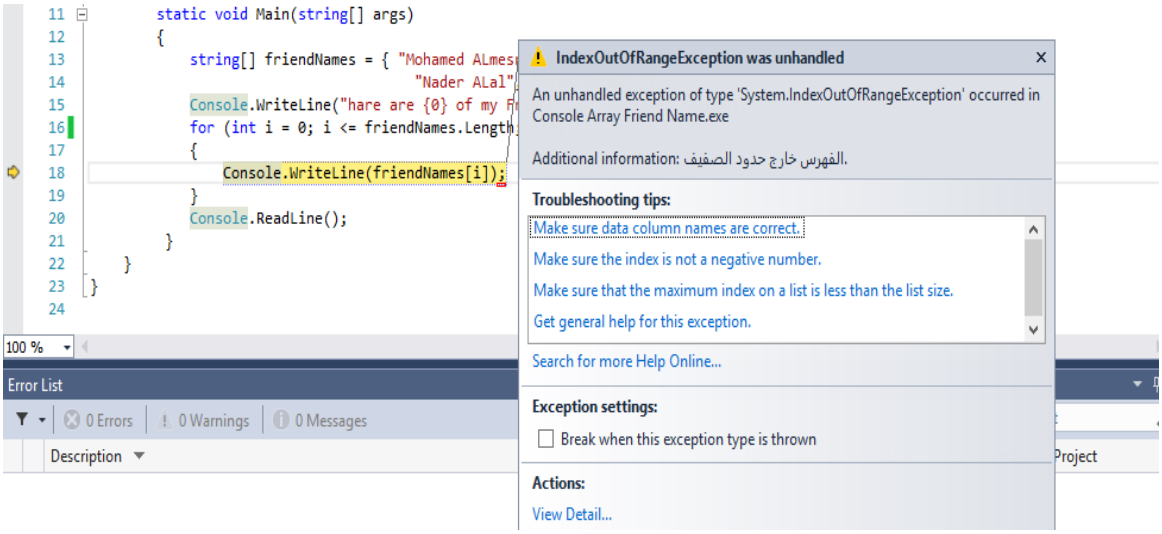
```
Console.WriteLine("hare are {0} of my Friend", friendNames.Length);
```

إن تلك الطريقة مفيدة للحصول على حجم المصفوفة.

أما طباعة قيم المصفوفة بواسطة حلقة for فهو أمر سهل للغاية لكن احتمال الوقوع في خطأ في تعليمة for كبير إذا غيرنا < إلى => كما يلي:

```
for (int i = 0; i <= friendNames.Length; i++)
{
    Console.WriteLine(friendNames[i]);
}
```

حيث سيؤدي ذلك إلى ظهور رسالة خطأكما في الشكل (15-5):



الشكل (15-5)

فبالعودة إلى هذه الحلقة نلاحظ أن الدورة الأخيرة لهذه الحلقة هي الدورة التي تكون فيها قيمة i مساوية لـ 3 وهذا يعني أن ما سيطبع في هذه الدورة هو العنصر friendNames[3] وهو في الحقيقة العنصر الرابع من المصفوفة وهو غير موجود وذلك يعتبر محاولة للوصول إلى عناصر خارج حدود المصفوفة وسيؤدي ذلك إلى حدوث خطأ.

هناك نوع خاص من الحلقات تفيدنا في حالة كهذه ألا وهي حلقات foreach.

حلقات foreach:

foreach Loops:

تسمح لنا حلقة foreach الوصول إلى جميع العناصر ضمن مصفوفة بصورة خطية ودون الاهتمام بعدد العناصر ضمنها ولهذه الحلقة الصيغة التالية:

```
foreach (<baseType> <name> in <array>)
{
// can use <name> for each element
}
```

ستقوم هذه الحلقة بالمرور عبر كل عنصر من عناصر المصفوفة وعند كل دورة من دورات هذه الحلقة سيمثل <name> الواجهة التي سنتعامل من خلالها مع العنصر وبهذه الطريقة لا يمكننا أن نقع في خطأ محاولة الوصول إلى عناصر خارج المصفوفة وبالتالي يمكننا تعديل الشيفرة السابقة لتأخذ الشكل التالي:

```
namespace Console_Array_Friend_Name
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] friendNames = { "Mohamed ALmesri",
                                     "Nader ALal", "Omar Ahmad"};
            Console.WriteLine("hare are {0} of my Friend", friendNames.Length);
            foreach (string friendName in friendNames )
            {
                Console.WriteLine(friendName);
            }
         }
    }
}
```

إن خرج هذه الشيفرة هو تماما كخرج الشيفرة في التطبيق السابق.

إن الفرق الرئيسي بين استخدام هذه الطريقة واستخدام حلقة for العادية هو أن حلقة foreach تؤمن لنا وصولا محميا (للقراءة فقط) لمحتويات المصفوفة أي لا يمكننا تعديل محتويات المصفوفة ضمن الحلقة فلا يمكننا على سبيل المثال كتابة شيفرة كهذه:

```
            foreach (string friendName in friendNames )
            {
                friendName="Gader Amen";
            }
```

إن هذه الشيفرة ستسبب في حدوث خطأ من المترجم أما باستخدام حلقة for العادية فإنه يمكننا تعديل محتوى المصفوفة كما نريد.

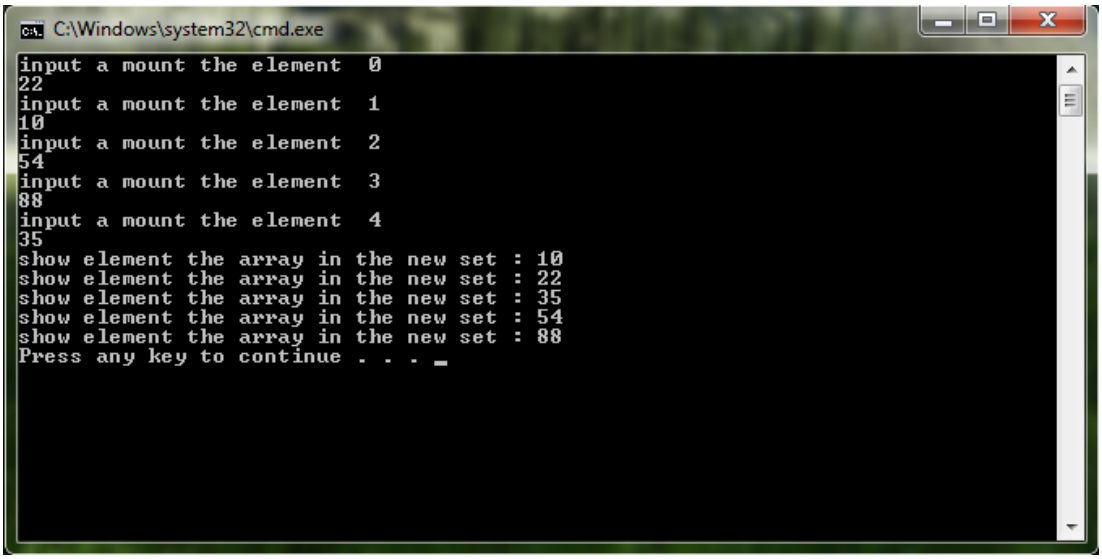
تطبيق أخر حول استخدام المصفوفات:

1- قم بإنشاء تطبيق Console جديد باسم Console Array New Set.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

```
        static void Main(string[] args)
        {
            int[] Array = new int[5];
```

```
        int i, j, Temp;
        for (i = 0; i < Array.Length; i++)
        {
            Console.WriteLine("input a mount the element  " + i);
            Array[i] = int.Parse(Console.ReadLine());
        }
        for (i = 0; i < Array.Length; i++)
        {
            for (j = i + 1; j < Array.Length; j++)
            {
                if (Array[i] > Array[j])
                {
                    Temp = Array[i];
                    Array[i] = Array[j];
                    Array[j] = Temp;
                }
            }
        }
        foreach (int element in Array )
        {
            Console.WriteLine("show element the array in the new set : "
                + element);
        }
    }
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 فيظهر الشكل (5-16).



الشكل (5-16)

كيفية العمل:

How it Works:

في هذا التطبيق قمنا بالتصريح عن مصفوفة أحادية البعد خطية تحوي خمس عناصر من النوع int باستخدام التعليمة:

```
int[] Array = new int[5];
```
ومن ثم قمنا بالتصريح عن ثلاث متحولات من النوع int.

ومن ثم قمنا ببناء حلقة إدخال تطلب من المستخدم إدخال قيمة كل عنصر من عناصر المصفوفة الخمس حيث تم هنا اسناد القيمة المدخلة إلى كل العنصر المقابل لها وذلك بعد تحويل الادخال على النوع int32 وذلك باستخدام طريقة جديدة للتحويل باستخدام العامل Parse وذلك باستخدام الشيفرة التالية:

```
for (i = 0; i < Array.Length; i++)
{
    Console.WriteLine("input a mount the element" + i);
    Array[i] = int.Parse(Console.ReadLine());
}
```
ومن ثم قمنا بتعشيش حلقتين الأولى تبدأ من قيمة العنصر الأول للمصفوفة والثانية تبدأ من قيمة العنصر الثاني إلى أن تنتهيان بأخر عنصر من المصفوفة حيث أن العمليات ضمن الحلقات تعمل على ترتيب عناصر المصفوفة ترتيبا تصاعديا من الاصغر إلى الأكبر وذلك باستخدام الشيفرة التالية:

```
for (i = 0; i < Array.Length; i++)
{
    for (j = i + 1; j < Array.Length; j++)
    {
        if (Array[i] > Array[j])
        {
            Temp = Array[i];
            Array[i] = Array[j];
            Array[j] = Temp;
        }
    }
}
```
ومن ثم عملنا أخيرا على طباعة عناصر المصفوفة بالترتيب الجديد وذلك باستخدام تعليمة foreach.

المصفوفات متعددة الأبعاد:

Multi-dimensional Arrays:

قد تظن من العنوان الذي يشير له هذا القسم أننا نتحدث عن إضافة عملية رياضية إلى لغة C# فبالإضافة إلى المصفوفات الخطية (linear arrays) التي تناولناها في السابق فإن لغة C# تدعم المصفوفات ثنائية وثلاثية البعد والمصفوفات ذات N بعد أيضا.

لنفترض على سبيل المثال أننا نود تخزين المسافات بين المدن السورية الأربعة التالية: دمشق, حلب, حمص, اللاذقية إن تخزين معلومات كهذه يتطلب جدولا عناوين أعمدته و عناوين صفوفه تمثل اسماء المدن و أما ما داخل الجدول فهو يمثل المسافة بالكيلومترات بين كل مدينتين إن جدولا كهذا سيأخذ الشكل التالي:

	دمشق	حلب	اللاذقية	حمص
دمشق	0	400	300	150
حلب	400	0	350	200
اللاذقية	300	350	0	250
حمص	150	200	250	0

ملاحظة:

البيانات الموجودة في هذا الجدول غير دقيقة ولا تطابق الواقع.

ولكي نستطيع تمثيل جدول كهذا فإن علينا أن نستخدم مصفوفة ثنائية البعد إن تصريحا لمصفوفة ثنائية البعد يأخذ الصيغة التالية:

<baseType> [,] <name>;

وبالنسبة لمصفوفات ذات أبعاد أكبر فإننا سنضع فواصل إضافية حيث تمثل كل فاصلة بعدا جديدا على سبيل المثال:

<baseType> [, , ,] <name>;

يمثل ذلك تصريحا لمصفوفة رباعية البعد.

إن إسناد القيم لهذه المصفوفة يأخذ الصيغة نفسها أيضا وللتصريح عن مصفوفة ثنائية البعد وتهيئتها وفقا لجدول المسافات بين المدن ولتكن باسم distances فإننا سنكتب ما يلي:

```
Double[,] distances = new double[i, j];
```

حيث تمثل i عدد الاسطر وj هو عدد الاعمدة يمكننا أن نضع القيم مباشرة ضمن المصفوفة مع أخذ الفواصل بعين الاعتبار عند تحديد موقع العنصر سنستخدم هنا أقواس "{}" لتحديد القيم الحرفية لتلك المصفوفة لكل سطر:

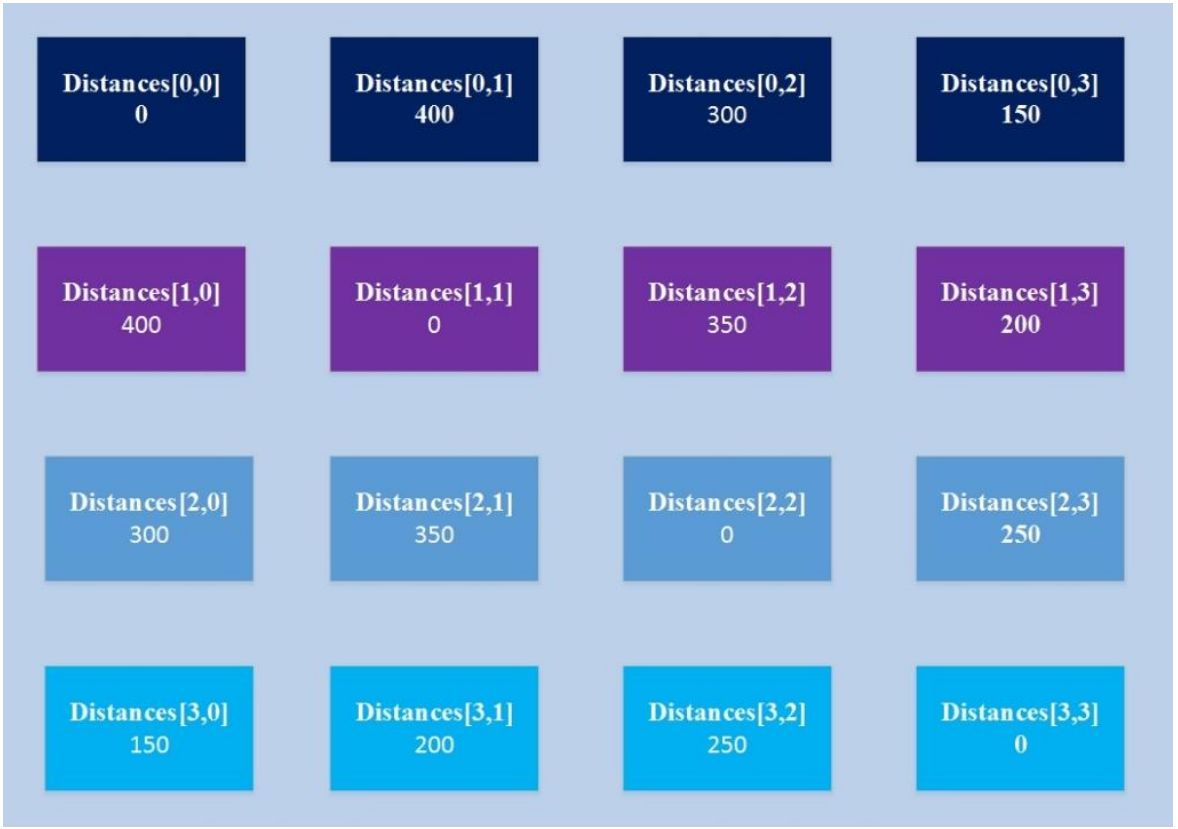
```
Double[,] distances ={{0,400,300,150},{400,0,350,200},
{300,350,0,250},{150,200,250,0}};
```

ولكي نتمكن من الوصول إلى العناصر كلا على حدة ضمن مصفوفة كهذه فإننا سنستخدم تعبيرا كهذا:

```
double x;
x=distances [2,1];
```

سيأخذ المتحول x هنا القيمة 350 وذلك لأننا حددنا العنصر الثالث رقم (2) من البعد الأول والثاني رقم (1) من البعد الثاني وهذا يوافق العنصر ذو القيمة 350.

والشكل (17-5) يبين كيفية تمثيل جدول المسافات السابق ضمن مصفوفة distances:



الشكل (5-17)

تمكننا حلقة foreach من الوصول إلى جميع العناصر بطريقةمتعددة الأبعاد كما في المصفوفة الخطية أحادية البعد على سبيل المثال:

```
Double[,] distances ={{0,400,300,150},{400,0,350,200},
{300,350,0,250},{150,200,250,0}};
foreach (double distance in distances)
{
    Console.WriteLine(distance);
}
Console.ReadLine();
```

ووفقا لذلك فإن طباعة العناصر سيأخذ الشكل التالي:

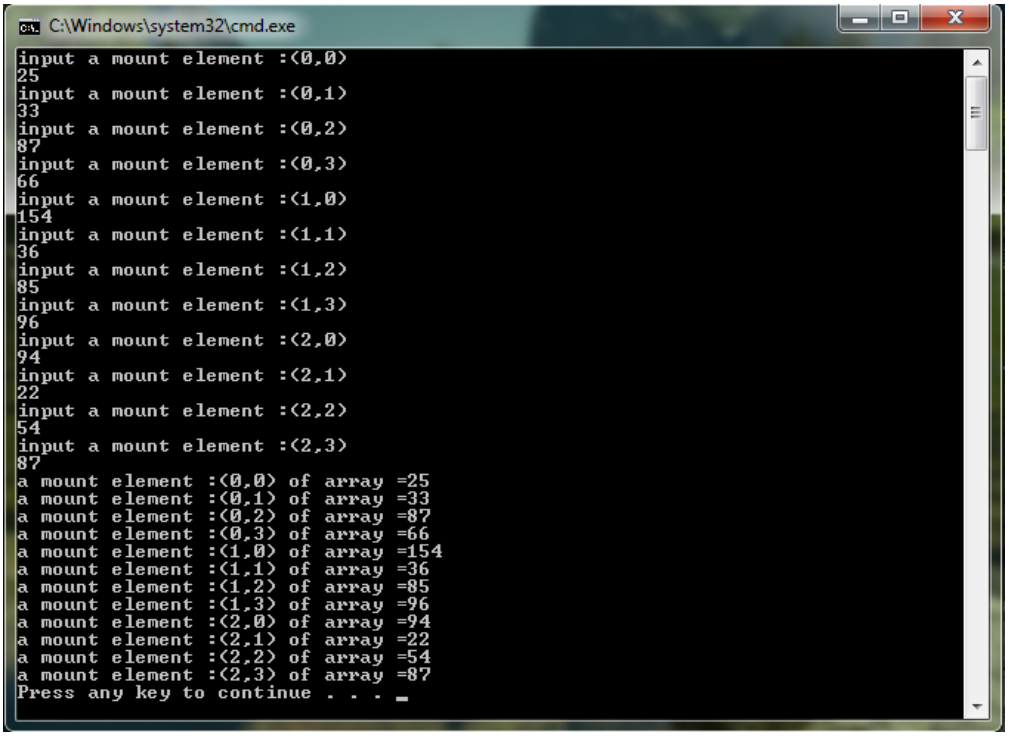
```
distances[0,0]
distances[0,1]
distances[0,2]
distances[0,3]
distances[1,0]
distances[1,1]
distances[1,2]

.......
```

تطبيق حول استخدام المصفوفات متعددة الأبعاد:

1- قم بإنشاء تطبيق Console جديد باسم Console Multi-dimensional Arrays.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

```
static void Main(string[] args)
{
    int[,] Matrice = new int[3, 4];
    int i, j;
    for (i = 0; i <= 2; i++)
    {
        for (j = 0; j <= 3; j++)
        {
            Console.WriteLine("input a mount element :("
                + i + "," + j + ")");
            Matrice[i, j] = int.Parse(Console.ReadLine());
        }
    }
    for (i = 0; i <= 2; i++)
    {
        for (j = 0; j <= 3; j++)
        {
            Console.WriteLine("a mount element :("
                + i + "," + j + ") of array =" + Matrice[i, j]);
        }
    }
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 فيظهر الشكل (5-18).



الشكل (5-18).

كيفية العمل:

How it Works:

في هذا التطبيق قمنا بالتصريح عن مصفوفة ثنائية البعد ومن ثم استخدمنا تعشيش الحلقات من أجل إدخال قيم عناصر هذه المصفوفة ومن ثم قمنا أيضا باستخدام تعشيش الحلقات لطباعة عناصر هذه المصفوفة.

مصفوفات المصفوفات:

Arrays of Arrays:

إن المصفوفات متعددة الأبعاد التي تناولناها في القسم السابق تأخذ شكلا مستطيلا يسمى هذا النوع من المصفوفات بالمصفوفات المستطيلة (rectangular arrays) وذلك لأن كل صف له الحجم نفسه ولقد رأينا ذلك في مثال المسافات بين المدن.

هناك طريقة في C# لاستخدام مصفوفات غير متجانسة () وهي المصفوفات التي تكون لها صفوف بأحجام مختلفة وللقيام بذلك فإننا نحتاج لمصفوفة يمثل فيها كل عنصر مصفوفة بحد ذاتها أي أنه يمكننا أن نشكل مصفوفة عناصرها عبارة عن مصفوفات أيضا والشرط الوحيد لصحة ذلك هو أن تكون الأنواع الأساسية لهذه المصفوفات واحدة.

إن صيغة التصريح عن مصفوفة مصفوفات يأخذ الصيغة التالية حيث أننا سنحدد مجموعة من الأقواس "[]" كما يلي:

```
int[][] jaggedIntArry;
```

للأسف فإن تهيئة مصفوفة كهذه ليس بسهولة تهيئة المصفوفات متعددة الأبعاد فلا يمكننا على سبيل المثال كتابة التالي:

```
jaggedIntArray =new int [3] [4];
```

وحتى إن استطعنا القيام بذلك فإن هذا لن يفيد باعتبار أن هذا يعني إنشاء مصفوفة لمصفوفات متجانسة وهو الأمر الذي نستطيع تحقيقه باستخدام مصفوفات متعددة الابعاد بجهد أقل من ذلك ولا يمكننا كتابة التالي:

```
jaggedIntArray = {{1,2,3},{1},{1,2}};
```

لدينا خياران لتهيئة مصفوفة المصفوفات يمكننا أن نهيئ المصفوفة التي تتضمن مصفوفات أخرى (سأسمي هذه المصفوفات بالمصفوفات الفرعية للتبسيط) ومن ثم تهيئة المصفوفات الفرعية وفقا لما يلي:

```
jaggedIntArray = new int[2][];
jaggedIntArray[0] = new int[3];
jaggedIntArray[1] = new int[4];
```

حيث تقرا هذه المصفوفة بأنها مصفوفة مصفوفات تحوي على مصفوفتين (الرقم 2) المصفوفة رقم 0 هي مصفوفة خطية احادية البعد تحوي على ثلاثة عناصر والمصفوفة رقم 1 هي مصفوفة خطية أحادية البعد تحوي على أربعة عناصر.

أو يمكننا استخدام نموذج معدل لأسلوب التهيئة السابق بحيث يتضمن إسناد القيم الحرفية إلى المصفوفات مباشرة:

```
jaggedIntArray={new int[] {1,2,3},new int [] {1},new int [] {1,2}};
```

يمكننا استخدام حلقات foreach مع المصفوفات غير المتجانسة إلا أننا نحتاج إلى تعشيش حلقتين من حلقات foreach للحصول على عناصر المصفوفات الفرعية للمصفوفة لنفترض أن لدينا مصفوفة غير متجانسة تتضمن عشرة مصفوفات وكل مصفوفة فرعية من هذه المصفوفات هي مصفوفة أعداد صحيحة تمثل الأعداد التي تقبل القسمة على موقع هذه المصفوفة الفرعية من المصفوفة الرئيسة (على سبيل المثال المصفوفة الخامسة) ذات الرقم 4 تتضمن عنصرين فقط 1 و5 لأن العدد 5 لا يقبل القسمة غلا على هذين الرقمين فقط:

```
int [][] divisors1To10 = {new int [] {1},
                          new int [] {1,2},
                          new int [] {1,3},
                          new int [] {1,2,4},
                          new int [] {1,5},
                          new int [] {1,2,3,6},
                          new int [] {1,7},
                          new int [] {1,2,4,8},
                          new int [] {1,3,9},
                          new int [] {1,2,5,10}};
```

إن الشيفرة التالية ستخفق في العمل (وستعطي خطأ):

```
foreach (int divisor in divisor1TO10)
{
    Console.WriteLine(divisor);
}
```

إن ذلك يعود على أن المصفوفة divisors1To10 تتضمن عناصر من النوع int[] وليست عناصر من النوع int وبالتالي فإننا سنحتاج إلى حلقتي foreach الخارجية للمرور على المصفوفات والداخلية للمرور على عناصر كل مصفوفة كما يلي:

```
foreach (int divisorofInt in divisor1TO10)
{
        foreach (int divisor in divisorofInt)

        {
          Console.WriteLine(divisor);
        }
}
```

توضيح:

لقد نوهنا قبل ذلك أن بإمكاننا عدم وضع شيفرة جسم الحلقة أو جسم العبارة الشرطية ضمن كتلة برمجية أي وضع الشيفرة ضمن الاقواس "{}" وذلك إذا كانت هذه الشيفرة تمثل سطرا برمجيا واحدا فقطوبالتالي فإن الشيفرة التالية مطابقة تماما للسابقة:

```
foreach (int divisorofInt in divisor1TO10)
        foreach (int divisor in divisorofInt)
          Console.WriteLine(divisor);
```

وكما ترى فإن صيغة استخدام المصفوفات غير المتجانسة معقدة نوعا ما وفي معظم الاحيان يكون من الافضل استخدام المصفوفات المستطيلة أو استخدام ابسط للتخزين على كل حال هناك أوضاع قد تجبر فيها على استخدام هذا النوع من المصفوفات.

خصائص ودوال المصفوفات:

Array properties:

تحتوي المصفوفات على بعض الخصائص والدوال التي قد تساعدك في العمل عليها وأشهر هذه الخصائص والدوال وأكثرها استخداما مبين في الجدول التالي:

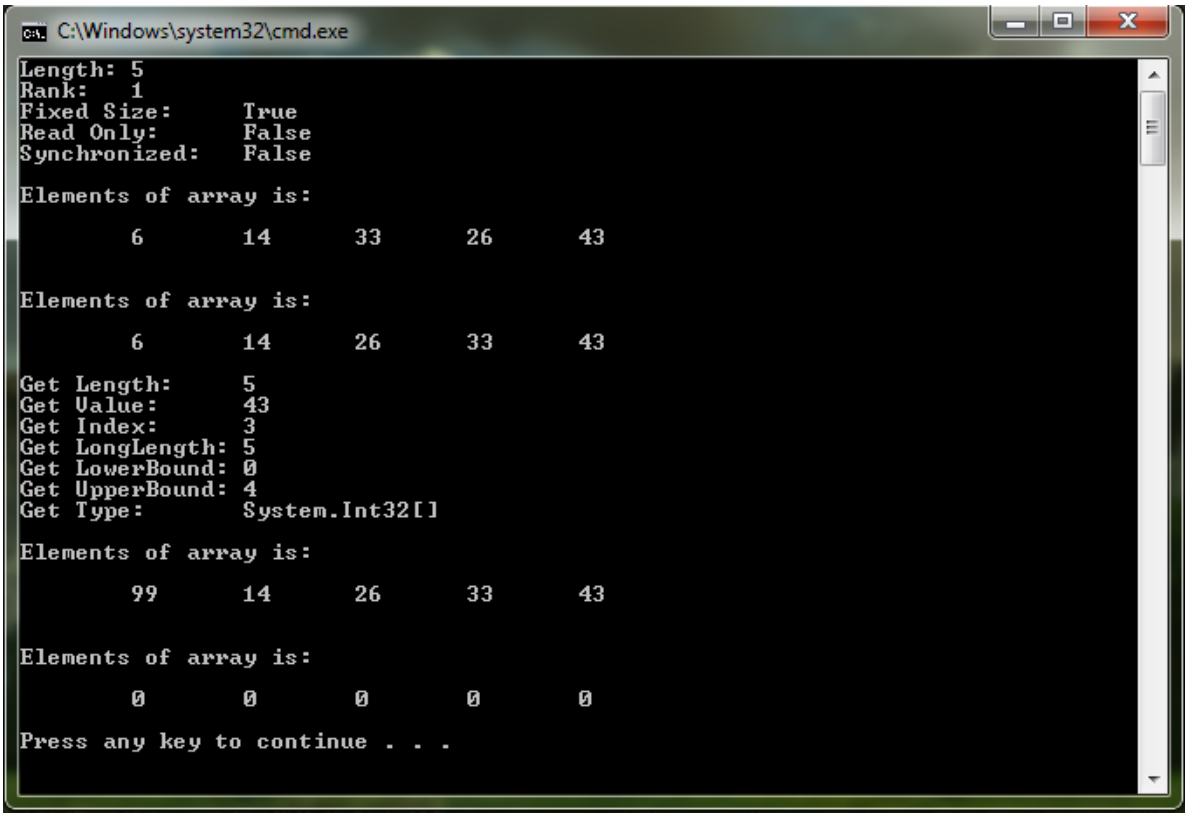
الخاصية أو الدالة	الاستخدام
Length	تحدد طول المصفوفة عدد العناصر بعدد صحيح 32 بت
Sort	تقوم بترتيب عناصر المصفوفة تصاعديا وترتيب هجائي إذا كانت نصية.
Reverse	تقوم بعكس ترتيب عناصر المصفوفة
ToString	لتحويل المصفوفة إلى متغير نصي
Rank	تحدد عدد الابعاد في المصفوفة وتسمى رتبة المصفوفة
IsFixedSize	يعطي قيمة منطقية True أو False تعبر فيما إذا كان حجم المصفوفة متغير أم ثابت.
IsReadOnly	يعطي قيمة منطقية True أو False تعبر فيما إذا كانت المصفوفة للقراءة فقط
IsSynchronized	يعطي قيمة تشير فيما إذا كان الوصول إلى المصفوفة تتم مزامنة (موضوع آمن).
LongLength	تحدد طول عناصر المصفوفة بعدد صحيح 64 بت
SyncRoot	يعطي كائن يمكن استخدامه لمزامنة الوصول إلى المصفوفة.
GetLowerBound	يعطي دليل العنصر الأول في البعد المحدد من المصفوفة
GetUpperBound	يعطي دليل العنصر الأخير في البعد المحدد من المصفوفة
IndexOf	يعطي دليل عنصر من المصفوفة طبعا دليل أول عنصر مشابه
GetValue	يعطي قيمة عنصر من مصفوفة بتحديد دليله
GetType	يعطي نوع بيانات المصفوفة.
SetValue	يعمل على استبدال عنصر من المصفوفة بعنصر أخر يحدده المستخدم.

تطبيق حول استخدام خصائص ودوال المصفوفات:

1- قم بإنشاء تطبيق Console جديد باسم Console Arrays properties.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

```
class Program
{
    static void printArray(int[] arr)
    {
        Console.WriteLine("\nElements of array is:\n");
        foreach (int i in arr)
        {
            Console.Write("\t{0}", i);
        }
        Console.WriteLine("\n");
    }
    static void Main(string[] args)
    {
        int[] arr1 = new int[5] { 43, 26, 33, 14, 6 };
        int[] arr2 = new int[5];
        int len, rank;
        bool fixedSize, readOnly, SyncHronized;
        len = arr1.Length;
        Console.WriteLine("Length:\t{0}", len);
        rank = arr1.Rank;
        Console.WriteLine("Rank:\t{0}", rank);
        fixedSize = arr1.IsFixedSize;
        Console.WriteLine("Fixed Size:\t{0}", fixedSize);
        readOnly = arr1.IsReadOnly;
        Console.WriteLine("Read Only:\t{0}", readOnly);
        SyncHronized = arr1.IsSynchronized;
        Console.WriteLine("Synchronized:\t{0}", SyncHronized);
        Array.Reverse(arr1);
        printArray(arr1);
        Array.Sort(arr1);
        printArray(arr1);
        Console.WriteLine("Get Length:\t{0}", arr1.GetLength(0));
        Console.WriteLine("Get Value:\t{0}", arr1.GetValue(4));
        Console.WriteLine("Get Index:\t{0}", Array.IndexOf(arr1, 33));
        Console.WriteLine("Get LongLength:\t{0}", arr1.GetLongLength(0));
        Console.WriteLine("Get LowerBound:\t{0}", arr1.GetLowerBound(0));
        Console.WriteLine("Get UpperBound:\t{0}", arr1.GetUpperBound(0));
        Console.WriteLine("Get Type:\t{0}", arr1.GetType());
        arr1.SetValue(99, 0);
        Array.Copy(arr1, arr2, 5);
        printArray(arr2);
        Array.Clear(arr1, 0, 5);
        printArray(arr1);
    }
}
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 فيظهر الشكل (5-19).



الشكل (5-19)

كيفية العمل:

How it Works:

في البداية قمنا ببناء دالة أو تابع ضمن Class يقوم هذا التابع عند استدعائه بطباعة عناصر المصفوفة وفق ترتيب معين لا تقلق سوف نتعلم هذا الأمر في الفصل التالي إن شاء الله والكود التالي هو لبناء تابع الطباعة ضمن Class:

```
class Program
{
    static void printArray(int[] arr)
    {
        Console.WriteLine("\nElements of array is:\n");
        foreach (int i in arr)
        {
            Console.Write("\t{0}", i);
        }
        Console.WriteLine("\n");
    }
```

في الشيفرة التالية التي كتبناها قمنا بتعريف مصفوفتين arr1 و arr2 من نوع int تحوي كل منهما خمس عناصر إلا أن المصفوفة arr1 قمنا بإسناد قيم العناصر لها ومن ثم قمنا بالتصريح عن متحولين len, rank من النوع int كما صرحنا عن ثلاثة متحولات أخرى fixedSize, readOnly, SyncHronized من النوع bool كما في الشيفرة التالية:

```
        int[] arr1 = new int[5] { 43, 26, 33, 14, 6 };
        int[] arr2 = new int[5];
```

```
int len, rank;
bool fixedSize, readOnly, SyncHronized;
```

من ثم أسندنا للمتحول len طول المصفوفة arr1 باستخدام الخاصية Length كما أسندنا للمتحول rank رتبة المصفوفة arr1 باستخدام الخاصية Rank كما أسندنا للمتحول المنطقي fixedSize قيمة منطقية باستخدام الخاصية IsFixedSize التي تختبر المصفوفة arr1 فإذا كانت ثابتة تعطي القيمة True وإلا فإنها تعطي القيمة False ومن ثم أسندنا للمتحول المنطقي readOnly قيمة منطقية باستخدام الخاصية IsReadOnly التي تختبر المصفوفة arr1 فإذا كانت للقراءة فقط فإنها تعطي القيمة True وإلا فإنها تعطي القيمة False وكذلك الأمر أسندنا للمتحول المنطقي SyncHronized قيمة منطقية باستخدام الخاصية IsSynchronized التي تختبر أمان التزامن فإذا كان أمن تعطي القيمة True وإلا فإنها تعطي القيمة False حيث عملنا على إظهار ناتج المتحولات الخمسة بعد إسناد القيم لها على شاشة Console باستخدام الشيفرة التالية:

```
len = arr1.Length;
Console.WriteLine("Length:\t{0}", len);
rank = arr1.Rank;
Console.WriteLine("Rank:\t{0}", rank);
fixedSize = arr1.IsFixedSize;
Console.WriteLine("Fixed Size:\t{0}", fixedSize);
readOnly = arr1.IsReadOnly;
Console.WriteLine("Read Only:\t{0}", readOnly);
SyncHronized = arr1.IsSynchronized;
Console.WriteLine("Synchronized:\t{0}", SyncHronized);
```

من ثم عملنا على عكس ترتيب عناصر المصفوفة باستخدام الخاصية Reverse وإعادة ترتيب عناصر المصفوفة تصاعديا باستخدام الخاصية Sort حيث قمنا بطباعة النتيجة باستدعاء دالة الطباعة printArray التي بنيناها سابقا في Class كما في الشيفرة التالية:

```
Array.Reverse(arr1);
printArray(arr1);
Array.Sort(arr1);
printArray(arr1);
```

من ثم قمنا بطباعة نتيجة بعض خواص المصفوفة arr1 منها الخاصية GetLength(0) وهي تشبه الخاصية Length إلا أننا هنا يمكننا في حالة المصفوفة متعددة الأبعاد يمكننا تحديد البعد المراد تحديد طوله وفي حالتنا هنا لا يوجد سوى بعد واحد لذا وضعنا القيمة 0 ولو أردنا ان نحدد البعد الثاني في مصفوفة ثنائية البعد لوضعنا الرقم 1 وهكذا.
كما قمنا بطباعة نتيجة الخاصية GetValue(4) في المصفوفة arr1 حيث تعمل هذه الخاصية على إظهار قيمة العنصر ذو الدليل n في مثالنا هذا طلبنا قيمة العنصر ذو الدليل رقم 4 أي العنصر رقم 5 من المصفوفة فكانت قيمته تساوي 43 حاول أن تطلب قيم مختلفة من الدليل لاكن انتبه أن الترقيم يبدا من صفر فلا تحاول أن تطلب قيمة دليل خارج طول المصفوفة لأن ذلك سوف يؤدي لحدوث خطأ.
أما الخاصية IndexOf (arr1, 33) فهي ترجع قيمة الدليل للعنصر ذو القيمة 33 من المصفوفة arr1 والخاصية GetLongLength(0) فهي تعطي قيمة طول المصفوفة arr1 إلا أن الناتج يكون من نوع بيانات int64 أما الخاصية GetLowerBound(0) فهي تعطي دليل أول عنصر من المصفوفة والخاصية GetUpperBound(0) تعطي دليل أخر عنصر من المصفوفة arr1 طبعا هاتين الخاصيتان تعطيان قيم تماثل الترقيم الذي تعتمده لغة C#.

أما الخاصية GetType() فهي تظهر نوع البيانات المخزنة ضمن المصفوفة arr1 والشيفرة التالية تبين طرق إسناد هذه الخواص:

```
Console.WriteLine("Get Length:\t{0}", arr1.GetLength(0));
Console.WriteLine("Get Value:\t{0}", arr1.GetValue(4));
Console.WriteLine("Get Index:\t{0}", Array.IndexOf(arr1, 33));
Console.WriteLine("Get LongLength:\t{0}", arr1.GetLongLength(0));
Console.WriteLine("Get LowerBound:\t{0}", arr1.GetLowerBound(0));
Console.WriteLine("Get UpperBound:\t{0}", arr1.GetUpperBound(0));
Console.WriteLine("Get Type:\t{0}", arr1.GetType());
```

في النهاية قمنا باستبدال قيمة أحد عناصر المصفوفة عن طريق تحديد دليله والقيمة الجديدة المراد تخزينها باستخدام الخاصية SetValue كما قمن بنسخ قيم عناصر المصفوفة arr1 إلى المصفوفة arr2 باستخدام أمر النسخ Copy ز أخير قمن بتصفير قيم المصفوفة arr1 باستخدام الأمر Clear ومن ثم قمنا بطباعة الخرج لكل الأوامر السابقة باستدعاء الدالة printArray كما تبين ذلك الشيفرة التالية:

```
arr1.SetValue(99, 0);
Array.Copy(arr1, arr2, 5);
printArray(arr2);
Array.Clear(arr1, 0, 5);
printArray(arr1);
```

ملاحظة:

استخدمنا في هذا التطبيق رمز الجدولة الافقية "\t " من أجل تحسين إظهار الطباعة على شاشة Console راجع الفصل الثالث.

ملاحظة:

للتوسع في خصائص المصفوفات يمكنك مراجعة مكتبات msdn للغة البرمجة C# على الرابط:

http://msdn.microsoft.com/en-us/library/system.array_properties(v=vs.110).aspx

اللوائح:

Lists:

تشبه اللوائح إلى حد كبير المصفوفات إلا أن الاختلاف بينها هو أن المصفوفات تملك حجما ثابتا نقوم بالإعلان عنه أثناء التصريح عن المصفوفة أما اللوائح فإنها متغيرة الحجم أي أنها ديناميكية وبالتالي فإننا لسنا مضطرين لتحديد حجم اللوائح.

التصريح عن اللوائح:

Declaring Lists:

يمكننا التصريح عن اللوائح وفقا للصيغة التالية:

List<BaseType> <name>= new List<Type> ();

حيث أن <baseType> تمثل نوع بيانات العناصر ويمكن أن تمثل أي نوع كان سواء كان نوع بسيطا أو تعداد أو بنية أما <name> فتمثل اسم اللائحة (أو اسم متحول اللائحة).

ويمكن إضافة عناصر للائحة أثناء التصريح كما يلي:

```
List<string> friendName = new List<string> {"Ahmad","Kaled","Amen"};
```

حيث قمنا هنا بالتصريح عن لائحة من النوع string باسم friendName تحوي على ثلاثة أسماء.
كما يمكننا إضافة عناصر لها باستخدام الدالة Add كما في الشيفرة التالية:

```
List<string> friendName = new List<string>();
friendName.Add("Ahmad");
friendName.Add("Kaled");
friendName.Add("Amen");
```

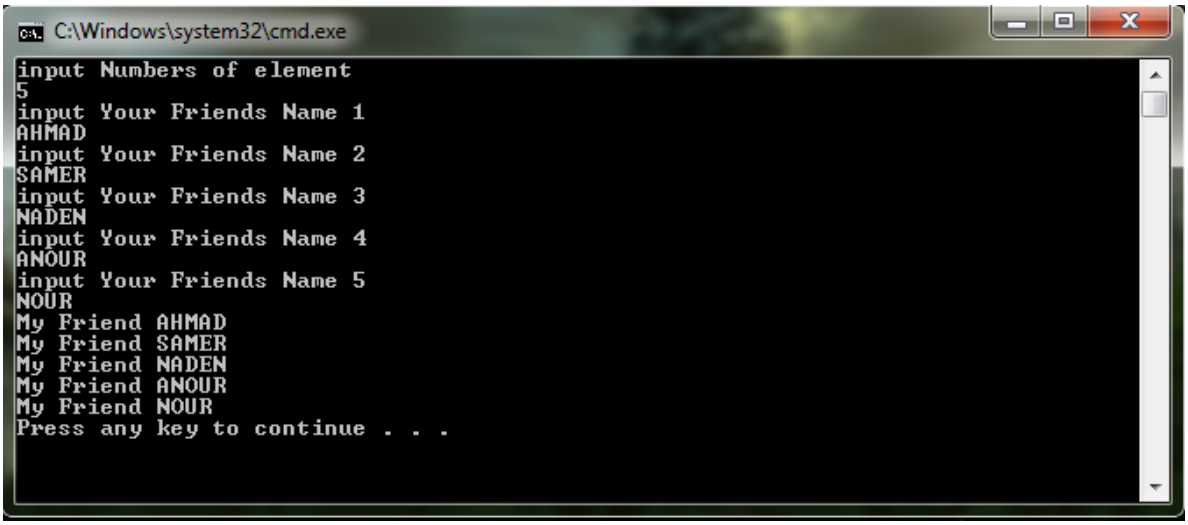
أما إظهار العناصر فهو مشابه تماما لإظهار العناصر في المصفوفات.

تطبيق حول استخدام اللوائح:

1- قم بإنشاء تطبيق Console جديد باسم Console Lists.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

```
static void Main(string[] args)
{
    int i, j;
    string friendName;
    List<string> friendNames = new List<string>();
    Console.WriteLine("input Numbers of element");
    j = Convert.ToInt32(Console.ReadLine());
    for (i = 1; i <= j; i++)
    {
        Console.WriteLine("input Your Friends Name {0}",i);
        friendName = Convert.ToString(Console.ReadLine());
        friendNames.Add(friendName);
    }
    foreach (string friend in friendNames)
    {
        Console.WriteLine("My Friend {0}",friend);
    }
}
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 فيظهر الشكل (5-20).



الشكل (5-20)

كيفية العمل:

How it Works:

في هذا التطبيق قمنا بالتصريح عن متحولين من النوع int وهما المتحولين i وj كما صرحنا عن متحول أخر من النوع string باسم friendName وقمنا بالتصريح عن لائحة من النوع string باسم friendNames دون أن نحدد عناصر هذه اللائحة كما تبين ذلك الشيفرة التالية:

```
int i, j;
string friendName;
List<string> friendNames = new List<string>();
```

ومن ثم طلبنا من المستخدم أن يدخل عدد العناصر التي يرغب بإدخالها ضمن اللائحة حيث استخدمنا الرقم المدخل ببناء حلقة تسمح لنا بإدخال العناصر ومن ثم تسند العناصر المدخلة بعد تحويلها إلى النوع string إلى اللائحة friendName وذلك باستخدام الشيفرة التالية:

```
Console.WriteLine("input Numbers of element");
j = Convert.ToInt32(Console.ReadLine());
for (i = 1; i <= j; i++)
{
    Console.WriteLine("input Your Friends Name {0}",i);
    friendName = Convert.ToString(Console.ReadLine());
    friendNames.Add(friendName);
}
```

بعد ذلك استخدمنا حلقة foreach من أجل طباعة عناصر اللائحة على نافذة الخرج كما تبين ذلك الشيفرة:

```
foreach (string friend in friendNames)
{
    Console.WriteLine("My Friend {0}",friend);
}
```

خصائص ودوال اللوائح:

Lists properties:

تحتوي اللوائح على بعض الخصائص والدوال التي قد تساعدك في العمل عليها وأشهر هذه الخصائص والدوال وأكثرها استخداما مبين في الجدول التالي:

الخاصية أو الدالة	الاستخدام
Count	تحدد عدد عناصر اللائحة (طول) بعدد صحيح.
Sort	تقوم بترتيب عناصر اللائحة تصاعديا إذا كانت رقمية وترتيب هجائي إذا كانت نصية.
Reverse	تقوم بعكس ترتيب عناصر المصفوفة
add	يضيف عنصر للائحة يتوضع في نهاية اللائحة.
Insert	يضيف عنصر جديد للائحة بحيث يحدث إزاحة للعنصر الذي سيحل مكانه ويأخذ دليله.
Remove	لإزالة عنصر من اللائحة
Capacity	يحدد حجم اللائحة.
GetType	يحدد نوع عناصر اللائحة.
Sum	يقوم بجمع عناصر اللائحة إذا كانت من نوع int.
Contains	يقوم باختبار وجود قيمة ما ضمن اللائحة.
RemoveAt	يقوم بإزالة عنصر من اللائحة عن طريق تحديد دليله
RemoveRange	يقوم بمسح مجال من العناصر عن طريق تحديد عرض المجال ونقطة البداية.
Clear	يقوم بمسح جميع عناصر اللائحة.

تطبيق حول استخدام خصائص ودوال اللوائح:

1- قم بإنشاء تطبيق Console جديد باسم Console Lists Properties.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

```
class Program
{
    static void printList(List <int> lists)
    {
        Console.WriteLine("\nElements of List is:\n");
        foreach (int i in lists)
        {
            Console.Write("\t{0}", i);
        }
        Console.WriteLine("\n");
    }

    static void Main(string[] args)
```

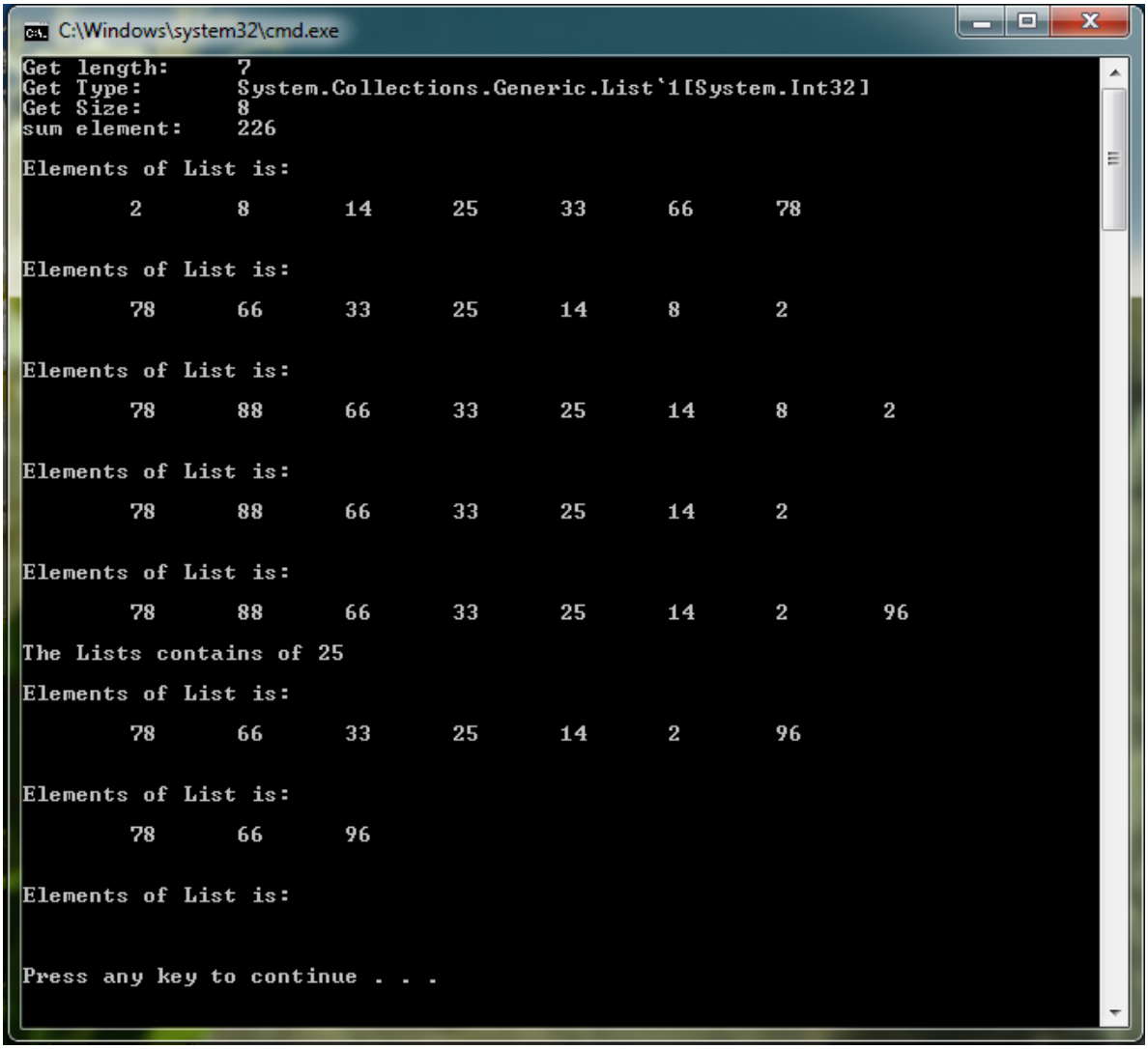
```
    {
        List<int> numbers = new List<int> { 14, 25, 33, 66, 78, 2, 8 };
        Console.WriteLine("Get length:\t{0}", numbers.Count);
        Console.WriteLine("Get Type:\t{0}", numbers.GetType());
        Console.WriteLine("Get Size:\t{0}", numbers.Capacity);
        Console.WriteLine("sum element:\t{0}", numbers.Sum());
        numbers .Sort();
        printList(numbers);
        numbers.Reverse();
        printList(numbers);
        numbers.Insert(1, 88);
        printList(numbers);
        numbers.Remove(8);
        printList(numbers);
        numbers.Add(96);
        printList(numbers);
        if (numbers.Contains  (25))
        {
            Console .WriteLine ("The Lists contains of 25");
        }
        numbers.RemoveAt(1);
        printList(numbers);
        numbers.RemoveRange(2, 4);
        printList(numbers);
        numbers.Clear();
        printList(numbers);
    }
}
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 فيظهر الشكل (21-5).

كيفية العمل:

How it Works:

لن أخوض في تفاصيل هذا التطبيق لأنني اجزم بأنك أصبحت قادرا على قراءة الشيفرات لذا سأترك الأمر لك.



الشكل (21-5)

معالجة السلاسل النصية:

String Manipulation:

لقد اقتصر استخدامنا للسلاسل النصية منذ بداية هذا الكتاب على طباعة هذه السلاسل النصية على نافذة الخرج أو قراءة سلاسل نصية من دخل المستخدم بالإضافة إلى ضم السلاسل النصية مع بعضها بواسطة العامل +.

ومع تقدم عملية تطويرك للتطبيقات ستحتاج لمعالجات إضافية على السلاسل النصية وليس مجرد قراءة وكتابة السلاسل النصية من وإلى نافذة Console لقد خصصنا هذا القسم لتناول كيفية معالجة السلاسل النصية والتي سنستخدمها بكثرة في البرمجة بلغة C# من خلال الفصول القادمة لهذا الكتاب.

وكبداية لمناقشتنا هذه من الأفضل أن ننوه إلى أنه يمكننا أن نتعامل مع السلسلة النصية على أنها عبارة عن مصفوفة من الرموز أي يمكننا تمثيل المتحول من نوع String بمصفوفة نوعها الأساسي هو char هذا يعني أنه يمكننا الوصول إلى رموز السلسلة النصية كلا على حدة باستخدام صيغة كما يلي:

```
string myString = "A String";
char myChar = myString[1];
```

إن قراءة الرموز بهذه الصورة أمر ممكن ولكن لا يمكننا استخدام هذه الطريقة لإسناد الرموز إلى السلسلة النصية أي أن الشيفرة التالية غير مقبولة:

```
myString[1] = 'a';
```

ولكي نتمكن من الحصول على مصفوفة char يمكننا الكتابة إليها فإننا سنستخدم الأمر ToCharArray() والذي يعيد مصفوفة رموز تمثل مصفوفة رموز تمثل السلسلة النصية:

```
string myString = "a String";
char[] myChar = myString.ToCharArray();
```

عندئذ يمكننا أن نعالج مصفوفة char بالأسلوب الذي نريد فيمكننا أن نغير في عناصرها ونكتب ضمنها كما نريد.

يمكننا أن نستخدم السلاسل النصية ضمن حلقات foreach أيضا على سبيل المثال:

```
foreach (char character in myString )
{
    Console.WriteLine("{0}", character);
    Console.ReadLine();
}
```

وكما في المصفوفات فإنه يمكننا أن نحصل على عدد الرموز المكونة للسلسلة النصية وذلك باستخدام الأمر Length كما في المثال التالي:

```
Console.WriteLine("input any statement");
string myString = Console.ReadLine();
Console.WriteLine("You type {0} characters.",myString.Length );
Console.ReadLine();
```

هناك الكثير من التقنيات الأساسية لمعالجة السلاسل النصية والتي تستخدم أوامر مشابهة لشكل الأمر <string>.ToCharArray فهناك الأمر ToLower و ToUpper ومهمتهما تحويل أحرف السلسلة النصية جميعها إلى حروف صغيرة أو كبيرة.

إن هذين الأمرين مهمان جدا ولكي تدرك ذلك لنفترض أن تطبيقنا ينتظر دخلا من المستخدم وبناء على هذا الدخل فإن هناك شيفرة برمجية معينة سيتم تنفيذها فلنفترض على سبيل المثال أن المستخدم أدخل السلسلة النصية "yes" عندئذ إذا قمنا بتحويل السلسلة النصية التي أدخلها المستخدم إلى حروف صغيرة فإننا سنتمكن من معالجة الحالات التي يمكن للمستخدم فيها أن يدخل "Yes" أو "yeS" أو "YES" وهي حالات واردة جدا:

```
Console.WriteLine("input any statement");
string userResponse = Console.ReadLine();
if (userResponse.ToLower() == "yes")
    Console.WriteLine("your input True");
   else
    Console .WriteLine ("your input False");
Console.ReadLine();
```

لكن إذا افترضنا عدم استخدام مثل هذا الأمر عندئذ علينا أن نتأكد من جميع الحالات التي يمكن للمستخدم أن يدخل الكلمة "yes" عندها تخيل الشيفرة التالية ومدى تعقيدها إذا حاولنا أخذ كافة احتمالات كتابة الكلمة "yes" بعين الاعتبار:

```
Console.WriteLine("input any statement");
string userResponse = Console.ReadLine();
if (userResponse == "yes" || userResponse == "Yes"
    || userResponse == "yEs" || userResponse == "yeS"
    || userResponse == "YEs" || userResponse == "yES"
    || userResponse == "YeS" || userResponse == "YES")
    Console.WriteLine("your input True");
else
    Console.WriteLine("your input False");
Console.ReadLine();
```

لاحظ أن هذا الأمر مثله مثل بقية أوامر السلاسل النصية لا يغير السلسلة النصية التي نطبق الأمر عليها وإنما يحفظ نتيجة الأمر في سلسلة نصية أخرى تمثل ناتج تطبيق هذا الأمر حيث سنستخدمها لمقارنة ما (كما في الشيفرة السابقة) أو لإسنادها إلى متحول آخر أو ربما إلى نفس المتحول كما في المثال:

```
UserResponse = userResponse.ToLower();
```

تلك هي نقطة هامة جدا يجب أن ننتبه إليها فكتابة الشيفرة التالية:

```
userResponse.ToLower();
```

لا يعني أننا حولنا حروف السلسلة النصية في المتحول userResponse إلى حروف صغيرة.

لنرى ماذا يمكننا أن نقوم به مع السلاسل النصية أيضا ماذا لو أن المستخدم قام بوضع فراغات إضافية إلى بداية أو نهاية سلسلة الدخل كأن يكتب مثلا:" yes " وفي حالة كهذه لن نتمكن من الحصول على الإجابة yes من المستخدم لحسن الحظ فإن هناك الأمر Trim() والذي يمكننا تطبيقه على السلسلة النصية حيث يقوم باستئصال جميع الفراغات من السلسلة النصية إليك المثال التالي:

```
Console.WriteLine("input any statement");
string userResponse = Console.ReadLine();
userResponse = userResponse.Trim();
if (userResponse.ToLower() == "yes")
    Console.WriteLine("your input True");
   else
    Console .WriteLine ("your input False");
Console.ReadLine();
```

وبهذا يمكننا أن نعالج حالات إدخال مثل:

" yes "
" yes"
"yes "

ويمكننا أن نستخدم هذا الأمر لإزالة أية رموز من السلسلة النصية وذلك بواسطة تحديد مصفوفة من نوع char كبارامتر لهذا الأمر إليك المثال الذي يوضح ذلك:

```
Console.WriteLine("input any statement");
char [] trimChars={' ','e','s'};
string userResponse = Console.ReadLine();
userResponse =userResponse .ToLower ();
userResponse = userResponse.Trim(trimChars );
if (userResponse.ToLower() == "y")
    Console.WriteLine("your input True");
else
    Console.WriteLine("your input False");
Console.ReadLine();
```

وبهذه الطريقة فإننا سنتخلص من اية رموز زائدة تمت كتابتها بصورة خاطئة إن هذا سيؤدي إلى معالجة حالة إدخال مثل:

"yeeeeeeeeeees"
" y"
"yesssssssssssss'

يمكننا استخدام الأمرين TrimStart() و TrimEnd() واللذين سيقومان بالتخلص من الفراغات في بداية أو نهاية السلسلة النصية ويمكن أن يتقبلا مصفوفة من الرموز char كبارامتر لها في حال أردنا إزالة رموز معينة من بداية أو نهاية السلسلة النصية.

هناك أمران أخران لمعالجة السلاسل النصية ويمكننا استخدامهما لأمور متعلقة بالفراغات في السلاسل النصية وهما: PadLeft() و PadRight() يسمح هذان الأمران بإضافة عدد من الفراغات إلى يمين أو يسار السلسلة النصية وذلك لإيصال السلسلة النصية إلى طول محدد يمكننا استخدام هذين الأمرين كما في المثال التالي:

```
string myString="Hussam";
myString = myString.PadLeft(10);
```

ستؤدي هذه الشيفرة إلى وضع أربعة فراغات في بداية الكلمة "Hussam" في المتحول myString.

يمكن لهذه المناهج أن تفيدنا في محاذاة السلاسل النصية ضمن أعمدة وهو أمر مهم عند وضع سلاسل الأرقام تحت بعضها البعض.

وكما في الأمر Trim() فإن هذين الأمرين يتقبلان بارمترا آخر بالإضافة إلى البارامتر <desiredLength> ألا وهو الرمز الذي نود أن نضعه في المساحة الزائدة للسلسلة النصية بدلا من الفراغات على سبيل المثال:

```
string myString="Hussam";
myString = myString.PadLeft(10,'-');
```

النتيجة هي أن المتحول myString سيحتوي على السلسلة النصية"----Hussam".

هناك العديد من أوامر معالجة السلاسل النصية الكثير منها مهم في أوضاع محددة وخاصة سوف نتحدث عن هذه الأوامر عندما نستخدمها في الفصول القادمة وقبل أن ننتقل إلى تطبيق عملي حول معالجة السلاسل النصية فإننا سنستعرض جدولا بالأوامر المطبقة على السلاسل النصية والتي مرت معنا في هذا الفصل:

الدالة أو الخاصية	الوصف
ToCharArray	يعيد هذا الامر مصفوفة من نوع char تمثل عناصرها الرموز المكونة للسلسلة النصية.
ToLower	تحويل جميع الحروف اللاتينية للسلسلة النصية إلى حروف صغيرة.
ToUpper	تحويل جميع الحروف اللاتينية للسلسلة النصية إلى حروف كبيرة.
Trim	إزالة أي فراغات من السلسلة النصية ويمكننا استخدام الامر لإزله رموز محددة من السلسلة النصية.
TrimStart	إزالة أي فراغات من بداية السلسلة النصية ويمكننا استخدام الامر لإزلهرموز محددة من بداية السلسلة النصية إن وجدت.
TrimEnd	إزالة أي فراغات من نهاية السلسلة النصية ويمكننا استخدام الامر لإزلهرموز محددة من نهاية السلسلة النصية إن وجدت.
PadLeft	إضافة عدد من الفراغات إلى يسار السلسلة النصية ويمكننا استخدام الأمر لإضافة رمز محدد عدد من المرات إلى يسار السلسلة النصية.

PadRight	إضافة عدد من الفراغات إلى يمين السلسلة النصية ويمكننا استخدام الأمر لإضافة رمز محدد عدد من المرات إلى يمين السلسلة النصية.
Split	يعود بمصفوفة منوع char عبارة عن حروف الكلمة إذا كانت من نوع char كما يعود بمصفوفة من نوع string عبارة عن كلمات الجملة منفصلة عن بعضها بفراغ إن كانت من نوع string.
Length	يعود بقيمة صحيحة هي عبارة عن عدد الاحرف في المصفوفة.
Compare	هذا الأمر يستخدم للمقارنة بين أكثر من كلمة ويعطي القيمة 0 إذا كانت الكلمتان متساويتان ويعطي الرقم 1 إذا كانت الأولى أكبر من الثانية ويعطي الرقم -1 إذا كانت الثانية أكبر من الاولى وهذا الأمر حساس لحالة الأحرف في اللغة اللاتينية يعنى waled لا تشبه WALED ويرجع القيمة -1 لأنه يعتبر الحروف الكبيرة أكبر من الحروف الصغيرة أما بالنسبة للحروف العربية فيمكن تجاهل التشكيل والكشيدة يعنى وليـــد تساوى وليد.
Concat	يعمل هذا الأمر على دمج نصين أو كلمتان مع بعض استخدامه مثل استخدام + ويكون الناتج الكلمتين أو النصين معا.
Join	يعمل هذا الأمر على دمج نصين أو كلمتان مع بعض لاكن الفرق بينه وبين Concat أنه يملك بارمترا يسمح بوضع فاصلة بين النصين أو فاصلةمنقوطة أو نقطتين فوق بعض أو أي رمز.
Contains	هذا الأمر يسمح بالتأكد من وجود نص أو كلمة ضمن نص معين ويعود بقيمة منطقية إما True أو False.
Copy	هذا الأمر يعمل على نسخ محتوى سلسلة نصية إلى سلسلة نصية أخرى.
CopyTo	يعمل هذا الأمر على نسخ سلسلة نصية إلى مصفوفة من نوع char وهو يحوي على أربع بارامترات الأول int (بداية الإضافة بالنسبة للنص المضاف) والثاني مصفوفة من نوع char (يضاف إليها string) والثالث من نوع int (بداية الإضافة بالنسبة للمضاف إليه) والرابع من نوع int (نهاية الإضافة من المضاف).
EndsWith	يعطي قيمة منطقية للاستعلام عما إذا كان النص ينتهي بكلمة معينة.
StartsWith	يعطي قيمة منطقية للاستعلام عما إذا كان النص يبدأ بكلمة معينة.
Equals	يعطي قيمة منطقية إذا كان النصين متساويان.
Remove	يستخدم هذا الأمر لإزله نص معين إما بتحديد نقطة البداية فقط أو بتحديد نقطة البداية والمجال.
Replace	يستخدم هذا الأمر لاستبدال حرف بحرف أو كلمة بكلمة.
Substring	يعود هذا الأمر بنص جديد من النص القديم وذلك بتحديد نقطة البداية وممكن تحديد نقطة البداية والنهاية

IndexOf	يعمل هذا الأمر في البحث عن حرف أو حروف ويرجع بقيمة عددية و هي مكان الحرف في النص أو أول حرف في الكلمة (لو كنت يبحث عن كلمة كاملة) ويعود بسالب واحد لو غير موجود الحرف أو الكلمة في النص.

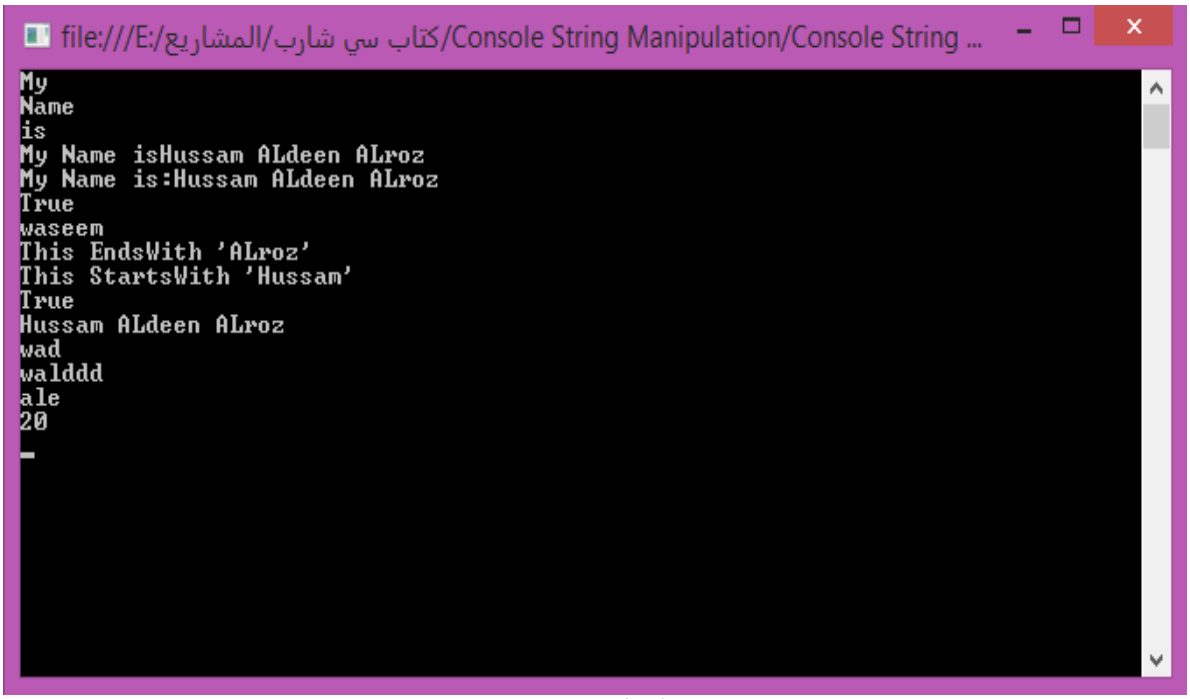
تطبيق حول معالجة النصوص:

1- قم بإنشاء تطبيق Console جديد باسم Console String Manipulation.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

```
static void Main(string[] args)
{
    string statement = "My Name is";
    string name = "Hussam ALdeen ALroz";
    string name1 = "HUSSAM ALDEEN ALROZ";
    char[] simpol = { ' ' };
    string[] myWords;
    myWords = statement.Split(simpol);
    foreach (string myWord in myWords)
    {
        Console.WriteLine(myWord );
    }
    Console.WriteLine(string.Concat(statement, name));
    Console.WriteLine(string.Join (":",statement, name));
    string search = "Name";
    bool b = statement.Contains(search) ? true : false;
    Console.WriteLine(b.ToString());
    string reuslt = null;
    string Mystr = "baseem";
    char[] c = new char[] { 'w', 'a', 'l', 'e', 'e', 'd' };
    Mystr.CopyTo(1, c, 1, Mystr.Length-1);
    for (int i = 0; i < c.Length; i++)
    {
        reuslt += c[i].ToString();
    }
    Console.WriteLine(reuslt);
    if (name.EndsWith("ALroz"))
    {
        Console.WriteLine("This EndsWith 'ALroz' ");
    }
    if (name1.StartsWith("Hussam", StringComparison.OrdinalIgnoreCase))
    {
        Console.WriteLine("This StartsWith 'Hussam' ");
    }
    bool w = name.Equals(name1,
        StringComparison.OrdinalIgnoreCase) ? true : false;
    Console.WriteLine(w);
    name1 = string.Copy(name);
    Console.WriteLine(name1);
    string str = "waleed";
    string reuslt1 = str.Remove(2, 3);
    Console.WriteLine(reuslt1);
    string reuslt2 = str.Replace('e', 'd');
    Console.WriteLine(reuslt2);
```

```
        string reuslt3 = str.Substring(1, 3);
        Console.WriteLine(reuslt3);
        string str1 = "private void button click sender EventArgs e";
        int index = str1.IndexOf("click");
        Console.WriteLine(index .ToString ());
    }
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 فيظهر الشكل (22-5).



الشكل (22-5)

كيفية العمل:

How it Works:

سوف اترك لك أيضا أمر التعليق على هذا التطبيق أعتقد أنه بات من السهل عليك قراءته وتفهم ما يقوم به.

الإكمال التلقائي للتعليمات في Visual Studio:

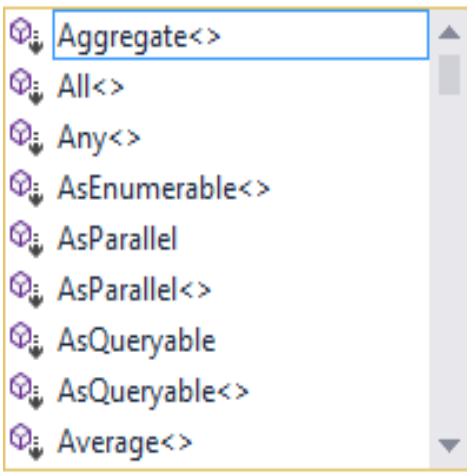
AutoComplete in Visual Studio:

1- قم بإنشاء تطبيق Console جديد باسم Console AutoComplete.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C#.

```
static void Main(string[] args)
{
    string myString = "This is a test.";
```

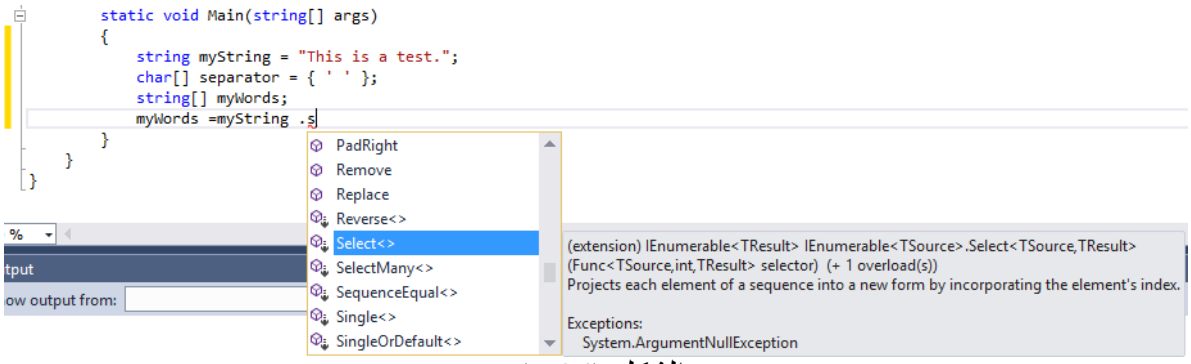
```
        char[] separator = { ' ' };
        string[] myWords;
        myWords =myString .
    }
```

3- لاحظ بعد كتابتك للنقطة الأخيرة في الشيفرة (أي عند myString.) ستلاحظ ظهور قائمة منسدلة كما في الشكل (5-23).



الشكل (5-23)

4- اضغط على المفتاح s دون أن تحرك المؤشر. عندئذ سيتغير موقع التحديد في القائمة المنسدلة إلى بند آخر وسيظهر تلميح ضمن إطار رمادي بجانب القائمة المنسدلة كما في الشكل (5-24).



الشكل (5-24)

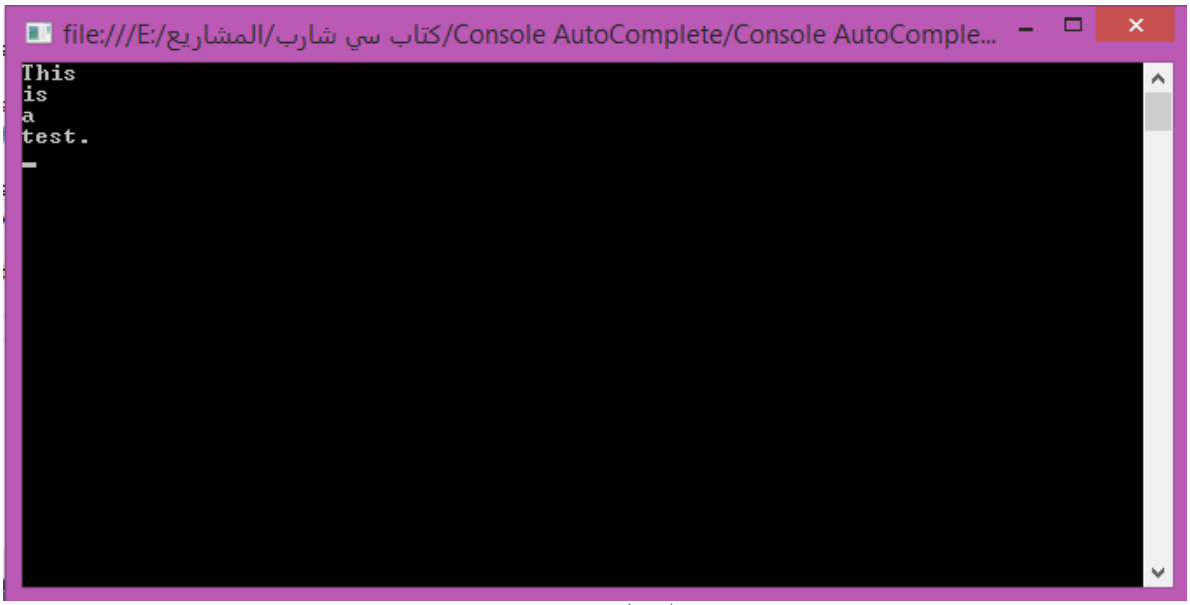
5- اضغط على المفتاح p دون ان تحرك المؤشر عندئذ سيتغير موقع التحديد في القائمة المنسدلة إلى بند آخر باسم split وسيظهر تلميح ضمن إطار رمادي بجانب القائمة المنسدلة وبضغطك على مفتاح المسطرة ستلاحظ أنه تمت عملية إضافة الكلمة تلقائيا دون أن تكون بحاجة لكتابتها كاملة.

6- أكمل الشيفرة ولاحظ ظهور القائمة المنسدلة وكيف ستساعدك في كتابة الشيفرة بحيث نحصل على الشيفرة البرمجية التالية:

```
    static void Main(string[] args)
    {
        string myString = "This is a test.";
        char[] separator = { ' ' };
        string[] myWords;
```

```
        myWords = myString.Split(separator);
        foreach (string word in myWords)
        {
            Console.WriteLine(word);
        }
        Console.ReadKey();
    }
```

7- نفذ التطبيق بالضغط على مفتاح F5 فيظهر الشكل (25-5).



الشكل (25-5)

أخيرا بعد العديد من التطبيقات المتعددة التي كتبناها بواسطة Visual Studio لاحظ ظهور قائمة منسدلة عند كتابة أي أمر ويعود السبب في ذلك إلى تغذية الحس الذكي (interlines) في فيجوال ستوديو وتوفير الكثير من الجهد و العناء عليك فعند وضع النقطة بعد المتحول myString سيتعرف البرنامج على نوع المتحول وبما أن المتحول من نوع string فسيقوم بعرض جميع الأوامر التي يمكننا تطبيقها على السلاسل النصية وعند هذه النقطة يمكننا أن نتوقف عن الكتابة ومن ثم تحديد الأمر الذي نود باستخدام مفاتيح الأسهم إلى أعلى أو أسفل وعند كل تحديد لأمر ما من القائمة المنسدلة سيظهر تلميح ضمن إطار أصفر يعطي شرحا مختصرا عن الأمر وعن كيفية استخدامه وماهي البارامترات التي يمكن أن يتقبلها.

عندما نبدأ بكتابة المزيد من رموز الأمر عند هذه النقطة سينتقل التحديد في القائمة المنسدلة إلى الأمر الذي تود الوصول إليه مباشر إن هذه الميزة في فيجوال ستوديو مهمة ومفيدة جدا خصوصا عندما لا نكون متأكدين من صحة كتابة الأوامر أو نود التعرف على جميع الأوامر التي تتقبلها أنواع محددة من المتحولات كالمتحولات من نوع string.

الخلاصة:

Summary:

لقد استفدنا من معلوماتنا التي حصلنا عليها في الفصول السابقة عن المتحولات في فصلنا هذا للتعرف على أنواع أخرى أكثر تعقيدا لقد تحدثنا في البداية عن تحويل الأنواع باعتبار أننا سنحتاج لاستخدام هذه التقنية في مواضيع كثيرة أثناء برمجة التطبيقات.

لقد تعرفنا على التعدادات ورأينا كيف تمثل أسلوبا ممتازا لتعريف أنواع من المتحولات لا تقبل إلا قيما محددة وكذلك تعرفنا على البنى وعلى كيفية الاستفادة منها لإنشاء تراكيب من المتحولات ضمن متحول واحد كما تعرفنا على المصفوفات بكافة أشكالها إن تلك الأنواع رغم كونها معقدة قليلا إلا أنها مفيدة جدا وضرورية في الكثير من المواضيع.

كما تعرفنا بعد ذلك على خواص هذه المصفوفات وكيف لنا أن نستفيد منها في بناء تطبيقاتنا المختلفة.

وتعرفنا على اللوائح وخواصها وأخير تحدثنا عن معالجة السلاسل النصية وتناولنا بعض التقنيات والمبادئ الأساسية في ذلك ولقد شرحنا بعضا منها فقط وليس جميعها ويمكنك أن تجرب تلك التي لم نتناولها والتعرف عليها.

الفصل السادس

التوابع أو الدوال

أن جميع الأمثلة البرمجية التي تناولناها حتى الان لم تكن تمثل إلا كتلة برمجية واحدة وربما تخلل هذه الكتلة بعض الحلقات لتكرار أسطر معينة من الشيفرة أو بعض التفرع لتنفيذ تعليمات اعتمادا على تحقيق شرط ما لكن إذا كنا بحاجة للقيام بعمليات ما على بياناتنا فإن هذا يعني وضع الشيفرة التي ستقوم بهذه المهمة في الموضع الذي تود لها أن تنفذ عنده.

لا تزال بنية التطبيقات التي يمكننا تطويرها حتى الأن محددة نوعا ما فعلى سبيل المثال قد نحتاج إلى إيجاد القيمة الأكبر في مصفوفة وقد نحتاج للقيام بذلك في نقاط مختلفة من البرنامج يمكننا أن ننسخ الشيفرة ذاتها ونلصقها في الموضع الذي نود أن نستخدمها فيه إلا أن لذلك مشاكل عدة فإذا اضطررت إلى إحداث تعديل بسيط في هذه الشيفرة فإن عليك أن تقوم بهذا التعديل في عدة أماكن من البرنامج وإذا نسيت تعديل أحد المواضع فإنك قد تحصل على نتائج مخيبة للآمال هذا بالإضافة إلى الطول الزائد للشيفرة.

إن الحل لهذه المشكلة يقضي باستخدام التوابع (functions) تعرف التوابع في لغة C# بانها عبارة عن كتل من الشيفرة البرمجية التي يمكننا أن ننفذها في أي موضع من البرنامج.

توضيح:

تسمى أنواع محددة من التوابع بالمناهج (methods) إن لهذا المصطلح معنى خاصا جدا في البرمجة ضمن .NET. وسوف تتوضح لك الأمور أكثر لاحقا في القسم التالي من هذا الكتاب لذا سنتجنب استخدام هذا المصطلح هنا.

على سبيل المثال يمكننا أن نكتب تابعا لاحتساب القيمة العظمى في مصفوفة ويمكننا أن نستخدم هذا التابع في أي موضع من الشيفرة واستخدام الاسطر البرمجية نفسها في كل مرة وبما أننا نكتب هذه الشيفرة مرة واحدة ونستدعيها عدة مرات فإن أي تعديل على الشيفرة سيؤثر على النتائج متى استخدم هذا التابع إن هذا يعطي انطباعا على الشيفرة القابلة لإعادة الاستخدام (reusable code).

تجعل التوابع شيفرتنا البرمجية مقروءة أكثر وذلك باعتبار أننا نقوم بتجميع مهام محددة ضمن كتلة برمجية واحدة وعند قيامنا بذلك سيصبح جسم برنامجنا قصير جدا وذلك لأن العمل الحقيقي موزع على كتل

برمجية أخرى إن هذا المبدأ مشابه لطي وتوسعة مناطق من الشيفرة في Visual Studio ويعطي ذلك بنية منطقية أكثر لتطبيقنا.

ويمكن أن تستخدم التوابع لإنشاء شيفرة متعددة الأغراض (multi-purpose) بحيث يمكن للتوابع أن تؤدي المهام نفسها على بيانات مختلفة. فيمكننا أن نزود التوابع بالمعلومات التي ستعمل عليها وفي تلك الحالة تسمى هذه المعلومات بالوسطاء أو البارامترات (parameters) ويمكننا ان نحصل على النتائج من هذه التوابع وتسمى هذه النتائج بالقيم المعادة (return values).

وفي مثالنا السابق يمكننا أن نزود التابع بالمصفوفة كبارامتر حيث سيقوم التابع بالبحث عن القيمة الأكبر ومن ثم سنحصل على هذه القيمة على شكل قيمة معادة من هذا التابع.

هذا يعني أنه يمكننا أن نستخدم التابع نفسه لإيجاد القيمة الأكبر لأي مصفوفة خلال البرنامج إن البارامترات والقيمة المعادة للتابع تعرف ما يسمى بتوقيع التابع (signature).

ثم سنتناول موضوع مدى المتحولات (variable scope) ويشتمل على مفهوم محلية المتحولات واقتصار استخدامها على مناطق محددة من الشيفرة وهو موضوع هام جدا خصوصا عند تجزئة الشيفرة على توابع متعددة.

بعد ذلك سنلقي نظرة عميقة على تابع مهم جدا في C# ألا وهو التابع Main () سنرى كيف يمكننا أن نستخدم هذا التابع لبناء سلوك لتطبيقنا بحيث يستخدم بارامترات الدخل من سطر الأوامر مباشرة (كما في أوامر DOS) وهو ما يمكننا من تمرير المعلومات على التطبيقات أثناء تنفيذها مباشرة.

سننتقل بعد ذلك إلى تناول ميزة إضافية لأنواع البنى string التي رأيناها في الفصل السابق وهي إمكانية تزويد البنى بالتوابع على هيئة أعضاء ضمن تعريف البنية.

وأخيرا سنعود إلى موضوع التوابع وسنتحدث عن بعض المواضيع التقدمية المتعلقة بهما: التحميل الزائد للتوابع (function overloading) والمفوضات (delegates) والاستدعاء التعاودي (recursion) يمثل التحميل الزائد للتوابع تقنية تمكننا من توفير توابع عديدة لها الاسم نفسه ولكن بتواقيع مختلفة (أي ببارامترات وقيم معادة مختلفة) أما المفوض فهو نوع من المتحولات يسمح لنا باستخدام التوابع بصورة غير مباشرة فالمفوض نفسه يمكن استخدامه لاستدعاء أي تابع له توقيع محدد مما يعطي إمكانية الاختيار من بين توابع عديدة أثناء تنفيذ البرنامج وأما التعاودية فهي استدعاء التابع ضمن شيفرته نفسها أي استدعاء التابع ذاتيا.

تعريف واستخدام التوابع:

Defining and Using Functions:

سنتعلم في هذا الفصل كيف نضيف التوابع إلى تطبيقاتنا ومن ثم استدعاء هذه التوابع من ضمن شيفرتنا البرمجية. سوف نبدأ بتوابع بسيطة بتبادل أية بيانات بينها وبين الشيفرة التي تستدعيها ومن ثم سننتقل إلى استخدام أكثر تقدما للتوابع.

تطبيق حول تعريف واستخدام توابع أساسية:

7- قم بإنشاء تطبيق Console جديد باسم Console Defining and Using Functions.
8- أضف الشيفرة البرمجية التالية إلى محرر نصوص C# في class Program:

```
class Program
{
    static void write()
    {
        Console.WriteLine("Text output from Function.");
        Console.ReadKey();
    }
```

9- أضف الشيفرة البرمجية التالية إلى محرر نصوص C# في جسم التابع الرئيسي Main لتصبح الشيفرة الكلية بالشكل:

```
namespace Console_Defining_and_Using_Functions
{
    class Program
    {
        static void write()
        {
            Console.WriteLine("Text output from Function.");
            Console.ReadKey();
        }
        static void Main(string[] args)
        {
            write();
        }
    }
```

10- نفذ التطبيق بالضغط على مفتاح F5 فيظهر الشكل (1-6).

الشكل (1-6).

كيفية العمل:

How it Works:

تقوم أسطر الشيفرة الخمسة التالية بتعريف تابع جديد باسم write ():

```
    static void write()
    {
        Console.WriteLine("Text output from Function.");
        Console.ReadKey();
    }
```

تقوم الشيفرة الموجودة ضمن التابع write () بإخراج نص بسيط على نافذة الخرج لن نهتم الآن بالشيفرة المكتوبة ضمن التابع بقدر ما نحن مهتمين بآلية تعريف واستخدام التوابع.

يتضمن تعريف التابع هنا على ما يلي:

- ✓ كلمتان مفتاحيتان هما static و void.
- ✓ اسم التابع متبوع بقوسين write ().
- ✓ كتلة من الشيفرة موضوعة ضمن القوسين "{}".

نلاحظ أن الشيفرة التي استخدمناها لتعريف التابع write () مشابهة جدا لشيفرة أخرى في تطبيقاتنا:

```
static void Main(string[] args)
{
    .....
}
```

ويعود ذلك إلى ان جميع الشيفرات البرمجية في أمثلتنا السابقة كانت تمثل جزءا من تابع وهذا التابع له الاسم Main () وهو يمثل نقطة الدخول (entry point) الأولى لتطبيقات Console فعند ترجمة شيفرة C# لتطبيق Console سيتم تنفيذ الشيفرة الموجودة ضمن تابع نقطة الدخول وذلك باستدعاء هذا التابع داخليا وعند انتهاء تنفيذ هذا التابع سينتهي التطبيق من العمل إن جميع شيفرات C# التنفيذية يجب أن تحتوي على نقطة دخول.

ملاحظة:

نقصد بشيفرة C# التنفيذية هي شيفرة البرنامج التي يتم تنفيذها مباشرة مثل الشيفرة البرمجية للملفات التنفيذية ذات الامتداد exe.

إن الاختلاف الوحيد بين التابع Main () والتابع write () الذي عرفناه للتو (بغض النظر عن الشيفرة الموجودة ضمن التابعين) هو ان التابع Main () يتضمن بعض الشيفرة ضمن القوسين "()" بعد اسم التابع بينما لا يتضمن تابعنا write () أي شيء ضمن هذين القوسين هنا هو المكان الذي نعرف ضمنه بارامترات التابع وهو ما سنناقشه بعد قليل.

وكما نو هنا مسبقا فإننا استخدمنا الكلمتين static و void لتعريف التابع write () وهما نفس الكلمتين المستخدمتين لتعريف التابع Main () تشير الكلمة static إلى مفاهيم متعلقة بالبرمجة كائنية التوجه وهو ما سنتحدث عنه في فصول لاحقة من هذا الكتاب وحاليا سنكتفي بالتنويه إلى وجوب وضع هذه الكلمة في جميع التوابع التي سسننشأها في هذا الفصل.

أما كلمة void فهي أبسط مما يمكننا شرحه فهذه الكلمة تشير إلى أن هذا التابع لا يعيد أي قيمة سوف نجد لاحقا في هذا الفصل ما سنحتاج لكتابته عندما نريد للتابع أن يعيد قيمة ما.

وبالعودة إلى شيفرة التطبيق نلاحظ ان التابع Main () لا يتضمن إلا على سطر برمجي وحيد:

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

ذاك السطر هو السطر الذي يقوم باستدعاء التابع write () وكما تلاحظ فإننا سنكتب ببساطة اسم التابع متبوعا بقوسين من الشكل "()" وعندما يصل تنفيذ الشيفرة إلى هذا الموضع سيتم تنفيذ الشيفرة الموجودة ضمن التابع write () وعند الانتهاء منها سيعود التحكم إلى التابع Main () لما بعد سطر استدعاء التابع.

ملاحظة:

إن وجود الأقواس "()" حتى ولو كانت خالية ضمن توقيع التابع او استدعائه واجب وإذا حاولت إزالة هذه الأقواس فإن الشيفرة لن تترجم.

القيم المعادة:

Return Values:

إن الطريقة الأبسط لتبادل البيانات مع التوابع تقتضي باستخدام قيمة معادة وتعامل التوابع التي تعيد قيما كما تعامل المتحولات تماما وهذا يعني أن بإمكاننا استخدام التوابع ضمن التعابير كما نستخدم المتحولات وسوف نرى ذلك لاحقا.

على سبيل المثال لنفترض أن لدينا تابعا باسم getString() وهو يعيد سلسلة نصية ما عندئذ يمكننا كتابة شيفرة كما يلي:

```
string myString;
myString = getString();
```

ولنفرض أيضا أن لدينا تابعا باستخدام getVal() وهو يعيد قيمة من نوع double عندئذ يمكننا كتابة التعبير الحسابي التالي:

```
double myVale;
double multipler = 5.3;
myVale = getVal() * multipler;
```

إذا أردنا للتابع أن يعيد قيمة فإن علينا القيام بما يلي:

- تحديد نوع القيمة المعادة وذلك ضمن تصريح أو توقيع التابع مكان الكلمة void.
- استخدام الكلمة return لإنهاء تنفيذ التابع ونقل القيمة المعادة إلى الشيفرة التي استدعت التابع.

وبناء على ذلك يمكننا وضع صيغة للتوابع التي تعيد قيمة كما يلي:

```
static <returnType><functionType>()
{
 ......
```

```
return <returnValue>;
}
```

يمثل <returnType> نوع بيانات القيمة التي سيعيدها التابع ويمكن أن يأخذ أي نوع كان سواء كان نوعا بسيطا مثل string أو int أو نوعا مركبا مثل بنى struct أو تعدادا أما <returnValue> فيجب أن تكون من نفس نوع <returnType> أو يمكن أن تحول بصورة مطلقة إلى هذا النوع.

يمكننا أن نكتب تابعا كما يلي مثلا:

```
class Program
{
    static double getVal()
    {
        return 5.2;
    }
```

إن هذه الشيفرة مقبولة تماما إلا أنك لن تستخدم تابعا لإعادة قيمة معلومة (يمكننا تحقيق ذلك باستخدام المتحولات الثابتة (أي التي يصرح عنا بالكلمة const)) وإنما سنستخدم التوابع لإعادة نتيجة معالجة ما يقوم التابع بها.

عندما يصل التنفيذ إلى الكلمة return سيعود التحكم إلى الشيفرة المستدعاة مباشرة ولن تنفذ أية شيفرة في التابع بعد هذه التعليمة لكن هذا لا يعني أن تعليمة return لا تتوضع إلا في نهاية جسم التابع يمكننا أن نستخدم أي عدد من تعليمات return داخل جسم التابع ويمكن ان يكون هذا الاستخدام منطقيا خصوصا ضمن بنى التفرع أو الحلقات ولكن تذكر انه متى تم الوصول إلى هذه التعليمة فسيتوقف التابع عن التنفيذ مباشرة وينتقل إلى النقطة التي تم استدعاءه منها على سبيل المثال:

```
class Program
{
    static double getVal()
    {
        double checkVal;
        if (checkVal <5)
            return 4.2;
        return 5.2;
    }
```

هنا سيتم إعادة قيمة واحدة من القيمتين التي يمكن أن يعيدها التابع (4.2أو5.2) وذلك بحسب قيمة checkVal إن التقييد الوحيد هنا هو أن تعليمة return يجب أن تنفذ قبل الوصول إلى قوس إغلاق جسم التابع "}" أي أن الشيفرة التالية غير مقبولة:

```
class Program
{
    static double getVal()
    {
        double checkVal;
        if (checkVal <5)
        return 5.2;
    }
```

فإذا كان checkVal>=5 فلن تنفذ تعليمة return أبدا وهو امر غير مسموح به.

وكملاحظة أخيرة يمكننا أن نستخدم الكلمة return مع التوابع المصرح عنها بالكلمة void أي التوابع التي لا تعيد قيمة وإذا قمنا بذلك فسيتوقف التابع عن العمل عندما يصادف هذه الكلمة مباشرة لكن في هذه الحالة لا يمكننا ان نضع أي شيء بين الكلمة return والفاصلة المنقوطة التي تليها وإلا فسينتج عن ذلك خطا من المترجم.

البارامترات:

Parameters:

عندما نود أن نزود التابع ببارامترات أو وسطاء فإن علينا أن نحدد التالي:

- لائحة البارامترات التي يتقبلها التابع بالإضافة إلى نوع كل بارامتر على حدة حيث يفصل بين كل بارامتر برمز الفاصلة "،".
- لائحة البارامترات مطابقة للائحة في تصريح التابع وذلك عند استدعاء هذا التابع حيث يفصل بين كل بارامتر برمز الفاصلة "،" أيضا ولكن دون ذكر النوع.

أي أن للتابع ذو البارامترات الصيغة التالية:

```
static <returnType><functionType>(<parmType> <pramName>,..)
{
 ......
return <returnValue>;
}
```

ويمكننا ان نحدد أي عدد من البارامترات لكل واحدة منها اسم ونوع وتمثل هذه البارامترات متحولات يمكن الوصول إليها ضمن جسم التابع فقط.

على سبيل المثال يمكن أن يكون لدينا تابع يقوم باحتساب مساحة مستطيل كما يلي:

```
class Program
{
    static double recArea(double height,double width)
    {
        return height * width;
    }
```

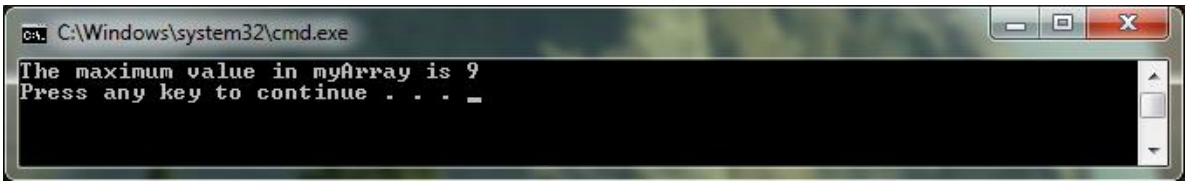
تطبيق حول تبادل البيانات مع التوابع:

1- قم بإنشاء تطبيق Console جديد باسم Console Parameters.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C# في class Program:

```
class Program
{
    static int MaxValue(int[] intArray)
    {
        int maxVal = intArray[0];
        for (int i = 1; i < intArray.Length; i++)
        {
```

```
            if (intArray[i] > maxVal)
                maxVal = intArray[i];
        }
        return maxVal;
    }
    static void Main(string[] args)
    {
        int[] myArray = { 1, 8, 3, 6, 2, 5, 9, 3, 0, 2 };
        int mxVal = MaxValue(myArray);
        Console.WriteLine("The maximum value in myArray is {0}", mxVal);
    }
}
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 فيظهر الشكل (2-6).



الشكل (2-6)

كيفية العمل:

How it Works:

تتضمن هذه الشيفرة التابع الذي يقوم بالمثال الذي تحدثنا عنه في بداية هذا الفصل حيث يتقبل هذا التابع مصفوفة من الأرقام كبارامتر ويقوم بإعادة القيمة الأكبر في المصفوفة التابع هو MaxValue() وهو كما يلي:

```
class Program
{
    static int MaxValue(int[] intArray)
    {
        int maxVal = intArray[0];
        for (int i = 1; i < intArray.Length; i++)
        {
            if (intArray[i] > maxVal)
                maxVal = intArray[i];
        }
        return maxVal;
    }
```

لهذا التابع بارامتر وحيد وهو عبارة عن مصفوفة من نوع int باسم intArray ولهذا التابع قيمة معادة أيضا وهي من نوع int إن شيفرة جسم التابع بسيطة وهي عبارة عن خوارزمية لإيجاد القيمة الأكبر في مصفوفة سيتم أولا تهيئة المتحول maxVal بأول قيمة من المصفوفة ويتم بعد ذلك مقارنة قيمة هذا المتحول مع جميع القيم الأخرى في المصفوفة وعندما نجد أن هناك قيمة أكبر من القيمة الحالية للمتحول MaxVal سيأخذ هذا المتحول هذه القيمة وستتكرر عملية المقارنة مع القيم المتبقية إلى آخر عنصر في المصفوفة وبالتالي عندما تنتهي الحلقة سيتضمن المتحول mxValue أكبر قيمة في المصفوفة وستعاد هذه القيمة بواسطة تعليمة return.

أما الشيفرة في التابع Main() فهي تقوم بالتصريح عن مصفوفة عددية من نوع int وتهيئتها بقيم أولية وذلك لتمريرها كبارامتر للتابع MaxValue():

```
int[] myArray = { 1, 8, 3, 6, 2, 5, 9, 3, 0, 2 };
```

بعد ذلك سيتم اسناد القيمة المعادة من التابع MaxValue() إلى المتحول mxVal ذو النوع int (لاحظ كيف قمنا بتمرير المصفوفة إلى هذا التابع):

```
int mxVal = MaxValue(myArray);
```

وأخيرا سنقوم بطباعة القيمة الأكبر على الخرج بواسطة الأمر:

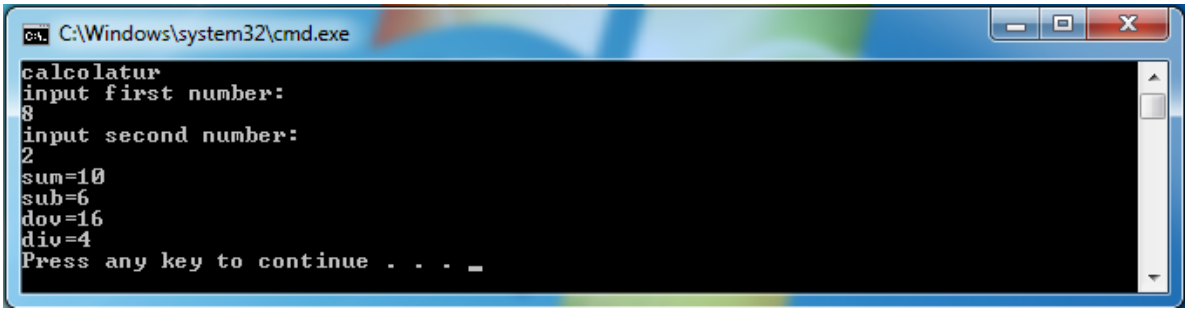
```
Console.WriteLine("The maximum value in myArray is {0}", mxVal);
```

تطبيق أخر حول تبادل البيانات مع التوابع:

1- قم بإنشاء تطبيق Console جديد باسم Console function calculator.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C# في class Program:

```
namespace Console_function_calculator
{
    class Program
    {
        static double sum(double n1, double n2)
        {
            return n1 + n2;
        }
        static double Sub(double n1,double n2)
        {
            return n1 - n2;
        }
        static double dov(double n1,double n2)
        {
            return n1 * n2;
        }
        static double div(double n1,double n2)
        {
            return n1 / n2;
        }
        static void inpu()
        {
            Console.WriteLine("calcolatur");
        }
        static void Main(string[] args)
        {
            inpu();
            Console.WriteLine("input first number:");
            double number1 = double.Parse(Console.ReadLine());
            Console.WriteLine("input second number:");
            double number2 = double.Parse(Console.ReadLine());
            Console.WriteLine("sum={0}\nsub={1}\ndov={2}\ndiv={3}",
                sum(number1, number2), Sub(number1, number2),
                dov(number1, number2), div(number1, number2));
        }
    }
}
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 فيظهر الشكل (3-6).



الشكل(3-6)

كيفية العمل:

How it Works:

أظن أن المثال سهل القراءة سأترك لك التعليق على التطبيق.

تطابق البارامترات:

Parameters Matching:

عندما نستدعي تابعا ما فإن علينا مطابقة البارامترات المحددة في سطر تصريح التابع ويسمى بسطر توقيع التابع مع الأنواع المحررة إليه ضمن سطر استدعاء التابع إن هذا التطابق يشتمل على تطابق عدد البارامترات المحررة وعلى تطابق الأنواع لكل بارامتر وبالتالي إذا كان لدينا تابع بالصورة الآتية:

```
class Program
{
    static void myFunction (string myString,double myDouble)
    {
      ...
    }
```

فإننا لا يمكننا استدعاء هذا التابع كما يلي:

```
myFunction(2.6, "Hello");
```

فلقد حاولنا هنا أن نمرر قيمة من نوع double كبارامتر أول وهو غير مطابق لنوع البارامتر الأول في سطر توقيع التابع كما أننا مررنا قيمة من نوع string كبارامتر ثاني وهو غير مطابق لنوع البارامتر الثاني والذي هو من نوع double.

كما أنه لا يمكننا كتابة:

```
myFunction("Hello");
```

فلقد قمنا فقط بتمرير قيمة للبارامتر الأول ولم نمرر قيمة للبارامتر الثاني.

إن محاولة استدعاء التابع السابق وفقا للاستدعاءين السابقين سيؤدي إلى حصول خطأ من المترجم.

بالعودة إلى التطابق السابق فإن هذا يعني أن التابع MaxValue () لا يمكن أن يستخدم إلا لإيجاد القيمة الأكبر في مصفوفة من نوع int فقط فإن استبدلنا الشيفرة الموجودة في التابع Main () بالشيفرة التالية:

```
static void Main(string[] args)
{
    double [] myArray = { 1.3, 8.9, 3.3, 6.6, 2, 5.5, 9.4};
    double mxVal = MaxValue(myArray);
    Console.WriteLine("The maximum value in myArray is {0}", mxVal);
    Console.ReadKey();
}
```

فإن هذا سيؤدي إلى حدوث خطأ باعتبار البارامتر الممرر مختلف عن النوع المعروف في توقيع التابع.

سوف نتعلم لاحقا في هذا الفصل تقنية التحميل الزائد للعوامل والتي تمكننا من الالتفاف حول هذه المشكلة.

مصفوفة البارامترات:

Parameters Array:

هناك نوع واحد خاص للبارامترات في لغة C# ويجب أن يأتي هذا البارامتر في نهاية لائحة البارامترات دائما ويعرف بمصفوفة البارامترات (parameter array) تسمح مصفوفة البارامترات باستدعاء التوابع بعد تزويدها بعدد متغير من البارامترات ويتم تعريف هذا النوع من البارامترات بواسطة الكلمة params.

يمكن لمصفوفة البارامترات أن تمثل طريقة مفيدة لتبسيط شيفرتنا فبدلا من تمرير المصفوفات إلى التوابع يمكننا أن نمرر عدة بارامترات من نفس النوع إلى التابع حيث ستعامل هذه البارامترات على شكل عناصر لمصفوفة ضمن التابع.

تبين الشيفرة التالية صيغة إنشاء تابع يتقبل بارامترات عادية ومصفوفة بارامترات:

```
static <returnType> <functionName> (<p1Type> <p1Name>,..,
params <typeof> [] <name>)
{
  ...
return <returnValue>;
}
```

ويمكننا أن نستدعي هذا التابع كما يلي:

```
<functionName> (<p1>,..,<val1>,<val2>,..);
```

تمثل <val1> و<val2> في هذه الصيغة قيما من النوع <type> الذي استخدمناه في تعريف صيغة التابع السابقة ليس هناك أية محدودات لعدد البارامترات التي يمكننا تحديدها هنا والتقييد الوحيد هنا هو أن تكون جميع عناصر مصفوفة البارامترات من نوع واحد كما هو محدد في تصريح التابع ويمكننا ألا نضع أية بارامترات بالمرة.

تطبيق آخر حول تبادل البيانات مع التوابع:

1- قم بإنشاء تطبيق Console جديد باسم Console Parameters2.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C# في class Program:

```
class Program
{
    static int sumVals(params int[] vals)
    {
        int sum = 0;
        foreach (int val in vals)
            sum += val;
        return sum;
    }
    static void Main(string[] args)
    {
        int sum = sumVals(1, 5, 2, 9, 8);
        Console.WriteLine("summed Values={0}", sum);
    }
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 فيظهر الشكل (4-6).

الشكل (4-6)

كيفية العمل:

How it Works:

لقد عرفنا بارامترا في التابع sumVals باستخدام الكلمة params وذلك لكي يتقبل أي عدد من القيم من النوع int (فقط):

```
    static int sumVals(params int[] vals)
    {
....
    }
```

أما شيفرة جسم التابع sumVals () فستقوم بالمرور على جميع عناصر المصفوفة vals (أي على جميع البارامترات الممررة) وستجمع هذه القيم مع بعضها البعض ومن ثم ستعيد ناتج الجمع كقيمة معادة للتابع sumVals ().

أما التابع Main () فلقد استدعيناه باستخدام خمسة بارامترات (قيم) رقمية صحيحة:

```
    int sum = sumVals(1, 5, 2, 9, 8);
```

في الحقيقة يمكننا ببساطة أن نستدعي هذا التابع دون تمرير أي بارامتر أو بارامتر واحد أو اثنين أو مئات البارامترات فليس هناك كما قلنا حدود لكمية البارامترات الممررة إلى المصفوفة البارامترية.

بارامترات المرجع وبارامترات القيمة:

Reference and Value Parameters:

إن جميع التوابع التي تعرفنا عليها حتى الان لها بارامترات بالقيمة (value parameters) إن ما أعنيه بذلك هو أنه عندما استخدمت هذه البارامترات فإنني قمت بتمرير القيمة إلى المتحول المستخدم ضمن التابع وأية تعديلات على هذا المتحول ضمن التابع لن تؤثر على القيمة الفعلية لهذا البارامتر المحدد في استدعاء التابع على سبيل المثال لنفترض تابعا يقوم بمضاعفة قيمة البارامتر الممرر إليه:

```
static void showDouble (int val)
{
    val *= 2;
        Console .WriteLine ("val double={0}",val);
}
```

لقد قمنا هنا بمضاعفة قيمة البارامتر val في هذا التابع إذا قمنا باستدعاء هذا التابع بالصورة التالية:

```
static void Main(string[] args)
{
    int myNumber = 5;
    Console.WriteLine("myNumber={0}", myNumber);
    showDouble(myNumber);
    Console.WriteLine("myNumber={0}", myNumber);
    Console.ReadKey();
}
```

أن الخرج سيكون كما في الشكل (5-6).

الشكل (5-6).

إن استدعاء التابع showDouble () مع المتحول myNumber كبارامتر لا يؤثر على قيمة المتحول myNumber في التابع Main () حتى إن تمت مضاعفة قيمة البارامتر الداخلي للتابع أي قيمة البارامتر val.

حسنا ماذا لو أردنا أن تتغير قيمة المتحول myNumber نتيجة لتغير قيمته عندما مررناه إلى التابع عندئذ علينا أن نحول التابع السابق إلى تابع يستطيع أن يعيد قيمة من نوع int ونكتب شيفرة كما يلي:

```
static void Main(string[] args)
{
    int myNumber = 5;
    Console.WriteLine("myNumber={0}", myNumber);
    myNumber = showDouble(myNumber);
    Console.WriteLine("myNumber={0}", myNumber);
    Console.ReadKey();
}
```

إن هذه شيفرة غير مريحة وسيزداد الأمر تعقيدا إذا كان التابع يعيد قيمة ما في الأساس مختلفة عن قيمة البارامتر val ونحن نعلم أن التابع لا يعيد إلا قيمة واحدة فقط.

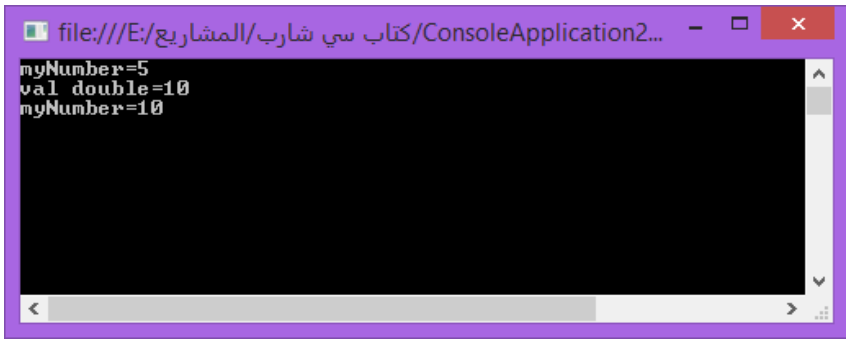
بدلا من هذا التعقيد فإننا سنمرر البارامتر بالمرجع (by reference) وهذا يعني أن التابع سيتعامل مع المتحول الممرر له مباشرة وليس مع نسخة منه لها نفس القيمة وبالتالي فإن أي تغيير لهذا المتحول ضمن جسم التابع ستنعكس على قيمة المتحول المستخدم كبارامتر ولكي نتمكن من استخدام بارامترات المرجع فإن علينا وضع الكلمة ref قبل نوع واسم البارامتر في سطر تصريح التابع كما يلي:

```
static void showDouble (ref int val)
{
    val *= 2;
        Console .WriteLine ("val double = {0}",val);
}
```

ويجب أن نشير إلى ذلك أيضا ضمن سطر استدعاء التابع كما يلي:

```
static void Main(string[] args)
{
    int myNumber = 5;
    Console.WriteLine("myNumber = {0}", myNumber);
    showDouble(ref myNumber);
    Console.WriteLine("myNumber = {0}", myNumber);
    Console.ReadKey();
}
```

والخرج في هذه الحلة سيصبح كما في الشكل (6-6).



الشكل (6-6)

لقد تغيرت قيمة المتحول myNumber هذه المرة من قبل التابع showDouble ().

هناك محددات حول المتحولات المستخدمة كبارامترات مرجعية أولا يمكن أن تخضع القيمة الممررة بالمرجع لتعديل ما ضمن التابع وبالتالي لا يمكننا أن نمرر متحولا ثابتا كبارامتر مرجعي إلى التابع أي أن الشيفرة التالية غير مقبولة:

```
static void Main(string[] args)
{
    const int myNumber = 5;
    Console.WriteLine("myNumber={0}", myNumber);
    showDouble(ref myNumber);
    Console.WriteLine("myNumber={0}", myNumber);
    Console.ReadKey();
}
```

ثانيا علينا أن نستخدم متحولا متهيئا فلا تسمح لغة C# بافتراض أن بارامتر المرجع سيهيئ في التابع الذي يستخدمه أي أن الشيفرة التالية غير مقبولة أيضا:

```
static void Main(string[] args)
{
    int myNumber;
    Console.WriteLine("myNumber={0}", myNumber);
    showDouble(ref myNumber);
    Console.WriteLine("myNumber={0}", myNumber);
    Console.ReadKey();
}
```

بارامترات الخرج:

Out Parameters:

بالإضافة لتمرير القيم بالمرجع يمكننا أن نخصص بارامترا ما على أنه بارامتر خرج باستخدام الكلمة out وتستخدم هذه الكلمة تماما مثل الكلمة ref (كمقيد للبارامتر في سطر التصريح عن التابع وفي سطر استدعاء التابع أيضا) لبارامترات الخرج سلوك مشابه لسلوك بارامترات المرجع حيث أن قيمة البارامترات ستعاد إلى المتحول المستخدم في استدعاء التابع عند الانتهاء من تنفيذ التابع في الحقيقة هناك اختلافات مهمة بين بارامترات الخرج وبارومترات المرجع فبينما من غير المقبول استخدام متحول غير مهيئ لم تسند إليه قيمة محددة كبارامتر مرجعي إلا أنه يمكننا استخدام متحول غير مهيئ كبارامتر خرج وبالإضافة إلى ذلك فإننا يجب أن نتعامل مع بارامتر الخرج على أساس أنه ليست هناك قيمة مسندة إليه وذلك بالنسبة للتابع الذي سيستخدمه هذا يعني أنه على الرغم من إمكانية استخدام متحول له قيمة مسندة إليه كبارامتر خرج إلا أن القيمة المخزنة في هذا البارامتر ستفقد عند تنفيذ التابع وستتوقع تغيرا لقيمته.

وكمثال على ذلك لنفترض تنفيذ التابع MaxValue () الذي يعيد أكبر قيمة في مصفوفة والذي تناولناه مسبقا سنقوم بتعديل هذا التابع قليلا بحيث يعطينا دليل العنصر ذو القيمة الأعلى ضمن المصفوفةولتبسيط الأمور فإننا سنقوم بإيجاد دليل أول عنصر له القيمة الأعلى وذلك في حال وجود قيمتين تمثلان القيمة العليا في المصفوفة.

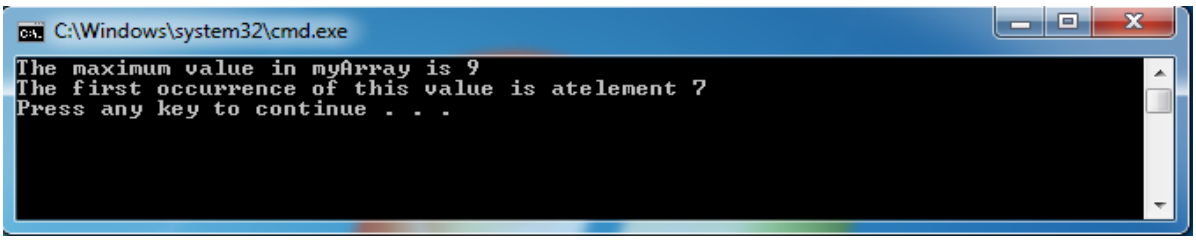
للقيام بذلك سنضيف بارامتر خرج بتعديل التابع كما يلي:

```
static int MaxValue(int[] intArray, out int maxIndex)
{
    int maxVal = intArray[0];
    maxIndex = 0;
    for (int i = 1; i < intArray.Length; i++)
    {
        if (intArray[i] > maxVal)
        {
            maxVal = intArray[i];
            maxIndex = i;
        }
    }
    return maxVal;
}
```

وسنستخدم هذا التابع بالصورة التالية:

```
static void Main(string[] args)
{
    int[] myArray = { 1, 8, 3, 6, 2, 5, 9, 3, 0, 2 };
    int maxIndex;
    Console.WriteLine("The maximum value in myArray is {0}",
        MaxValue(myArray, out maxIndex));
    Console.WriteLine("The first occurrence of this value is at"
        + "element {0}", maxIndex + 1);
    Console.ReadKey();
}
```

والخرج سيكون كما في الشكل (7-6).



الشكل (7-6).

ملاحظة:

لاحظ أنني أضفت 1 إلى القيمة maxIndex التي سيتم عرضها على الشاشة وذلك كي نحصل على الموقع المنطقي للعنصر ذو القيمة الأكبر فإذا كان العنصر الأكبر هو العنصر الأول فإن ما سيطبع هو العنصر رقم 1 وليس العنصر رقم 0.

مدى المتحول:

Variable Scope:

ربما تساءلت خلال القسم السابق من هذا الفصل عن ضرورة تبادل البيانات مع التوابع السبب في ذلك يعود إلى أن المتحولات في C# لا يمكن الوصول إليها إلا ضمن مناطق محددة ومحلية في الشيفرة وبالتالي فإن ما نقصده بمدى المتحول (Scope) هو المنطقة التي يمكننا أن نقرأ ونسند القيم من وإلى هذا المتحول ضمن شيفرة التطبيق.

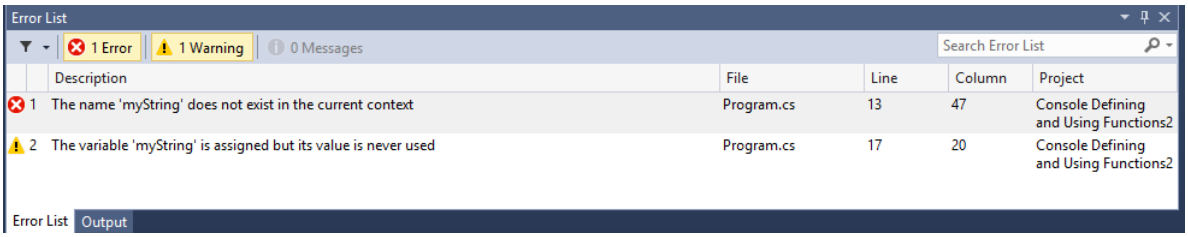
إن موضوع مدى المتحولات مهم جدا ومن الأفضل أن نتناول هذا الموضوع مع بعض الأمثلة.

تطبيق حول تعريف واستخدام تابع بسيط:

1- قم بإنشاء تطبيق Console جديد باسم Console Defining and Using Functions2.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C# في class Program:

```
class Program
{
    static void Write()
    {
        Console.WriteLine("myString={0}", myString);
    }
    static void Main(string[] args)
    {
        string myString = "String defined in Main()";
        Write();
    }
}
```

3- ترجم الشيفرة ولاحظ رسالة الخطأ والتحذير التي ستظهر في لائحة المهام الشكل (8-6).



الشكل (8-6)

كيفية العمل:

How it Works:

أين الخطأ في الشيفرة السابقة؟ حسنا إن المتحول myString معرف ضمن التابع Main () ولا يمكن الوصول إلى هذا المتحول ضمن التابع Write ().

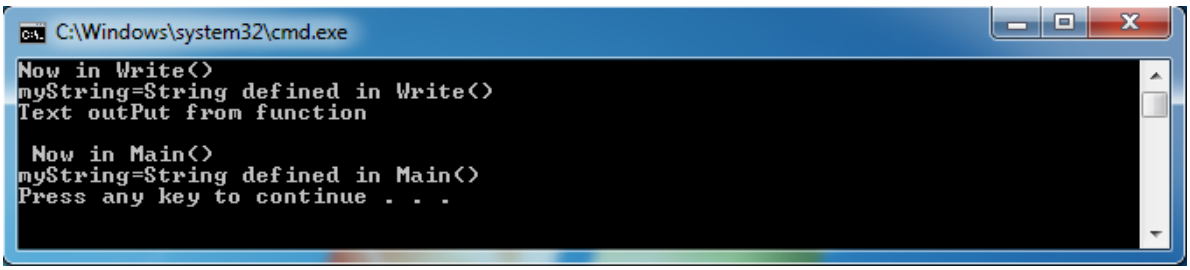
يعود السبب في ذلك إلى أن الوصولية إلى هذه المتحولات لها مدى يحدد المنطقة التي يمكن أن نصل ضمن حدودها إلى تلك المتحولات إن هذا المدى متعلق بكتلة الشيفرة التي تم التصريح عن المتحول ضمنها ولأي كتل معششة ضمن هذه الكتلة وبالنسبة للتوابع فإن كتل شيفرتها منفصلة عن بعضها البعض.

وبالتالي فإن المتحول myString بالنسبة للتابع Write () هو متحول غير مصرح عنه فهذا المتحول مصرح عنه ضمن التابع Main () وبالتالي لا يمكن الوصول إلى هذا المتحول إلا ضمن جسم التابع Main () فقط.

في الحقيقة يمكن أن يكون لدينا متحول باسم myString مختلفة تماما عن المتحول myString المصرح عنه في التابع Main () حاول تعديل الشيفرة السابقة إلى الشكل التالي:

```
class Program
{
    static void Write()
    {
        string myString = "String defined in Write()";
        Console.WriteLine("Now in Write()");
        Console.WriteLine("myString={0}", myString);
        Console.WriteLine("Text outPut from function");
    }
    static void Main(string[] args)
    {
        string myString = "String defined in Main()";
        Write();
        Console.WriteLine("\n Now in Main()");
        Console.WriteLine("myString={0}", myString);
    }
}
```

ستترجم هذه الشيفرة بصورة سليمة وستعطي خرجا كما في الشكل (9-6):



الشكل (9-6)

إن العمليات التي تمت في هذه الشيفرة هي كالآتي:

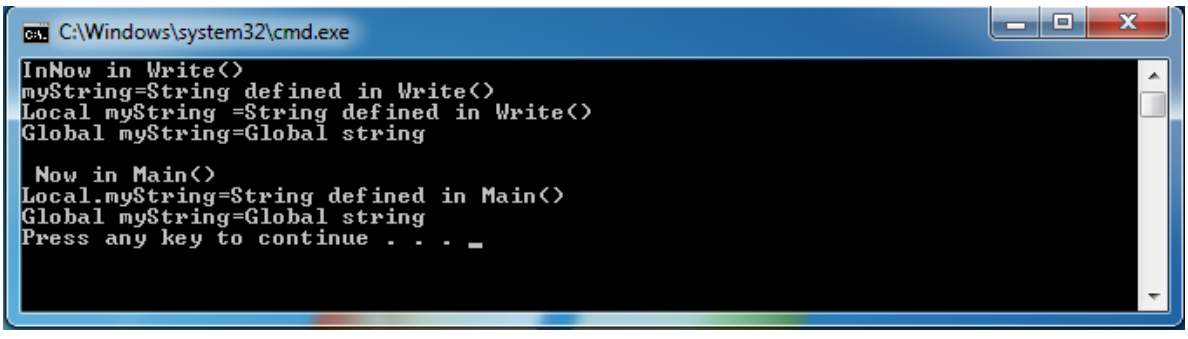
- تعريف وتهيئة متحول من نوع string باسم myString ضمن التابع Main ().
- نقل التحكم من التابع Main () إلى التابع Write () وذلك باستدعاء هذا التابع.
- تعريف وتهيئة متحول من نوع string باسم myString ضمن التابع Write () ومتحول مختلف تماما عن المتحول المصرح عنه في التابع Main ().

- طباعة سلسلة نصية على نافذة Console تتضمن قيمة المتحول myString المعرف في التابع Write ().
- نقل التحكم من التابع Write () إلى التابع Main () بسبب الوصول إلى نهاية التابع Write ().
- طباعة سلسلة نصية على نافذة Console تتضمن قيمة المتحول myString المعرف في التابع Main ().

تسمى المتحولات التي يقتصر مداها على التوابع المعرفة ضمنها بالمتحولات المحلية (Local variables) ومن الممكن أن يكون لدينا متحولات عامة (global variables) بحيث يغطي مداها توابع عدة عدل الشيفرة كما يلي:

```
class Program
{
    static string myString;
    static void Write()
    {
        string myString = "String defined in Write()";
        Console.WriteLine("InNow in Write()");
        Console.WriteLine("myString={0}", myString);
        Console.WriteLine("Local myString ={0}",myString );
        Console.WriteLine("Global myString={0}", Program.myString);
    }
    static void Main(string[] args)
    {
        string myString = "String defined in Main()";
        Program.myString = "Global string";
        Write();
        Console.WriteLine("\n Now in Main()");
        Console.WriteLine("Local.myString={0}", myString);
        Console.WriteLine("Global myString={0}", Program.myString );
    }
}
```

والخرج سيكون كما في الشكل (10-6).



الشكل (10-6)

لقد أضفنا هنا متحولا جديدا باسم myString أيضا لكن هذه المرة صرحنا عن هذا المتحول في أعلى هرمية التطبيق وذلك خارج أية توابع:

```
static string myString;
```

لاحظ أننا استخدمنا الكلمة static هنا أيضا للتصريح عن المتحولات كما استخدمناها للتصريح عن التوابع لقد قلت إنني لن أتحدث عن هذه الكلمة الآن ولكن يكفي أن أقول إننا يجب أن نضع إما static او const بالنسبة للمتحولات العامة في تطبيقات Console كهذه فإذا أردنا أن نستخدم متحولا ذو قيمة ثابتة (لا يمكن أن تتغير وإن حاولنا تغييرها فسيحدث خطأ) فإننا سنستخدم الكلمة const بدلا من static.

ولكي نفرق بين هذا المتحول والمتحولات المحلية ذات الاسم myString نفسه المعرفة ضمن التابعين Write() و Main() فإننا سنكتب اسم المتحول الكامل (لقد تحدثنا عن ذلك في الفصل الثالث عند حديثنا عن فضاءات الأسماء) لقد قمنا هنا بالإشارة إلى متحول عام باسم Program.myString لاحظ أن هذه التسمية ضرورية فقط في الحالة التي نواجه فيها تضاربا بين أسماء المتحولات العامة والمتحولات المحلية – كما في حالتنا هنا – فإن لم تكن هناك متحولات محلية باسم myString في الكتلة البرمجية فيمكننا أن نشير إلى المتحول العام ضمن هذه الكتلة بمجرد ذكر الاسم myString دون ذكر الصنف الذي يتبع له هذا المتحول (أي دون كتابة Program.myString) فعندما يكون لدينا متحول محلي له اسم مطابق لاسم متحول عام عندئذ سيحجب المتحول العام ضمن مدى المتحول المحلي هذا.

لقد حددنا قيمة للمتحول العام في التابع Main ():

```
Program.myString = "Global string";
```

ولقد تمكنا من الوصول إلى هذا المتحول العام ضمن التابع Write ():

```
Console.WriteLine("Global myString={0}", Program.myString );
```

قد تتساءل الآن عن الداعي لاستخدام المتحولات العامة لتبادل البيانات مع التوابع وذلك بدلا من تمرير البارامترات على هذه التوابع كما رئينا ذلك مسبقا هناك حالات نحتاج إليها بالفعل إلى هذه الطريقة لتبادل البيانات مع التوابع إلا أن هناك الكثير من الحالات إن لم تكن معظمها التي لا تتطلب استخدام هذه التقنية لتبادل البيانات بين التوابع المشكلة مع المتحولات العامة هي أنها غير مناسبة للتوابع ذات الأغراض العامة والتي يمكنها التعامل مع أية بيانات تزودها بها وليست محدودة بالبيانات في المتحولات العامة سوف نتحدث عن هذه النقطة بتفصيل أكبر لاحقا في هذا الفصل.

مدى المتحولات في بنى أخرى:

Variable Scope in Other Structures:

لقد ذكرنا مسبقا أن مدى المتحولات يقتصر على كتلة الشيفرة التي تم التصريح عن المتحول ضمنها وما تتضمن هذه الكتلة من كتل أخرى معششة ضمنها إن ذلك مطبق أيضا على كتل الشيفرة الأخرى مثل بنى التفرع والحلقات لنأخذ الشيفرة التالية مثلا:

```
static void Main(string[] args)
{
    int i;
    for (i = 0; i < 10; i++)
    {
        String text = "Line:" + Convert.ToString(i);
```

```
            Console.WriteLine(text);
        }
        Console.WriteLine("Last text output in loop:{0}", text);
    }
```

إن المتحول text هنا يمثل متحولا محليا بالنسبة لحلقة for إن هذه الشيفرة لن تترجم وذلك لأننا نقوم في استدعائنا للأمر Console.WriteLine() الموجود خارج الحلقة بمحاولة قراءة قيمة المتحول text وهو متحول واقع خارج المدى وذلك لأن مدى المتحول text هنا مقتصر على الحلقة فقط.

لنعدل الشيفرة السابقة كما يلي:

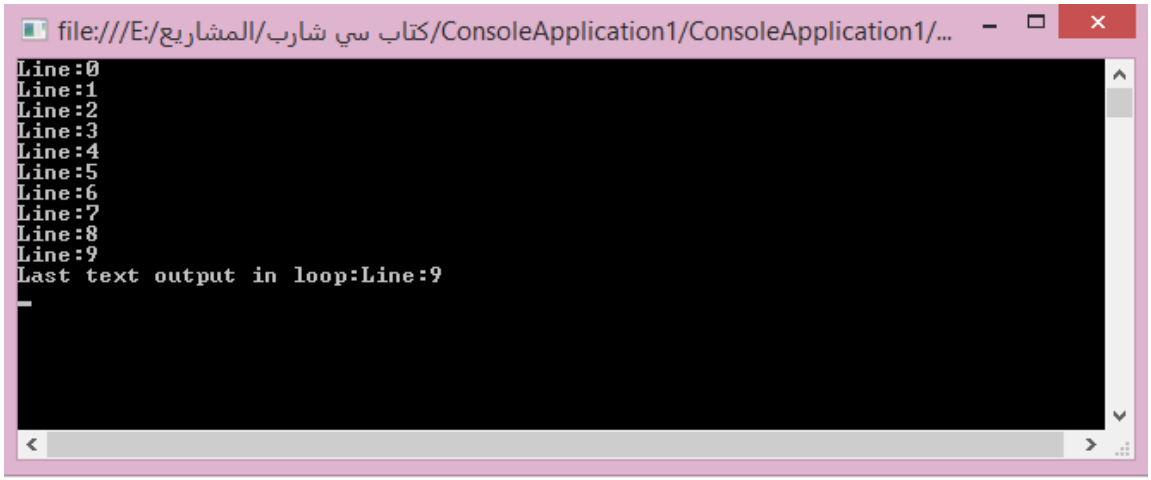
```
static void Main(string[] args)
{
    int i;
    String text;
    for (i = 0; i < 10; i++)
    {
        text = "Line:" + Convert.ToString(i);
        Console.WriteLine(text);
    }
    Console.WriteLine("Last text output in loop:{0}", text);
}
```

سوف تخفق هذه الشيفرة أيضا والسبب هو أن علينا أن نصرح عن المتحولات وأن نهيئها قبل استخدامها والمتحول text تمت تهيئته ضمن حلقة for فقط فإن القيمة المسندة إلى المتحول text ستفقد عند الخروج من الحلقة.

لنعدل الشيفرة الآن بالصورة التالية:

```
static void Main(string[] args)
{
    int i;
    String text=" ";
    for (i = 0; i < 10; i++)
    {
        text = "Line:" + Convert.ToString(i);
        Console.WriteLine(text);
    }
    Console.WriteLine("Last text output in loop:{0}", text);
}
```

ستنفذ هذه الشيفرة دون أخطاء هذه المرة وناتج تنفيذ هذه الشيفرة كما في الشكل (11-6):



الشكل (6-11)

لاحظ إمكانية الوصول من خارج الحلقة إلى القيمة الأخيرة التي أسندت إلى المتحول text ضمن الحلقة إن نتيجة عدم تنفيذ الشيفرة التي تسبق هذه الأخيرة يعود إلى حجز الذاكرة للمتحول text أو لأي متحول كان في الحقيقة إن مجرد التصريح عن متحول من نوع بسيط لا يعني حدوث الكثير وإنما عندما يتم إسناد القيم إلى المتحول فإن هذه القيم ستخزن في مكان ما من الذاكرة وعندما يستخدم هذا الجزء من الذاكرة ضمن الحلقة ستعرف هذه القيمة أنها محلية بالنسبة للحلقة وستخرج هذه القيمة عن المدى عند الخروج من جسم الحلقة وحتى إن كان المتحول غير محلي فإن قيمته محلية بينما يضمن لنا إسناد قيمة خارج الحلقة من أن هذه القيمة محلية بالنسبة للشيفرة الرئيسية وهذا يعني أن كتلة الحلقة ستستخدم نفس موقع الذاكرة الذي استخدم لتهيئة المتحول بالقيمة الأولية لتخزين قيم المتحول text ولذا فإننا سنتمكن من الوصول إلى هذه القيمة خارج الحلقة.

ولحسن الحظ فإن مترجم C# سيكتشف مشاكل مدى المتحولات وسيتجاوب مع رسائل الخطأ التي يولدها نتيجة لذلك وهو ما سيساعدك في الحقيقة لفهم موضوع مدى المتحولات.

وكملاحظ أخيرة فإننا ننصح بالتصريح عن جميع المتحولات وتهيئتها أيضا قبل أن نستخدمها في أي كتلة برمجية والاستثناء الوحيد لذلك هو عندما نصرح عن متحولات الحلقات كجزء من كتلة الحلقة كما في المثال:

```
for (int i = 0; i < 10; i++)
```

إن المتحول i هنا هو متحول محلي لكتلة الحلقة وهذا أمر طبيعي باعتبار أنه من النادر أن نستخدم متحول الحلقة في كتلة برمجية خارج الحلقة.

ملاحظة:

في الحقيقة إن أكثر الأخطاء البرمجية الشائعة التي يقع فيها المبرمجون المبتدئون بلغة C# مشابهة للسيناريو السابق لذا يجب عليك استيعاب مفهوم مدى المتحولات جيدا.

البارامترات والقيم المعادة مقابل البيانات العامة:

Parameters and Return Values vs Global Data:

سوف نلقي في هذا القسم نظرة عميقة على تبادل البيانات ضمن التوابع عبر البيانات العامة وعبر البارامترات والقيم المعادة وللإيجاز سنناقش الفرق بين شيفرة كالآتية:

```
class Program
{
    static void showDouble(ref int val)
    {
        val *= 2;
        Console.WriteLine("val doubled={0}", val);
    }
    static void Main(string[] args)
    {
        int val = 5;
        Console.WriteLine("val={0}", val);
        showDouble(ref val);
        Console .WriteLine ("val={0}",val);
    }
}
```

والشيفرة التالية:

```
class Program
{
    static int val;
    static void showDouble()
    {
        val *= 2;
        Console.WriteLine("val doubled={0}", val);
    }


    static void Main(string[] args)
    {
        val = 5;
        Console.WriteLine("val={0}", val);
        showDouble();
        Console .WriteLine ("val={0}",val);
        Console .ReadKey ();
    }
}
```

إن نتيجة تنفيذ هاتين الشيفرتين متطابقة تماما.

والآن ليست هناك أبه قواعد أو أمور إجبارية تضطرك لاستخدام طريقة بدلا من الأخرى وكلا التقنيتين المستخدمتين صحيحتان تمام إلا أن هناك بعض النقاط الرئيسية التي يجب التنويه عنها هنا.

وكبداية فإننا نذكر بأننا قلنا ان إصدارة التابع showDouble () التي تستخدم قيمة عامة لن تستخدم إلا متحولا واحدا فقط وهو المتحول val دائما وبناء على ذلك فإن علينا أن نستخدم هذا المتحول العام في كل

مرة نود تنفيذ التابع showDouble () إن هذا يؤثر على تعددية الاستخدام للتابع showDouble() قليلا ويعني أن علينا أن ننسخ قيمة المتحول العام val إلى متحولات أخرى إذا أردنا أن نحتفظ بالنتائج وبالإضافة إلى ذلك يمكن للبيانات العامة الموجودة في المتحول val أن تعدل في مكان آخر من التطبيق مما يمكن أن يؤدي إلى نتائج غير متوقعة.

على كل حال إن هذا النوع من فقدان تعددية الاستخدام للتوابع يمكن أن يكون في صالحنا أحيانا فهناك حالات لا نحتاج فيها إلا لاستخدام التابع لغرض واحد فقط واستخدام متحولات البيانات العامة في هذه الحالة سيقلص من احتمال حدوث خطأ ما في استدعاء التابع وربما ينتج من تمرير متحول خاطئ إلى التابع.

وبالطبع فإن هذا النوع من التبسيط يمكن أن يقودنا إلى كتابة شيفرة أكثر تعقيدا وأصعب فهما إن تحديد البارامترات بصورة صريحة يسمح لنا برؤية ما يتغير بلمحة سريعة فعندما نرى استدعاء لتابع بالشكل myFunction (val1 , out val2) فإن هذا يعني أن المتحولين val1 و val2 هما المتحولان الأكثر أهمية في هذا التابع وأن المتحول val2 سيأخذ قيمة جديدة عند الانتهاء من تنفيذ التابع وبالعكس فإن لم يستخدم هذا التابع أية بارامترات فإن ذلك سيضطرنا إلى تخمين أي شيء حول البيانات التي ستعالج ضمن التابع.

وأخيرا يجب ان ننوه إلى أن استخدام البيانات العامة ليس ممكنا دوما سوف نرى لاحقا في هذا الكتاب شيفرة مكتوبة ضمن ملفات مختلفة وتنتمي إلى فضاءات أسماء مختلفة تتصل فيما بينها عبر التوابع وفي حالات كهذه فإن الشيفرة ستجزأ إلى درجة تجعل من المستحيل حفظ البيانات في أماكن عامة من الشيفرة.

لذا وخلاصة لذلك إنك حر في استخدام التقنية التي تريحك إلا أنه من الواضح في مواضع كثيرة أن استخدام البارامترات والقيم المعادة أفضل بكثير من استخدام البيانات العامة إلا أن هناك مواضع خاصة ستحتاج فيها إلى ستحتاج فيها إلى استخدام البيانات العامة وهو أمر ليس بالخطأ إذا أردنا استخدام هذه التقنية.

التابع Main ():

The Main () Function:

لقد غطينا حتى الآن معظم التقنيات البسيطة المستخدمة في إنشاء واستخدام المتحولات سنسلط الضوء في هذا القسم على التابع Main ().

لقد قمنا سابقا في هذا الفصل أن التابع Main () يمثل نقطة دخول لتطبيق C# وأن تنفيذ التطبيق يبدأ بتنفيذ هذا التابع وينتهي بانتهائه لقد رأينا أيضا أن لهذا التابع بارامتر وحيد وهو string [] args إلا أننا لم نتعرض إلى هذا البارامتر حتى الآن سوف نتعلم في هذا القسم فائدة هذا البارامتر وكيف يمكن استخدامه.

يمثل البارامتر args للتابع Main () طريقة للحصول على المعلومات من خارج التطبيق حيث يتم تحديدها أثناء التنفيذ تمثل هذه المعلومات بارامترات لسطر الأوامر وذلك عند تنفيذ التطبيق من خلال سطر الأوامر أو محث DOS.

لقد كان استخدام سطر الأوامر السبيل الوحيد لتنفيذ التطبيقات في بيئة DOS أما اليوم فإن بيئة Windows الرسومية تغنينا عن ذلك باعتبار أن بإمكاننا تنفيذ التطبيقات والقيام بأعقد الأمور بمجرد النقر بزر الفأرة ومع ذلك فلقد أتاح لنا نظام Windows فرصة استخدام محث الأوامر.

لتأخذ برنامج المفكرة (Notepad) على سبيل المثال يمكننا أن نشغل هذا البرنامج بمجرد كتابة notepad في نافذة محث الأوامر أو ضمن الإطار الذي يظهر عند اختيار "تشغيل" (Run) من قائمة "ابدأ" (Start) ويمكننا كتابة أمر كما يلي:

Notepad "myfile.txt"

وفي حالة كهذه سيتم تحميل الملف myfile.txt في برنامج المفكرة (Notepad) عند بدء تشغيل البرنامج مباشرة أو سيتم إنشاء هذا الملف إن لم يكن موجودا يمثل النص "myfile.txt" هنا وسيط في سطر الأوامر يمكننا ان نكتب تطبيقات Console بلغة C# تعمل بأسلوب مشابه لهذا السلوك وذلك باستخدام البارامتر args.

ملاحظة:

يجب أن ننوه إلى أن هناك توقيعان يمكن للتابع () Main أن يأخذها وهي:

```
*       static void Main()
*       static void Main(string[] args)
```

أي أنه يمكننا إذا أردنا حذف البارامترات args الذي سنناقشه هنا إن سبب استخدامنا لهذا البارامتر يعود إلى وجوده بصورة تلقائية عند إنشاء تطبيقات Console في Visual Studio 2013.

عندما ينفذ تطبيق Console ما سيتم وضع أية بارامترات في سطر الأوامر ضمن المصفوفة args يمكننا ان نستخدم هذه المصفوفة بعد ذلك ضمن تطبيقنا حسب الحاجة لاستخراج قيم هذه البارامترات.

تطبيق حول بارامترات سطر الاوامر:

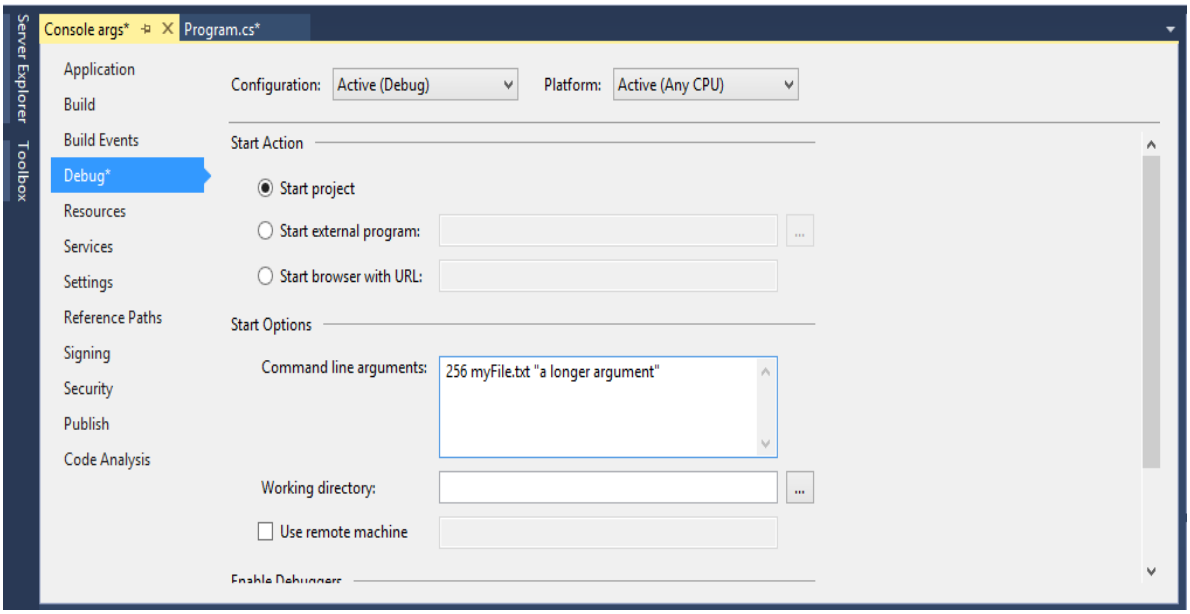
1- قم بإنشاء تطبيق Console جديد باسم Console args.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C# في class Program:

```
class Program
{
    static void Main(string[] args)
    {
        Console .WriteLine ("{0} command Line arguments were specified:"
            ,args .Length );
        foreach (String arg in args)
```

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

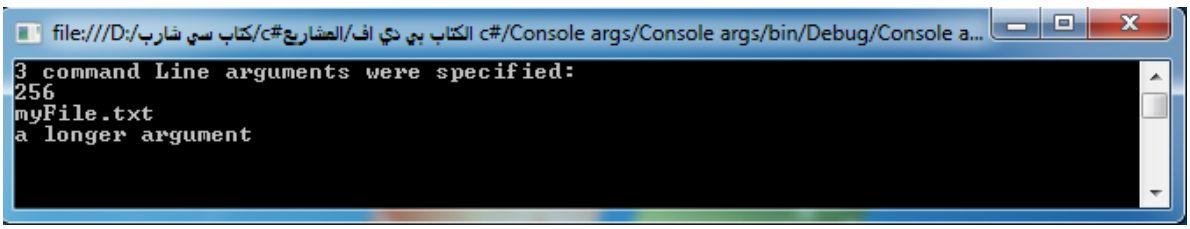
3- افتح خصائص المشروع (وذلك بالنقر بزر الماوس الأيمن على اسم المشروع Console args في إطار مستكشف الحلول solution Explorer ومن ثم اختيار الأمر Properties من القائمة المنسدلة).

4- اختر الصفحة *Debug ثم أضف أية وسطاء لسطر الأوامر وذلك ضمن الخاصية Command Line Arguments الشكل (12-6).



الشكل (12-6)

5- نفذ التطبيق بالضغط على مفتاح F5 فيظهر الشكل (13-6).



الشكل (13-6)

كيفية العمل:

How it Works:

الشيفرة المستخدمة هنا بسيطة للغاية:

```
Console .WriteLine ("{0} command Line arguments were specified:"
    ,args .Length );
foreach (String arg in args)
    Console.WriteLine(arg);
Console.ReadKey();
```

وهي لا تقوم إلا باستخدام البارامتر args كما لو كان كأية مصفوفة بارامترية أخرى في الحقيقة لا تقوم الشيفرة بأي شيء سوى طباعة بارامترات سطر الأوامر على نافذة الخرج.

لقد مررنا بارامترات سطر الأوامر في هذا المثال من ضمن نافذة خصائص المشروع في Visual Studio 2013 إن تلك تمثل طريقة مفيدة لاستخدام بارامترات سطر الأوامر نفسها متى أردنا تنفيذ التطبيق من ضمن Visual Studio 2013 وذلك بدلا من كتابة هذه الوسطاء ضمن محث سطر الأوامر في كل مرة.

توابع البنية Struct:

Struct Function:

لقد تحدثنا عن البنية Struct في الفصل السابق ووجدنا أنها تمثل أسلوبا ممتازا لحفظ عناصر البيانات المرتبطة ببعضها في مكان واحد في الحقيقة إن للبنية struct قدرات أكبر من ذلك وأحد هذه القدرات هي إمكانية احتوائها على توابع كتضمينها للبيانات إن ذلك يظهر غريبا نوعا ما من الوهلة الأولى إلا أنه أمر مفيد جدا.

وكمثال بسيط لنأخذ البنية التالية:

```
struct customerName
{
    public string firstName, lastName;
}
```

إذا كان لدينا متحولات من النوع customerName فإننا قد نود طباعة الاسم الكامل المتضمن على firstName وlastName عندئذ نحن مجبرين على تجميع هذين الاسمين مع بعضهما البعض وفي حالة كهذه فإننا سنستخدم شيفرة كما يلي:

```
customerName myCustomer;
myCustomer.firstName = "Hussam";
myCustomer.lastName = "ALdeen ALroz";
Console.WriteLine("{0} {1}", myCustomer.firstName, myCustomer.lastName);
```

عند إضافة التوابع إلى البنية struct فإننا سنبسط أمرا كهذا عبر تمركز المهام الشائعة المتعلقة ببيانات هذه البنية كتلك ضمن توابع في البنية نفسها يمكننا أن نضع تابعا مناسبا للبنية customerName كما يلي:

```
struct customerName
{
    public string firstName, lastName;
    public string Name()
    {
        return firstName + " " + lastName;
```

```
    }
}
```

إن هذا يظهر مشابها للتوابع التي تناولناها في هذا الفصل فيما عدا أننا لم نستخدم هنا الكلمة static سيتضح سبب ذلك لاحقا خلال هذا الكتاب أما الآن يكفي أن تعلم أن هذه الكلمة غير مطلوبة لتوابع struct يمكننا أن نستخدم هذا التابع كما يلي:

```
customerName myCustomer;
myCustomer.firstName = "Hussam";
myCustomer.lastName = "ALdeen ALroz";
Console.WriteLine(myCustomer .Name ());
Console.ReadKey();
```

إن هذه الصيغة أبسط بكثير من السابق وأسهل فهما أيضا.

هناك نقطة هامة يجدر الإشارة إليها هنا وهي أن التابع Name () يمكن أن يصل مباشرة إلى الأعضاء firstName وlastName في البنية Struct أي أن تلك الأعضاء عامة ضمن البنية customerName.

التحميل الزائد للتوابع:

Overloading Function:

لقد رأينا في هذا الفصل ضرورة مطابقة استدعاء التابع مع توقيعه وذلك كي نتمكن من تنفيذ التابع دون أخطاء ويتضمن ذلك حاجتنا لتوابع منفصلة للقيام بعمليات على أنواع مختلفة من المتحولات لقد حلت لنا تقنية التحميل الزائد للعوامل هذه المشكلة وذلك بتوفير إمكانية إنشاء توابع متعددة لها الاسم نفسه إلا أن لكل منها توقيع مختلف.

على سبيل المثال لقد استخدمنا الشيفرة التالية مسبقا والتي تتضمن التابع MaxValue ():

```
class Program
{
    static int MaxValue(int[] intArray)
    {
        int maxVal = intArray[0];
        for (int i = 1; i < intArray.Length; i++)
        {
            if (intArray[i] > maxVal)
                maxVal = intArray[i];
        }
        return maxVal;
    }
    static void Main(string[] args)
    {
        int[] myArray = { 10, 84, 35, 99, 21, 8, 0, 48, 11, 1 };
        int mxVal = MaxValue(myArray);
        Console.WriteLine("The maximum value in myArray is {0}", mxVal);
        Console.ReadKey();
    }
```

```
}
```

يمكن أن يستخدم التابع MaxValue () هنا مع المصفوفات ذات النوع int فقط يمكننا أن نعطي هذا النوع من التوابع أسماء مختلفة بحسب أنواع البارامترات التي يمكن أن تتقبلها فلربما نسمي التابع MaxValue () بالاسم intArrayMaxValue () ومن ثم نضيف توابع أخرى مثل DoubleArrayMaxValue () وذلك كي نطبق التابع MaxValue () نفسه على مصفوفة من نوع double لكن يمكننا كبديل لهذا التعقيد أن نضيف الشيفرة التالية إلى التطبيق السابق:

```
static double  MaxValue(double [] intArray)
{
    double  maxVal = intArray[0];
    for (int i = 1; i < intArray.Length; i++)
    {
        if (intArray[i] > maxVal)
            maxVal = intArray[i];
    }
    return maxVal;
}
```

الاختلاف هنا هو أن التابع MaxValue () أصبح يتقبل مصفوفة تتضمن على قيم من نوع double وهو نفس اسم التابع الذي يتقبل مصفوفة تتضمن على قيم من نوع int إلا أن لهذين التابعين توقيعان مختلفان إن هذا أمر مقبول في C# باعتبار أن لكلا التابعين توقيع مختلف ولكن لا يمكننا أن نكتب تابعين لهما نفس الاسم والتوقيع في آن واحد.

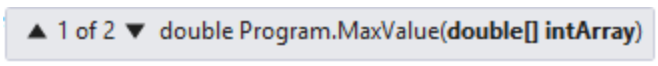
أصبح لدينا الآن إصدارتين من التابع MaxValue () أحدهما يتقبل مصفوفة int ويعيد قيمة من نوع int والآخر يتقبل مصفوفة double ويعيد قيمة من نوع double وهي القيمة الأكبر في المصفوفة وذلك بالنسبة لكلا التابعين.

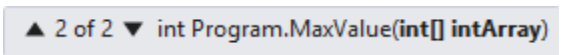
إن جمالية هذا النوع من الشيفرة هو أننا لسنا بحاجة للإشارة صراحة إلى أي من هذين التابعين سنستخدم عند استدعاء التابع باسمه فنحن سنقوم بتمرير بارامتر المصفوفة وسيتم استدعاء التابع المناسب تلقائيا بالاعتماد على نوع البارامتر الممرر.

ويجدر بنا أن ننوه على ميزة أخرى من ملكات الذكاء الموجودة في Visual Studio 2013 فإذا كان لدينا هذين التابعين وواصلنا كتابة اسم التابع ضمن التابع Main () سيقوم Visual Studio 2013 بعرض التحميلات الزائدة لهذا التابع تلقائيا فإذا كتبنا الشيفرة التالية مثلا:

```
double result=MaxValue ()
```

سيعطي Visual Studio 2013 في هذه الحالة تلميحا ضمن إطار رمادي يستعرض فيه توقيع كلا التابعين.





في الحقيقة يمكننا أن نستخدم جميع الجوانب المختلفة لتواقيع التوابع عند التحميل الزائد لها على سبيل المثال يمكن ان يكون لدينا تابعين أحدهما يأخذ البارامترات بالقيمة والآخر يأخذ البارامترات بالمرجع:

```
static void showDouble(ref int val)
{
    ...
}

static void showDouble(int val)
{
    ...
}
```

وفي تلك الحالة فإن تحديد استخدام أي من هذين الإصدارين مقتصر على شكل الاستدعاء أي إن تضمن الاستدعاء الكلمة ref أولا فالاستدعاء التالي سينفذ التابع ذو بارامتر المرجع:

```
showDouble(ref val);
```

وأما الاستدعاء التالي فسينفذ التابع ذو بارامتر القيمة:

```
showDouble(val);
```

المفوّضات:

Delegates:

المفوّض (delegate) هو نوع يمكنك من حفظ مراجع للتوابع وعلى الرغم من أن هذا يظهر غريبا نوعا ما إلا أن آلية عمل المفوّضات بسيطة للغاية لن تتمكن من فهم الغرض الأساسي للمفوّضات إلا في فصول لاحقة من هذا الكتاب وذلك عندما نتحدث عن الأحداث (events) ومعالجتها إلا أنه يمكننا أن نعطي لمحة بسيطة عن المفوضات هنا وعندما نستخدمها لاحقا ستظهر مألوفة بالنسبة إليك مما يبسط عليك أمورا كثيرة.

يصرح عن المفوّضات بشكل مشابه للتصريح عن التوابع إلا أنه ليس للمفوّض جسم كما في التوابع. تستخدم الكلمة delegate للتصريح عن المفوّض ويمثل التصريح توقيعا لتابع حيث يتألف من قيمة معادة ولائحة بارامترات وبعد تعريف المفوّض يمكننا أن نصرح عن متحول من نوع هذا المفوّض يمكننا أن نهيئ هذا المتحول ليمثل مرجعا لأي تابع له نفس توقيع المفوّض ومتى قمنا بذلك يمكننا أن نستدعي هذا التابع باستخدام متحول delegate تماما كما لو أننا نستخدم التابع مباشرة.

بما أن لدينا متحول يشير إلى تابع فإن بإمكاننا القيام بأشياء كثيرة بناء على ذلك على سبيل المثال يمكننا أن نمرر متحول المفوّض delegate إلى تابع على شكل بارامتر حيث سيستخدم التابع هذا المفوّض لاستدعاء التابع الذي يشير إليه المتحول.

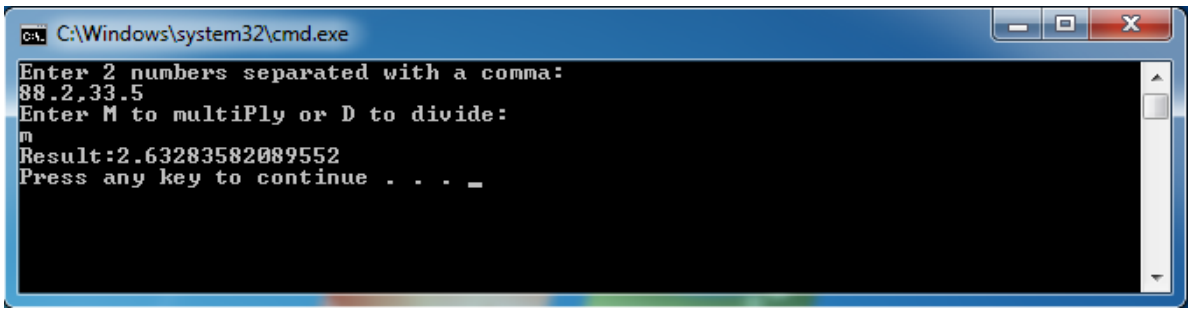
تطبيق حول استخدام المفوّض لاستدعاء تابع:

1- قم بإنشاء تطبيق Console جديد باسم Console delegate.

2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C# في class Program:

```
class Program
{
    delegate double processDelegate(double param1, double param2);
    static double multiply(double param1, double param2)
    {
        return param1 * param2;
    }
    static double divide(double param1, double param2)
    {
        return param1 / param2;
    }
    static void Main(string[] args)
    {
        processDelegate process;
        Console.WriteLine("Enter 2 numbers separated with a comma:");
        string input = Console.ReadLine();
        int commaPos =input .IndexOf (',');
        double param1=Convert .ToDouble (input .Substring (0,commaPos));
        double param2=Convert .ToDouble (input .Substring
            (commaPos +1,input .Length -commaPos -1));
        Console .WriteLine ("Enter M to multiPly or D to divide:");
        input=Console .ReadLine ();
        if (input =="M")
            process =new processDelegate (multiply );
        else
            process =new processDelegate (divide );
        Console .WriteLine ("Result:{0}",process (param1 ,param2 ));
    }
}
```

3- نفذ التطبيق بالضغط على مفتاح Ctrl+F5 فيظهر الشكل (14-6).



الشكل (14-6)

كيفية العمل:

How it Works:

تقوم هذه الشيفرة بتعريف مفوض (يحمل الاسم processDelegate) حيث يتطابق توقيعه مع توقيعي التابعين multiply () و divide () يأخذ هذا المفوض التصريح التالي:

```
delegate double processDelegate(double param1, double param2);
```

تحدد الكلمة delegate أن هذا التعريف لمفوض وليس لتابع (لاحظ أن شكل التعريف مشابه جدا لتعريف التابع) بعد ذلك هناك توقيع المفوض والذي يتضمن على قيمة معادة من نوع double وعلى بارامترين من نوع double ويمكنك أن تسمي المفوضات وبارامتراتها بأي اسم تريد لقد سمينا المفوض هنا بالاسم processDelegate والبارامترين بالاسم param1 وparam2.

تبدأ الشيفرة الموجودة في التابع Main() بالتصريح عن متحول للمفوض السابق:

```
static void Main(string[] args)
{
    processDelegate process;
```

بعد ذلك هناك شيفرة C# مألوفة من قبل حيث قمنا بطباعة نص على نافذة Console واستخدمنا طريقة جديدة للحصول على البيانات من سلسلة الدخل:

```
Console.WriteLine("Enter 2 numbers separated with a comma:");
string input = Console.ReadLine();
int commaPos =input .IndexOf (',');
double param1=Convert .ToDouble (input .Substring (0,commaPos));
double param2=Convert .ToDouble (input .Substring
    (commaPos +1,input .Length -commaPos -1));
```

ملاحظة:

لاحظ انني لم استخدم هنا أية شيفرة للتحقق من إدخال المستخدم وإذا كانت هذه شيفرة لتطبيق حقيقي فمن اللازم أن نتحقق من جميع احتمالات إدخال المستخدم التي لا تتناسب مع متطلبات التطبيق.

توضيح:

لقد استخدمنا هنا أمرين جديدين من أوامر معالجة السلاسل النصية لم نتحدث عنهما في الفصل السابق لقد طلبنا من المستخدم إدخال قيمتين رقميتين بحيث يفصل بينهما برز الفاصلة ","، لقد قلنا أن ما نحصل عليه من المستخدم لا يمثل إلا سلسلة نصية (قيمة من نوع String)حتى لو ظهر لنا غير ذلك كأن يقوم المستخدم بإدخال قيمة رقمية فقط نحتاج هنا لفصل القيمتين عن بعضهما و التعامل معهما كلا على حدى استخدمنا هنا الأمر IndexOf() والذي يتقبل قيمة من نوع Char كبارامتر له ويعيد قيمة رقمية من نوع int تمثل أول موقع للرمز Char في السلسلة النصية لقد احتفظنا بهذا الموقع ضمن المتحول commaPos حتى الآن لم نقم بفصل القيمتين عن بعضهما البعض والخطوة التالية هي استخدام الأمر Substring () والذي يعطينا سلسلة نصية ويتقبل بارامترين يمثل الأول موقع الرمز الذي يمثل بداية السلسلة الجزئية ويمثل الثاني طول هذه السلسلة الجزئية وبالتالي فإن القيمة الأولى التي أدخلها المستخدم تبدأ من أول رمز في السلسلة النصية ذو الدليل 0 وصولا إلى موقع الفاصلة أي أن طول السلسلة الجزئية هو نفسه دليل الفاصلة وأما القيمة الثانية فهي تبدأ من بعد الفاصلة أي عند الدليل commpos+1 ووصولا إلى نهاية السلسلة أي بطول input.Length-commpos-1.

بعد ذلك سنطلب من المستخدم أن يحدد ما إذا كان يود طباعة حاصل ضرب القيمتين أم حاصل قسمة الأول على الثاني:

```
Console .WriteLine ("Enter M to multiPly or D to divide:");
input=Console .ReadLine ();
```

وبالاعتماد على دخل المستخدم سيتم تهيئة متحول المفوض process:

```
if (input =="M")
    process =new processDelegate (multiply );
else
    process =new processDelegate (divide );
```

لقد استخدمنا هنا أسلوبا غريبا نوعا ما لإسناد مرجع لتابع إلى متحول المفوض process وكما في إسناد قيم المصفوفة فإن علينا استخدام الكلمة new لإنشاء مفوض جديد وبعد هذه الكلمة سنحدد نوع المفوض وتزويده ببارامترات يشير إلى التابع الذي نود استخدامه لاحظ أن هذا البارامتر لا يتطابق مع توقيع المفوض وإنما يمثل اسم التابع الذي سيمثله المتحول.

وأخيرا يمكننا أن نستدعي التابع باستخدام مفوضه حيث يمكننا أن نستخدم متحول المفوض تماما كما لو أننا استخدمنا التابع نفسه:

```
Console .WriteLine ("Result:{0}",process (param1 ,param2 ));
```

ليس هذا وحسب بل ويمكننا أن نقوم بعمليات إضافية على هذا المتحول كتمريره إلى تابع على شكل بارامتر كمثال على ذلك:

```
static void executeFunction (processDelegate process)
 {
     process(2.2, 3.3);
 }
```

هذا يعني أن بإمكاننا التحكم بسلوك التوابع بتمرير هذه التوابع على شكل مفوضات على سبيل المثال يمكن أن يكون لدينا تابع يقوم بترتيب مصفوفة من نوع string أبجديا هناك العديد من الطرق لترتيب اللوائح والتي يختلف أداؤها بحسب صفات اللائحة المراد ترتيبها وباستخدام المفوضات يمكننا تحديد الطريقة المستخدمة وذلك بتمرير مفوض تابع خوارزمية ترتيب محددة إلى تابع الترتيب.

هناك الكثير من الاستخدامات للمفوضات إلا أنها تستخدم بصورة خاصة لمعالجة الأحداث وهذا ما سنراه في المستقبل.

الاستدعاء التعاودي:

Recursion Calling:

هنا يمكننا أن نستدعي التابع ضمن جسم التابع نفسه؟! الجواب ليس هذا وحسب بل إن ذلك يعد أحد تقنيات البرمجة المتقدمة وفي مراحل متقدمة من البرمجة قد تستخدمها يمكن أن نسمي ذلك بالاستدعاء الذاتي للتوابع ما يطلق على هذا الأسلوب البرمجي بالتعاودية (recursion).

لنأخذ على سبيل المثال الشيفرة التالية:

```
static int factorial(int val)
{
    if (val <= 0)
        return 1;
    else
        return val * factorial(val - 1);
}
```

في الحقيقة يمثل هذا التابع خوارزمية احتساب العاملي فلنفرض مثلا أننا استدعينا هذا التابع كما يلي:

```
static void Main(string[] args)
{
    int x;
    x = factorial(5);
    Console.WriteLine("5!={0}", x);
    Console.ReadKey();
}
```

لنرى ما سيحدث ضمن التابع سيأخذ البارامتر val القيمة 5 وبما أن هذه القيمة لا تحقق الشرط val<=0 فسوف ينفذ السطر:

```
        return val * factorial(val - 1);
```

أي سيتم استدعاء التابع factorial() من ضمن التابع نفسه هذه المرة ببارامتر مختلف عن السابق factorial(5-1) وبالتالي سيأخذ البارامتر val في الاستدعاء الثاني القيمة 4 وبما أن 4 ليست أصغر أو تساوي 0 فسيعود التابع لاستدعاء نفسه مجددا لكن هذه المرة وفقا للاستدعاء التالي:

factorial(4-1) وتستمر عجلة الاستدعاء بهذه الصورة إلى أن تصل قيمة val للصفر عندها سيتوقف التابع والمحصلة هي إعادة النتيجة:

```
=Val*(Val-1)*(Val-2)....*1
=5*4*3*2*1
=120
```

في الحقيقة أن هناك عدة أمور يجب مراعاتها عند استخدام الاستدعاء التعاودي وأهم هذه الأمور هي أن يكون هناك مخرج من التعاودية دائما ولهذا السبب وجد الشرط:

```
if (val <= 0)
```

في الشيفرة السابقة فإذا كتبنا التابع السابق بالشكل:

```
static int factorial(int val)
{
        return val * factorial(val - 1);
}
```

فإن ذلك سيؤدي إلى حدوث خطأ في التنفيذ نتيجة لاستمرارية الاستدعاء التعاودي دون توقف مما سيؤدي إلى إنهاك مكدس النظام وتوقف البرنامج عن العمل.

إن الاستدعاء التعاودي موضوع برمجي متقدم ولقد أحببت ان انوه عنه هنا في حديثنا عن التوابع لكن إذا أردت ان تقرأ في موضوع كهذا يمكنك البحث عن كتب تتحدث عن الخوارزميات البرمجية.

الخلاصة:

Summary:

لقد أخذنا في هذا الفصل لمحة كاملة حول كيفية استخدام التوابع في لغة C# ولقد تعرفنا على الكثير من المزايا الإضافية المتعلقة بالتوابع مثل المفوضات والاستدعاء التعاودي لكن للحديث بقية عن التوابع وإليك ملخصا عما تناولناه في هذا الفصل:

- ✓ تعريف واستخدام التوابع في تطبيقات Console.
- ✓ تبادل البيانات مع التوابع بواسطة القيم المعادة والبارامترات.
- ✓ المصفوفات البارامترية.
- ✓ تمرير البارامترات بالمرجع أو القيمة.
- ✓ استخدام بارامترات الخرج لإعادة قيم إضافية.
- ✓ مفهوم مدى المتحولات.
- ✓ التابع Main () وكيفية استخدام بارامترات سطر الأوامر.
- ✓ استخدام التوابع ضمن بنية Struct.
- ✓ التحميل الزائد للتوابع.
- ✓ المفوضات.
- ✓ الاستدعاء التعاودي.

الفصل السابع

معالجة الأخطاء

لقد تناولنا في الفصول السابقة من الكتاب معظم أساسيات البرمجة بلغة C# وقبل أن ننتقل إلى الحديث عن البرمجة كائنية التوجه في الجزء التالي من الكتاب فإن الوقت قد حان للحديث عن تنقيح ومعالجة الأخطاء في شيفرة C#.

الأخطاء في الشيفرة هو شيء يحتمل إلى حد كبير أن تقع فيه أثناء البرمجة ومهما كنت مبرمجا جيدا فإنك على الأغلب سترتكب أخطاء برمجية وبالطبع فإن هناك أخطاء لا تؤثر على سير البرنامج كخطأ إملائي في كتابة اسم زر ما و هناك أخطاء تتسبب في توقف التطبيق عن العمل تماما تعرف هذه الأخطاء بالأخطاء الفادحة fatal errors تصنف الأخطاء الفادحة إلى أخطاء بسيطة ناتجة عن خطأ في كتابة صيغة ما syntax errors و هناك أخطاء منطقية logic errors وتحدث عندما يخفق تطبيقنا في القيام بمهمة ما كالقسمة على صفر مثلا.

في الحقيقة يمكن أن تصبح عملية معالجة الأخطاء كابوسا حقيقيا للمبرمج خصوصا إذا تذمر المستخدم من شيء يعمل بصورة غير منطقية في التطبيق في هذه الحالة عليك أن تبحث عن الخطأ خلال الشيفرة في محاولة لاكتشاف ما يحدث وكيف يجب أن تعدل شيفرتك لكي تتجاوز هذا الخطأ.

في حالات كهذه ستجد أن إمكانيات تنقيح الأخطاء في Visual Studio 2013 تقدم لك مساعدة أكثر من رائعة وفي الجزء الأول من هذا الفصل سنلقي نظرة على بعض التقنيات التي يقدمها لنا Visual Studio 2013 لحل بعض المشاكل الشائعة.

وبالإضافة إلى هذا فإننا سنتناول تقنيات معالجة الأخطاء المتوفرة في C# إن هذا سيمكننا من اتخاذ التدابير الوقائية في حالات يحتمل وقوع الأخطاء فيها وكتابة شيفرة برمجية مرنة بالصورة الكافية بحيث تمكننا من الالتفاف حول الأخطاء التي تمثل أن تكون أخطاء فادحة تتسبب في إيقاف التطبيق بصورة غير محببة إن التقنيات تمثل جزءا من لغة C# وليست بميزة في Visual Studio 2013 إلا أن Visual Studio 2013 يوفر بعض الأدوات التي تساعدنا في الاستفادة من هذه التقنية أيضا.

الأخطاء النحوية:

Syntax Errors:

إن هذا النوع من الأخطاء هو الاسهل وفي Visual Studio 2013 يتم اكتشاف هذه الأخطاء فورا مثال على هذا الخطأ كتابة الجملة التالية:

```
firstNumber = Convert.ToDuble(Console.ReadLine());
```

بالطبع ستجد رسالة خطأ قبل التنفيذ تخبرك بأن الخاصية ToDuble غير موجودة مثل هذه الأخطاء هي الاسهل ويتم اكتشافها من خلال بيئة لغة البرمجة التي تعمل عليها وفي Visual Studio 2013 أصبحت رسائل الخطأ واضحة للغاية ويمكن تفسيرها بسهولة وحلها.

الأخطاء المنطقية:

Logical Errors:

هذا النوع من الأخطاء هو الأصعب، فعلى صعيد كتابة الكود ربما لا يوجد خطأ نحوي ولكنه خطأ منطقي يظهر عند التنفيذ، ابسط مثال على هذا الخطأ هو كتابة كود كالتالي:

```
byte x = 2015446;
```

طبعاً تعرف ان حدود النوع byte أصغر من هذا الحد ولكن في الإصدارات القديمة لم يكن هذا ليظهر خطا حيث أن الجملة مكتوبة نحويا بشكل سليم كما ترى.

وتقسم الأخطاء المنطقية إلى ثلاثة أنواع أساسية وهي:

1- User Error:

أخطاء تنتج من استخدام البرنامج لو افترضنا المثال السابق لـ byte نقوم فيه بتخزين عمر المستخدم لكن المستخدم قام بإدخال رقم 10000 هذا الخطأ من المستخدم سيتسبب في المشاكل لك فيما لو لم تكن قد اضفت شرط التأكد من عدم تجاوز العمر لحد معين، ايضاً ادخال بيانات نصية في خانة العمر وخلافه تندرج تحت اسم أخطاء المستخدم.

2- Exceptions:

وهو النوع الأشهر من الأخطاء مثل محاولة فتح ملف او قاعدة بيانات غير موجودة مثلاً حيث لم يتم تحميلها بصورة صحيحة أو محاولة قراءة بيانات من قاعدة البيانات في حين انها تساوي null بدون استخدام nullable type أو محاولة الكتابة إلى ملف نصي Readonly وخلافه من الأخطاء المشهورة.

3- Bugs :

أكثر الأخطاء شهرة لا يمكن حصرها ولا عدها، وتوجد في جميع البرامج بما فيهم نسخة الويندوز التي تستخدمها في العادة لن يخلو برنامج منها ولكننا نحاول تفاديها قدر المستطاع، قد تحدث بسبب نسيان حذف متغير او قراءة متغير من قيمة موجودة اصلاً في الذاكرة ونحن نظن انها قيمة فارغة ... الخ، هذه الأخطاء قد لا تظهر لـ 99 % من المستخدمين ولكنا تظهر لمستخدم واحد فقط، لذا في العادة تكون هناك عدة نسخ تجريبية من اي برنامج لمحاولة معرفة أماكن امثال هذه الأخطاء وتعديلها قبل طرح النسخة الرسمية.

تنقيح الأخطاء في Visual Studio 2013:

Debugging in Visual Studio 2013:

عندما نقوم بتنفيذ برامجنا في نمط التنقيح debug mode فإن هناك أمور كثيرة تحدث خلف الكواليس وليس مجرد تنفيذ للشيفرة التي كتبناها وحسب. يقوم المنقح debugger ببناء معلومات رمزية symbolic information عن تطبيقك وبهذه المعلومات فإن VS سيصبح قادرا على معرفة ما يحدث عند تنفيذ كل سطر برمجي في الشيفرة إن ما نقصده بالمعلومات الرمزية هي معلومات متعلقة بأسماء المتحولات المستخدمة في الشيفرة قبل ترجمتها حيث يتم مطابقتها مع القيم الموجودة ضمن شيفرة التطبيق المترجمة أثناء التنفيذ فهذه الشيفرة المترجمة غير مفهومة من قبل الإنسان كالشيفرة التي نكتب بها تطبيقاتنا يتم الاحتفاظ بهذه المعلومات ضمن ملفات لها الامتداد pdb و هو ما ستجده في مجلد Debug ضمن مجلد المشروع إن ذلك يمكننا من القيام بالعديد من العمليات تتضمن:

- طباعة معلومات التنقيح إلى VS.
- قراءة قيم المتحولات وتحريرها أثناء تنفيذ التطبيق.
- الإيقاف المؤقت لتنفيذ التطبيق أو إعادة تنفيذه من جديد.
- إنهاء تنفيذ التطبيق تلقائيا عند الوصول إلى نقاط معينة من الشيفرة.
- تنفيذ شيفرة التطبيق سطرا بسطر.
- مراقبة المتغيرات التي تطرأ على قيم المتحولات خلال تنفيذ التطبيق.
- تعديل محتوى المتحولات أثناء تنفيذ التطبيق.
- إجراء استدعاءات تجريبية للتوابع.

سوف نلقي نظرة على هذه التقنيات في هذا الفصل وسنبين كيفية استخدامها وذلك كي نتمكن من تنقيح وتصحيح الأخطاء التي قد نرتكبها عند كتابة تطبيقاتنا.

سوف نقسم هذه التقنيات إلى قسمين وذلك وفقا لاستخداماتها وبصورة عامة تنقيح الشيفرة يتم إما بمقاطعة تنفيذ التطبيق أو إنشاء ملاحظات لتحليل المشاكل لاحقا وبالنسبة لـ VS فإن تنقيح الشيفرة يتم إما أثناء تنفيذ الشيفرة (نمط عدم المقاطعة) أو في نمط مقاطعة التطبيق (Break mode) أي عند إيقاف التنفيذ العادي للشيفرة مؤقتا سوف نناقش تقنيات التنقيح باستخدام نمط عدم التوقف أولا.

التنقيح في نمط عدم المقاطعة (العادي):

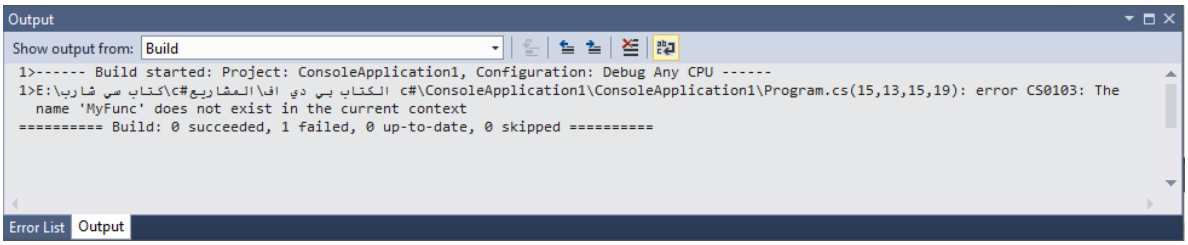
Debugging in Non-Break (Normal) Mode:

إن أحد أكثر الأوامر التي مرت معنا منذ بداية هذا الكتاب إلى الآن هو الأمر Console.WriteLine() والذي يقوم بطباعة نص ما على نافذة Console يمكننا أن نستفيد من هذا الأمر عندما نقوم بتطوير التطبيقات وذلك لمساعدتنا في تتبع ما يحدث أثناء تنفيذ الشيفرة على سبيل المثال:

```
Console.WriteLine("MyFunc() Function about to be called.");
MyFunc("Do something");
Console.WriteLine("MyFunc() Function execute completed.");
```

تستعرض هذه الشيفرة كيف يمكننا الحصول على معلومات إضافية متعلقة بالتابع MyFunc() إن القيام بذلك جيد جدا لكن سيظهر خرج Console غير منظم فليلا وكطريقة بديلة لذلك فإن بإمكاننا ان نخرج النصوص إلى مكان منفصل وهو إطار الخرج في VS المسمى بإطار Output.

لقد نوهنا على إطار Output في الفصول السابقة وهو موجود في الجزء السفلي من بيئة تطوير VS ويشاركه بنفس الموضع إطار Task List لقد تعرفنا على هذا الإطار أيضا ورأينا أنه يعرض معلومات متعلقة بالأخطاء والمهام الموجودة في الشيفرة ويتضمن ذلك الأخطاء التي تحصل أثناء كتابة الشيفرة يمكننا ان نستخدم هذه النافذة أيضا لعرض معلومات تفحصيه من عندنا وذلك بكتابتها مباشرة ضمن الإطار يمكننا أن نرى هذا الإطار كما في الشكل (1-7):



الشكل (1-7)

ملاحظة:

لاحظ أن للإطار Output ثلاثة أنماط يمكننا تحديد أحدهما من خلال تحديد اسم النمط في مربع القائمة المنسدلة يمكننا أن ننتقل بين أنماط بناء التطبيق Build وتنقيحه Debug وتنفيذه بصورة اختبارية Test Run يعطينا نمطي Build و Debug معلومات متعلقة بزمن تنفيذ التطبيق وبالأخطاء الحاصلة عندما سأشير إلى الكتابة إلى إطار Output في هذا القسم فإنني أعني الكتابة في نمط عرض Debug لإطار Output.

إن ما يقوم به VS من كتابة معلومات متعلقة بتطبيقنا الذي يجري تنفيذه حاليا في إطار Output لهو أمر مشابه لملف التسجيل logging file الذي تقوم بإنشائه لتسجيل ما يحدث أثناء تنفيذ تطبيقاتك إلا أننا هنا لسنا بحاجة للاهتمام بكيفية الوصول إلى الملف والكتابة ضمنه.

إخراج معلومات التنقيح:

Outputting Debugging Information:

إن كتابة النصوص في إطار Output بسيط للغاية فنحن ببساطة سنستبدل استدعاءات الأمر Console.WriteLine() بالاستدعاء المطلوب لكتابة النص في المكان الذي نريد ضمن إطار Output.

هناك أمران يمكننا استخدامهما لذلك:

الأمر Debug.WriteLine().

الأمر Trace.WriteLine().

إن هذين الأمرين متشابهان لدرجة كبيرة جدا إلا هناك اختلاف واحد بينهما وهو أن الأمر الأول يعمل فقط عند بناء التطبيق في مرحلة التنقيح debug build أما الأمر الثاني فهو يعمل عند بناء التطبيق في مرحلة الإطلاق release build في الحقيقة سيقوم المترجم بتخطي جميع أوامر Debug.WriteLine() عند بناء التطبيق في مرحلة الإطلاق فهذا الأمر سيختفي من شيفرة الإطلاق.

توضيح:

يمكن أن تكون لدينا إصدارتين من الشيفرة البرمجية ناتجة من ملف مصدري وحيد: إصدارة التنقيح debug version وإصدارة الإطلاق release version تستعرض إصدارة التنقيح جميع المعلومات التفحصية الإضافية التي نحتاج إليها في مرحلة تنقيح التطبيق أما إصدارة الإطلاق فهي لا تقوم بذلك ولن تعرض رسائل على المستخدمين يمكن أن تزعجهم وذلك باعتبار ان هذه الرسائل مفيدة جدا لنا أثناء تطوير تطبيق وليس للمستخدم النهائي.

لاحظ أن هذين الأمرين لا يعملان بنفس الصورة التي يعمل بها الأمر Console.WritLine() فالأمر Console.WritLine() يستخدم الصيغة {x} لربط النصوص مع قيم المتحولات بينما يجب أن نستخدم العامل + للقيام بهذا النوع من الربط عند استخدام الأمرين السابقين وذلك لأن هذين الأمرين لا يتقبلان ذلك أي أن هذين الأمرين يتقبلان بارامترا واحدا يمثل السلسلة النصية التي نود طباعتها في إطار Output ويمكن أن يتقبلا بارامترا ثانيا حيث يستخدم لعرض تصنيف النص المطبوع إن ذلك يمكننا من معرفة أية رسائل يتم طباعتها في إطار Output في حال كانت هناك رسائل متشابهة في أماكن مختلفة من التطبيق.

إن الخرج العام لهذين الأمرين هو كما يلي:

<category>:<message>

على سبيل المثال لنفترض اننا كتبنا في موضوع ما من شيفرة التابع MyFunc السطر التالي:

```
Debug.WriteLine("Added 1 to i", "MyFunc");
```

عندئذ فإن الخرج في إطار Output سيكون كما يلي:

MyFunc: Added 1 to i

في الحقيقة إننا لن نستخدم أمري التنقيح لطباعة نصوص وحسب وإنما سنستخدمها في الغالب لطباعة قيمة متحول ما.

تطبيق حول طباعة نص في إطار Output:

1- قم بإنشاء تطبيق Console جديد باسم Console Output.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C# في class Program:

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

namespace Console_Output
{
    class Program
    {
        static int Maxima(int[] integers, out int[] indices)
        {
            Debug.WriteLine("Maximum value search started.");
            indices = new int[1];
            int maxVal = integers[0];
            indices[0] = 0;
            int count = 1;
            Debug.WriteLine("Maximum value initialized to" + maxVal +
                ",at element index 0.");
            for (int i = 1; i < integers.Length; i++)
            {
                Debug.WriteLine("Now looking at element at index" + i + ".");
                if (integers[i] > maxVal)
                {
                    maxVal = integers[i];
                    count = 1;
                    indices = new int[1];
                    indices[0] = i;
                    Debug.WriteLine("New maximum found.New value is" +
                        maxVal + ",at element index" + i + ".");
                }
                else
                {
                    if (integers[i] == maxVal)
                    {
                        count++;
                        int[] oldIndices = indices;
                        indices = new int[count];
                        oldIndices.CopyTo(indices, 0);
                        indices[count - 1] = i;
```

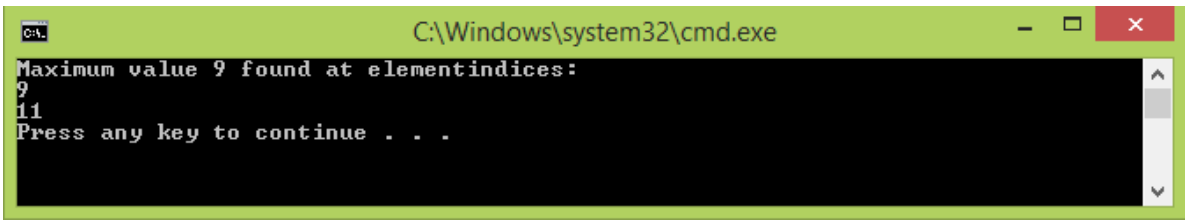
```
                    Debug.WriteLine("Duplicate maximum found at element" +
                        "index" + i + ".");
                }
            }
        }
        Trace.WriteLine("Maximum value" + maxVal + "found with" +
            count + "occurrences.");
        Debug.WriteLine("Maximum value search completed.");
        return maxVal;
    }
    static void Main(string[] args)
    {
        int[] testArray = { 4, 7, 4, 2, 7, 3, 7, 8, 3, 9, 1, 9 };
        int[] maxValIndices;
        int maxVal = Maxima(testArray, out maxValIndices);
        Console.WriteLine("Maximum value {0} found at element" +
            "indices:", maxVal);
        foreach (int index in maxValIndices)
        {
            Console.WriteLine(index);
        }
    }
  }
}
```

3- نفذ التطبيق في نمط التنقيح باختيار الأمرStart أو بالضغط على الزر F5

ملاحظة:

لاحظ أنه يمكنك تحديد نمط تنفيذ الشيفرة من خلال مربع القائمة المنسدلة Solution Configurations الموجود في شريط الأدوات.

ستظهر نافذة Console وستختفي بسرعة ولو أننا نفذنا الشيفرة بدون تنقيح Start without Debugging أو بالضغط على مفتاح Ctrl+ F5 فيظهر الشكل (2-7).



الشكل (2-7)

4- انظر الآن إلى محتويات إطار Output في نمط التنقيح Debug يمكنك اختيار نمط التنقيح من خلال مربع القائمة المنسدلة في أعلى النافذة حيث لا تتضمن هذه القائمة سوى ثلاثة خيارات Build وBuild Order وDebug سيظهر الخرج كما يلي:

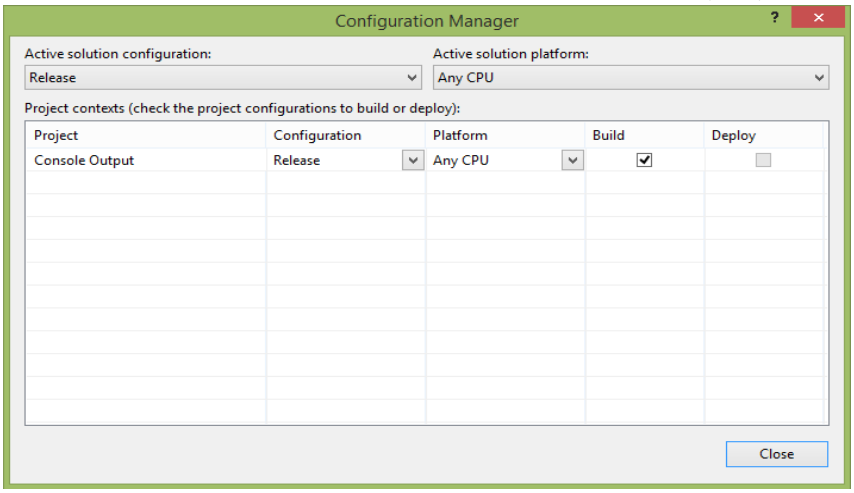
```
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\Microsoft.Net\assembly\GAC_32\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\mscorli
```

```
b.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just My
Code' is enabled.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\assembly\GAC_MSIL\Microsoft.VisualStudio.HostingProcess.Utilities\12.0.0.0__b
03f5f7f11d50a3a\Microsoft.VisualStudio.HostingProcess.Utilities.dll'. Skipped loading
symbols. Module is optimized and the debugger option 'Just My Code' is enabled.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Windows.Forms\v4.0_4.0.0.0__b77a5c5619
34e089\System.Windows.Forms.dll'. Skipped loading symbols. Module is optimized and the
debugger option 'Just My Code' is enabled.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Drawing\v4.0_4.0.0.0__b03f5f7f11d50a3a
\System.Drawing.dll'. Skipped loading symbols. Module is optimized and the debugger
option 'Just My Code' is enabled.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System\v4.0_4.0.0.0__b77a5c561934e089\System.
dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just My Code'
is enabled.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\assembly\GAC_MSIL\Microsoft.VisualStudio.HostingProcess.Utilities.Sync\12.0.0
.0__b03f5f7f11d50a3a\Microsoft.VisualStudio.HostingProcess.Utilities.Sync.dll'. Skipped
loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\assembly\GAC_MSIL\Microsoft.VisualStudio.Debugger.Runtime\12.0.0.0__b03f5f7f1
1d50a3a\Microsoft.VisualStudio.Debugger.Runtime.dll'. Skipped loading symbols. Module is
optimized and the debugger option 'Just My Code' is enabled.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded 'D:\كتاب
المشاريع\اف دي بي الكتاب#c\شارب سي c#\Console Output\Console Output\bin\Debug\Console
Output.vshost.exe'. Skipped loading symbols. Module is optimized and the debugger option
'Just My Code' is enabled.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Core\v4.0_4.0.0.0__b77a5c561934e089\Sy
stem.Core.dll'. Skipped loading symbols. Module is optimized and the debugger option
'Just My Code' is enabled.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Xml.Linq\v4.0_4.0.0.0__b77a5c561934e08
9\System.Xml.Linq.dll'. Skipped loading symbols. Module is optimized and the debugger
option 'Just My Code' is enabled.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Data.DataSetExtensions\v4.0_4.0.0.0__b
77a5c561934e089\System.Data.DataSetExtensions.dll'. Skipped loading symbols. Module is
optimized and the debugger option 'Just My Code' is enabled.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\Microsoft.CSharp\v4.0_4.0.0.0__b03f5f7f11d50a
3a\Microsoft.CSharp.dll'. Skipped loading symbols. Module is optimized and the debugger
option 'Just My Code' is enabled.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\Microsoft.Net\assembly\GAC_32\System.Data\v4.0_4.0.0.0__b77a5c561934e089\Syst
em.Data.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just
My Code' is enabled.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Xml\v4.0_4.0.0.0__b77a5c561934e089\Sys
tem.Xml.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just
My Code' is enabled.
The thread 0x12bc has exited with code 259 (0x103).
The thread 0x688 has exited with code 259 (0x103).
The thread 0xa40 has exited with code 259 (0x103).
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded 'D:\كتاب
المشاريع\اف دي بي الكتاب#c\شارب سي c#\Console Output\Console Output\bin\Debug\Console
Output.exe'. Symbols loaded.
```

```
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Configuration\v4.0_4.0.0.0__b03f5f7f11
d50a3a\System.Configuration.dll'. Skipped loading symbols. Module is optimized and the
debugger option 'Just My Code' is enabled.
Maximum value search started.
Maximum value initialized to4,at element index 0.
Now looking at element at index1.
New maximum found.New value is7,at element index1.
Now looking at element at index2.
Now looking at element at index3.
Now looking at element at index4.
Duplicate maximum found at elementindex4.
Now looking at element at index5.
Now looking at element at index6.
Duplicate maximum found at elementindex6.
Now looking at element at index7.
New maximum found.New value is8,at element index7.
Now looking at element at index8.
Now looking at element at index9.
New maximum found.New value is9,at element index9.
Now looking at element at index10.
Now looking at element at index11.
Duplicate maximum found at elementindex11.
Maximum value9found with2occurrences.
Maximum value search completed.
The thread 0x1250 has exited with code 259 (0x103).
The thread 0x424 has exited with code 259 (0x103).
The program '[1712] Console Output.vshost.exe: Program Trace' has exited with code 0
(0x0).
The program '[1712] Console Output.vshost.exe' has exited with code 0 (0x0).
```

5- والآن انتقل إلى نمط الاطلاق release وذلك باتباع الخطوات التالية:

❖ اختر الأمر Configuration Manager من القائمة Build ستظهر نافذة كما في الشكل(3-7):



الشكل (3-7)

❖ تأكد من اختيار Release من العمود Configuration.

❖ والآن اختر Release من مربع القائمة المنسدلة Active Solution Configurations.

6- نفذ التطبيق مرة أخرى لكن هذه المرة سيحدث التنفيذ في نمط الإطلاق ثم ألق نظرة على إطار Output عند انتهاء التنفيذ:

```
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\Microsoft.Net\assembly\GAC_32\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\mscorli
b.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just My
Code' is enabled.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\assembly\GAC_MSIL\Microsoft.VisualStudio.HostingProcess.Utilities\12.0.0.0__b
03f5f7f11d50a3a\Microsoft.VisualStudio.HostingProcess.Utilities.dll'. Skipped loading
symbols. Module is optimized and the debugger option 'Just My Code' is enabled.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Windows.Forms\v4.0_4.0.0.0__b77a5c5619
34e089\System.Windows.Forms.dll'. Skipped loading symbols. Module is optimized and the
debugger option 'Just My Code' is enabled.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Drawing\v4.0_4.0.0.0__b03f5f7f11d50a3a
\System.Drawing.dll'. Skipped loading symbols. Module is optimized and the debugger
option 'Just My Code' is enabled.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System\v4.0_4.0.0.0__b77a5c561934e089\System.
dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just My Code'
is enabled.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\assembly\GAC_MSIL\Microsoft.VisualStudio.HostingProcess.Utilities.Sync\12.0.0
.0__b03f5f7f11d50a3a\Microsoft.VisualStudio.HostingProcess.Utilities.Sync.dll'. Skipped
loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\assembly\GAC_MSIL\Microsoft.VisualStudio.Debugger.Runtime\12.0.0.0__b03f5f7f1
1d50a3a\Microsoft.VisualStudio.Debugger.Runtime.dll'. Skipped loading symbols. Module is
optimized and the debugger option 'Just My Code' is enabled.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded 'D:\كتاب
المشاريع\اف دي بي الكتاب#c\شارب سي c#\Console Output\Console Output\bin\Release\Console
Output.vshost.exe'. Skipped loading symbols. Module is optimized and the debugger option
'Just My Code' is enabled.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Core\v4.0_4.0.0.0__b77a5c561934e089\Sy
stem.Core.dll'. Skipped loading symbols. Module is optimized and the debugger option
'Just My Code' is enabled.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Xml.Linq\v4.0_4.0.0.0__b77a5c561934e08
9\System.Xml.Linq.dll'. Skipped loading symbols. Module is optimized and the debugger
option 'Just My Code' is enabled.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Data.DataSetExtensions\v4.0_4.0.0.0__b
77a5c561934e089\System.Data.DataSetExtensions.dll'. Skipped loading symbols. Module is
optimized and the debugger option 'Just My Code' is enabled.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\Microsoft.CSharp\v4.0_4.0.0.0__b03f5f7f11d50a
3a\Microsoft.CSharp.dll'. Skipped loading symbols. Module is optimized and the debugger
option 'Just My Code' is enabled.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\Microsoft.Net\assembly\GAC_32\System.Data\v4.0_4.0.0.0__b77a5c561934e089\Syst
em.Data.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just
My Code' is enabled.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Xml\v4.0_4.0.0.0__b77a5c561934e089\Sys
tem.Xml.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just
My Code' is enabled.
The thread 0x37c has exited with code 259 (0x103).
```

```
The thread 0x1274 has exited with code 259 (0x103).
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded 'D:\كتاب
المشاريع\اف دي بي الكتاب#c\شارب سي c#\Console Output\Console Output\bin\Release\Console
Output.exe'. Symbols loaded.
'Console Output.vshost.exe' (CLR v4.0.30319: Console Output.vshost.exe): Loaded
'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Configuration\v4.0_4.0.0.0__b03f5f7f11
d50a3a\System.Configuration.dll'. Skipped loading symbols. Module is optimized and the
debugger option 'Just My Code' is enabled.
Maximum value9found with2occurrences.
The thread 0x738 has exited with code 259 (0x103).
The thread 0x12c0 has exited with code 259 (0x103).
The program '[4132] Console Output.vshost.exe' has exited with code 0 (0x0).
The program '[4132] Console Output.vshost.exe: Program Trace' has exited with code 0
(0x0).
```

توضيح:

لقد اضطررنا هنا إلى تحديد نمط الإطلاق Release في موضعين ضمن VS لقد استخدمنا في هذا المثال نافذة Configuration Manager لتحديد نمط بناء المشروع واستخدمنا شريط الأدوات لتحديد نمط بناء الحل (Solution) ككل ويمكننا تحديد نمط الحل من خلال نافذة Configuration Manager أيضا وذلك ضمن مربع القائمة المنسدلة Active Solution Configuration.

إن ذلك لأن مشروعنا أو تطبيقنا يمثل جزءا من مشروع أكبر يسميه Visual Studio.NET 2013 بالحل أو Solution حيث يمكن أن يتضمن الحل عدة مشاريع.

كيفية العمل:

How it Works:

إن هذا التطبيق هو عبارة عن مثال موسع للتطبيق الذي تناولناه في الفصل السابق فهو يقوم باستخدام تابع لاحتساب القيمة العظمى في مصفوفة عددية إلا أن هذا التابع يعيد مصفوفة تتضمن الأدلة التي يحتلها العنصر الأكبر في المصفوفة وبالتالي يمكن للشيفرة التي استدعت التابع أن تعالج هذه العناصر.

لتناول الشيفرة السابقة بتفصيل أكبر كبداية لاحظ أننا استخدمنا تعليمة using جديدة كجزء من الشيفرة:

```
using System.Diagnostics;
```

إن ذلك يبسط الوصول إلى الأوامر التي تحدثنا عنها قبل البدء بهذا المثال وذلك لأنها موجودة ضمن فضاء الأسماء System.Diagnostics فعند استخدام تعليمة using هذه فإنه يمكننا كتابة أوامر التنقيح كما يلي:

```
Debug.WriteLine("Maximum value search started.");
```

وبدون هذه التعليمة فإن علينا تحديد الاسم التام للأمر أو التابع كما يلي:

```
System.Diagnostics.Debug.WriteLine("Maximum value search started.");
```

إن استخدام تعليمة using يبسط من شيفرتنا البرمجية ويقلص من حجمها.

تقوم الشيفرة الموجودة ضمن التابع Main() بتهيئة مصفوفة اختبارية بقيم من نوع int وتسمى هذه المصفوفة بالاسم testArray() يقوم هذا التابع أيضا بالتصريح عن مصفوفة من نوع int أخرى تسمى maxValIndices لحفظ دليل العنصر الأكبر في مصفوفة ضمنها حيث سيتم طباعة هذه الأدلة والقيمة العظمى على نافذة Console.

أما التابع Maxima() فهو معقد قليلا إلا أنه يستخدم معظم الصيغ التي تعرفنا عليها في الفصول السابقة إن خوارزمية البحث المستخدمة في هذا التابع مشابهة لتلك المستخدمة في التابع MaxVal() الذي تعرفنا عليه في الفصل السابق فيما عدا أن التابع Maxima() يقوم هنا بحفظ مواقع وجود القيمة العليا في المصفوفة.

ولربما يجدر بنا أن ننوه إلى أن المنهج الذي استخدمناه في انشاء مصفوفة الأدلة فبدلا من إنشاء مصفوفة حجمها مساو لحجم المصفوفة الأصلية فإننا قمنا بإنشاء مصفوفة حجمها مطابق تماما لعدد المواقع التي وجد فيها العنصر الأكبر في المصفوفة لاحظ انه لا يمكننا تغيير حجم المصفوفة متى تم إنشائها ولكننا هنا قمنا بإعادة إنشاء المصفوفة في كل مرة نحصل فيها على موقع جديد للعنصر الأكبر.

تبدأ عملية البحث بافتراض أن العنصر الأول من المصفوفة الأصلية (تسمى هذه المصفوفة ضمن التابع بالاسم integers) هو العنصر ذو القيمة الأكبر من المصفوفة وهو موجود في مكان واحد فقط من المصفوفة وهو الدليل 0 مبدئيا سنقوم بحفظ القيم العظمى التي وجدناها ضمن متحول باسم count وهو ما سيمكننا من تتبع المصفوفة indices.

إن الجسم الرئيسي للتابع متمثل بحلقة for والتي تقوم بالمرور خلال جميع العناصر في المصفوفة integers بدءا من العنصر الثاني ذو الدليل رقم 1 سيتم مقارنة كل قيمة في المصفوفة مع قيمة المتحول maxVal والذي يمثل القيمة الأكبر الآنية في المصفوفة فإذا كانت قيمة maxVal عندئذ سيأخذ المتحول maxVal قيمة ذلك العنصر وسيتم تعديل المصفوفة indices تتضمن دليل هذا العنصر أيضا ولكن إن كانت قيمة العنصر مساوية لقيمة maxVal فذا يعني أننا حصلنا على موقع آخر للعنصر الأكبر في المصفوفة وبالتالي سيتم إضافة موقع هذا العنصر كقيمة في المصفوفة indices إن هذه المصفوفة الجديدة هي اكبر بعنصر واحد من المصفوفة indices القديمة.

أن الشيفرة التي تقوم بهذه العملية هي كما يلي:

```
if (integers[i] == maxVal)
{
    count++;
    int[] oldIndices = indices;
    indices = new int[count];
    oldIndices.CopyTo(indices, 0);
    indices[count - 1] = i;
    Debug.WriteLine("Duplicate maximum found at element" +
        "index" + i + ".");
}
```

لاحظ أننا نحتفظ بالمصفوفة indices القديمة ضمن مصفوفة جديدة باسم oldIndices المحلية لاحظ أيضا أنه سيتم نسخ القيم في المصفوفة oldIndices إلى المصفوفة indices الجديدة وذلك باستخدام

التابع <array>.CopyTO() يقوم هذا التابع بنسخ جميع القيم الموجودة في المصفوفة <array> ولصقها في مصفوفة أخرى جديدة من نفس نوع المصفوفة <array>.

هناك العديد من النصوص التي تم طباعتها خلال هذه الشيفرة وذلك باستخدام الأمرين Debug.WriteLine() و Trace.WriteLine() والمحصلة النهائية لهذين الأمرين عند تنفيذ التطبيق في نمط Debug هو سجل كامل للخطوات التي تمت ضمن الحلقة اما في نمط Release فإننا رأينا فقط نتيجة العملية برمتها وذلك لأنه لن ينفذ أي أمر Debug.WriteLine() في هذا النمط.

لقد لاحظنا هنا وجود تابعي Write() خلال هذا المثال وهما مشابهان للتابع Console.Write() من الناحية الوظيفية:

- التابع Debug.Write().
- التابع Trace.Write().

ويستخدم هذان التابعان نفس صيغة التابع WriteLine() أي بارامتر أو بارامترين يتضمن الأول الرسالة أو النص المراد إظهارها والثاني صنف الرسالة إلا أنهما لا يؤديان للانتقال إلى سطر جديد أي أن الكتابة تتم في نفس السطر.

هناك أوامر أخرى أيضا مثل:

- التابع Debug.WriteLineIf().
- التابع Trace.WriteLineIf ().
- التابع Debug.WriteIf().
- التابع Trace.WriteIf ().

تستخدم هذه التوابع البارامترات نفسها وإليك صيغة أحد هذه التوابع وليكن Debug.WriteLineIf():

```
Debug.WriteLineIf(<condition>,<textToOutputIfTrue>,<category>);
```

لاحظ أن صيغة هذا التابع مشابهة لصيغة التابع Debug.WriteLine (). فيما عدا أن هناك بارامترا إضافيا هو ويمثل هذا البارامتر شرطا منطقيا أي يعيد قيمة من نوع bool فإن كانت نتيجة الشرط true فسيتم عندئذ طباعة النص في البارامتر والصنف في إطار Output أما إذا لم يتحقق الشرط فلن يتم طباعة أي شيء.

على سبيل المثال يمكننا أن نستخدم الأمر Debug.WriteLineIf() لطباعة نص على إطار Output عند حدوث حالة معينة وبالتالي يمكننا أن نستخدم هذا الأمر في أشياء كثيرة ضمن الشيفرة وذلك بالاعتماد على الشرط فإن لم يتحقق الشرط لن يظهر شيء على إطار Output وإن تحقق فسيظهر النص الموافق للأمر المحدد في البارامتر.

توضيح:

لاحظ أن أوامر Debug وTrace جميعها ليس لها أي تأثير على نسخة التطبيق النهائية فعند إنهاء التطبيق وتوزيعه فإن هذه الأوامر لن تقدم أو تؤخر من شيء في نسخة التطبيق الموزعة وإنما هي للأغراض البرمجية فقط.

التنقيح في نمط المقاطعة:

Debugging in Break Mode:

تعمل تقنيات التنقيح المتبقية في نمط المقاطعة ويمكن الدخول في هذا النمط بواسطة عدة طرق وجميعها تؤدي إلى توقف البرنامج عن التنفيذ بشكل مؤقت إن أول ما سنتعلمه في هذا الفصل هو كيفية الدخول في نمط المقاطعة ومن ثم سنتحدث عما يمكننا تحقيقه في نمط المقاطعة.

الدخول في نمط المقاطعة:

Entering Break Mode:

الطريقة الأبسط للدخول في نمط مقاطعة التنفيذ للتطبيق هي بالنقر على زر التوقف Break All أثناء تنفيذ التطبيق يمكنك أن تجد هذا الزر ضمن شريط أدوات Debug إن شريط الأدوات هذا لا يظهر بصورة افتراضية ويجب أن تظهره بنفسك وللقيام بذلك انقر بزر الفارة الأيمن على منطقة اشرطة الأدوات ثم اختر من القائمة المنسدلة البند Debug كما في الشكل (4-7).

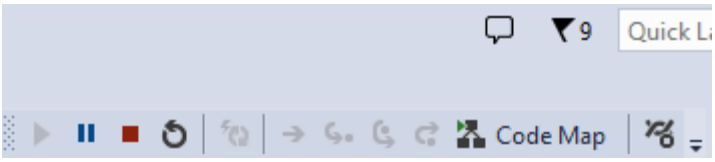
الشكل (4-7)

سيظهر عندئذ شريط أدوات Debug و هو كما في الشكل (5-7):

الشكل (5-7)

تمكننا الأزرار الأربعة الأولى من اليسار بالتحكم اليدوي بمقاطعة التطبيق ونلاحظ في الشكل أن هناك ثلاث ازرار مظللة بلون رمادي هذا يعني أنه لا يمكننا استخدامها في الوقت الحالي وذلك لأن التطبيق غير منفذ حاليا أما الزر الممكن هنا فهو Start والذي يقوم بتنفيذ التطبيق تماما كما لو أننا ضغطنا على المفتاح F5 أو اخترنا الأمر Run من القائمة Debug أما بقية الأزرار فسوف نتحدث عنها عندما نحتاج إلى استخدامها.

وعند تنفيذ التطبيق سيتغير مظهر شريط الأدوات هذا إلى الشكل (6-7):



الشكل (6-7)

والآن فإن الأزرار الثلاثة التي كانت مظللة مسبقا أصبحت ممكنة و هي تسمح لنا بالقيام بما يلي:

- إيقاف التطبيق مؤقتا والدخول في نمط المقاطعة.
- إيقاف التطبيق كليا (إن هذا لا يؤدي إلى الدخول في نمط المقاطعة وإنما إلى إنهاء تنفيذ التطبيق تماما).
- إعادة تشغيل التطبيق.

ربما يعد الإيقاف المؤقت للتطبيق من أبسط الطرق للدخول في نمط المقاطعة إلا أن ذلك لا يعطينا تحكما بالموضع الذي نود التوقف عنده تماما يسمح لنا هذا النوع من التوقف بمقاطعة التطبيق أثناء طلبه إدخالا من المستخدم مثلا ولكن في العديد من الحالات قد نحتاج إلى التوقف خلال معالجة طويلة أو خلال حلقة مثلا و هذا لا يمكن تحقيقه باستخدام الإيقاف المؤقت بشكل مباشر.

وبصورة عامة فإنه من الأفضل في حالة كتلك أن نستخدم نقاط المقاطعة.

نقاط المقاطعة:

Breakpoints:

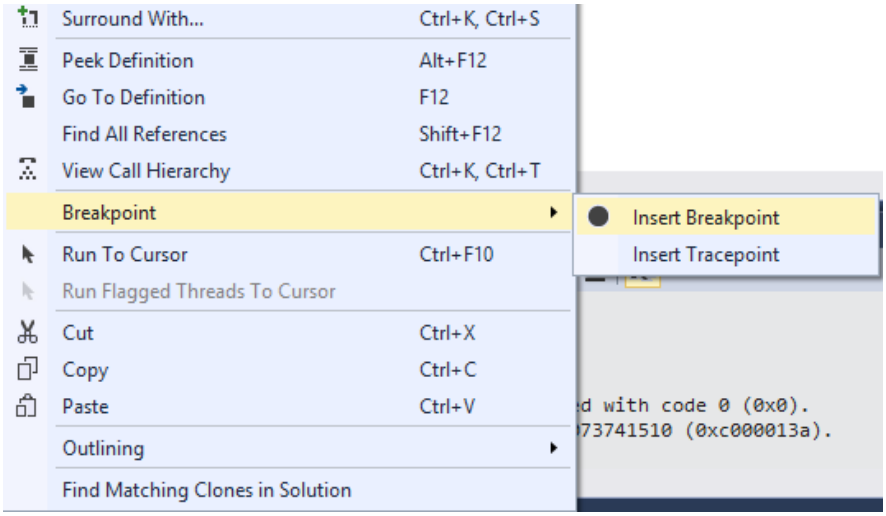
تمثل نقاط المقاطعة علامات في شيفرتك المصدرية تتم مقاطعة تنفيذ التطبيق عند الوصول إليها يمكننا إعداد نقاط المقاطعة وفقا لما يلي:

- ✓ الدخول في نمط المقاطعة فورا عند الوصول إلى نقطة المقاطعة.
- ✓ الدخول في نمط المقاطعة عند الوصول إلى نقطة المقاطعة إذا أخذ تعبير منطقي القيمة True.
- ✓ الدخول في نمط المقاطعة عند الوصول إلى نقطة المقاطعة لعدد محدد من المرات.
- ✓ الدخول في نمط المقاطعة عند الوصول إلى نقطة المقاطعة وقد تغيرت قيمة متحول ما منذ آخر مرة تم فيها الوصول إلى نقطة المقاطعة نفسها.

ملاحظة:

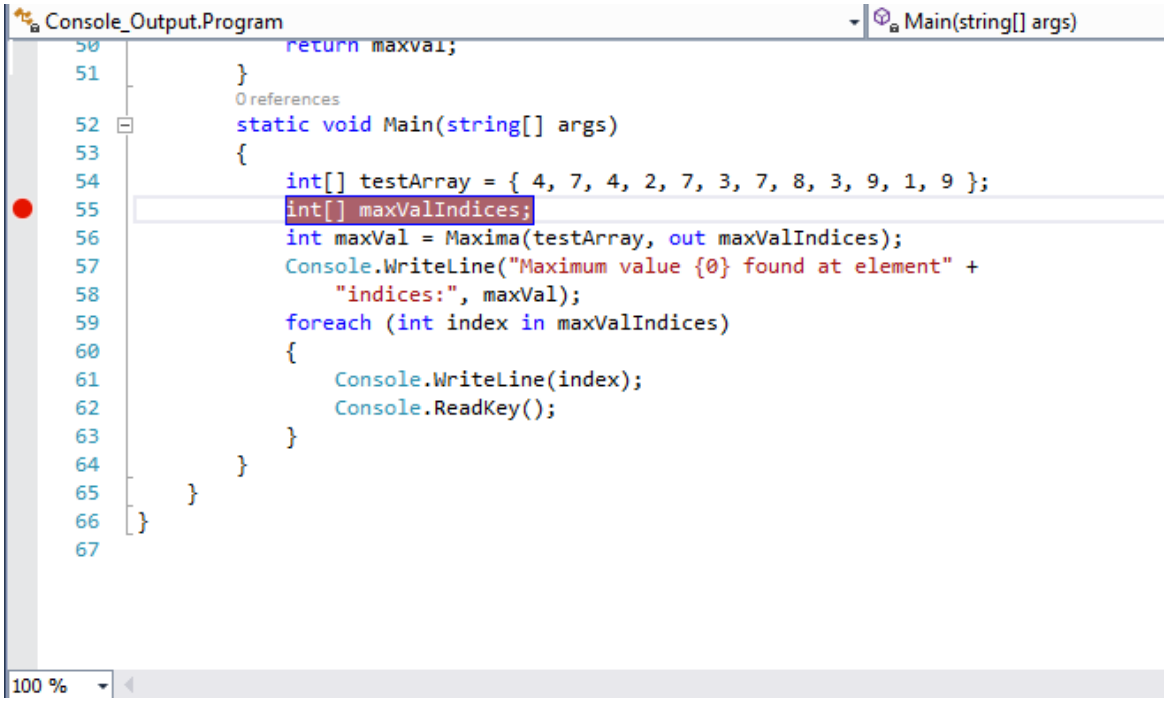
تذكر أن نقاط المقاطعة متوفرة فقط أثناء بناء التطبيق في نمط التنقيح Debug أي أنه لن يتم الاستجابة إلى نقاط المقاطعة عند ترجمة التطبيق في نمط الإطلاق حيث سيتم تجاهل جميع نقاط المقاطعة وكأنها غير موجودة.

هناك عدة طرق لإضافة نقاط المقاطعة فلإضافة نقطة مقاطعة بسيطة تقوم بمقاطعة تنفيذ التطبيق مباشرة عند الوصول إليها سنقوم بالنقر بزر الفأرة الأيسر على الشريط الرمادي الموجود إلى يسار إطار محرر الشيفرة أو بالنقر بزر الفأرة الأيمن على السطر البرمجي ثم اختيار الأمر Insert Breakpoint كما في الشكل (7-7):



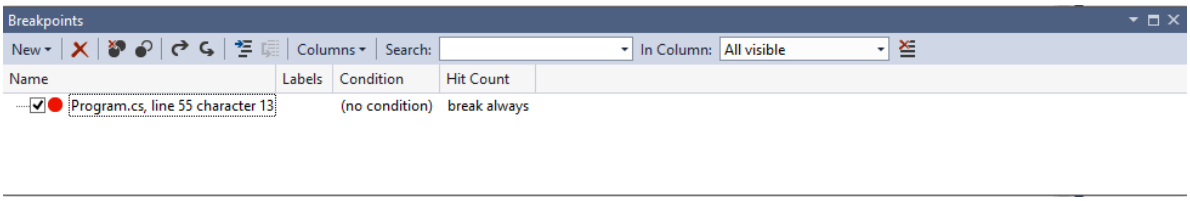
الشكل (7-7)

ستظهر نقطة المقاطعة كدائرة حمراء بجانب السطر الذي ستحدث عنده وسيتم تظليل ذلك السطر باللون الأرجواني كما في الشكل (7-8):



الشكل (7-8)

يمكننا تتبع نقاط المقاطعة الموجودة في الملف وذلك باستخدام إطار نقاط المقاطعة Breakpoints لكن علينا ان نعرض هذا الإطار أولا وذلك باختيار الأمر Breakpoints من القائمة الفرعية Windows من القائمة Debug سيظهر عندئذ إطار مشابه للشكل (7-9) وسيتوضع في نفس موضع إطار Task List وإطار Output:

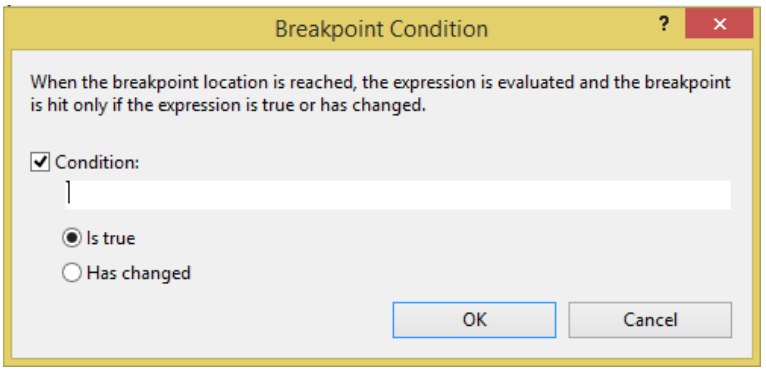


الشكل (7-9)

يمكننا باستخدام هذا الإطار حجب نقاط مقاطعة محددة وذلك بإزالة الإشارة الموجودة إلى يسار وصف نقطة المقاطعة حيث ستظهر نقطة المقاطعة المحجوبة على شكل دائرة حمراء مفرغة ويمكننا حذف نقاط المقاطعة أو تحرير خصائصها.

نلاحظ في إطار Breakpoints الخاصتين Condition وHit Count وهما الخاصيتان المتوفرتان إلا أنهما مفيدتان جدا يمكننا تحرير هاتين الخاصيتين بالنقر بزر الفارة الأيمن على نقطة المقاطعة ضمن الشيفرة او من خلال هذا الإطار ومن ثم اختيار الأمر Properties من القائمة المنسدلة يمكننا عندئذ استخدام ثلاثة بنود Function وFile وAddress وذلك لتغيير موضع نقطة المقاطعة يسمح لنا البند Address بتحديد عنوان مطلق في الذاكرة لنقطة المقاطعة وهناك الزران Condition وHit Count المستخدمتان لتغيير هاتين الخاصيتين الاتي نوهنا عنهما.

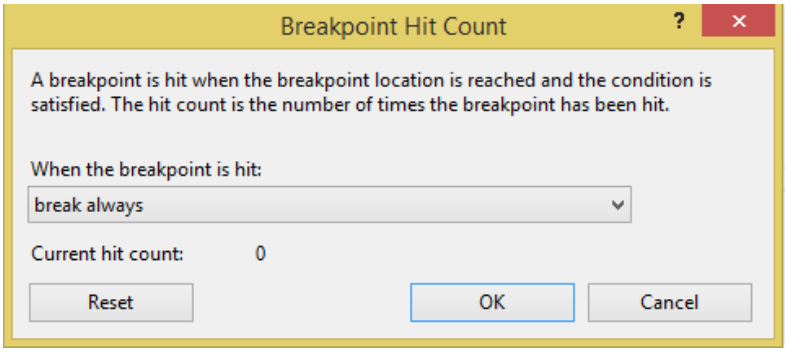
وباختيارك لزر Condition سيظهر إطار كما في الشكل(7-10):



الشكل (7-10)

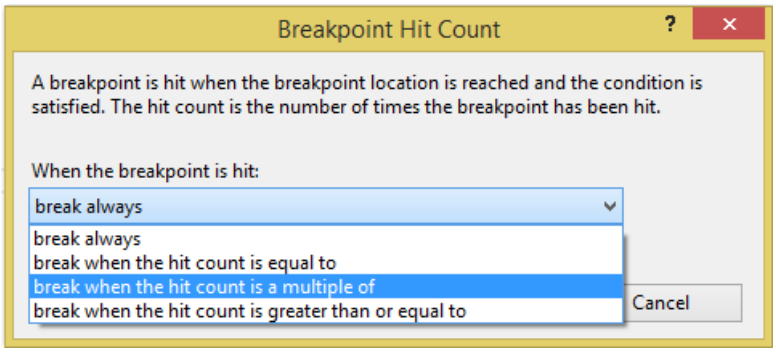
يستخدم هذا الإطار لكتابة تعبير منطقي يستخدم كشرط لمقاطعة التنفيذ عند نقطة المقاطعة تلك حيث يمكنك هنا كتابة هذا التعبير ويمكنك أن تضمّن التعبير أية متحولات ضمن مدى نقطة المقاطعة.

وعند اختيار الزر Hit Count سيظهر الإطار كما في الشكل (7-11):



الشكل (7-11)

يمكننا هنا أن نحدد عدد المرات التي سينفذ عنده سطر نقطة المقاطعة قبل أن تقدح الشكل (7-12) يبين الخيارات التي يوفرها مربع القائمة المنسدلة في هذا الإطار:



الشكل (7-12)

إن الخيار الذي سيتم تحديده في مربع القائمة هنا مرتبط بقيمة مربع النص المجاور والذي سيحدد ضمنه قيمة موافقة لهذا الخيار.

إن ميزة المقاطعة تلك مفيدة جدا خصوصا في الحلقات الطويلة فيمكننا مثلا أن نقاطع تنفيذ حلقة ما بعد الـ 5000 دورة الأولى.

طرق أخرى للدخول في نمط المقاطعة:

Other Ways of Entering Break Mode:

هناك طريقتان إضافيتان للدخول في نمط المقاطعة الأولى تقتضي بمقاطعة التطبيق عند رمي اعتراض في معالج unhandled exception وسوف نغطي موضوع الاعتراضات لاحقا في هذا الفصل عند الحديث عن معالجة الأخطاء واما الطريقة الأخرى فهي مقاطعة التنفيذ عند طرق توليد توكيد assertion ما.

والتوكيدات assertions عبارة عن تعليمات يمكن ان تقاطع تنفيذ التطبيق برسالة ما وتستخدم التوكيدات عادة اثناء تطوير التطبيق وذلك للتأكد من ان الأمور تسير على ما يرام على سبيل المثال يمكننا في نقطة ما من التطبيق أن نحتاج لأن تكون قيمة متحول ما اقل من 10 يمكننا ان نستخدم توكيدا لتفحص ما إذا كان ذلك قد حدث فعلا ام لا وإن لم يحدث ذلك سيتم مقاطعة التطبيق وعند حدوث التوكيد فإننا امام ثلاثة خيارات إما إلغاء التوكيد Abort مما سيؤدي إلى إنهاء التطبيق أو إعادة المحاولة Retry مما سيتسبب في الدخول في نمط المقاطعة او التجاهل Ignore حيث سيكمل تنفيذ التطبيق بصورة طبيعية.

وكما رأينا ان هناك توابع تستخدم للتنقيح Debug فإن هناك تابعان للتوكيد:

- التابع Debug.Assert().
- التابع Trace.Assert().

وكما هو الحال مع توابع التنقيح فإن هذين التابعين سيترجمان فقط أثناء بناء التطبيق في نمط التنقيح Debug.

يأخذ هذان التابعان ثلاثة بارامترات فقط يمثل الأول قيمة منطقية فإن كانت القيمة false سيتم قدح التوكيد أما البارامتر الثاني والثالث فهما يمثلان بارامترين نصيين حيث ستكتب هذه النصوص ضمن رسالة التوكيد وضمن إطار Output أيضا إليك مثالا يبين كيفية استخدام التابع Debug.Assert:

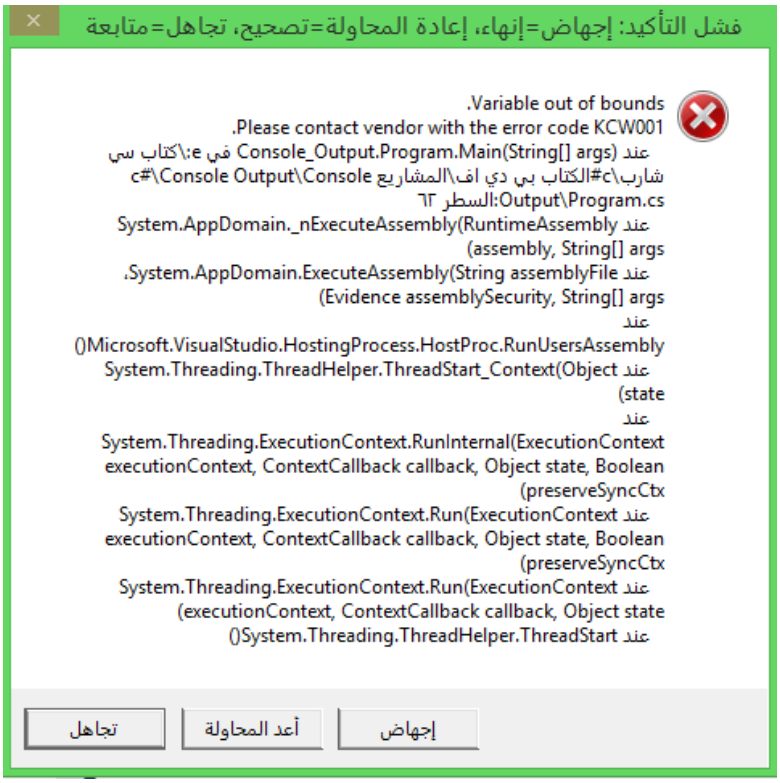
```
Debug.Assert(maxVal<10 ,"maxval is 10 or greater.",
                        "Assertion occurred in Main().");
```

إن التوكيدات مفيدة في أغلب الأحيان عند المراحل الأولى لاختيار التطبيق يمكننا أن نوزع تطبيقنا في نمط الإطلاق بحيث يتضمن التابع Trace.Assert() حيث يمكن للمستخدمين أن يكونوا على دراية بما يحصل في التطبيق وبالتالي يستطيعون أن ينقلوا هذه المعلومات إلى المطورين عندئذ سيتمكن المطورون من حل المشكلة حتى لو لم يعرفوا كيف سارت الأمور بصورة خاطئة.

يمكننا على سبيل المثال توفير معلومات موجزة عن الخطأ في السلسلة النصية الأولى وبتعليمات عما يجب القيام به في السلسلة النصية الثانية:

```
Trace.Assert(maxVal < 5, "Variable out of bounds.",
    "Please contact vendor with the error code KCW001.");
```

حيث ستظهر الرسالة التالية على المستخدم كما في الشكل (13-7):



الشكل (13-7)

فإن كانت بيئة تطوير Visual Studio.NET 2013 مثبتة على جهاز المستخدم وضغط على زر Retry ضمن نسخة من التطبيق تعمل في نمط الإطلاق Release عندئذ لن يعرض على المستخدم شيفرتنا البرمجية وإنما ستعرض أمامهم تعليمات بلغة التجميع assembly language تمثل شيفرة تطبيقنا إليك مثالا لبعض التعليمات التي قد تراها بالنسبة للشيفرة السابقة:

```
00000196 nop
00000197 pop ebx
00000198 pop esi
00000199 pop edi
0000019a mov esp , ebp
0000019c pop ebp
0000019d ret 4
```

إن تلك ليست بشيفرة سهلة يمكنك فهمها كشيفرة C# وليس هناك أمل في فهم هذه التعليمات إلا من قبل خبراء بلغة التجميع هذا يعني أن شيفرتنا البرمجية محمية من معظم عيون المتطفلين.

ستركز الأقسام التالية على ما يمكننا القيام به عند مقاطعة تنفيذ التطبيق وبصورة عامة فإننا لن ندخل في نمط المقاطعة إلا عندما نود أن نتتبع الأخطاء في الشيفرة أو فقط لنتأكد من أن الأمور تسير على ما يرام ومتى كنا ضمن نمط مقاطعة التطبيق فإن هناك الكثير من التقنيات التي يمكننا استخدام معظمها في تحليل شيفرتنا ومعرفة الحالة الفعلية لتطبيقنا عند النقطة التي توقفنا عندها.

مراقبة محتوى المتحولات:

Monitoring Variable Content:

إن مراقبة محتوى المتحولات ليس إلا مثالا واحدا لما يمكن أن يقدمه لنا VS من مساعدة كبرى لجعل الأمور بسيطة قدر المستطاع والطريقة الأبسط للتحقيق من قيمة متحول هي بتحريك مشيرة الفارة فوق اسم المتحول في الشيفرة البرمجية وذلك طبعا ضمن نمط المقاطعة سيظهر عندئذ تلميح ضمن إطار يعرض معلومات عن المتحول بالإضافة إلى القيمة التي يأخذها المتحول حاليا يمكننا أن نرى ذلك كما في الشكل (14-7):

```
56          int maxVal = Maxima(testArray, out maxValIndices);
57          Console.WriteLine("Maximum value {0} found at element" +
58              "indices:", maxVal);
59          foreach (int index in maxValIndices)
60          {
61              Console.WriteLine(index);
62              Console.ReadKey();          index 9
63
```

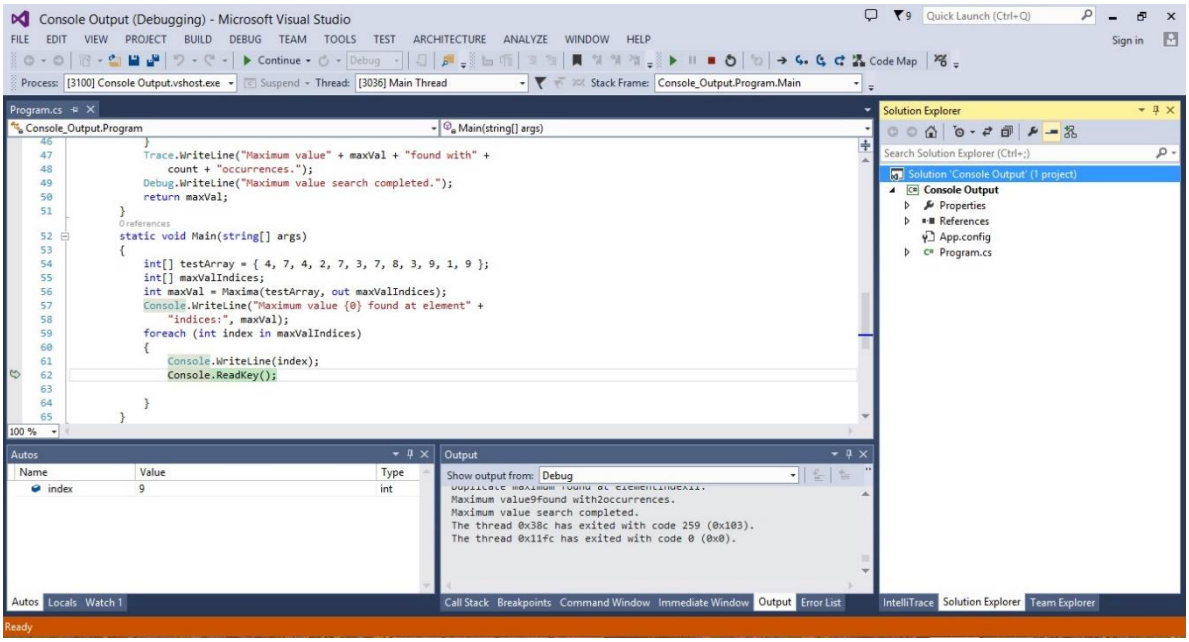
الشكل (14-7)

ويمكننا أيضا أن نركز على تعابير كاملة وليس مجرد متحول للحصول على معلومات حول نتائج هذه التعابير وبنفس الطريقة إن هذه التقنية محددة نوعا ما فهي لا تمكننا من استعراض محتوى مصفوفة على سبيل المثال.

ولان ربما قد لاحظت اننا عندما ننفذ تطبيقا من خلال VS فإن هناك تغيرا في عدد من إطارات VS وبصورة افتراضية إليك ما سيحدث أثناء تنفيذ التطبيق:

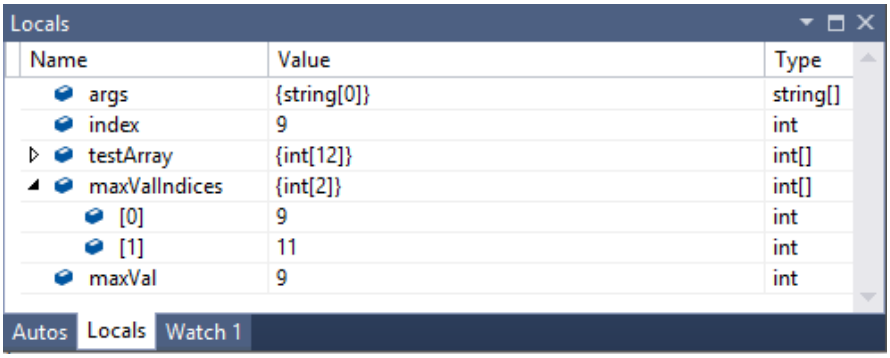
- ✓ اختفاء إطار الخصائص Properties.
- ✓ تغير حجم إطار Output حيث سيحتل نصف الجزء السفلي من الشاشة وظهور إطار جديد إن الإطار الجديد الذي سيظهر في زمن تنفيذ المشروع مفيد جدا لعمليات التنقيح فهو يسمح بتتبع ومراقبة قيم المتحولات في تطبيقنا عند مقاطعة التنفيذ ويتضمن هذا الإطار ثلاثة بنود:
  - البند Autos: حيث يستعرض المتحولات المستخدمة في التعليمات الحالية.
  - البند Locals: يستعرض جميع المتحولات الموجودة في المدى.

- البند Watch N: تخصيص عدد من المتحولات والتعابير لمراقبتها حيث تتراوح N بين 1 و 4 إليك الشكل الافتراضي للشاشة في زمن تنفيذ المشروع الشكل (15-7):



الشكل (15-7)

تعمل جميع هذه الإطارات بنفس المبدأ تقريبا مع بعض المزايا الإضافية لكل إطار وذلك بالاعتماد على وظيفة الإطار وبصورة عامة فإن كل إطار يتضمن لائحة من المتحولات تشتمل على المعلومات التالية: اسم المتحول، قيمة المتحول، ونوع هذا المتحول. أما بالنسبة لمتحولات أكثر تعقيدا كالمصفوفات فإنها قد تتضمن معلومات إضافية يمكننا استعراضها أو طيها بالنقر على رمز السهم الأفقي أو رمز السهم المائل المجاور لاسم المتحول على سبيل المثال يبين الشكل (16-7) إطار تتبع المتحولات: البند Locals الناتج عن وضع نقطة مقاطعة في شيفرة التطبيق السابق بعد استدعاء التابع Maxima() مباشرة:

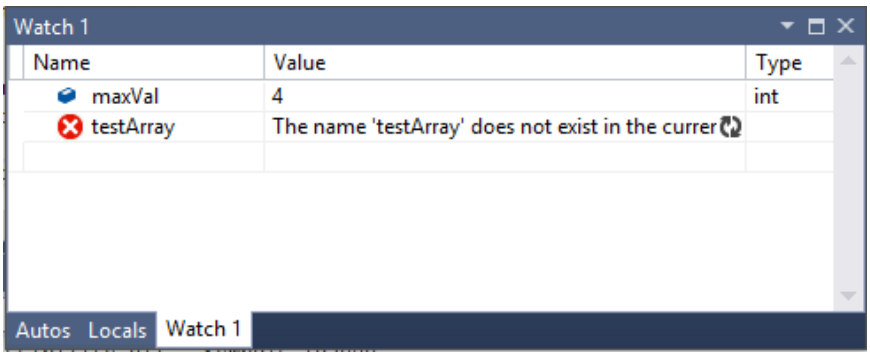


الشكل (16-7)

لاحظ هنا كيفية عرض متحولات المصفوفات (لقد قمت بتوسعة المصفوفة maxValIndices).

يمكننا تعديل محتوى المتحولات بواسطة هذا الإطار أيضا وعند قيامنا بذلك فإننا سنتجاوز أي إسناد لهذه المتحولات يمكن أن يحدث ضمن الشيفرة وللقيام بذلك فإننا ببساطة سنكتب القيمة الجديدة ضمن عمود value للمتحول الذي نود تغيير قيمته.

أما البند Watch فإنه يمكننا من مراقبة متحولات أو تعابير محددة ولاستخدام ها الإطار فإننا سنكتب اسم المتحول أو التعبير ضمن عمود Name ومن ثم سنلحظ النتائج لاحظ أنه لا يمكن لجميع المتحولات الموجودة في التطبيق أن تكون ضمن المدى دائما على سبيل المثال يبين الشكل (7-17) إطار Watch لعدد من المتحولات والتعابير ولقد استخدمنا شيفرة المثال الأخير هنا أيضا وقد قاطعنا تنفيذ التابع Maxima():



الشكل (7-17)

إن المصفوفة testArray () هي مصفوفة محلية بالنسبة للتابع Main () ولا يمكننا أن نرى قيمة لها هنا وبدلا من ذلك فإننا سنحصل على رسالة خطأ تبين أن المتحول خارج المدى.

يمكننا أن نضيف المتحولات إلى إطار Watch بسحب اسم المتحول من الشيفرة المصدرية وإسقاطه ضمن الإطار.

ولكي نضيف إطارات أخرى فإننا سنستخدم الأمر Watch N من القائمة الفرعية Watch من Windows من القائمة Debug حيث يمكننا أن نستخدم أربعة إطارات Watch كحد أقصى وبأمر القائمة هذا يمكننا أن ننتقل بين هذه الإطارات فكل إطار يمكن أن يتضمن على مجموعة منفردة من المتحولات والتعابير لمراقبتها وبالتالي يمكننا تجميع المتحولات المرتبطة مع بعضها البعض للوصول إليها بسهولة وبالإضافة إلى إطارات Watch الأربعة تلك فإن هناك الإطار QuickWatch أيضا والذي يعطينا معلومات تفصيلية عن متحول في الشيفرة المصدرية بسرعة ولاستخدام هذا الإطار فإننا سننقر بزر الفأرة الأيمن على المتحول ونختار الامر QuickWatch من القائمة المنسدلة.

هناك نقطة أخيرة يجب أن نذكرها قبل إنهاء حديثنا عن إطارات Watch وهي أننا لسنا بحاجة لإضافة المتحولات أو التعابير إلى إطار Watch في كل مرة نقوم فيها بتنفيذ التطبيق فسوف يتذكر VS المتحولات التي طلبنا مراقبتها وذلك لمراقبتها عند التنفيذ التالي للشيفرة.

الخطو خلال الشيفرة:

Stepping Through Code:

لقد رأينا ما يحدث في تطبيقنا عند النقطة التي يدخل فيها نمط المقاطعة والآن سوف نرى كيف نستخدم VS للخطو خلال الشيفرة أثناء مقاطعة التنفيذ مما يسمح لنا برؤية النتائج الحالية للشيفرة المنفذة.

عند الدخول في نمط المقاطعة سيظهر سهم على اليسار محرر الشيفرة عند السطر الذي يوشك على أن ينفذ يمكن أن يظهر بشكل أولي ضمن الدائرة الحمراء لنقطة المقاطعة هذا إن استخدمت نقطة مقاطعة للدخول في نمط مقاطعة التنفيذ:

```
51          }
            0 references
52          static void Main(string[] args)
53          {
54              int[] testArray = { 4, 7, 4, 2, 7, 3, 7, 8, 3, 9, 1, 9 };
55              int[] maxValIndices;
56              int maxVal = Maxima(testArray, out maxValIndices);
57              Console.WriteLine("Maximum value {0} found at element" +
58                  "indices:", maxVal);
59              foreach (int index in maxValIndices)
60              {
61                  Console.WriteLine(index);
62                  Console.ReadKey();
```

الشكل (18-7)

يمكننا هنا رؤية النقطة التي وصل التنفيذ عندها أثناء دخولنا لنمط المقاطعة وعند هذه النقطة يمكننا أن نختار الاستمرار بالتنفيذ سطرا بسطر وللقيام بذلك فإننا نستخدم أزرار شريط أدوات Debug الذي تعرفنا عليه في السابق وكما تشير الأسهم فإن بإمكاننا ان نتحكم في سير التطبيق وهو في نمط المقاطعة بواسطة هذه الأيقونات الثلاث وهي:

- زر Step Into حيث سيتم تنفيذ السطر الحالي والانتقال على السطر التالي لتنفيذه.
- زر Step Over كما في الزر السابق إلا اننا لن ندخل في الكتل البرمجية المعششة.
- زر Step Out تنفيذ الأسطر البرمجية إلى نهاية الكتلة البرمجية والاستمرار في نمط المقاطعة عند التعليمة التي تلي هذه الكتلة البرمجية.

فإن أردنا ان نتفحص كل عملية يقوم بها التطبيق فغننا سنستخدم Step Into لتتبع تنفيذ التعليمات واحدة تلو الأخرى عن هذا يتضمن الانتقال إلى داخل التوابع كالتابع ()Maxima مثلا وبالنقر على هذه الايقونة عند وصول السهم على استدعاء التابع ()Maxima سيؤدي ذلك لانتقال السهم على اول سطر في التابع ()Maxima اما بالنقر على Step Over عند هذا السطر فسيتم نقل السهم على السطر الذي يلي استدعاء التابع ()Maxima دون الدخول إلى شيفرة التابع ()Maxima وإن خطونا ضمن التابع لا نود ان نراقب ما يجث ضمنه يمكننا ببساطة ان نضغط على Step Out حيث سنعود على الشيفرة التي استدعت هذا التابع.

ملاحظة:

انتبه من ان تختلط عليك الأمور فتنفيذ الشيفرة له علاقة بالخطو هنا وإن ما سيحدث هنا هو فقط تحديد إيقاع معين للدخول في نمط المقاطعة ليس أكثر.

ومع تخطينا لأسطر الشيفرة البرمجية وانتقالنا من سطر إلى آخر فإن هناك احتمالا لتغيير قيم المتحولات في هذه الشيفرة وبتتبع وتفحص الإطارات التي تحدثنا عنا مسبقا يمكننا أن نرى ما يحدث في تطبيقنا بسهولة.

عندما نواجه شيفرة يشتبه في وجود أخطاء ضمنها تصبح هذه التقنية الأكثر أهمية هنا يمكننا ان نخطو ضمن الشيفرة إلى النقطة التي نتوقع حصول المشاكل عندها حيث ستولد الأخطاء تماما كما لو اننا ننفذ البرنامج بصورة عادية وخلال ذلك يمكننا ان نتتبع البيانات لرؤية ما يسير بصورة خاطئة سوف نستخدم هذه التقنية لاحقا في هذا الفصل وذلك لرؤية ما يحدث في التطبيق.

هناك بضعة إطارات أخرى لم نتحدث عنها بعد فهناك عدة إطارات تظهر ضمن إطار Task List/Output خلال التنقيح وهما إطار Command Window وإطار Call Stack وإطار Breakpoints وإطار Immediate Window.

الأوامر الفورية:

Immediate Commands:

الإطار Command window يسمح لنا بالقيام بوظائف وعمليات VS دون استخدام الأدوات او القوائم وأما الإطار Immediate Window فهو يسمح لنا بتنفيذ شيفرة ضمن الشيفرة الموجودة في الملف المنفذ حاليا ستلاحظ وجود الرمز (>) عندما نكون ضمن إطار Command Window وذلك عند بداية كل سطر في هذا الإطار يمكننا أن ننتقل إلى إطار Immediate Window من إطار Command window بكتابة immed في هذا الإطار والضغط على مفتاح Enter أما عندما نكون ضمن إطار Immediate Window ونريد الذهاب للإطار Command window فإننا نقوم بكتابة (>cmd) ضمن الإطار Command window والضغط على مفتاح Enter طبعا يمكنك استخدام الفارة للتنقل بين الإطارين وللتذكير هذين الإطارين يعملان في حالة التنقيح فقط.

سوف نركز على إطار Immediate Window هنا وذلك لأن الإطار Command window مفيد فقط بالنسبة للتعابير المعقدة الطريقة الابسط لاستخدام هذا الإطار هو بتنفيذ التعابير وللقيام بذلك فإننا سنكتب التعبير مباشرة ثم نضغط على مفتاح Enter ستظهر المعلومات المطلوبة عندئذ على سبيل المثال انظر الشكل (7-19):

الشكل (7-19)

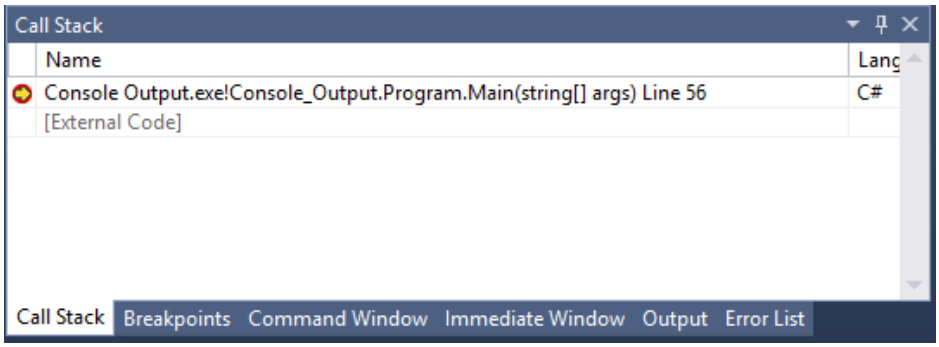
وفي معظم الحالات فإننا سنستفيد من إطارات مراقبة المتحولات الإطار Watch أكثر من هذا الإطار إلا أن هذه التقنية تبقى مفيدة أحيانا وذلك لاختبار التعابير مثلا.

الإطار Call Stack:

The Call Stack Window:

الإطار الأخير الذي سنتناوله هنا يستعرض لنا النقطة التي وصلنا عندها أثناء تنفيذ التطبيق وبصورة أبسط فإن هذا الإطار يبين لنا التابع الذي يجري تنفيذه حاليا بالإضافة إلى التابع الذي قام باستدعائهوالتابع الذي استدعى ذلك التابع أيضا و هكذا وبالتالي فغننا سنحصل على قائمة شجرية من استدعاءات التوابع تسمى بمكدس الاستدعاءات وسيتم تحديد النقطة التي حدث عندها استدعاء التابع أيضا.

وفي مثالنا السابق إذا دخلنا نمط المقاطعة في التابع Maxima() أو بالانتقال إلى هذا التابع باستخدام تقنية تخطي الشيفرة فسيؤدي ذلك إلى ظهور النتائج التالية على إطار Call Stack:



الشكل (7-20)

إن هذا الإطار مفيد جدا عند المرحلة الأولية لاستكشاف الأخطاء باعتبار أنه يسمح لنا برؤية ما يحدث قبل حدوث الخطأ مباشرة وذلك باعتبار أن معظم الأخطاء تحدث ضمن التوابع مما سيساعدنا على تحديد مصدر الخطأ أي التابع الذي حصل عنه الخطأ.

ملاحظة:

لاحظ ان هذا الإطار يمكن أن يعرض معلومات مركبة فيمكن مثلا أن يحدث الخطأ خارج التطبيق على سبيل المثال كاستخدام توابع خارجية بصورة خاطئة وفي حالات كهذه فإنك ستجد لائحة طويلة من المعلومات في هذا الإطار.

معالجة الأخطاء:

Error Handling:

لقد تعلمنا في الجزء الأول من هذا الفصل كيفية اكتشاف الأخطاء وتصحيحها خلال تطوير التطبيق وذلك باعتبار أنها لن تحدث في مرحلة الإطلاق (أي عندما يصل التطبيق لأيدي المستخدمين) لكن هناك حالات يمكننا أن نقدر حدوث الأخطاء عندها وليس هناك أي طريقة نضمن بها عدم حدوث الأخطاء 100% وفي حالات كهذه فإنه لمن الأفضل أن نعالج هذه الأخطاء وأن نكتب شيفرة قوية بالصورة التي يمكنها أن تتعامل مع هذه الأخطاء بصورة مثمرة دون مقاطعة تنفيذ التطبيق بصورة مزعجة للمستخدم.

الاعتراضات:

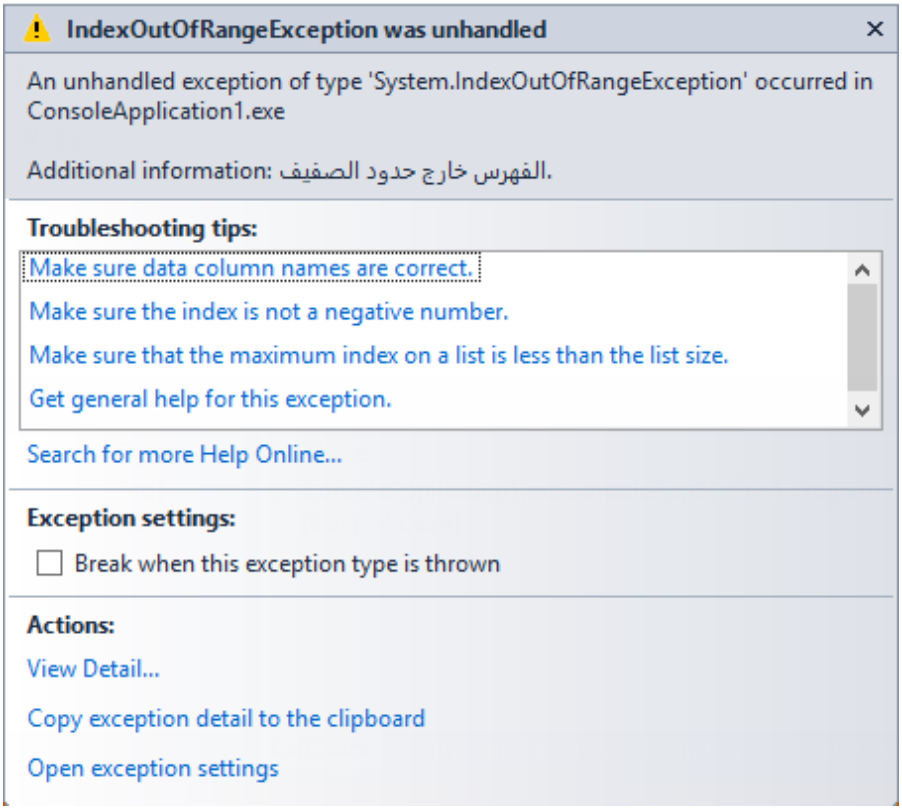
Exceptions:

يمثل الاعتراض خطأ حدث أثناء تنفيذ التطبيق وتم توليده إما من شيفرتنا أو من تابع استدعته شيفرتنا إن وصف الخطأ هنا أكثر غموضا من الوصف الذي استخدمناه مسبقا وذلك لأن الاعتراضات يمكن أن تولد بصورة يدوية في التوابع إلى غير ذلك من أمور على سبيل المثال يمكننا أن نولد اعتراضا في تابع إذا لم يبدأ أحد البارامترات النصية لتابع بالحرف "a" إن هذا لا يمثل خطأ بحد ذاته وإنما وبحكم سياق ومنطق التطبيق فإننا سنتعامل مع حالات كهذه على أنها تمثل خطأ.

لقد تناولنا الاعتراضات عددا من المرات في هذا الكتاب ولربما أبسط مثال لذلك هو عند محاولة الإشارة إلى عنوان عنصر في مصفوفة خارج مجالها على سبيل المثال:

```
int[] myArray = { 4, 7, 4, 2 };
int myElem=myArray[4];
```

سيولد ذلك رسالة اعتراض وسيتوقف التطبيق عندها بالرسالة الشكل (21-7):



الشكل (21-7)

تعرف الاعتراضات في فضاءات الأسماء ولمعظمها أسماء تدل على معانيها على سبيل المثال إن الاعتراض المولد في الشيفرة السابقة يأخذ الاسم System.IndexOutOfRangeException ومن خلال هذا الاسم يمكننا أن نستكشف ان الخطأ نتج عن استخدام دليل خارج المجال الممكن للمصفوفة myArray.

لا تظهر هذه الرسالة ولا يتوقف التطبيق عن العمل إلا إذا لم يكن الاعتراض معالجا إذن ماذا علينا ان نفعل لكي نتمكن من معالجة الاعتراضات؟

التركيب try..catch..finally:

try..catch..finally:

تتضمن لغة C# صيغة لمعالجة الاعتراضات البنيوية Structured Exception Handling أو اختصار SHE وهي عبارة عن كلمات مفتاحية تستخدم للإشارة إلى أن الشيفرة قابلة لمعالجة الاعتراضات مضمنة إياها تعليمات ستنفذ عند حدوث اعتراض ما إن هذه الكلمات الثلاث هي try,catch,finally إن كل كلمة من هذه الكلمات لها كتلة برمجية ويجب أن تستخدم هذه الكلمات الثلاث بصورة متعاقبة وبترتيب ثابت والبنية الأساسية لهذه الصيغة هي كما يلي:

```
Try
{
..
Catch (<exceptionType> e)
{
..
}
Finally
{
..
}
```

ومن الممكن أن تكون هناك كتلة try وكتلة finally فقط دون وجود الكتلة catch أو أن يكون هناك كتلة try وعدة كتل catch فإن كان هناك كتلة أو أكثر من كتل catch عندئذ فإن وجود الكتلة finally هو أمر اختياري وإلا فإن وجودها إجباري.

- الكتلة try تتضمن الشيفرة التي يمكن ان تتسبب برمي الاعتراضات حيث تعني الكلمة رمي throw في لغة C# توليد الخطأ أو الشيفرة المتسببة في حدوث الخطأ.
- الكتلة catch وتتضمن الشيفرة التي ستنفذ عند رمي الاعتراضات يمكن أن تعد كتل catch بطريقة تستجيب عندها لأنواع اعتراضات محددة مثل الاستجابة للاعتراض System.IndexOutOfRangeException فقط وذلك بتحديد نوع الاعتراض ضمن <exceptionType> حيث يمكننا توفير عدة كتل catch تعالج كل واحدة منها نوعا من الاعتراضات أو الأخطاء ومن الممكن تجاهل البارامتر وفي حالة كهذه فإنه يكون لدينا كتل catch عامة تستجيب لجميع الاعتراضات.
- الكتلة finally والتي تتضمن الشيفرة التي ستنفذ دوما سواء بعد الكتلة try إذا لم يحدث أي اعتراض أو بعد الكتلة catch عند حدوث الاعتراض ومعالجته ضمنها أو قبل أن ينهي الاعتراض غير المعالج تنفيذ التطبيق.

ملاحظة:

في الحقيقة إن تنفيذ كتلة finally قبل إنهاء تنفيذ التطبيق نتيجة اعتراض غير معالج هو السبب الرئيسي لوجود الكلمة finally وذلك باعتبار أنه يمكننا ان نضع الشيفرة بعد صيغة SHE بصورة مباشرة لكي تنفذ.

أما تسلسل الأحداث الذي يمكن أن يحدث بعد حدوث الاعتراض في الشيفرة ضمن الكتلة try فهو كما يلي:

- سيتوقف تنفيذ الكتلة try عند النقطة التي حدث عندها الاعتراض.
- سيتم البحث عن كتل catch المخصصة لمعالجة الاعتراض الحاصل حيث سيتم تفحص ما إذا كانت أحد هذه الكتل مطابقة لنوع الاعتراض الذي رمي فإن لم تتوفر هناك كتل catch موافقة لهذا الاعتراض سيتم تنفيذ الشيفرة الموجودة في الكتلة finally وهو ما يعلل وجوب وجود الكتلة finally عند عدم توفر كتلة catch موافقة لنوع الاعتراض الحاصل.
- وإن طابقت كتلة catch نوع الاعتراض الحاصل عندئذ سيتم تنفيذ شيفرة هذه الكتلة ومن ثم تنفيذ شيفرة الكتلة finally إن وجدت هذه الكتلة في تلك الحالة.

لقد حان الوقت الآن لكي نتناول مثالا يستعرض كيفية معالجة الاعتراضات:

تطبيق حول كتابة نص في نافذة الخرج:

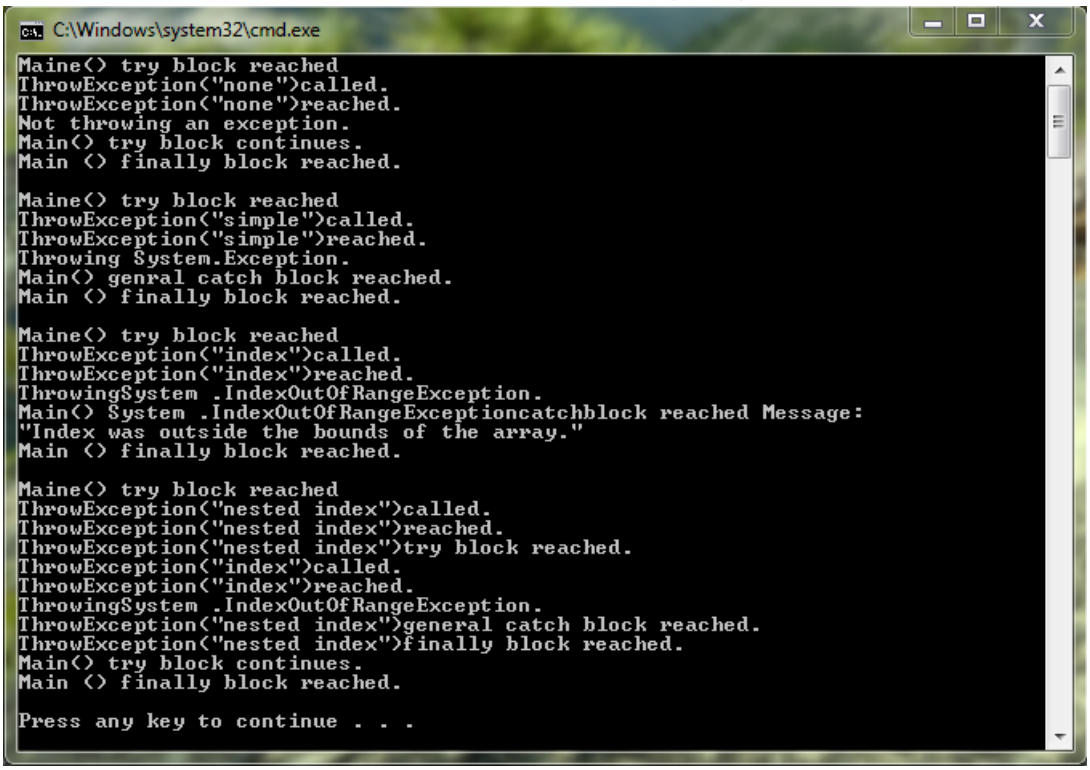
1- قم بإنشاء تطبيق Console جديد باسم Console TryCatchFinally.
2- أضف الشيفرة البرمجية التالية إلى محرر نصوص C# في class Program:

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

namespace Console_TryCatchFinally
{
    class Program
    {
        static string[] eTypes = { "none", "simple", "index", "nested index" };
        static void ThrowException(string exceptionType)
        {
            //line14
            Console.WriteLine("ThrowException(\"{0}\")reached.", exceptionType);
            switch (exceptionType)
            {
                case "none":
                    Console.WriteLine("Not throwing an exception.");
                    break;
                    //line21
                case "simple":
                    Console.WriteLine("Throwing System.Exception.");
```

```
                throw (new System.Exception());
                //line25
                break;
            case "index":
                Console.WriteLine("Throwing" +
                    "System .IndexOutOfRangeException.");
                eTypes[4] = "error";   //line30
                break;
            case "nested index":
                try           //line33
                {
                    Console.WriteLine("ThrowException(\"nested index\")" +
                        "try block reached.");
                    Console.WriteLine("ThrowException(\"index\")called.");
                    ThrowException("index");   //line38
                }
                catch      //line40
                {
                    Console.WriteLine("ThrowException(\"nested index\")" +
                        "general catch block reached.");
                }
                finally
                {
                    Console.WriteLine("ThrowException(\"nested index\")" +
                        "finally block reached.");
                }
                break;
        }
    }

    static void Main(string[] args)
    {
        foreach (string eType in eTypes)
        {
            try
            {
                Console.WriteLine("Maine() try block reached");//line60
                Console.WriteLine("ThrowException(\"{0}\")called."
                    , eType);//line61
                ThrowException(eType);
                Console.WriteLine("Main() try block continues.");//line64
            }
            catch (System.IndexOutOfRangeException e)//line66
            {
                Console.WriteLine("Main() System .IndexOutOfRangeException" +
                    "catch" + "block reached Message:\n\"{0}\"", e.Message);
            }
            catch
                //line72
            {
                Console.WriteLine("Main() genral catch block reached.");
            }
            finally
            {
                Console.WriteLine("Main() finally block reached.");
            }
            Console.WriteLine();
```

```
            }
        }
    }
}
```

3- نفذ التطبيق في نمط التنقيح باختيار الأمر Start without Debugging أو بالضغط على مفتاح Ctrl+ F5 فيظهر الشكل (7-22).



الشكل (7-22)

كيفية العمل:

How it Works:

لهذا التطبيق كتلة try في التابع Main() الذي يستدعي التابع ThrowException() ويمكن لهذا التابع أن يرمي اعتراضات وذلك بحسب البارامتر الممرر إليه.

أي سيتم التصرف بحسب البارامتر الممرر وذلك وفقا لما يلي:

- الاستدعاء ThrowException("none") لن يؤدي إلى رمي أي اعتراض.
- الاستدعاء ThrowException("simple") سيولد اعتراضا عاما.
- الاستدعاء ThrowException("index") سيولد الاعتراض System.IndexOutOfRangeException.

❖ الاستدعاء ThrowException("nested index") يتضمن على كتلة try خاصة به والتي تتضمن شيفرة تقوم باستدعاء ThrowException("index") والذي سيولد الاعتراض System.IndexOutOfRangeException.

إن كل قيمة نصية من قيم البارامتر مخزنة ضمن المصفوفة العامة eTypes حيث سيتم المرور خلال هذه المصفوفة واستدعاء التابع ThrowException() مرة واحدة وفقا لهذه القيم النصية وخلال ذلك سيتم طباعة عدد من الرسائل على نافذة Console للإشارة إلى ما يحدث.

تعطي هذه الشيفرة فرصة مناسبة لاستخدام تقنيات الخطو التي رأيناها في هذا الفصل فبالانتقال سطرا بسطر خلال هذه الشيفرة يمكننا أن نلحظ كيفية توليد أو رمي الاعتراضات.

قم بإضافة نقطة مقاطعة عند السطر رقم 60 الذي يتضمن على التعليمة:

```
Console.WriteLine("Maine() try block reached");
```

ملاحظة:

لاحظ أنني سأشير إلى الشيفرة وفقا لأرقام الأسطر والتي تظهر ضمن الشيفرة الموجودة في الكتاب على هيئة تعليقات من الشكل //Line xx.

نفذ التطبيق في نمط التنقيح Debug.

ما أن تنفذ التطبيق حتى يدخل في نمط المقاطعة مباشرة حيث سيظهر سهم عند السطر رقم 60 وإذا اخترت البند Locals في إطار مراقبة المتحولات ستجد أن قيمة eType الحالية هي none استخدم الأيقونة Step Into لتنفيذ الأسطر 60 و61 ومن ثم تفحص السطر الأول من النص الذي تمت طباعته على نافذة Console بعد ذلك استخدم الأيقونة Step Into مرة أخرى للانتقال إلى استدعاء التابع ThrowException() على السطر 63.

ومتى انتقل التنفيذ إلى التابع ThrowException() في السطر رقم ستتغير محتويات الإطار Locals حيث لم يعد المتحولان args أو eType ضمن المدى فهما متحولان محليان للتابع Main() وبدلا منهما سنجد المتحول المحلي exceptionType وهو بارامتر التابع حيث يعامل كالمتحول المحلي في هذا التابع وهو يأخذ القيمة none طبعا استمر بالضغط على أيقونة Step Into إلى ان تصل إلى تعليمة switch التي تتفحص قيمة exceptionType وتنفيذ الشيفرة التي ستقوم بطباعة النص: Not throwing an exception على الشاشة وعند تنفيذ تعليمة break في السطر رقم 21 سيتم الخروج من التابع والاستمرار في تنفيذ التابع Main() في السطر رقم 60 لاحظ أنه لم يتم رمي أي اعتراض وبالتالي سيستمر التنفيذ في كتلة try.

بعد ذلك سيستمر التنفيذ مع الكتلة finally استمر بالنقر على أيقونة Step Into عددا من المرات لإنهاء تنفيذ كتلة finally والدورة الأولى لحلقة foreach وعند الوصول التالي للسطر رقم 60 سيتم استدعاء التابع ThrowException() ببارامتر جديد هو simple.

وبالاستمرار في الضغط على الايقونة Step Into خلال التابع ThrowException() ستلاحظ أننا وصلنا بعد عدد من النقرات على ايقونة Step Into إلى السطر رقم 25.

```
throw (new System.Exception());
```

لقد استخدمنا الكلمة throw لتوليد اعتراض بصورة يدوية ولاحظ أننا هنا قمنا بتوليد اعتراض جديد باستخدام الكلمة new بتحديد ذلك كبارامتر لهذه الكلمة لقد استخدمنا هنا اعتراضا آخر من فضاء الأسماء System.Exception.

وعندما تنفيذ هذه التعليمة في نمط Step Info سنجد أنفسنا ضمن كتلة catch العامة الموجودة في السطر رقم 72 فليس هناك أي كتلة catch مطابقة لهذا النوع من الاعتراضات وبالتالي سيتم تنفيذ كتلة catch العامة وبالخطو خلال شيفرة كتلة catch وصولا للكتلة finally والعودة إلى دورة الحلقة التي استدعت التابع ThrowException() بالبارامتر simple في السطر رقم 60 وهذه المرة فإن البارامتر الممرر للتابع ThrowException() هو index.

وفي هذه المرة سيولد التابع ThrowException() اعتراضا في السطر رقم 30:

```
eTypes[4] = "error"
```

إن المصفوفة eTypes هي مصفوفة عامة وبالتالي يمكننا الوصول إليها من ضمن أي تابع في هذا الملف لكننا هنا نحاول الوصول إلى العنصر الخامس في المصفوفة تذكر أن ترقيم العناصر يبدا من الصفر وهو ما سيولد الاعتراض System.IndexOutOfRangeException.

وفي هذه المرة هناك كتلة catch مطابقة في التابع Main() وبالخطو ضمن الشيفرة سننتقل إلى هذه الكتلة التي تبدأ بالسطر رقم 66.

يقوم التابع Console.WriteLine() في هذه الكتلة بطباعة رسالة مخزنة ضمن الاعتراض وذلك باستخدام القيمة e.Message يمكننا التعامل مع الاعتراض الحاصل عبر بارامتر الكتلة catch وبتخطي هذه الشيفرة باستخدام Step Into سنجد أنفسنا ضمن الكتلة finally وليس ضمن كتلة catch الثانية وذلك لأنه قد تمت معالجة الاعتراض في كتلة catch ومن ثم العودة إلى دورة الحلقة في Main() من جديد واستدعاء التابع ThrowException() مجددا في السطر رقم 60.

وعند الوصول إلى بنية switch في التابع ThrowException() فإننا سندخل هذه المرة في كتلة try جديدة بدء بالسطر رقم 33 وعند الوصول إلى السطر رقم 38 سنقوم باستدعاء تعاودي ذاتي للتابع ThrowException() أيضا وهذه المرة بالبارامتر index وإذا أحببت يمكنك استخدام الزر Step Over لتخطي أسطر الشيفرة التي ستنفذ هنا باعتبار أننا اطلعنا عليها وكما رأينا مسبقا فإن هذا الاستدعاء سيولد اعتراضا من النوع System.IndexOutOfRangeException وهذه المرة سيتم معالجة هذا الاعتراض باستخدام البنية try..catch..finally الموجودة ضمن التابع ThrowException() وليس لهذه البنية تطابق صريح من نوع الاعتراض الحاصل وبالتالي سيتم تنفيذ كتلة catch العامة بدء بالسطر رقم 72 وسيستمر التنفيذ ضمن الكتلة catch إلى الكتلة finally أيضا إلى الوصول لنهاية التابع المستدعى.

هناك أمر مهم هنا فعلى الرغم من أن الاعتراض قد رمى في التابع Main() إلا أنه قد تمت معالجته ضمن التابع ThrowException() وهذا يعني أنه ليس هناك أية اعتراضات لمعالجتها في التابع Main() لذا سينتقل التنفيذ مباشرة إلى الكتلة finally ثم سينتهي تنفيذ التطبيق بعد ذلك.

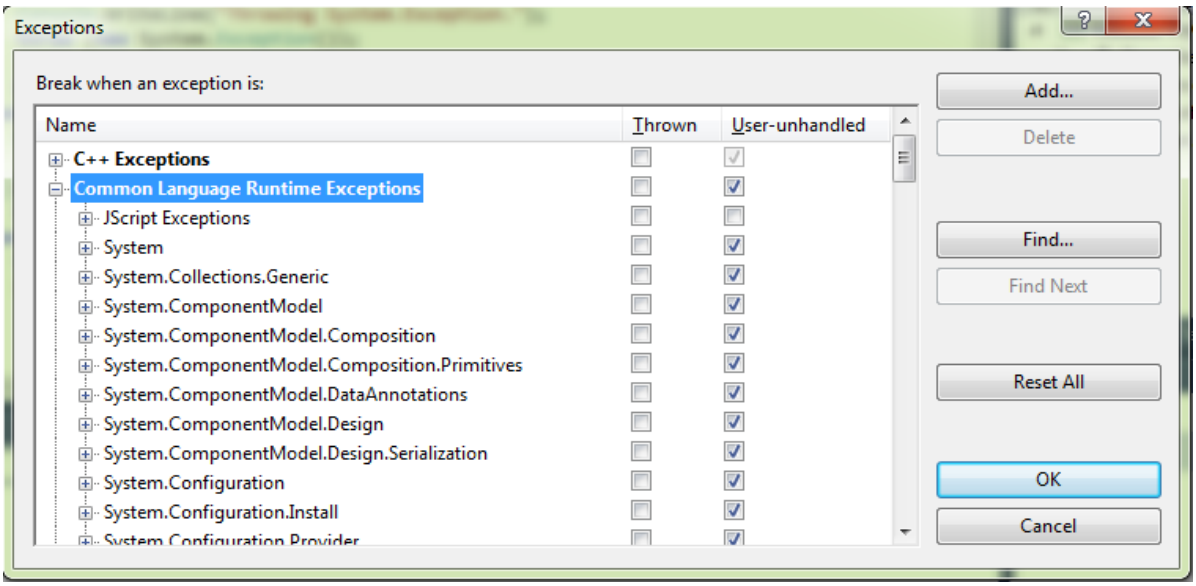
ملاحظة:

لاحظ أنه عند تنفيذ هذه الشيفرة سيظهر لنا بند ضمن قائمة المهام في إطار Task List يشير إلى أن هناك تعليمة ضمن السطر لا يمكن الوصول إليها تلك التعليمة هي break وهذا منطقي باعتبار ان الاعتراض سيرمى باستخدام الأمر throw والذي سيؤدي إلى كتلة catch أو finally مباشرة.

سرد وإعداد الاعتراضات:

Listing and Configuring Exceptions:

يتضمن إطار عمل NET. عددا كبيرا من أنواع الاعتراضات ويمكننا رمي ومعالجة أي من هذه الاعتراضات بحرية ضمن شيفرتنا البرمجية يقدم Visual Studio 2013 صندوق حوار لتفحص وتحرير الاعتراضات المتوفرة ويمكننا عرض صندوق الحوار هذا باختيار Exceptions من القائمة Debug أو بالضغط على المفاتيح Ctrl+Alt+E.



الشكل (7-23)

تصنف الاعتراضات ضمن هذه النافذة وفقا لنوعها وفضاء أسماء مكتبة .NET. ويمكننا أن نرى الاعتراضات الموجودة في فضاء الأسماء System بتوسعة البند Common Language Runtime Exceptions ومن ثم اختيار البند System عندئذ ستعرض لائحة من الاعتراضات تتضمن الاعتراض IndexOutOfException الذي استخدمناه مسبقا.

يمكننا اعداد الاعتراضات باستخدام زر الخيار في أسفل النافذة فمعظم هذه الاعتراضات معدة بالخيار Use parent setting بصورة افتراضية وهذا يعني أنها تستخدم خيارات مستوى الصنف الذي تنتمي إليه يمكننا أن نستخدم مجموعات الخيارات الأولى وهو When the exception is thrown لتحديد كيف سيتم التصرف عند رمي الاعتراض ومجموعة الخيارات الثانية وهي When the exception is not handled والتي تمكننا من تحديد كيف سيتم التصرف عند رمي الاعتراض غير المعالج وذلك إما باستمرار التنفيذ أو بمقاطعة التنفيذ للتنقيح وفي معظم الحالات تكون الإعدادات الافتراضية مناسبة.

ملاحظات حول معالجة الاعتراضات:

Notes on Exception Handling:

لاحظ أن علينا توفير كتل catch خاصة دائما لأنواع محددة من الاعتراضات بدلا من استخدام كتلة catch العامة فغذا حصل خطأ ما دون معالجته فغن هذا سيسبب في إخفاق التطبيق عن الاستمرار بالتنفيذ.

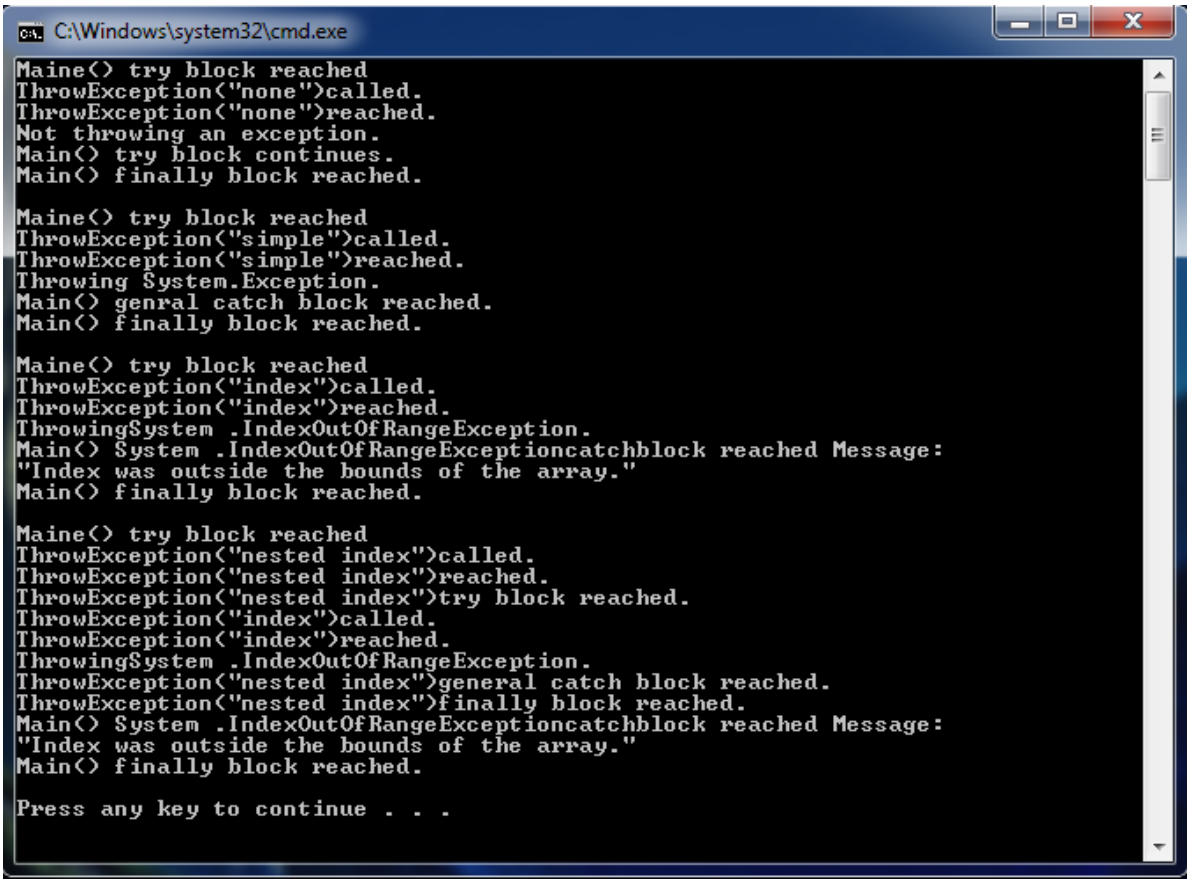
لاحظ أيضا أن بإمكاننا رمي الاعتراضات ضمن كتل catch أيضا إما باستخدام الطرق التي تعلمناها مسبقا أو بمجرد كتابة throw.

وإذا رمينا الاعتراضات بهذه الطريقة فإننا لن نتمكن من معالجتها ضمن بنية try..catch..finally إلا أنه يمكننا ذلك بواسطة الشيفرة الرئيسية لكن تذكر أن شيفرة كتلة finally ضمن البنية المعششة ستنفذ.

على سبيل المثال إذا غيرنا الكتلة try..catch..finally في التابع ThrowException() كما يلي:

```
try           //line33
{
    Console.WriteLine("ThrowException(\"nested index\")" +
        "try block reached.");
    Console.WriteLine("ThrowException(\"index\")called.");
    ThrowException("index");   //line38
}
catch      //line40
{
    Console.WriteLine("ThrowException(\"nested index\")" +
        "general catch block reached.");
    throw;
}
finally
{
    Console.WriteLine("ThrowException(\"nested index\")" +
        "finally block reached.");
}
```

عندئذ سيستمر التنفيذ في بادئ الأمر للكتلة finally هنا ثم مع الكتلة catch المطابقة الموجودة في التابع Main() والشكل التالي يبين تغير النص المطبوع على الخرج:



الشكل (7-24)

نلاحظ في هذه الشاشة أن هناك بعض الأسطر الإضافية في الخرج ناتجة من التابع Main() وذلك لأن الاعتراض System.IndexOutOfRangeException قد تم اصطياده caught.

مصطلح جديد:

رمي thrown واصطياد caught يمثل المصطلح thrown حدث حصول اعتراض ما أما caught فيمثل حدث معالجة هذا الاعتراض.

الخلاصة:

Summary:

لقد ركزنا في هذا الفصل على التقنيات التي يمكنك استخدامها لتنقيح تطبيقاتك هناك العديد من التقنيات التي تحدثنا عنها في هذا الفصل قد تحتاج إلى استخدام معظمها في نوع من التطبيقات التي ستقوم ببنائها وليس تطبيقات Console فقط.

لقد تناولنا في هذا الفصل:

- استخدام الأمر Debug.WriteLine() و Trace.WriteLine() لكتابة نصوص في إطار Output.
- نمط المقاطعة وكيفية الدخول فيه وما يتضمن ذلك من استخدام لنقاط المقاطعة.
- إطار معلومات التنقيح في Visual Studio 2013.
- الخطو ضمن الشيفرة.
- معالجة الاعتراضات باستخدام البنية try..catch..finally.

لقد غطينا حتى الآن كل شيء نحتاجه لإنشاء تطبيقات Console بسيطة بالإضافة إلى كيفية تنقيح هذه التطبيقات ومعالجة الاعتراضات الحاصلة فيها سوف نتعرف في الجزء التالي من هذا الكتاب على تقنية البرمجة كائنية التوجه القوية.