الجزء الثاني

الإبحار في لغة C#.NET

Visual Studio 2013

البرمجة كائنية التوجه "OOP"

إعداد : المهندس حسام الدين الرز

الجزء الثاني

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..

إلى الحبيبة الغالية سوريا..

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

إلى أمي التي أعجز عن شكرها..

إلى والدي الذي حملني مسؤلية الحياة مبكراً..

إلى إخوتي وأخواتي..

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

إلى من تشاطرني هموم الحياة و متاعبها زوجتي الغالية..

حسام الدين الرز

في حال وجود أي استفسار يرجى التواصل على الإيميل:

freesyria.syria123@gmail.com

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 بما 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 وJava 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 وC++ 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 وC++ 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 الكبسلةcapsulation والتوارث Inheritance وتعدد الأوجه Polymorphism 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.

الجزء الثاني

البرمجة كائنية التوجه

"OOP"

الفصل الأول

مدخل إلى البرمجة كائنية التوجه

لقد تناولنا في الجزء الأول أساسيات البرمجة في C# وصيغها ويمكننا بعد قراءة الجزء الأول كتابة تطبيقات Console مفيدة لكن كي نتمكن من الوصول إلى القوة الحقيقية للبرمجة بلغة C# وإطار عمل .NET. فإن علينا أن نستخدم تقنيات البرمجة كائنية التوجه Object-Oriented Programming أو اختصارا OOP وفي الحقيقة لقد استخدمنا هذه التقنيات مسبقا في أمثلتنا إلا أننا لم نشر إلى ذلك أثناء شرحنا للأمثلة بهدف تبسيط الأمور في البداية.

سوف نتجنب في هذا الفصل الشيفرات البرمجية وسنركز على مبادئ البرمجة كائنية التوجه وهو ما سيقودنا إلى كتابة شيفرة بلغة C# بعد ذلك باعتبار أن علاقتها بالبرمجة كائنية التوجه وثيقة جدا سوف نتكلم عن جميع المبادئ التي سنتناولها في هذا الفصل بتفصيل أكبر في الفصول القادمة ومع الكثير من التطبيقات والأمثلة لذا لا داعي لأن تقلق إذا لم تستطع استيعاب كافة الأمور في هذا الفصل من الوهلة الأولى.

وكبداية فإننا سنتحدث عن أساسيات البرمجة كائنية التوجه ويتضمن ذلك الإجابة على أسئلة جوهرية مثل "ما هو الكائن؟" سوف تكتشف وجود العديد من المصطلحات المرتبطة بالبرمجة كائنية التوجه التي قد تظهر مربكة نوعا ما في البداية إلا أنك ستجد الكثير من الشرح الوافي لها. ستجد أيضا أن استخدام البرمجة كائنية التوجه يتطلب منا رؤية مختلفة للبرمجة.

وبالإضافة إلى مناقشة المبادئ الأساسية للبرمجة كائنية التوجه فإننا سنتناول إحدى المناطق التي يكون فيها فهم OOP أمرا جوهريا وهي تطبيقات Windows Forms يوفر هذا النوع من التطبيقات رؤية واضحة للبرمجة كائنية التوجه حيث سنستعرض النقاط الأساسية في OOP ضمن بيئة تطبيقات Windows Forms.

توضيح:

نقصد بتطبيقات Windows Forms تلك التطبيقات التي تستفيد من المزايا التي يقدمها نظام Windows وبيئته الرسومية من قوائم وأشرطة أدوات...الخ.

ملاحظة:

لاحظ أن OOP المقدمة في هذا الفصل هي OOP.NET حقيقية وهذا يعني أن بعضا من التقنيات المعروضة في هذا الفصل لا تنطبق على بيئات OOP الأخرى إن التنويه عن ذلك سيبسط علينا الأمور لاحقا في هذا الفصل خصوصا إذا كنت تبرمج بلغة برمجة أخرى.

ما هي البرمجة كائنية التوجه؟

What is Object Oriented Programming?

تعتبر البرمجة كائنية التوجه أسلوبا جديدا في إنشاء تطبيقات الحاسب والذي يتجاوز العديد من المشاكل التي قد نواجهها باستخدام أسلوب البرمجة التقليدية إن نوع البرمجة الذي تعرفنا عليه حتى الآن يسمى بالبرمجة الإجرائية procedural programming والذي ينتج عنه تطبيقات موحدة وهذا يعني أن جميع وظائف التطبيق موجودة ضمن عدد من الوحدات البرمجية وفي الغالب لن يكون هناك سوى وحدة برمجية واحدة أما في OOP فإن الوضع مختلف فباستخدام OOP يمكننا استخدام العديد من وحدات الشيفرة بحيث تقدم وحدة وظائفها الخاصة وبحيث تكون هذه الوحدة مستقلة بصورة كاملة أو جزئية عن الوحدات الأخرى يوفر لنا هذا النهج المعياري للبرمجة جوانب برمجية عديدة بالإضافة إلى توفير الفرصة لإعادة استخدام الشيفرة في أكثر من تطبيق.

ولكي نصف ذلك بصورة أدق تخيل تطبيقا ما ذا أداء عال في جهازك كسيارة السباق إن تطبيقنا الذي استخدمنا فيه تقنيات البرمجة التقليدية مشابه لسيارة السباق المكونة من وحدة واحدة فإذا أردنا أن نحسن من هذه السيارة فإن علينا إما استبدالها بسيارة أخرى جديدة وذلك بإرسالها إلى المصنع لكي يتمكن خبراء ميكانيك السيارات من تحديثها وإما أن نشتري واحدة جديدة أما إذا كنا قد استخدمنا تقنيات OOP في هذا التطبيق فإن ذلك مشابه لمجرد شراء محرك جديد من المصنع وإتباع تعليمات استبدال المحرك القديم بهذا الجديد.

يكون سير التنفيذ في التطبيقات التقليدية عادة بسيطا وخطيا حيث يتم تعديل التطبيق إلى الذاكرة ومن ثم بدء التنفيذ من النقطة A وإنهاء التنفيذ عند النقطة B ومن ثم إزالة التطبيق من الذاكرة وخلال هذا الطريق هناك العديد من الأشياء المستخدمة مثل الملفات ووسائط التخزين أو إمكانية بطاقة العرض لكن الجسم الرئيسي للمعالجة يتم في مكان واحد وخلال هذه الشيفرة سنجد العديد من عمليات معالجة البيانات وفقا لمبادئ رياضية ومنطقية وطرق معالجة البيانات تكون بسيطة وتستخدم أنواع بيانات بسيطة أيضا مثل القيم العددية والمنطقية لبناء تمثيلات للبيانات أكثر تعقيدا.

إن الأمور نادرا ما تسير في OOP بصورة خطية فعلى الرغم من أن تقنية OOP تحقق النتائج نفسها التي تحققها التقنيات التقليدية إلا أن تقنيات OOP معدة وفقا لبنية البيانات ومعانيها بالإضافة إلى التفاعل بين البيانات والبيانات الأخرى يبدو ذلك أن استخدام تقنيات OOP يتطلب جهدا أكبر في تصميم التطبيقات في الحقيقة إن هذا صحيح إلا أننا سنستفيد عندها من عدة مزايا كمجالات التوسع والتطوير المستقبلية

فمتى استطعنا إنشاء نوع وأسلوب معين لاستخدامه في تطبيق ما فإن بإمكاننا استخدام نفس النوع والأسلوب في أي تطبيق أخر.

يسمى هذا النوع في تقنية OOP بالكائن Object إذا ما هو الكائن؟

ما هو الكائن؟

What is Object?

الكائن Object عبارة عن كتلة برمجية لتطبيق OOP تغطي هذه الكتلة جزء من التطبيق حيث يمكن لهذا الجزء أن يعالج كمية من البيانات سواء كانت هذه البيانات محددة أو مجردة.

يمثل الكائن في أبسط صوره البنية struct التي سبق وتحدثنا عنها في الفصول السابقة والتي تتضمن أعضاء من متحولات وتوابع يمكن ان تمثل المتحولات الموجودة ضمن هذه البنية. البيانات التي يحتفظ بها الكائن. وأما التوابع فهي تعطينا إمكانية الوصول إلى وظائف هذا الكائن. هناك كائنات أكثر تعقيدا حيث لا تتضمن اية بيانات وإنما يمكنها أن تحتوي على توابع فقط على سبيل المثال يمكننا ان نستخدم كائنا يمثل تخاطب التطبيق مع الطابعة وبالتالي يمكن لهذا الكائن أن يحتوي على توابع تتحكم بصورة كاملة في الطابعة كطباعة مستند ما أو طباعة ورقة تجريبية...الخ.

يتم إنشاء الكائنات في C# من الأنواع تماما كما نقوم بإنشاء المتحولات يطلق على نوع الكائن في OOP بالصنف (الفئة) class يمكننا أن نستخدم تعريفات الصنف لتهيئة الكائنات وهذا يعني إنشاء حالة instance محددة وحقيقية من هذا الصنف.

ملاحظة:

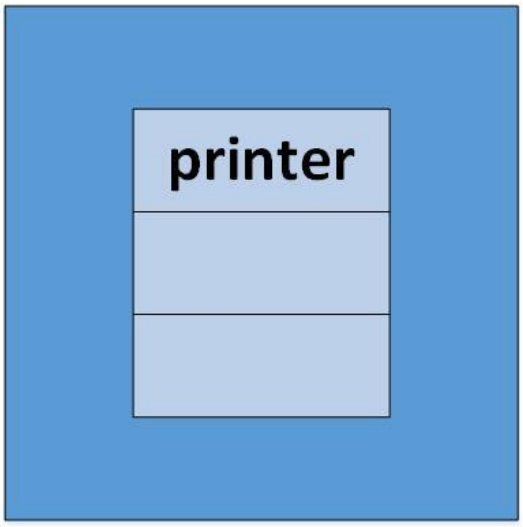
لاحظ ان تعبير كائن وحالة من الصنف يمثلان الشيء ذاته لكن لاحظ ان تعبير كائن وصنف هما شيئان مختلفان فالكائن يتبع صنفا محددا تماما كما تتبع المتحولات لأنواعها.

سوف نصور الأصناف والكائنات في هذا الفصل باستخدام صيغة Universal Modeling Language (UML) فلغة UML هي لغة مصممة لتشكيل التطبيقات من الكائنات التي تكونها ووصولا للعمليات التي تؤديها واستخدام الحالات المتوقعة لهذه الكائنات سوف نستخدم هنا أساسيات هذه اللغة وسنركز على OOP دون الخوض في تفاصيل أكثر تعقيدا.

ملاحظة:

لقد تم تصميم الأشكال في هذا الفصل بواسطة Microsoft Visio.

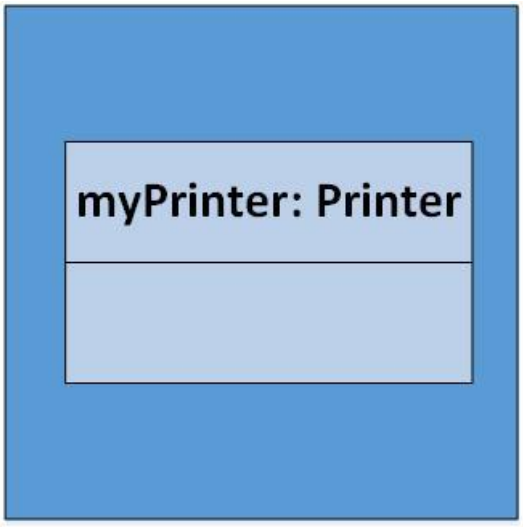
يمثل الشكل (1-1) تمثيلا بلغة UML لصنف الطابعة وهو باسم Printer:



الشكل (1-1)

نلاحظ أن اسم الصنف ظاهر في القسم العلوي من المخطط سنهتم بالأقسام السفلى لاحقا.

وتمثيل UML التالي يستعرض حالة للصنف Printer حيث لهذه الحالة الاسم myPrinter:



الشكل (1-2)

نلاحظ هنا أن اسم الحالة ظاهر في القسم العلوي من المخطط متبوعا باسم الصنف حيث تم فصل كلا الاسمين باستخدام الرمز ":".

الخصائص والحقول:

Properties and Fields:

توفر الخصائص والحقول إمكانية الوصول على البيانات الموجودة ضمن الكائن إن بيانات الكائن هي ما يميز كل كائن عن الآخر أي أنه يمكن أن يكون لدينا عدة كائنات من صنف واحد بحيث تختلف هذه الكائنات عن بعضها بالقيم المخزنة ضمن خصائصها وحقولها.

إن مختلف أجزاء البيانات الموجودة ضمن الكائن تمثل حالة state ذلك الكائن.

لنفترض أننا نود تمثيل فنجان من القهوة باستخدام مفاهيم البرمجة كائنية التوجه سنحتاج في هذه الحالة على كائن ليمثل فنجان القهوة ويجب ان يتبع هذا الكائن لصنف يعرف جميع كائنات فناجين القهوة سنسمي هذا الصنف باسم CupOfCoffee وعندما نقوم بإنشاء كائن من هذا الصنف علينا أن نزود هذا الكائن بحالة معينة لكي يكون لهذا الكائن مدلول ومعنى سنستخدم هنا الخصائص والحقول الجديدة لتحديد نوع القهوة المستخدمة مثلا سواء كانت قهوة بحليب أو بسكر أو أنها نسكافيه ...الخ عندئذ سيكون لكائن فنجان القهوة حالة معينة مثلا فنجان قهوة بن برازيلي بالحليب وبقطعتين من السكر تمثل الخصائص والحقول أماكن تخزينية لذا يمكننا ان نحفظ ضمنها معلومات كالمتحولات النصية من نوع String أو قيما من نوع int أو أي نوع آخر من أنواع البيانات ويمكن أن تمثل هذه المعلومات كائنات تابعة لأصناف أخرى ولكن تختلف الخصائص عن الحقول في طريقة قراءة وإسناد المعلومات إليها فالحقول كالمتحولات تمكننا من الوصول إلى قيمها وتعديلها بصورة مباشرة أما الخصائص فهي توفر لنا ألية معينة تمكننا من القيام بأمور محددة عند قراءة أو إسناد القيم إليها فإذا استخدمنا حقلا لحفظ عدد قطع السكر في كائن CupOfCoffee فإن المستخدمين يمكن أن يضعوا أي قيمة يرغبون بها ضمن هذا الحقل وذلك بحسب نوع الحقل طبعا أما إذا استخدمنا خاصية فإن بإمكاننا التقيد بمجال محدد لقيمة هذه الخاصية لنفرض مثلا قيمته ما بين 0 و2 فقط.

وبصورة عامة يفضل استخدام الخصائص بدلا من الحقول عند الوصول إلى حالة الكائن باعتبار ان لدينا تحكما اكبر بالقيم المخزنة ضمنها يمكن لخصائص معينة ان تكون قابلة للقراءة فقط read-only أي أن بإمكاننا قراءة محتوياتها لكن لا يمكننا تعديل قيمتها على الأقل ليس بصورة مباشرة ويعد ذلك تقنية مفيدة إذا أردنا قراءة أجزاء عديدة من حالة الكائن في آن واحد قد يكون لدينا خاصية للقراءة فقط مفيدة إذا أردنا قراءة أجزاء عديدة من حالة الكائن في آن واحد قد يكون لدينا خاصية للقراءة فقط ضمن الصنف CupOfCoffee تسمى بـ Description حيث تعيد هذه الخاصية سلسلة نصية تمثل حالة الكائن في هذا الصنف كوصف لفنجان القهوة وذلك عند الطلب يمكننا ان نحصل على النتيجة نفسها بتجميع البيانات الموجودة في عدة خصائص إلا أن هذه الخاصية توفر علينا الوقت والجهد والشيفرة الزائدة ويمكن أن تكون هناك خصائص للكتابة فقط write-only وهي تعمل بصورة مشابهة لخصائص القراءة فقط.

وبالإضافة إلى ميزة القراءة/الكتابة في الوصول إلى الخصائص فإنه يمكننا ان نحدد أسلوبا مختلفا للوصول إلى الحقول والخصائص أيضا تحدد هذه الوصولية ماهي أجزاء الكائن التي يمكننا الوصول إليها في

شيفرتنا البرمجية أي إما أن تكون هذه الأجزاء متوفرة لكل الشيفرة البرمجية أي أنها عامة أو أن تكون هذه الأجزاء مقتصرة على شيفرة الصنف فقط أي أنها خاصة يمكننا مثلا إنشاء حقول خاصة private وتوفير الوصول إليها من خلال خصائص عامة public مثلا هذا يعني أن الشيفرة الموجودة ضمن الصنف يمكن أن تصل إلى هذه الحقول بصورة مباشرة أما الخاصية العامة فإنها وسيلة التخاطب مع الشيفرة خارج الصنف والواقي من تعديل هذه البيانات بصورة خاطئة تعرف الأعضاء العامة في الصنف بأنها الأعضاء المكشوفة منه.

إن ذلك مشابه لمبدأ مدى المتحولات فالحقول والخصائص الخاصة هي أعضاء محلية بالنسبة للكائن بينما مدى الحقول والخصائص العامة يشتمل على الشيفرة خارج الكائن.

سنستخدم القسم الثاني من تمثيل UML للصنف لعرض الخصائص والحقول وذلك كما يلي:

الشكل (3-1)

ذلك يمثل تمثيل الصنف CupOfCoffee حيث يتضمن على خمسة أعضاء خصائص أو حقول حيث لم نحدد ذلك في الشكل (3-1) معرفة كما شرحنا ذلك مسبقا ولكل عضو من هذه الأعضاء المعلومات التالية:

- ✓ الوصولية: يستخدم الرمز+ للأعضاء العامة أما الرمز- فيستخدم للأعضاء الخاصة وبصورة عامة فإنني لن أعرض الأعضاء الخاصة ضمن الأشكال في هذا الفصل وذلك باعتبار أن هذه المعلومات هي معلومات داخلية للصنف.
- ✓ اسم العضو.
- ✓ نوع العضو.

وقد استخدم الرمز ":"لفصل اسم العضو عن النوع.

المناهج:

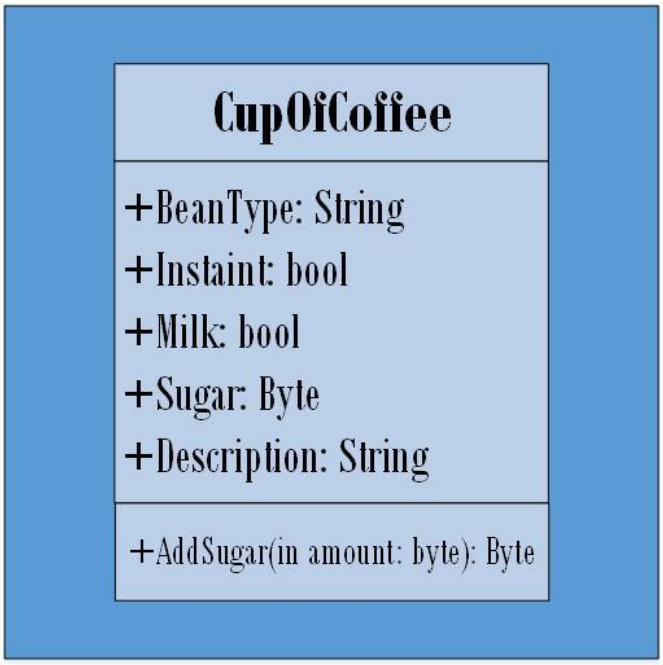
Methods:

يشير مصطلح المنهج method إلى التوابع التي يحتويها الصنف يمكننا أن نستدعيها بنفس الصورة التي نستدعي فيها أي تابع آخر ويمكننا ان نستخدم القيم المعادة والبارامترات بنفس الصورة أيضا عد إلى الفصل السادس من الجزء الأول لمزيد من التفاصيل حول ذلك.

ملاحظة:

إن جميع التوابع التي أنشأناها في الفصول السابقة بالإضافة إلى التابع Main() عبارة عن مناهج وتتبع هذه المناهج إلى الصنف الذي يمثل التطبيق.

تستخدم المناهج للوصول إلى وظائف الكائنات وكما في الخصائص والحقول فإن المناهج أيضا يمكن ان تكون عامة او خاصة بحيث يقتصر استخدامها ضمن الكائن فقط أي لا يمكننا استدعائها من خارج شيفرة الصنف أو يمكن أن تستخدم من قبل شيفرة خارجية وفي أغلب الأحيان تعتمد هذه المناهج على حالة الكائن (قيم خصائصه وحقوله) ومن الجدير بالذكر التنويه إلى أن لهذه المناهج إمكانية الوصول إلى الأعضاء الخاصة ضمن الكائن على سبيل المثال يمكن لصنفنا CupOfCoffee ان يعرف منهجا باسم AddSugar() بحيث يقوم بزيادة قيمة حقل عدد قطع السكر بمقدار معين (حتى إن كان هذا الحقل خاصا) وباستخدام تمثيل UML يمكننا ان نعرض المناهج بالصورة التالية:



الشكل (4-1)

إن الصيغة هنا مشابهة لنفس صيغة الحقول والخصائص فيما عدا أن النوع الذي يظهر في النهاية يمثل نوع القيمة المعادة بالإضافة إلى وجود لائحة من البارامترات.

ملاحظة:

يتم عرض كل بارامتر في UML بواسطة واحد من المحددات التالية: in أو out أو inout وتستخدم هذه المحددات للإشارة إلى اتجاه سير البيانات حيث أن المحددين inout وout يوافقان كلمتي C# المقابلتين لهما وهما ref وout على الترتيب أما الكلمة in فتوافق سلوك لغة C# عندما لا نضع أي من هاتين الكلمتين.

كل شيء عبارة عن كائن:

Everything's an Object:

لقد حان الوقت الآن كي نوضح لك الأمور أكثر في الحقيقة إن كل شيء في لغة C# وإطار عمل .NET هو عبارة عن كائن فالتابع Main () الموجود ضمن تطبيق Console يمثل منهجا لصنف وكل نوع متحول رأيناه هو عبارة عن صنف إن كل أمر استخدمناه كان يمثل إما خاصية أو منهجا مثل الخاصية <String>. Length والمنهج <String>.ToUpper() ...الخ إن النقطة "."في الامر هنا تفصل اسم الكائن عن اسم الخاصية أو المنهج.

ستجد الكائنات في كل مكان أثناء البرمجة بلغة C# وصيغة استخدامها بسيطة للغاية في أغلب الأحيان من الآن وصاعدا سنركز على الكائنات بتفصيل أكبر ليبقى في بالك ان المفاهيم التي سنتحدث عنها هنا ومهما كانت صعوبة فهمها بالنسبة إليك إلا أنها مطبقة على أبسط العناصر في C# مثل النوع int الذي استخدمناه بكثرة ولحسن الحظ يمكننا ان نستخدم هذه النقطة لصالحنا في محاولة لفهم هذه المفاهيم الأساسية.

دورة حياة الكائن:

The Lifecycle of an Object:

لكل كائن دورة حياة تبدأ بالتصريح عن الكائن وتنتهي بتدمير هذا الكائن وبالإضافة إلى مرحلة قيد الاستخدام فإن هناك مرحلتان رئيسيتان في دورة حياة الكائن:

- مرحلة البناء (construction): وهي المرحلة التي يتم تهيئة الكائن فيها وتسمى هذه التهيئة بالبناء وتتم ضمن منهج بناء الكائن.
- مرحلة التدمير (destruction): فعند عدم حاجتنا لاستخدام الكائن فإننا سنقوم ببعض المهام كتحرير جزء من الذاكرة مثلا وتلك مهمة منهج التدمير.

مناهج البناء:

Constructors:

تتم تهيئة الكائن بصورة تلقائية فنحن لن نخوض في تفاصيل وضع كائن جديد ضمن الذاكرة مثلا إلا أننا قد نحتاج عادة إلى القيام بمهام إضافية خلال مرحلة تهيئة الكائن على سبيل المثال يمكننا أن نضع قيما افتراضية ضمن خصائص هذا الكائن أثناء بناءه.

لجميع الكائنات تابع بناء افتراضي (default constructor) وهو منهج لا يتضمن أية بارامترات ويأخذ نفس اسم الصنف ويمكن أن يتضمن تعريف الصنف العديد من مناهج البناء ويمكن أن يكون لهذه المناهج تواقيع مختلفة يمكن استخدامها بواسطة الشيفرة الخارجية لتهيئة وإنشاء كائن من هذا الصنف.

تستخدم المناهج التي تتطلب بارامترات عادة لتزويد الكائن بقيم افتراضية للبيانات التي ستخزن ضمنه.

يتم استدعاء مناهج البناء في C# باستخدام الكلمة new على سبيل المثال يمكننا أن ننشئ كائنا من نوع String باستخدام الأسلوب التالي:

```
string myString = new string();
```

توضيح:

في السابق كنا نقول (متحول من نوع String) وأما الآن فإننا نقول (كائن من نوع String) والسبب في ذلك أن الأنواع عبارة عن أصناف.

ويمكننا إنشاء الكائنات باستخدام منهج بناء غير المنهج الافتراضي وكما في منهج البناء الافتراضي فإن مناهج البناء غير الافتراضية لها نفس اسم الصنف إلا انها تتضمن بارامترات وتستخدم بنفس صيغة استخدام منهج البناء الافتراضي كما يلي:

```
string myString = new string('a',10);
```

سيقوم منهج البناء هذا بإنشاء سلسلة نصية جديدة تأخذ القيمة "aaaaaaaaaa".

ويمكن لمناهج البناء كما في باقي مناهج الكائن الأخرى أن تكون عامة أو خاصة فلا يمكن لشيفرة خارجية أن تقوم بإنشاء كائن باستخدام منهج بناء خاص وإنما يجب أن تستخدم منهج بناء عام وبهذه الطريقة فإننا نخبر مستخدمي أصنافنا على استخدام منهج بناء غير المنهج الافتراضي.

قد لا يكون لبعض الأصناف مناهج بناء عامة مما يعني انه من المستحيل لشيفرة خارجية أن تقوم بإنشاء حالة من هذه الأصناف لكن لاحظ ان ذلك لا يعني ان تكون هذه الأصناف عديمة النفع كما سنرى لاحقا.

مناهج التدمير:

Destructors:

تستخدم مناهج التدمير من قبل .NET لعمليات التنظيف بعد إنهاء العمل بالكائن وبصورة عامة فإننا لا نقدم شيفرة لمنهج التدمير وإنما يستخدم منهج افتراضي لذلك ولكن يمكن أن نستخدم تعليمات محددة لأي شيء مهم نحتاج للقيام به قبل حذف الكائن.

ملاحظة:

من المهم ان تتعلم أن منهج تدمير الكائن لا يتم استدعاؤه عندما نتوقف عن استخدام الكائن.

عندما يخرج المتحول عن المدى فإننا لن نتمكن من الوصول إليه في الشيفرة الحالية إلا أنه لا يزال موجودا في مكان ما من ذاكرة الحاسوب ولن تتم إزالته نهائيا إلا عندما يقوم نظام CLR في .NET بتجميع نفايات تطبيقنا من الذاكرة عند تدمير الكائن نهائيا.

ملاحظة:

هذا يعني أن علينا ألا نعتمد على منهج التدمير لتحرير المصادر التي يشغلها الكائن فإن استهلك هذا الكائن مصادر النظام بصورة كبيرة فإن هذه المسألة تصبح حرجة بالنسبة إلينا على كل حال هناك حل لهذه المشكلة وسوف نتحدث عن كيفية التخلص من الكائنات لاحقا في هذا الفصل.

أعضاء الصنف الستاتيكية:

Static Class Members:

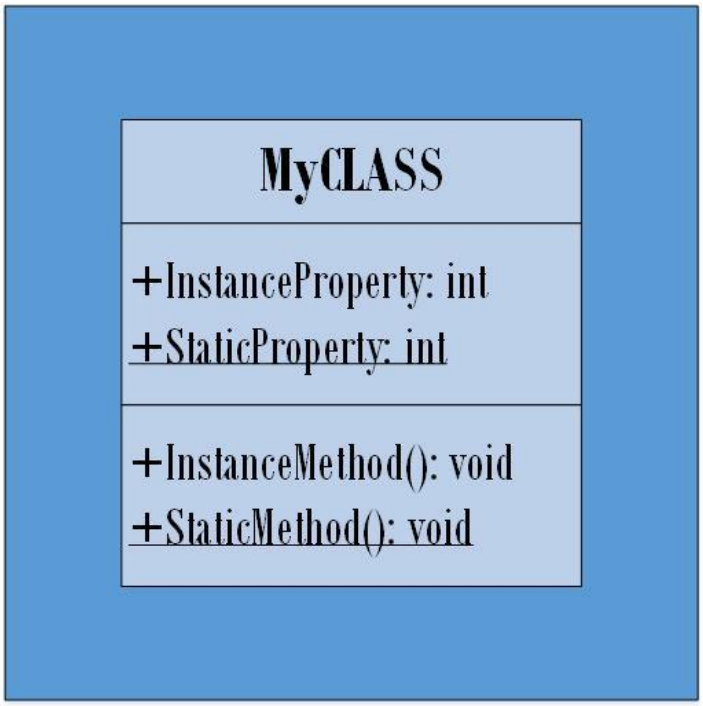
وبالإضافة إلى إمكانية ان يكون لدينا أعضاء كالخصائص والمناهج والحقول المخصصة لحالات محددة من الكائنات إلا أنه من الممكن أيضا أن يكون لدينا أعضاء ستاتيكية (static) وتسمى أيضا بالأعضاء المشتركة (shared) والتي يمكن ان تكون مناهج أو خصائص او حقول إن الأعضاء الستاتيكية مشتركة بين جميع الكائنات التي تنتمي إلى صنف واحد وبالتالي يمكننا ان نتأملها وكأنها أعضاء عامة لكائنات صنف ما تسمح لنا الخصائص والحقول الستاتيكية بالوصول إلى البيانات المستقلة عن اية حالة من الكائنات تمكننا المناهج الستاتيكية من تنفيذ الأوامر المرتبطة بنوع الكائن وليس بحالة محددة من الصنف وبالتالي إذا أردنا استخدام الأعضاء الستاتيكية فإننا لسنا بحاجة إلى إنشاء حالة من كائنات الصنف الذي يحوي هذه الأعضاء.

على سبيل المثال يمثل المنهجان Console.WriteLine() وConvert.ToString() مناهج ستاتيكية ونحن لسنا بحاجة هنا لإنشاء حالة من الأصناف Console أو Convert كي نتمكن من استخدام هذه

المناهج وإن حاولنا ذلك فإن هذا سيؤدي إلى حدوث خطأ في الترجمة وذلك لأن مناهج بناء هذه الأصناف خاصة (ليست عامة) كما شرحنا ذلك مسبقا.

هناك العديد من الأوضاع التي يمكننا استخدام المناهج والخصائص الستاتيكية فيها.

يمكننا أن نمثل الأعضاء الستاتيكية في الصنف بصيغة UML كما في الشكل التالي:



الشكل (5-1)

تقنيات البرمجة كائنية التوجه:

OOP Techniques:

لقد تناولنا حتى الآن الأساسيات وعرفنا ماهي الكائنات وكيف تعمل ولقد حان الوقت لكي نلقي نظرة على بعض المزايا الأخرى للكائنات سوف نتحدث في هذا القسم عن:

- الواجهات.
- الوراثة.
- تعددية الأشكال.
- العلاقات بين الكائنات.
- التحميل الزائد للعوامل.
- الاحداث.

الواجهات:

Interfaces:

تعرف الواجهة interface على أنها مجموعة من المناهج والخصائص العامة المطلقة مجمعة مع بعضها البعض بحيث تغلف وظيفة معينة ومتى عرفنا واجهة ما فإن بإمكاننا استخدامها ضمن صنف ما هذا يعني أن الصنف سيزود بجميع خصائص ومناهج الواجهة.

لاحظ أنه لا يمكن ان تتواجد الواجهات لوحدها فنحن لا يمكننا إنشاء حالة من الواجهة كما يمكننا ذلك مع الأصناف وبالإضافة إلى ذلك فإن الواجهات لا يمكن أن تتضمن أية شيفرة برمجية ضمن أعضائها فهي تحوي تعاريف الأعضاء فقط وأما كتابة شيفرة هذه الأعضاء فيتم عن طريق الصنف المزود بهذه الواجهة.

بالعودة إلى مثال فنجان القهوة السابق يمكننا على سبيل المثال أن نجمع الخصائص والمناهج ذات الأغراض العامة مثل (AddSugar() و Milk و Sugar) ضمن واجهة ولتكن واجهة المشروبات الساخنة IHotDrink تبدأ أسماء الواجهات عادة بالحرف I) يمكننا استخدام هذه الواجهة مع كائنات أخرى ربما استخدامها مع الصنف CupOfTea وبالتالي يمكننا أن نعامل هذه الكائنات بنفس الطريقة ويمكن لهذه الكائنات أن تكون لها خصائصها ومناهجها الخاصة أيضا (مثل خاصية نوع حبوب القهوة BeanType لصنف فنجان القهوة CupOfCoffee أو خاصية نوع ورق الشاي LeafType لصنف الشاي CupOfTea).

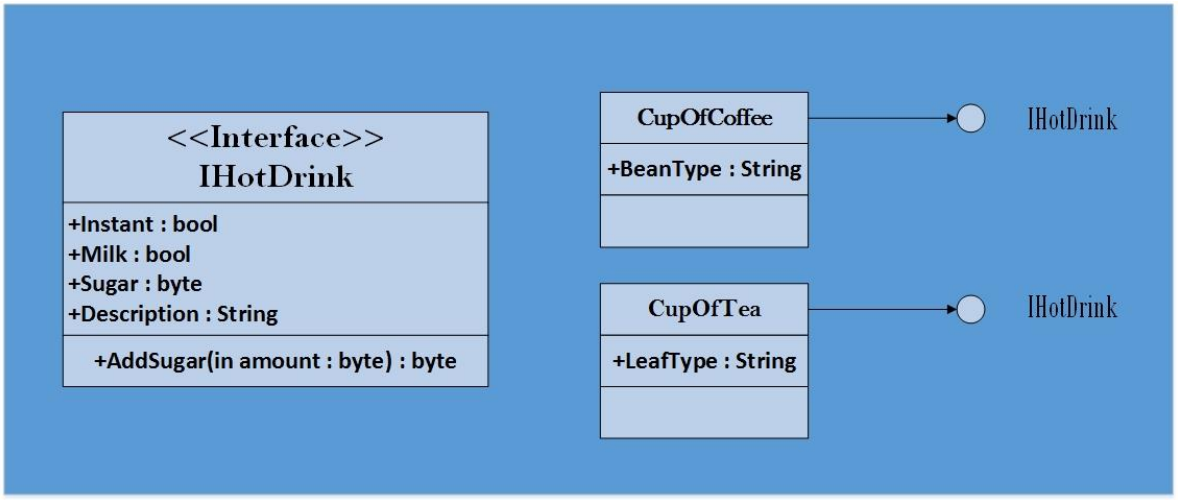
يمثل الشكل التالي تمثيل UML للواجهة IHotDrink وكيف بتم تزويد الصنفان CupOfTea وCupOfCoffee بها لاحظ أن تمثيل الواجهة مشابه تماما لتمثيل الصنف الشكل (6-1).

يمكن أن يدعم الصنف واجهات عديدة ويمكن لأصناف عديدة أن تزود بنفس الواجهة وإن استخدامنا للواجهات في البرمجة يوفر علينا الكثير من الجهد فبدلا من أن نعيد تعريف الأعضاء المشتركة بين صنفين يمكننا أن نضع هذه الأعضاء ضمن واجهة ومن ثم نزود هذين الصنفين بهذه الواجهة وسيصبح الأمر وكأن لهذين الصنفين أعضاء الواجهة بالإضافة إلى الأعضاء الخاصة لكل صنف على حدة.

توضيح:

ماذا نقصد بالتزويد؟ عندما نقوم بإنشاء صنف ما يتضمن منهجا وليكن A فإن علينا أن نزود هذا المنهج بشيفرة معينة يقوم بها هذا المنهج أما عندما نقوم بإنشاء واجهة ولتكن IA فإنه لا يمكننا كتابة شيفرة لمناهجها (على الرغم من أنه يمكن تعريف هذه المناهج فقط) عندما نود استخدام المناهج المعرفة ضمن الواجهة IA في الصنف A فإن علينا القيام بما يلي:

- ✓ **تزويد الصنف A بالواجهة IA وذلك ضمن سطر تعريف الصنف.**
- ✓ **تزويد الصنف A بالمناهج المقابلة الموجودة في IA أي تعريف هذه المناهج وكتابة شيفرة ضمنها.**



الشكل (6-1)

التخلص من الكائنات:

Disposable Objects:

إن إحدى الواجهات المهمة التي يجب أن ننوه عنها هنا هي IDisposable إن الكائن الذي يدعم هذه الواجهة يجب أن يزود بمنهج يسمى بـ Dispose() يمكن أن يستدعى هذا المنهج عندما لا نعود بحاجة للكائن ونود التخلص منه ولكن عملية التخلص من الكائنات ليست ضرورية دائما مع جميع الكائنات وإنما نحتاج أحيانا للتخلص من الكائنات التي تستهلك قدرا كبيرا من موارد النظام وذلك بعد انتهاءنا من استخدامها هل هذا يعني أن علينا تذكر استدعاء هذا المنهج دوما؟ في الحقيقة هناك أسلوب آخر يمكننا استخدامه للتخلص من الكائنات.

تسمح لنا لغة C# باستخدام بنية خاصة تمكننا من الاستفادة القصوة من هذا المنهج بطريقة غير مباشرة فباستخدام الكلمة Using يمكننا أن نهيئ كائنا يستخدم موارد حرجة (أي يستهلك موارد من النظام بشكل كبير كالذاكرة مثلا) حيث نستخدم هذا الكائن ضمن كتلة برمجية خاصة لها الصيغة التالية:

```
Using (<ClassName> <VariableName>=new <ClassName>())
{
...
}
```

وبهذه الطريقة فإن المنهج Dispose() سيستدعى تلقائيا عند الوصول إلى نهاية هذه الكتلة أي أن الكائن <VariableName> سيستخدم ضمن كتلة الشيفرة تلك فقط وسيتم التخلص منه تلقائيا عند نهاية الكتلة.

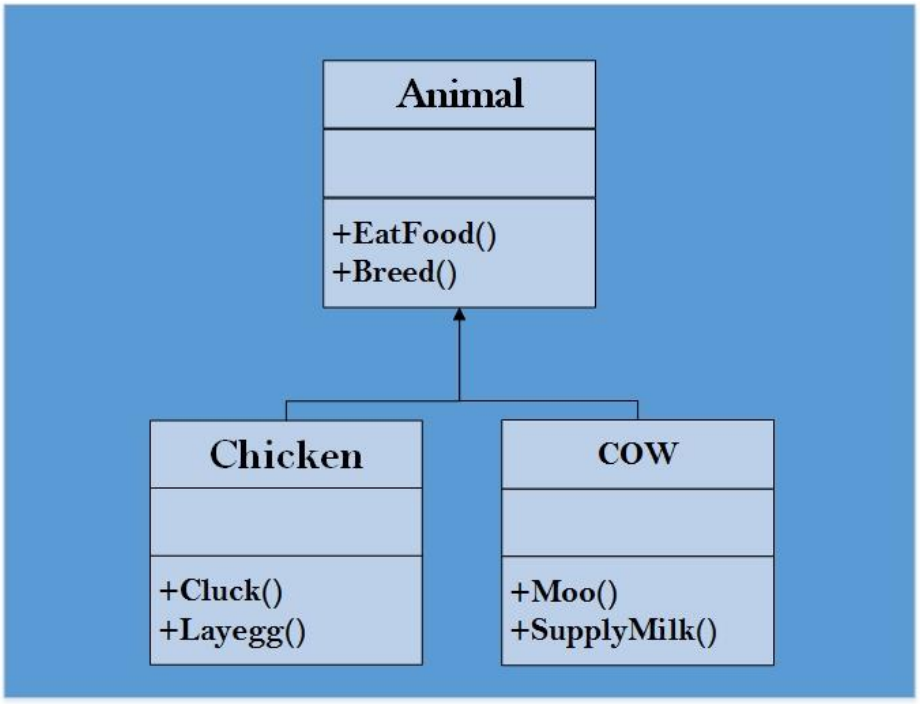
الوراثة:

Inheritance:

تعد الوراثة Inheritance من أهم مزايا البرمجة كائنية التوجه حيث يمكن لأي صنف أن يرث من صنف آخر ومعنى ذلك أن الأعضاء الموجودة في الصنف المورث كالخصائص والمنهج يمكن أن تستخدم في الصنف الوارث (المشتق) كما لو كانت معرفة ضمنية ويطلق على الصنف المورث بالصنف الأب parent ويسمى أيضا بالصنف الأساس (base) أما الأصناف التي ترث من الصنف الأساس فتسمى بالأصناف المشتقة (derived) وتنحدر جميع الكائنات في C# من صنف أساس واحد سنتعرف عليه في الفصل القادم.

تمكننا الوراثة من توسعة الأصناف أو إنشاء أصناف أكثر تخصصا من الصنف الأساس على سبيل المثال لنفترض أن هناك صنف الحيوانات يمكننا أن نسمي هذا الصنف بالاسم Animal ويتضمن مناهج مثل EatFood() و Breed() يمكننا إنشاء صنف مشتق يسمى بصنف البقر cow حيث يدعم جميع المناهج المتوفرة في الصنف Animal بالإضافة إلى أنه يدعم مناهجه الخاصة مثل Moo() و SupplyMilk() يمكننا إنشاء صنف آخر مشتق وليكن صنف الدجاج Chicken حيث يتضمن على المناهج Cluck() و LayEgg() مثلا.

يمكننا تمثيل الوراثة بواسطة مخطط UML كما في الشكل التالي:



الشكل (7-1)

ملاحظة:

لقد تجاهلت هنا أنواع القيم المعادة لتبسيط الأمور لا أكثر.

وعندما يرث صنف من الصنف الأساس تصبح مسألة وصولية الأعضاء أمرا هاما هنا لا يمكن الوصول إلى الأعضاء الخاصة للصنف الأساس من الصنف المشتق إلا أنه يمكننا الوصول إلى الأعضاء العامة في الحقيقة يمكننا الوصول إلى الأعضاء العامة من ضمن الصنف الأساس نفسه أو من ضمن أي صنف مشتق وكذلك من ضمن شيفرة خارجية لكن ماذا لو كنا بحاجة للوصول إلى الأعضاء ضمن الصنف المشتق دون التمكن من الوصول إليهم من الشيفرة الخارجية؟

لتحقيق ذلك هناك نوع ثالث من أنواع الوصولية يسمى بالنوع المحمي أو protected حيث يمكن فقط للأصناف المشتقة أن تصل إلى الأعضاء المحمية في الصنف الأساس.

وبالإضافة إلى تعريف مستوى الحماية لأعضاء الصنف الأساس فإن بإمكاننا أيضا تعريف سلوك الوراثة لهذه الأعضاء فيمكن لأعضاء الصنف الأساس أن تكون ظاهرية virtual بمعنى أنه يمكن تجاوز هذا العضو overridden من قبل الصنف المشتق إن ما نعنيه بذلك أنه ربما يكون لدى الصنف المشتق تزويد بديل لهذا العضو إن التزويد لهذا العضو لا يلغي شيفرة العضو الأصلي وإنما الوصول إليه من ضمن الصنف إلا أن ذلك يحجبه عن الشيفرة الخارجية فإن لم يكن هناك بديل لهذا العضو عندئذ ستتمكن الشيفرة الخارجية من الوصول إلى تزويد الصنف الأساس لهذا العضو.

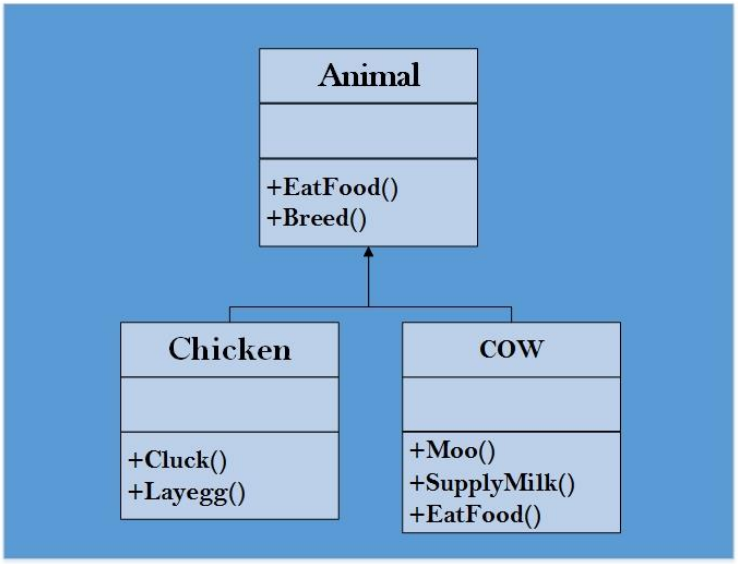
لاحظ أن الأعضاء الظاهرية لا يمكن أن تكون أعضاء خاصة لأن ذلك سيعتبر تناقضا فمن المستحيل أن نقول أن العضو يمكن أن يتجاوز من قبل الصنف المشتق في حين لا يمكن الوصول إلى هذا العضو في الصنف المشتق!

وبالعودة إلى مثال الحيوانات يمكننا ان نجعل المنهج EatFood() منهجا ظاهريا وذلك لأن بإمكان الحيوانات أن تأكل أشياء مختلفة وليس شيئا موحدا ومن ثم توفير تزويد جديد لهذا المنهج ضمن الصنف المشتق يمكننا تمثيل ذلك كما في الشكل (8-1).

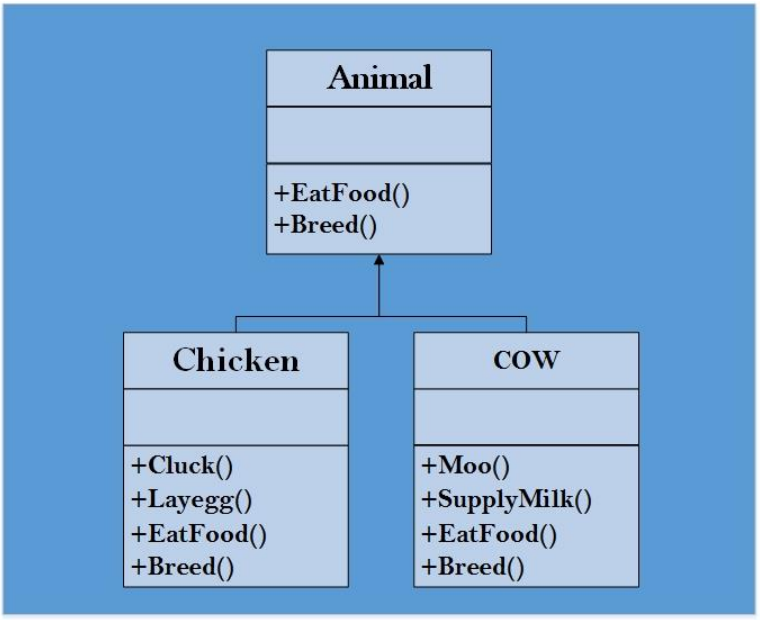
لاحظ أن هناك منهجا EatFood() أحدهما ضمن الصنف الأساس Animal والآخر ضمن الصنف المشتق cow لقد أضفنا المنهج EatFood() في الصنف cow وذلك لكي يكون للصنف المشتق تزويده الخاص لهذا المنهج.

يمكننا ان نعرف الأصناف الأساس على أنها أصناف مجردة abstract أي لا يمكننا أن ننشئ حالة من الصنف المجرد بصورة مباشرة ولكي نستخدم الصنف المجرد فإن علينا أولا أن نورثه لصنف غير مجرد ويمكن للأصناف المجردة أن تحوي على أعضاء مجردة أيضا وبالتالي ليس لهذه الأعضاء أي تزويد في الصنف الأساس أي يجب أن يكون هناك تزويد لهذه الأعضاء ضمن الصنف المشتق.

فإن كان الصنف Animals صنفا مجردا عندئذ سيصبح مخطط UML لمثال الحيوانات كما في الشكل (9-1).



الشكل (8-1)



الشكل (9-1)

وأخيرا يمكن للصنف أن يكون منغلقا sealed أي لا يمكننا استخدامه كصنف أساس وبالتالي لا يمكن ان نشتق منه أصنافا أخرى.

هناك صنف أساس مشترك في C# وهو يمثل الصنف الأساس لجميع الكائنات في C# ويسمى بالصنف object وهو اختصار للاسم الكامل System.object في إطار عمل .NET. سوف نتحدث عن هذا الصنف بتفصيل أكبر في الفصل القادم.

ملاحظة:

يمكن للواجهات التي تحدثنا عنها مسبقا في هذا الفصل أن ترث من واجهات أخرى وبعكس الأصناف فإن الواجهات يمكن أن ترث من واجهات أساس عديدة بنفس الصورة التي يمكن للأصناف أن تزود بواجهات عديدة.

توضيح:

لماذا نستخدم الواجهات ما دامت وراثة الأصناف تمكننا من القيام بنفس الوظائف تقريبا؟ حسنا لا بد أن الأمور بدأت تختلط عليك الآن إلا أننا سنبين متى نستخدم الواجهات ومتى نستخدم وراثة الأصناف في الفصل القادم.

تعددية الأشكال:

Polymorphism:

إن احد الأمور المهمة المتعلقة بالوراثة هنا هي أن الصنف المشتق يمكن أن يشترك مع الصنف الأساس بالمناهج والخصائص وبسبب ذلك فإنه من الممكن في معظم الأحيان أن نعامل الكائن الناشئ من الأصناف المشتقة من صنف أساس مشترك باستخدام صيغة مشتركة على سبيل المثال إذا كان للصنف الأساس Animal منهجا باسم EatFood() عندئذ فإن صيغة استدعاء هذا المنهج من الأصناف المشتقة cow و Chicken ستكون مماثلة أيضا لاستدعاء المنهج كما لو كان منتميا إلى الصنف المشتق:

```
cow myCow = new cow();
Chicken myChicken = new Chicken();
myCow.EatFood();
myChicken.EatFood();
```

إن تعددية الأشكال هنا تعطي للوراثة بعدا جديدا فيمكننا أن نسند لمتحول من نوع الصنف الأساس متحولا من نوع الصنف المشتق كما يلي:

```
Animal myAnimal=myCow;
```

ولسنا بحاجة لاستخدام التشكيل casting هنا يمكننا استدعاء مناهج الصنف الأساس من خلال هذا المتحول:

```
myAnimal.EatFood();
```

سيؤدي هذا الامر إلى استدعاء المنهج EatFood() المزود في الصنف المشتق لاحظ أنه لا يمكننا استدعاء المناهج المعروفة في الصنف المشتق بنفس الطريقة أي الشيفرة التالية لن تعمل بصورة صحيحة:

```
myAnimal.Moo();
```

على كل حال يمكن أن نستخدم التشكيل casting هنا حيث يمكننا تشكيل متحول النوع الأساس إلى متحول الصنف المشتق واستدعاء المنهج المعرف ضمن الصنف المشتق كما يلي:

```
cow myNewCow = (cow)myAnimal;
myNewCow.Moo();
```

سيتسبب هذا النوع من التشكيل برفع اعتراض إذا كان نوع المتحول myAnimal ليس بالصنف cow أو لم يكن أي صنف مشتق من الصنف cow هناك عدة طرق لمعرفة نوع الكائن إلا أننا سنترك ذلك للفصل القادم.

تمثل تعددية الاشكال تقنية مفيدة جدا لأداء مهام على الكائنات المختلفة المتحدرة من صنف واحد.

لاحظ أنه ليس مجرد الأصناف التي تشترك بصنف الأب نفسه وبشكل مباشر يمكن أن تستخدم تعددية الأشكال فمن الممكن أن نعالج ولنفرض صنف الابن وصنف الجد بنفس الطريقة أيضا (كاستخدام تعددية الاشكال بين صنف البقر cow وصنف الكائنات الحية وليكن له الاسم creature) وذلك طالما أن هناك صنف مشترك ضمن هرمية الوراثة.

وكملاحظة إضافية هنا تذكر أن جميع الأصناف في C# مشتقة من الصنف الأساس object وهذا يعني أن الصنف الأساس object هو الصنف الجذر في هرمية وراثة الأصناف وهذا يعني أن من الممكن أن تعالج جميع الكائنات كحالات من الصنف object.

ملاحظة:

لهذا السبب يمكن للأمر Console.WriteLine() استخدام أي نوع من البارامترات عند إنشاء نص الخرج فكل بارامتر بعد البارامتر الأول سيعالج كمتحول من نوع object مما سيسمح بطباعة القيمة المباشرة لأي كائن على نافذة الخرج.

تعددية الأشكال في الواجهات:

Interface Polymorphism:

لقد تحدثنا مسبقا عن مفهوم الواجهات ورأينا أنها تستخدم لتجميع المناهج والخصائص معا وعلى الرغم من أنه من غير الممكن إنشاء حالات من الواجهات إلا انه يمكن أن يكون لدينا متحول من نوع الواجهة يمكننا عندئذ استخدام هذا المتحول للوصول إلى المناهج والخصائص المقدمة من قبل هذه الواجهة على الكائنات التي تدعم تلك الواجهة.

على سبيل المثال لنفترض أنه بدلا من استخدام الصنف الأساس Animal لتزويد المنهج EatFood() فإننا سنضع المنهج EatFood() ضمن واجهة تسمى بـ ICoonsume عندئذ يمكن للصنفين Chicken وcow التزود بهذه الواجهة والفرق الوحيد في هذه الحالة هو أننا مجبرين على إيجاد تزويد للمنهج

EatFood() ضمن الصنف باعتبار أن الواجهات لا تتضمن أية تزويدات يمكننا عندئذ الوصول إلى هذا المنهج باستخدام شيفرة كما يلي:

```
cow myCow = new cow();
Chicken myChicken = new Chicken();
IConsume consumeInterface;
consumeInterface=myCow;
consumeInterface.EatFood();
consumeInterface=myChicken;
consumeInterface.EatFood();
```

يوفر ذلك طريقة بسيطة لاستدعاء كائنات عديدة بنفس الأسلوب حيث لا تعتمد على صنف أساس مشترك لقد قمنا في هذه الشيفرة باستدعاء المنهج consumeInterface.EatFood() والذي سيؤدي إلى استدعاء المنهج EatFood() الموجود ضمن الصنف cow والصنف Chicken تباعا وذلك بالاعتماد على متحول الحالة المسند إلى متحول الواجهة.

العلاقات بين الكائنات:

Relationship between Objects:

تمثل الوراثة علاقة بسيطة بين الكائنات حيث يتم كشف الصنف الأساس بصورة كاملة للصنف المشتق - الأعضاء العامة منه- ويمكن للصنف المشتق أن يتمكن من الوصول الداخلي إلى بعض أعضاء الصنف الأساس من خلال الأعضاء المحمية هناك أوضاع عديدة تكون فيها العلاقات بين الكائنات أمرا مهما.

سوف نتناول في هذا القسم ما يلي:

- الاحتواء: أي عندما يحتوي صنف ما صنفا آخر إن هذا مشابه للوراثة إلا أن الاحتواء يمكن الصنف الحاوي من التحكم بالوصول إلى أعضاء الصنف المحتوى بالإضافة إلى إمكانية القيام بعمليات إضافية قبل استخدام أعضاء الصنف المحتوى.
- المجموعات: حيث يعامل الصنف وكأنه حاو لحالات عديدة من صنف آخر إن هذا مشابه لحصولنا على مصفوفة من الكائنات إلا أن للمجموعات مزايا أخرى تمتاز بها عن المصفوفات فهي تتضمن آلية للفهرسة والترتيب وإمكانية تغيير الحجم وغيرها من المزايا.

الاحتواء:

Containment:

إن تطبيق الاحتواء أمر بسيط جدا ويتم باستخدام حقل ما لاستيعاب حالة كائن يمكن لهذا الحقل أن يكون عاما وبالتالي سيتمكن مستخدمو الكائن الحاوي من الوصول إلى مناهج وخصائص الكائن الموجود ضمن

الكائن الحاوي على شكل حقل إن هذا مشابه للوراثة إلا أننا هنا لن نتمكن من الوصول إلى الأعضاء الداخلية للصنف المحتوى عبر الصنف المشتق كما كنا نستطيع ذلك بالوراثة.

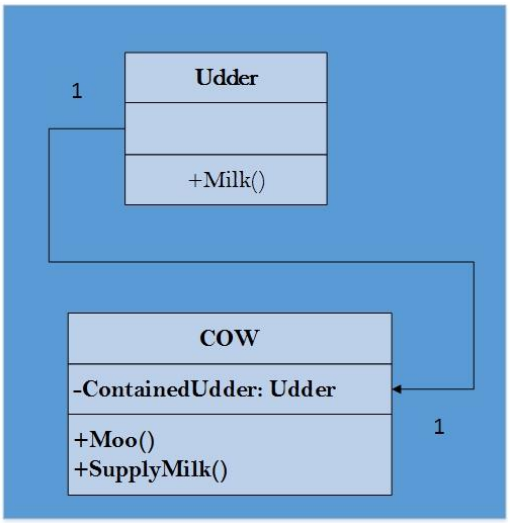
ملاحظة:

تذكر أن ما نقصده بالأعضاء الداخلية هي الأعضاء العامة والمحمية فقط فالأعضاء الخاصة في الصنف لا يمكننا الوصول إليها من الخارج.

وبصورة بديلة فإنه يمكننا جعل الكائن المحتوى عضوا خاصا ضمن الصنف الحاوي وإذا قمنا بذلك فلن نتمكن من الوصول إلى أي من أعضاء هذا الكائن مباشرة من خلال الشيفرة الخارجية وحتى إن كانت أعضاء الكائن المحتوى عامة وبدلا من ذلك فإن بإمكاننا ان نوفر وصولا إلى هذه الأعضاء باستخدام أعضاء الصنف الحاوي هذا يعني أن لدينا تحكما كاملا في كشف أعضاء الكائن المحتوى ضمن الصنف الحاوي بالإضافة إلى القيام بعمليات إضافية في أعضاء الصنف الحاوي قبل الوصول إلى أعضاء الصنف المحتوى.

على سبيل المثال يمكن لصنف البقر cow أن يتضمن على صنف الضرع Udder والذي له منهج عام يسمى بالحليب Milk() يمكن لكائن cow أن يستخدم هذا المنهج حسب الحاجة ولربما يقوم باستدعاء هذا المنهج كجزء من المنهج SupplyMilk() إلا ان تفاصيل هذا المنهج لن تكون ظاهرة أو مهمة لمستخدمي الكائن cow.

يمكننا ان نمثل الأصناف المحتواة في تمثيل UML باستخدام خط رابط بين الخاصية من نوع الصنف المحتوى والصنف المحتوي نفسه وسنضع على طرفي هذا الخط الرقم 1 وذلك للإشارة إلى أنها علاقة واحد – إلى – واحد أي أن كائن cow سيتضمن على كائن Udder واحد يمكننا أن نستعرض كائن الصنف Udder وكأنه حقل خاص في الصنف cow وذلك للتوضيح:



الشكل (1-10)

المجموعات:

Collections:

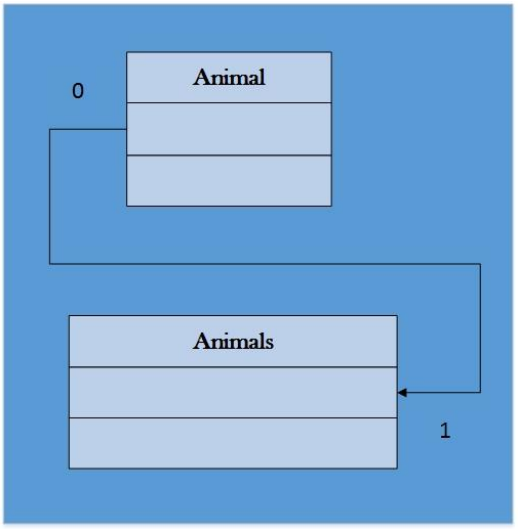
لقد تعلمنا في الفصل الخامس كيفية استخدام المصفوفات لحفظ قيم عديدة من نفس النوع في متحول واحد إن ذلك ينطبق أيضا على الكائنات تذكر أن أنواع المتحولات التي استخدمناها هي عبارة عن كائنات أيضا على سبيل المثال:

```
Animal [] animals = new Animal[5];
```

إن المجموعات عبارة عن مصفوفة مع بعض المزايا الإضافية تسمى المجموعات عادة بصيغة الجمع للكائنات التي تحتويها على سبيل المثال يمكن للصنف ذي الاسم Animals أن يحتوي على مجموعة من كائنات Animal.

إن الفرق الرئيسي بين المجموعات والمصفوفات يكمن في أن المجموعات تتضمن وظائف إضافية مثل المنهجين Add() وRemove() لإضافة وإزالة العناصر من المجموعة ويكون هناك عادة الخاصية Item والتي تمكننا من الوصول إلى كائن ما في المجموعة بالاعتماد على موقعه (دليله) منها. في الحقيقة هناك طرق تمكننا من استخدام هذه الخاصية للوصول إلى العناصر بصورة أكثر تفاعلية فعلى سبيل المثال يمكننا تصميم المجموعة Animals بحيث نتمكن من الوصول إلى كائنات Animal وفقا لأسمائها.

يبين تمثيل UML التالي المجموعات وارتباطها بعناصرها:



الشكل (11-1)

ملاحظة:

لم أضع الأعضاء هنا وذلك لأننا مهتمين الآن بالعلاقة بين الكائنات.

تبين الأرقام في أطراف الخط الواصل بين صنف المجموعة وصنف كائناتها بأن الكائن Animals يمكن ألا يتضمن أي كائن Animal ويمكن أن يتضمن أي عدد من كائنات Animal.

سوف نتحدث عن المجموعات بتفصيل أكبر في الرابع من هذا الجزء.

التحميل الزائد للعوامل:

Operator Overloading:

لقد استخدمنا العوامل في الفصول السابقة من هذا الكتاب لمعالجة أنواع المتحولات البسيطة لكن قد نجد في بعض الأحيان أنه من المنطقي استخدام العوامل مع كائنات الأصناف التي نقوم بإنشائها بأنفسنا إن ذلك ممكن وذلك بتزويد الأصناف بتعليمات تبين كيفية تفسير نتائج هذه العوامل.

على سبيل المثال يمكننا أن نضيف خاصية إلى الصنف Animal باسم Weight عندئذ إذا أردنا مقارنة أوزان الحيوانات باستخدام هذه الخاصية فإننا سنكتب شيفرة كما يلي:

```
if (cowA.Weight > cowB.Weight)
{
    ...
}
```

باستخدام التحميل الزائد للعوامل يمكننا أن نوفر المنطق الذي يعتمد على الخاصية Weight في مقارنة أوزان الحيوانات باستخدام أسماء الكائنات مباشرة وبالتالي يمكننا أن نكتب الشرط السابق كما يلي:

```
if (cowA > cowB)
{
    ...
}
```

لقد قمنا هنا باستخدام تقنية التحميل الزائد للعامل">" والعامل المحمل بصورة زائدة overloaded operator هو العامل الذي نكون قد كتبنا شيفرة تحدد سلوكه وذلك كجزء من تعريف أحد الأصناف التي تستخدمه فلقد استخدمنا في الشيفرة السابقة تعبيرا حديه هما كائنان من الصنف cow والعامل هو">" لاحظ أن هذا النوع من العمليات غير منطقي إلا إذا كان هناك تعريف جديد لهذا العامل يجعله منطقيا.

ولاحظ أيضا أن بإمكاننا استخدام تقنية التحميل الزائد مع عوامل C# المعروفة فقط فلا يمكننا ان ننشئ عوامل جديدة إلا أنه يمكننا أن نستخدم التحميل الزائد لهذه العوامل بشكلي هذه العوامل الأحادي والثنائي.

سوف نتحدث عن ذلك في الرابع من هذا الجزء.

الأحداث:

Events:

يمكن للكائنات ان تشهر الاحداث events كجزء من عملها إن الاحداث مهمة جدا فهي عبارة عن شيفرة برمجية معينة تنفذ عند حصول ظرف معين وهي مشابهة لطريقة عمل الاعتراضات إلا أنها أكثر قوة منها يمكننا على سبيل المثال تأدية مهمة معينة عندما نقوم بإضافة كائن Animal جديد إلى مجموعة الكائنات Animals بحيث لا تمثل الشيفرة التي تقوم بذلك جزءا من الصنف Animals أو الشيفرة التي تستدعي المنهج ()Add وللقيام بذلك فإن علينا إضافة معالج حدث event handler لشيفرتنا ومعالج الحدث عبارة عن نوع خاص من التوابع التي تستدعى عند حدوث ذلك الحدث وهو ما نسميه عندئذ بإشهار الحدث أو رفعه raise event وعلينا أن نعد معالج الحدث هذا لكي يستمع إلى الحدث الذي يخصه.

يمكننا إنشاء تطبيقات مقادة بالأحداث event-driven وهي أكثر تعقيدا مما تتوقع على سبيل المثال من المهم أن تعلم أن العديد من تطبيقات Windows تعتمد بصورة كلية على الأحداث فكل ضغطة بزر الفأرة أو سحب لشريط التمرير يؤدي إلى رفع حدث ما تتم معالجته بواسطة معالج الحدث الخاص به وهذا يعني أنه يمكننا توليد الأحداث من خلال الفأرة او لوحة المفاتيح أيضا.

سوف نرى لاحقا في هذا الفصل كيفية عمل آلية الحدث في تطبيقات Windows وسوف نتعمق في الحديث عن الأحداث في الخامس من هذا الجزء.

أنواع المرجع مقابل أنواع القيمة:

Reference vs. Value Types:

تحفظ البيانات بالمتحول في C# بطريقتين وذلك بحسب نوع المتحول ويمكننا الآن أن نصنف الأنواع في C# تحت بندين: أنواع المرجع (Reference Types) وأنواع القيمة (Value Types):

- نحفظ أنواع القيمة محتواها ضمن موضع واحد في الذاكرة(يسمى بالمكدس Stack).
- تحفظ أنواع المرجع بمرجع يشير إلى محتوى موجود في مكان آخر من الذاكرة (يسمى بالكومة Head).

أنت لست بحاجة للاهتمام لهذه التفاصيل عند البرمجة بلغة C# لقد استخدمنا متحولات String وهي أنواع مرجعية ومتحولات أخرى بسيطة مثل متحولات int ومعظمها أنواع قيمة وقد كان استخدامنا لهذين النوعين متشابها.

في الحقيقة ليس هناك إلا نوعين مرجعيين بسيطين فقط: النوع String والنوع object وهناك أيضا المصفوفات والتي تعتمد بطبيعتها على النوع الأساس لها إن كل صنف تقوم بإنشائه هو نوع مرجعي ولذا فإنني نوهت عن ذلك هنا في هذا الفصل.

البنية Struct:

Struct:

هناك نقطة هامة يجدر الإشارة إليها هنا فعلى الرغم من أن البنية Struct شبيهة جدا بالأصناف إلا أن هناك فرقا جوهريا بينهما فالبنية Struct نوع قيمة وأما الأصناف كما قلنا فهي نوع مرجعي.

ملاحظة:

إن التشابه بين الأصناف وبنى Struct واضح بالنسبة إلينا فلكل منهما أعضاء ويمكن لكل منهما أن يحتوي على توابع تسمى بالمناهج في الأصناف ومتحولات أيضا تسمى بالخصائص أو الحقول في الأصناف.

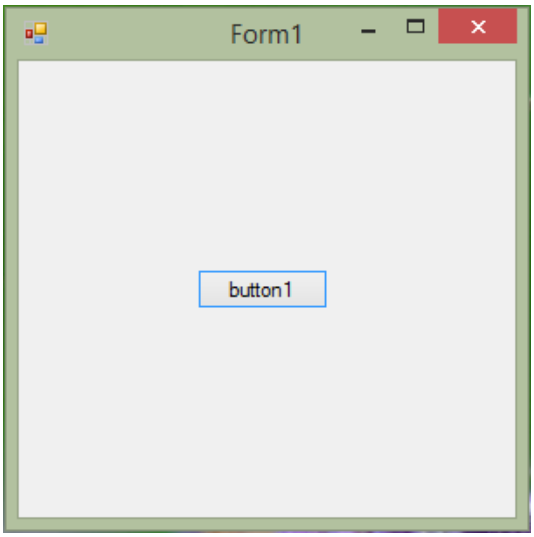
البرمجة كائنية التوجه في تطبيقات Windows:

OOP in Windows Applications:

لقد رأينا في الفصل الثاني من الجزء الأول لهذا الكتاب كيفية إنشاء تطبيق Windows بلغة C# إن تطبيقات Windows تعتمد بصورة كبيرة على تقنيات OOP وسوف نتناول في هذا القسم بعضا من النقاط التي تضع ما تحدثنا عنه في هذا الفصل موضعا عمليا.

تطبيق حول الكائنات في تطبيقات Windows:

1- قم بإنشاء تطبيق Windows Application جديد وقم بحفظه باسم Windows Application OOP.
2- أضف عنصر تحكم Button باستخدام صندوق الأدوات Toolbox وضعه في وسط النموذج Form1:



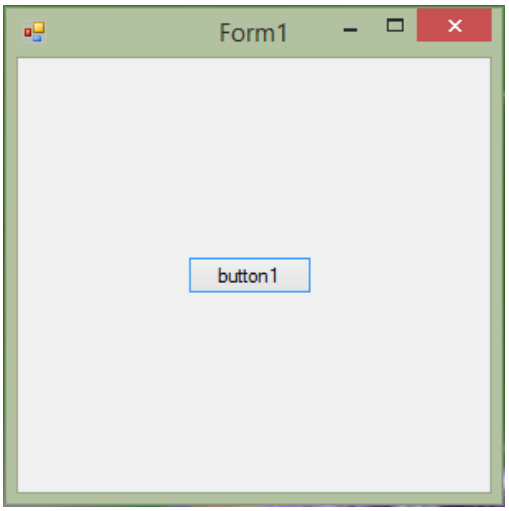
الشكل (12-1)

3- انقر نقرا مزدوجا على زر الامر لإضافة الشيفرة لحدث الضغط على هذا الزر بالفأرة عدل الشيفرة كما يلي:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Windows_Application_OOP
{

    public partial class Form1 : Form
    {

        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            ((Button)sender).Text = "اضغط الزر";
            Button newButton = new Button();
            newButton.Text = "زر جديد";
            newButton.Click += new EventHandler(newButton_Click);
            Controls.Add(newButton);
        }
```

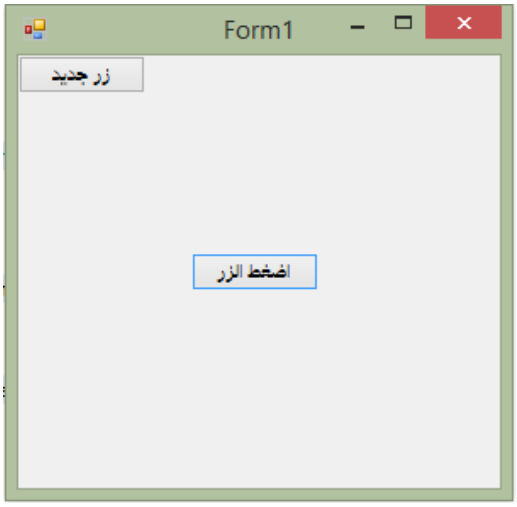
```
        private void newButton_Click(object sender, System.EventArgs e)
        {
            ((Button)sender).Text = "اضغط الزر";
        }
    }
}
```

4- نفذ التطبيق بالضغط على F5 من لوحة المفاتيح أو بالضغط على زر Start من شريط الأدوات فيظهر الشكل (1-12).



الشكل (1-12)

5- اضغط على الزر Button1 فيتغير اسم الزر إلى "اضغط هنا" ويظهر زر جديد باسم "زر جديد" كما في الشكل (1-13).



الشكل (1-13)

6- انقر على الزر "زر جديد" ولاحظ كيف يتغير اسمه إلى "اضغط هنا" كما في الشكل (14-1).

الشكل (14-1)

كيفية العمل:

How to Work:

بإضافة بعض الأسطر البرمجية فإننا قمنا بإنشاء تطبيق Windows يقوم بعمل ما ولقد استخدمنا بعضا من تقنيات OOP أيضا في الحقيقة "كل شيء عبارة عن كائن!" هي عبارة حقيقية جدا عند التعامل مع تطبيقات Windows فمن النموذج الذي سيعرض عند تنفيذ التطبيق وصولا إلى عناصر التحكم الموجودة على النموذج فإن جميعها تحتاج لاستخدام تقنيات OOP دائما وسوف نركز هنا على تقنيات OOP وعلى المفاهيم النظرية التي تحدثنا عنها في هذا الفصل لتتجلى أمامنا الصورة بشكل أوضح.

إن أول ما قمنا به في تطبيقنا هو إضافة زر أمر جديد إلى النموذج Form1 إن هذا الزر هو عبارة عن كائن يسمى Button1 وهو كائن من الصنف Button وبالنقر المزدوج على هذا الزر فإننا نكون قد أضفنا معالج حدث event handler ستنفذ عند النقر Click على الزر Button سيتم إضافة معالج الحدث هذا إلى الشيفرة ضمن الكائن Form1 الذي يغلف تطبيقنا وذلك كمنهج خاص private:

```
private void button1_Click(object sender, EventArgs e)
{

}
```

لقد استخدمنا هنا الكلمة private في بداية تعريف المنهج لا تقلق حول ذلك الآن فسوف نتحدث عما تعنيه هذه الكلمة وغيرها لاحقا في الفصل القادم.
إن السطر الأول الذي أضفناه هنا يقوم بتعديل نص عنوان الزر ولقد استخدمنا هنا تعددية الأشكال polymorphism يمثل الكائن Button1 الزر الذي إذا نقرنا عليه بزر الفأرة فإنه سيتم إرسال هذا

الحدث إلى معالج الحدث على شكل بارامتر من نوع object حيث سنقوم بتشكيل cast هذا الكائن إلى النوع Button إن ذلك ممكن باعتبار أن الكائن Button يرث من الكائن System.object والذي يمثل الاسم الكامل للكائن object في C# لقد غيرنا بعد ذلك قيمة الخاصية Text لهذا الكائن وذلك لتغيير عنوان ذلك الزر كما يلي:

```
((Button)sender).Text = "اضغط الزر";
```

بعد ذلك قمنا بإنشاء كائن Button جديد باستخدام الكلمة new وذلك ضمن الشيفرة التالية حيث سيأخذ زر الأمر الجديد العنوان "زر جديد"

```
Button newButton = new Button();
newButton.Text = "زر جديد";
```

توضيح:

لاحظ أننا استفدنا هنا من ميزة فضاءات الأسماء وذلك لتبسيط الصيغ ويمكنك ملاحظة هذا ضمن الأسطر الأولى من الملف Form.cs إن لم نستخدم تعليمات using تلك فإننا مضطرين إلى استخدام الصيغة الكاملة وبالتالي فإن علينا كتابة النوع Button كما يلي:

System.Windows.Forms.Button

هناك معالج حدث آخر في هذه الشيفرة وهو ما سنستخدمه للاستجابة لحدث النقر على زر newButton الجديد:

```
private void newButton_Click(object sender, System.EventArgs e)
{
    ((Button)sender).Text = "اضغط الزر";
}
```

وبالعودة إلى شيفرة معالج حدث النقر على الزر Button1 الأصلي فإننا قمنا بتسجيل معالج الحدث للزر الجديد باستخدام صيغة التحميل الزائد للعوامل ولقد قمنا بإنشاء كائن من نوع EventHandler جديد باستخدام منهج بناء غير افتراضي وذلك باستخدام نفس اسم منهج معالج الحدث الجديد:

```
newButton.Click += new EventHandler(newButton_Click);
```

وأخيرا فلقد استخدمنا كائن المجموعة Controls للنموذج وذلك لإضافة الزر الجديد إلى النموذج باستخدام المنهج Add():

```
Controls.Add(newButton);
```

يبين هذا المثال البسيط أننا استخدمنا معظم تقنيات البرمجة كائنية التوجه التي تحدثنا عنها في هذا الفصل وكما ترى من هذا المثال فإن تقنية OOP ليست معقدة إلى درجة كبيرة وإنما تتطلب منا نظرة مختلفة لطريقة البرمجة لا أكثر.

الخلاصة:

Summary:

لقد قدم هذا الفصل وصفا كاملا لتقنيات البرمجة كائنية التوجه لقد تحدثنا عن تقنيات OOP في سياق حديثنا عن البرمجة بلغة C# إلا اننا ركزنا هنا على OOP بصورة خاصة أكثر من تطرقنا لكيفية استخدامها في C#.

لقد بدأنا أولا بتغطية الأساسيات مثل معنى المصطلح object وكيفية إنشاء حالة من الكائن لصنف ما ثم رأينا بعد ذلك كيف يمكن للكائنات أن تتضمن أعضاء عديدة مثل الحقول fields والخصائص properties والمناهج methods وكيفية التحكم في الوصولية إلى هذه الأعضاء فيمكن ان تكون أعضاء عامة او خاصة او محمية بعد ذلك تحدثنا عن إمكانية ان تكون هذه الأعضاء ظاهرية او مجردة ولقد تحدثنا أيضا عن إمكانية وجود أعضاء ستاتيكية static مشتركة أو حالية instance.

تحدثنا بعد ذلك عن دورة حياة الكائن ورأينا اننا نستخدم مناهج البناء Constructors لإنشاء الكائنات ومناهج التدمير Destructors لحذف الكائنات ولقد تحدثنا أيضا حول كيفية تجميع الأعضاء في واجهات interfaces وتحدثنا عن كيفية استخدام طرق أخرى للتخلص من الكائنات disposable objects بواسطة الواجهة IDisposable.

ولقد تحدثنا عن الوراثة inheritance ورأينا كيف يمكن للأصناف أن ترث من أصناف أخرى ورأينا أيضا كيفية استخدام تعددية الاشكال polymorphism وذلك من خلال الأصناف الأساس والواجهات المشتركة ورأينا كيف يمكننا ان نستخدم الكائنات لاحتواء كائنات أخرى وذلك من خلال تقنيات الاحتواء containment والمجموعات collections وأخيرا فقد تحدثنا عن تقنية التحميل الزائد للعوامل لتبسيط صيغ استخدام العوامل مع الكائنات وكيف يمكن للكائنات أن ترفع الاحداث events.

لقد وضع الجزء الأخير من هذا الفصل المعلومات النظرية التي تناولناها منذ بداية هذا الفصل موضع الاستخدام العملي على الرغم من أننا لم نستخدم إلا بضعة أسطر برمجية وذلك ضمن تطبيق Windows بسيط جدا.

ربما لم تستوعب معظم ما تناولناه في هذا الفصل إلا أن الهدف من تقديم جميع تقنيات البرمجة كائنية التوجه في فصل كهذا لإعطائك فكرة عامة عن هذه التقنيات وليس من الضروري ان تستوعب كل ما تحدثنا عنه هنا إلا ان الأمور ستتوضح لك بعد قراءتك للفصول الأربعة القادمة.

الفصل الثاني

تعريف الأصناف

لقد تحدثنا في الفصل السابق عن مزايا البرمجة كائنية التوجه OOP وسوف نتناول في هذا الفصل بعضا من المعلومات النظرية التي تطرقنا إليها سابقا بتطبيقات عملية وموضوع هذا الفصل هو كيفية تعريف واستخدام الأصناف في C#.

لن نتحدث هنا عن كيفية تعريف أعضاء الأصناف وسنركز على تعريف الأصناف فقط وسنترك الحديث عن الأعضاء للفصل القادم.

وكبداية فإننا سنتحدث عن الصيغة الأساسية لتعريف الأصناف والكلمات المفتاحية التي سنستخدمها لتحديد وصولية الصنف ...الخ كما وسنتناول موضوع الوراثة بالتفصيل في هذا الفصل وسنتحدث أيضا عن الواجهات وكيفية تعريفها باعتبارها مشابهة للأصناف أما بقية هذا الفصل فسوف نتحدث ضمنه عن مواضيع عديدة متعلقة بتعريف الأصناف في C# يتضمن ذلك:

- ✓ الصنف System.object.
- ✓ أدوات مفيدة يوفرها Visual Studio.NET.
- ✓ مكتبات الأصناف.
- ✓ مقارنة بين الواجهات والأصناف المجردة.
- ✓ أنواع البنى Struct.
- ✓ نسخ الكائنات.

تعريف الأصناف في C#:

Class Definitions in C#:

تستخدم لغة C# الكلمة المفتاحية Class لتعريف الأصناف والبنية الأساسية لتعريف الأصناف لها الصيغة التالية:

```
Class <ClassName>
{
```

```
    //Class members
}
```

يشير <ClassName> هنا إلى اسم الصنف الذي نود إنشائه ومتى قمنا بتعريف الصنف فإن بإمكاننا إنشاء كائن من هذا الصنف في أي مكان من مشروعنا يستطيع الوصول إلى هذا الصنف ويتم التصريح عن الأصناف داخليا internal بصورة افتراضية مما يعني أن شيفرة المشروع الحالي فقط يمكنها الوصول إلى هذا الصنف ويمكننا أن نحدد ذلك بشكل صريح باستخدام الكلمة internal كما يلي:

```
internal class  MyClass
{
   //Class members
}
```

حيث قمنا في هذه الشيفرة بإنشاء صنف داخلي جديد باسم MyClass.

وإذا أردنا ان نستخدم الأصناف في شيفرة مشاريع أخرى فإن علينا التصريح عن الصنف على أنه صنف عام وذلك باستخدام الكلمة Public كما يلي:

```
public class  MyClass
{
   //Class members
}
```

تسمى الكلمات internal وpublic في OOP بمقيدات الوصول access modifiers وذلك لأنها تحدد كيفية الوصول إلى الأصناف أو أية عناصر أخرى.

ملاحظة:

لاحظ أن الأصناف المصرح عنها بهذه الصورة لا يمكن أن تكون خاصة private أو محمية protected فمن الممكن استخدام هذين المقيدين للتصريح عن الأصناف كأعضاء محتواه في صنف وهو ما سنتحدث عنه في الفصل القادم.

يمكننا إنشاء كائنات (حالات) من الأصناف التي نعرفها بأنفسنا بصورة افتراضية ويمكننا تغيير هذا السلوك باستخدام مقيدات وصول معينة فكما يمكننا تحديد مدى الوصول للصنف ضمن تعريفه فإنه يمكننا أيضا تحديد ما إذا كان الصنف مجردا abstract أو منغلقا sealed وللقيام بذلك فإننا سنستخدم الكلمتين abstract وsealed.

توضيح:

ونقصد بالصنف المجرد هو الصنف الذي لا يمكننا إنشاء حالة منه وإنما يمكننا توريثه لأصناف أخرى فقط أما الصنف المغلق فهو الصنف الذي لا يمكن ان ترثه أصناف أخرى.

وبالتالي إذا أردنا منع إمكانية إنشاء حالة من صنفنا العام MyClass فإننا سنعرف الصنف بالصورة التالية:

```
public abstract class  MyClass
{
   //Class members, may be abstract
}
```

لقد قمنا هنا بالتصريح عن صنف مجرد عام باسم MyClass ويمكننا أن نصرح عن صنف مجرد داخلي بنفس الأسلوب.

أما الأصناف المنغلقة تعرف بنفس الطريقة أيضا:

```
public sealed class  MyClass
{
   //Class members
}
```

وكما هو الحال بالنسبة للأصناف المجردة فإن الأصناف المنغلقة يمكن أن تكون داخلية أو عامة.

وإذا أردنا ان يرث صنفنا أعضاء صنف آخر فإن علينا تحديد ذلك ضمن سطر التصريح عن الصنف أيضا وللقيام بذلك فإننا سنضع الرمز ":" بعد اسم الصنف ونتبعه باسم الصنف الأساس الذي سيرث منه مباشرة على سبيل المثال:

```
Public Class MyClass :MyBase
{
  //Class members
}
```

لقد قمنا هنا بتعريف صنف عام باسم MyClass وهو قابل لأن نقوم بإنشاء حالات منه ويرث هذا الصنف أعضاء صنف أساس له الاسم MyBase.

لاحظ أنه لا يمكن للصنف وراثة إلا صنف واحد فقط في C# وإذا ورث من صنف مجرد فإن علينا تزويد صنفنا بجميع الأعضاء المجردة الموجودة في الصنف الأساس إلا إذا كان صنفنا المشتق هو أيضا صنف مجرد.

لن يسمح لنا المترجم بأن يكون الصنف المشتق ذو مدى وصول أكبر من مدى وصول الصنف الأساس إن هذا يعني أن الصنف الداخلي يمكن ان يرث من صنف أساس عام إلا أنه لا يمكن لصنف عام أن يرث من صنف أساس داخلي أن هذا يعني أن الشيفرة التالية صحيحة:

```
public class  MyBase
{
  //Class members
```

```
}
internal class  MyClass : MyBase
{
   //Class members
}
```

إلا ان الشيفرة التالية لن تترجم:

```
internal class  MyBase
{
  //Class members
}
public class  MyClass : MyBase
{
   //Class members
}
```

وعندما لا يرث الصنف أي صنف أساس أخر فهو يرث الصنف الأساس System.Object فقط والذي يمكننا كتابته بالاسم المختصر object فقط في الحقيقة إن جميع الأصناف في C# ترث الصنف الأساس System.object بصورة افتراضية فهو يعد الجذر في هرمية وراثة الأصناف سوف نتحدث عن هذا الصنف الأساسي لاحقا في هذا الفصل.

وبالإضافة إلى إمكانية تحديد أصناف الأساس بهذه الطريقة فإنه يمكننا تحديد الواجهات التي يدعمها صنفنا بعد الرمز ":" وإن كان صنفنا يرث من صنف أساس ومزود بواجهة أيضا فإن علينا ذكر اسم الصنف الأساس أولا ثم اسم الواجهة بحيث يفصل بين الاسمين بواسطة الفاصلة "," لاحظ انه يمكن للصنف التزود بعدة واجهات.

على سبيل المثال يمكننا إضافة واجهة للصنف MyClass كما يلي:

```
public class  MyClass : IMyInterface
{
   //Class members
}
```

يجب أن يزود الصنف بجميع أعضاء الواجهة التي يدعمها ولو كان هذا التزود فارغا أي بدون شيفرة مفيدة إذا لم نرغب باستخدام عضو واجهة معين.

لنفرض ان لدينا واجهة بالاسم IMyInterface وصنفا أساس بالاسم MyBase عندئذ فإن التصريح التالي غير صحيح:

```
public class  MyClass : IMyInterface , MyBase
{
   //Class members
}
```

والطريقة الصحيحة للتصريح السابق بالصورة التالية:

```
public class  MyClass :MyBase , IMyInterface
{
```

```
   //Class members
}
```

وتذكر أن بإمكاننا تزويد الصنف بالعديد من الواجهات لذا فإن التصريح التالي صحيح:

```
public class  MyClass :MyBase , IMyInterface ,IMySecondInterface
{
   //Class members
}
```

أي يمكننا وضع صيغة عامة لتعريف الأصناف كما يلي:

[<Mod1>] [<Mod2>] Class <ClassName> : [<BaseClass> , <Interfaces>]

{

// Class members

}

تشير الاقواس "[]" إلى أن هذا الجزء من الصيغة اختياري أما <Mode1> فتمثل أحد مقيدي الوصول internal أو public و <Mode2> فيمثل أحد مقيدي الوصول abstract او sealed اما <ClassName> فيمثل اسم الصنف وهو يخضع لنفس قواعد تسمية المتحولات في C# أما <BaseClass> فيمثل الصنف الأساس الذي يرثه الصنف <ClassName> وكذلك فإن <Interface> يمثل الواجهات التي يتزود بها الصنف.

يبين الجدول التالي الاحتمالات الممكنة لمقيدات الوصول في تعريف الصنف:

المقيد	المعنى
Internal أو بدون مقيد	**يمكن الوصول إلى الصنف ضمن المشروع الحالي فقط.**
Public	**سمكن الوصول إلى الصنف من أي مكان ضمن المشروع الحالي او خارجه.**
abstract او internal abstract	**يمكن الوصول إلى الصنف من ضمن المشروع الحالي فقط ولا يمكن إنشاء حالة من هذا الصنف وإنما يمكننا توريثه لأصناف أخرى.**
Public abstract	**يمكن الوصول إلى الصنف من أي مكان ولا يمكن إنشاء حالة من هذا الصنف وإنما يمكننا وراثته.**
Sealed أو internal sealed	**يمكن الوصول إلى الصنف من ضمن المشروع الحالي فقط ولا يمكننا وراثة هذا الصنف وإنما يمكننا إنشاء حالة منه.**
Public sealed	**يمكن الوصول إلى الصنف من أي مكان ولا يمكننا وراثة هذا الصنف وإنما يمكننا إنشاء حالة منه.**

تعريف الواجهات:

Interface Definitions:

يصرح عن الواجهات بطريقة مشابهة للتصريح عن الأصناف إلا أننا نستخدم الكلمة interface بدلا من الكلمة Class أي التصريح البسيط عن الواجهة يأخذ الصيغة التالية:

```
Interface <interfaceName>
{
  // interface members
}
```

ويستخدم مقيدا الوصول public و internal بنفس الأسلوب التي تستخدمه الأصناف أيضا وبالتالي يمكننا ان نتمكن من الوصول إلى الواجهة IMyInterface من خارج المشروع الحالي باستخدام التعريف التالي:

```
public interface IMyInterface
{
   // interface members
}
```

لا يمكننا استخدام المقيدات abstract وsealed مع الواجهات وذلك لأنه ليس لاستخدامها أي معنى مع الواجهات ذلك لأنه لا يمكننا استخدام الواجهات بصورة مباشرة وإنما يجب أن نورثها لصنف أو لواجهة أخرى كي نستفيد منها.

ويمكن للواجهات ان ترث واجهات أخرى بنفس أسلوب الوراثة في الأصناف إلا أن الفرق الأساسي هنا يكمن في عدم وجود صنف أساس على سبيل المثال:

```
public interface IMyInterface : IMyBaseInterface , IMyBaseInterface2
{
   // interface members
}
```

وجميع الواجهات ترث الصنف System.object كما في الأصناف أيضا ويوفر ذلك آلية استخدام تعددية الأشكال مع الواجهات أيضا وكما نوهنا مسبقا فإنه من المستحيل إنشاء حالات من الواجهات كما نقوم بذلك في الأصناف.

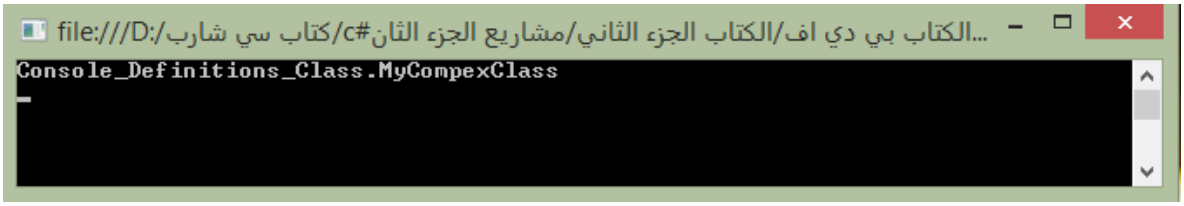
لنلق نظرة الآن على بعض تعريفات الأصناف في تطبيق بسيط.

تطبيق حول تعريف الأصناف:

1- قم بإنشاء تطبيق Console جديد باسم Console Definitions Class.
2- أضف الشيفرة التالية إلى الملف Program.cs.

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

namespace Console_Definitions_Class
{
    public abstract class MyBase
    {
    }
    internal class MyClass : MyBase
    {
    }
    public interface IMyBaseInterface
    {
    }
    internal interface IMyBaseInterface2
    {
    }
    internal interface IMyInterface : IMyBaseInterface, IMyBaseInterface2
    {
    }
    internal sealed class MyComplexClass : MyClass, IMyInterface
    {
    }
    class Program
    {
        static void Main(string[] args)
        {
            MyComplexClass myObj = new MyComplexClass();
            Console.WriteLine(myObj.ToString());
            Console.ReadLine();
        }
    }
}
```

3- نفذ الشيفرة بالضغط على زر F5 فيظهر الشكل (1-2).

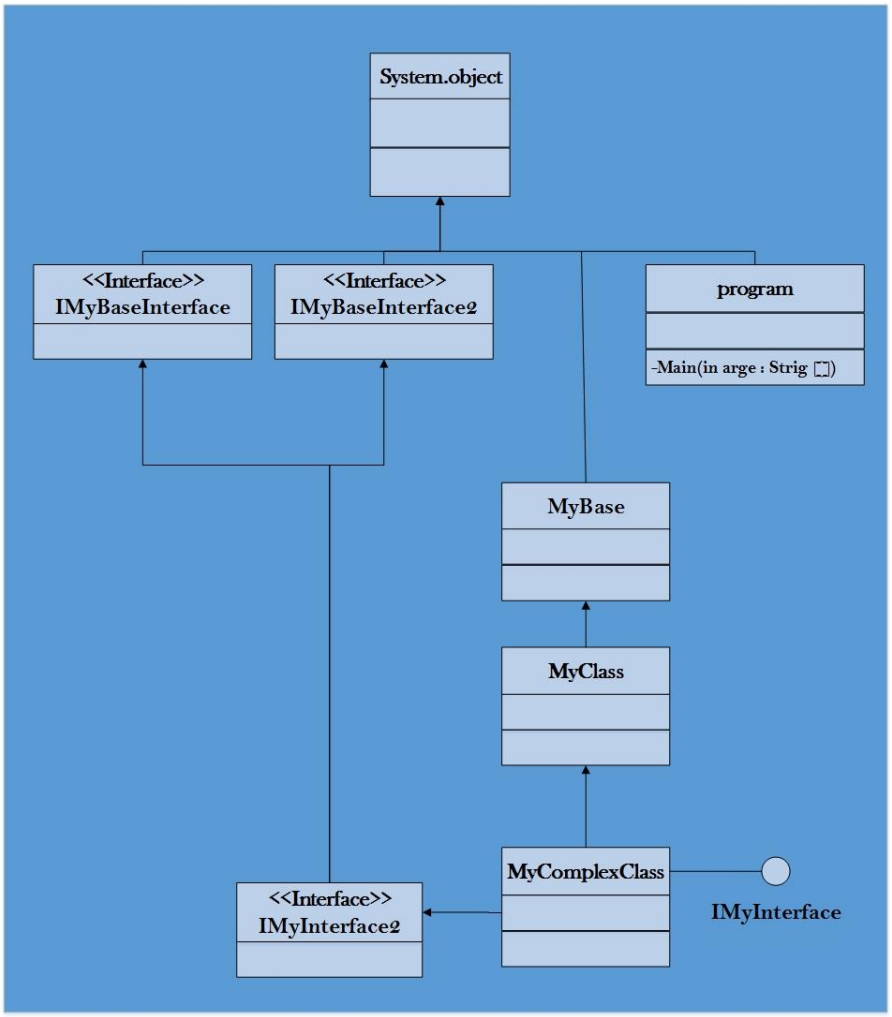


الشكل (1-2)

كيفية العمل:

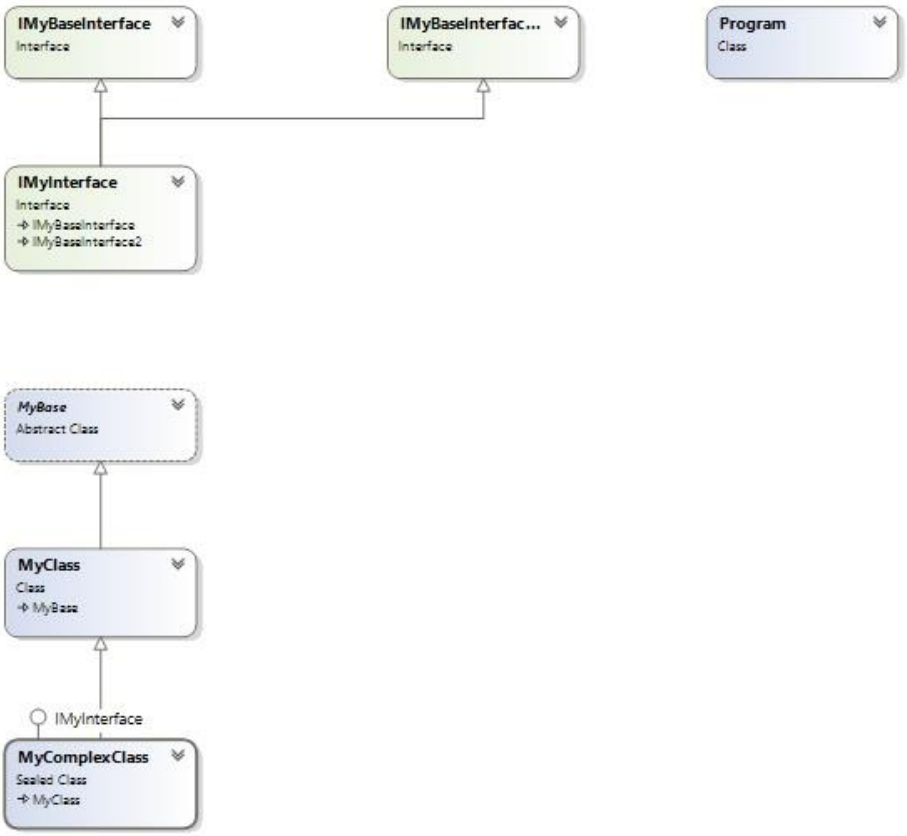
How to Work:

لقد قمنا في هذا المشروع بتعريف عدة واجهات واصناف وهي موضحة بتمثيل UML التالي حيث نبين هنا هرمية الوراثة لهذه العناصر:



الشكل (2-2)

إن إصدارة Visual Studio.NET 2013 تحوي على إطار يظهر العلاقة بين الفئات والواجهات وهي أشبه بمخطط UML ويمكننا إظهار مخطط مشروعنا هذا باختيار الملف Program.cs من نافذة Solution Explorer ومن ثم الضغط عليها بزر الفأرة الأيمن واختيار البند View Class Diagram فتظهر نافذة باسم *ClassDiagram.cd وتظهر ضمنها مخطط المشروع وفق تمثيل صندوقي كما في الشكل (2-3).



الشكل (3-2)

ملاحظة:

تمكننا *ClassDiagram.cd من تنفيذ عدة أمور سوف نتعرف عليها في الفصل القادم لذا لا تقلق.

تبين الأشكال السابقة جميع الأصناف والواجهات التي يتضمنها مشروعنا لاحظ أن الصنف System.object هو الصنف الذي ترث منه جميع العناصر في المشروع سواء الأصناف أو حتى الواجهات وأما المنهج Main () فهو يمثل نقطة الدخول للتطبيق ككل.

إن الصنف MyBase والواجهة IMyBaseInterface هما عنصران عامان وبالتالي يمكننا الوصول إليهما من خلال مشاريع أخرى أما الأصناف والواجهات الأخرى فهي داخلية لذا فإنه لا يمكننا الوصول إليها من داخل مشروعنا الحالي فقط.

يقوم التطبيق السابق باستدعاء المنهج ()ToString للكائن myObj والذي يمثل كائنا من نوع MyComplexClass:

```
MyComplexClass myObj = new MyComplexClass();
Console.WriteLine(myObj.ToString());
```

وعلى الرغم من عدم وجود أية أعضاء ضمن الصنف MyCompexClass إلا أننا قمنا باستخدام المنهج ToString() لكائن من نوع هذا الصنف وذلك لأن الصنف MyCompexClass ورث هذا المنهج من الصنف الأساس System.object ومهمة هذا المنهج هي إعادة اسم صنف الكائن على شكل نص (قيمة من نوع String).

ملاحظة:

إن هذا يعني أن بإمكاننا استخدام جميع خصائص ومناهج الصنف System.object مع أي صنف في C# ويمكنك ان تجرب ذلك بنفسك.

الكائن System.object:

System.object:

باعتبار أن جميع الأصناف ترث من الصنف System.object فإن جميع الأصناف يمكنها الوصول إلى الأعضاء العامة والمحمية لهذا الصنف وبالتالي من المهم ان نستعرض تلك الأعضاء يبين الجدول التالي المناهج التي يتضمنها الصنف System.object:

المنهج	القيم المعادة	منهج ظاهري	منهج ستاتيكي	الوصف
Object()	بدون	لا	لا	منهج بناء الكائنات من نوع System.object ويتم استدعائه بواسطة مناهج البناء للأنواع المشتقة.
~object() أو Finalize()	بدون	لا	لا	منهج تدمير الكائنات من نوع System.object ويتم استدعائه تلقائيا بواسطة مناهج التدمير للأنواع المشتقة.
Equals(object)	bool	نعم	لا	يأخذ بارامتر من نوع object ويقوم بمقارنة الكائن الذي تم استدعاء المنهج من خلاله مع كائن آخر (البارامتر) ويعيد القيمة true إن كان الكائنين متساويان ويمكن تجاوز هذا المنهج إذا أردنا مقارنة الكائنين بصورة مختلفة.
Equals(object , object)	bool	لا	نعم	يأخذ بارامترين من نوع object ويقوم بمقارنة هذين الكائنين ويعيد القيمة true إن كان الكائنين

				متساويان لاحظ أنه إذا لم يكن لهذين الكائنين وجود فإن التابع سيعيد القيمة true.
ReferenceEquals (object , object)	bool	لا	نعم	يقوم هذا المنهج بمقارنة كائنين (البارامترين) ويتفحص فيما إذا كان كلا البارامترين يشيران إلى الكائن نفسه.
ToString()	String	نعم	لا	يعيد سلسلة نصية تمثل الاسم الكامل لصنف الكائن إلا أنه يمكننا تجاوز هذا المنهج وإعادة سلسلة نصية أخرى.
MemberwiseClone()	object	نعم	لا	يقوم بنسخ الكائن بإنشاء كائن جديد ونسخ كافة الأعضاء أيضا لاحظ أن عملية نسخ الأعضاء لا تقوم بإنشاء حالات جديدة لتلك الأعضاء وبالتالي فإن أي أعضاء من نوع مرجعي في الكائن الجديد ستشير إلى الكائنات نفسها في الكائن الأصلي إن هذا المنهج محمي وبالتالي يمكننا استخدامه ضمن الصنف فقط أو ضمن أي صنف يشتقه.
GetType()	System. Type	لا	لا	يعيد نوع الكائن على هيئة كائن من نوع System.Type
GetHashCode()	int	نعم	لا	يستخدم كتابع فرز للكائنات التي تتطلب ذلك حيث يعيد هذا التابع قيمة تعرف الكائن بهيئة مضغوطة.

تلك هي المناهج الأساسية التي تدعمها أنواع الكائنات في إطار عمل NET. وقد لا نستخدم أيا منها بتاتا ويمكن أن نستخدم بعضها في حالات خاصة جدا مثل المنهج GetHashCode().

إن المنهج GetType() مفيد جدا عند استخدام تعددية الاشكال باعتباره يسمح لنا بالقيام بعمليات محددة على الكائنات بالاعتماد على أنواعها وذلك بدلا من استخدام الشيفرة نفسها لجميع أنواع الكائنات التي قد نواجهها في حالة ما على سبيل المثال إذا كان لدينا تابع يتقبل بارامتر من نوع object مما يعني أنه يستطيع تمرير أي نوع من الكائنات فإنه يمكننا أن نقوم بعمليات إضافية إذا تعرفنا على نوع الكائن بصورة أدق وباستخدام المنهج GetType() والعامل typeof() وهو عامل في C# يقوم بتحويل اسم الصنف الذي يتبع له الكائن إلى كائن من نوع System.Type فإنه يمكننا القيام بعمليات مقارنة كما في الشيفرة:

```
if (typeof(myObj)==typeof (MyComplexClass))
{
    // myObj is an instance of the class MyComplexClass
}
```

إن كائن System.Type المعاد يتضمن إمكانيات عديدة أكثر من هذه العملية إلا أننا لن نتحدث عنه هنا وسوف نتحدث عن ذلك بتفصيل أكبر في فصول لاحقة.

ويمكننا ان نستفيد بصورة كبيرة من تجاوز المنهج ToString() وذلك في الحالات التي نود فيها تمثيل محتوى الكائن على هيئة سلسلة نصية بسهولة.

سوف نتعرف على مناهج الكائن System.object بصورة متكررة في الفصول القادمة لذا فإننا سنترك تفاصيل هذا الكائن ومناهجه للحديث عنها عند الحاجة إلى ذلك.

مناهج البناء والتدمير:

Constructors and Destructors:

عندما نعرف صنفا في C# فإننا نحتاج إلى تعريف منهج بناء وتدمير لهذا الصنف وعلى الرغم من أن الصنف الأساس System.Object يتضمن تزويدا افتراضيا لهذين المنهجين مما يعني أن هناك منهج تدمير ومنهج بناء لأي صنف في C# بصورة افتراضية إلا أنه قد نحتاج في أوضاع معينة إلى إنشاء مناهج بناء وتدمير مخصصة.

يأخذ منهج البناء نفس اسم الصنف ويمكننا ان ننشئ منهج بناء بصورة بسيطة كما يلي:

```
class MyClass
{
    public MyClass ()
    {
        // Constructor code
    }
}
```

وكما تلاحظ فإن لمنهج البناء نفس اسم الصنف الذي يحتويه ونلاحظ أن منهج البناء هنا لا يحتوي أية بارامترات وهذا يعني أن هذا المنهج هو المنهج الافتراضي الذي ستنفذ شيفرته عند إنشاء كائن من هذا الصنف ونلاحظ اننا استخدمنا مقيد الوصول العام public مما يعني أنه يمكننا إنشاء حالة مع كائنات هذا الصنف عد للفصل السابق للمزيد من المعلومات حول هذا.

يمكننا أن نستخدم منهج بناء افتراضي خاص وهذا يعني أنه لا يمكننا إنشاء كائنات من هذا الصنف باستخدام منهج البناء هذا وذلك كما يلي:

```
class MyClass
{
    private MyClass ()
    {
```

```
        // Constructor code
    }
}
```

ويمكننا أن نضيف منهج بناء غير افتراضي لصنفنا بنفس الأسلوب ويتم التمييز بين منهج البناء الافتراضي ومناهج البناء غير الافتراضية من خلال تواقيع هذه المناهج فمنهج البناء الافتراضي لا يحتوي على بارامترات:

```
class MyClass
{
    public MyClass()
    {
        // Constructor code
    }
    public MyClass (int myInt)
    {
        // non - default Constructor code (uses myInt)
    }
}
```

وليس هناك أي حدود لعدد مناهج البناء التي يمكننا إنشائها في الصنف.

تستخدم مناهج التدمير صيغة مختلفة قليلة عن صيغة تعريف مناهج البناء يسمى منهج التدمير في .NET. والذي يزود من قبل الصنف System.Object بالمنهج Finalize() إلا أن هذا لا يمثل اسم المنهج الذي سنستخدمه في تعريف مناهج تدمير لأصنافنا وإنما سنستخدم الصيغة التالية:

```
class MyClass
{
    ~MyClass()
    {
        // destructor body
    }
```

ستنفذ شيفرة منهج التدمير عندما يقوم CLR بتجميع النفايات مما يسمح بتحرير جزء من الموارد التي يحتلها الكائن وبعد استدعاء منهج التدمير سيتم استدعاء مناهج التدمير للأنواع الأساس المورثة بصورة ضمنية واحدة تلو الآخر بالإضافة إلى منهج التدمير Finalize() الموجود في الصنف الجذر System.Object لاحظ انه يمكننا تجاوز المنهج Finalize() والذي يمثل آخر منهج يقوم بتدمير الكائن من الذاكرة تماما وعند قيامنا بذلك فإن علينا القيام بعملية التدمير من الذاكرة بصورة يدوية أي بشكل صريح وهو أمر خطير سوف نتعلم كيفية استدعاء مناهج الصنف الأساس في الفصل القادم.

تسلسل تنفيذ مناهج البناء:

Constructor Execution Sequence:

إذا قمنا بعدة عمليات في مناهج البناء لصنف ما فإنه من المفيد ان نضع شيفرة هذه العمليات في مكان واحد ولذلك نفس الفوائد التي يمكن أن نحصل عليها من تقسيم الشيفرة ضمن توابع كما رأينا ذلك سابقا

في هذا الكتاب يمكننا القيام بذلك باستخدام منهج ما إلا أن لغة C# توفر طريقة بديلة أفضل فيمكننا أن نعد أي منهج بناء لاستدعاء منهج بناء آخر قبل تنفيذ شيفرة منهج البناء نفسه.

وقبل أن نتناول كيفية ذلك فإن علينا أن نفهم ما يحصل عندما نقوم بإنشاء حالة من صنف ما.

لكي نتمكن من إنشاء حالة من صنف مشتق لصنف ما فإنه يجب إنشاء حالة من الصنف الأساس قبل ذلك ولكي نتمكن من إنشاء حالة من الصنف الأساس هذا فإن علينا إنشاء حالة من صنفه الأساس أيضا وهكذا إلى أن نصل لإنشاء حالة من الصنف الجذر System.Object ومحصلة ذلك بغض النظر عن منهج البناء المستخدم لإنشاء حالة من صنف ما فإن منهج البناء في الصنف System.Object سينفذ أولا.

إذا استخدمنا منهج بناء غير افتراضي لصنف ما فإن السلوك الافتراضي هو استخدام منهج بناء الصنف الأساس الذي يطابق توقيعه منهج بناء الصنف المشتق فإن لم يكن هناك منهج بناء مطابق عندها سيستدعى منهج البناء الافتراضي للصنف الأساس وهو الأمر الذي يحدث دائما عند الوصول إلى الصنف الأساس System.Object وذلك لأن ليس لهذا الصنف مناهج بناء غير منهج البناء الافتراضي.

لنأخذ مثالا سريعا يستعرض لنا تسلسل تنفيذ الأحداث لتكن لدينا هرمية الأصناف التالية:

```
public class MyBaseClass
{
    public MyBaseClass ()
    {
    }
    public MyBaseClass (int i)
    {
    }
}
 public class MyDerivedClass:MyBaseClass
 {
     public MyDerivedClass ()
     {
     }
     public MyDerivedClass (int i)
     {
     }
     public MyDerivedClass (int i,int j)
     {
     }
 }
```

إذا قمنا بإنشاء حالة من الصنف MyDerivedClass بالصورة التالية:

```
MyDerivedClass myObj = new MyDerivedClass();
```

فإن تسلسل ما سيحدث عند تنفيذ هذا السطر هو بالشكل التالي:

- تنفيذ منهج البناء System.Object.Object() .
- تنفيذ منهج البناءة MyBaseClass.MyBaseClass().

- تنفيذ منهج البناء MyDerivedClass.MyDerivedClass().

وبصورة بديلة إذا أنشأنا حالة من الصنف MyDerviedClass كما يلي:

```
MyDerivedClass myObj = new MyDerivedClass(4);
```

فإن تسلسل التنفيذ سيأخذ الشكل الآتي:

- تنفيذ منهج البناء System.Object.Object() .
- تنفيذ منهج البناءة MyBaseClass.MyBaseClass(int i).
- تنفيذ منهج البناء MyDerivedClass.MyDerivedClass(int i).

وأخيرا إذا أنشأنا حالة من الصنف نفسه بالشكل:

```
MyDerivedClass myObj = new MyDerivedClass(4 , 8);
```

فإن تسلسل ما سيحدث عند تنفيذ هذا السطر هو بالشكل التالي:

- تنفيذ منهج البناء System.Object.Object() .
- تنفيذ منهج البناءة MyBaseClass.MyBaseClass().
- تنفيذ منهج البناء MyDerivedClass.MyDerivedClass(int i , int j).

إن هذا النظام يعمل بشكل جيد ويتأكد من أن أي أعضاء موروثة يمكن أن تصل إلى منهاج البناء في الأصناف المشتقة إلا أن هناك في بعض الأحيان حالات نحتاج فيها لمزيد من التحكم في مناهج البناء المستدعاة على سبيل المثال قد نرغب في المثال الأخير بتنفيذ تسلسل يأخذ الصورة التالية:

- تنفيذ منهج البناء System.Object.Object() .
- تنفيذ منهج البناءة MyBaseClass.MyBaseClass(int i).
- تنفيذ منهج البناء MyDerivedClass.MyDerivedClass(int i , int j).

سيؤدي هذا النوع من التسلسل إلى استخدام البارامتر int i في منهج البناء MyBaseClass(int i) وهذا يعني أنه سيكون لمنهج البناء MyDerivedClass(int i , int j) عمل أقل سيقوم به ولن يحتاج في حالة كهذه إلا لمعالجة البارامتر j حيث سيتولى منهج البناء MyBaseClass(int i) مهمة معالجة البارامتر int i المذكور في بارامتر المنهج MyDerivedClass(int i , int j) وتمكننا لغة C# من اتباع هذا النوع من السلوك إذا شئت.

ملاحظة:

لقد افترضنا هنا أن يكون البارامتر int i في منهج البناء MyDerivedClass(int i , int j) له المعنى نفسه للبارامتر int i في منهج البناء MyBaseClass(int i) وهذا الأمر ليس صحيحا دائما إلا أن الأمر سيكون بهذه الصورة في أغلب الأحيان عندما نصادف حالة كهذه.

وللقيام بذلك فإن علينا ان نحدد ذلك ضمن منهج البناء للصنف المشتق وحسب المثال هذا فإننا ستعدل توقيع منهج البناء MyDerivedClass(int i , int j) كما يلي:

```
public class MyDerivedClass:MyBaseClass
{
    public MyDerivedClass ()
    {
    }
    public MyDerivedClass (int i)
    {
    }
    public MyDerivedClass (int i,int j) : base(i)
    {
    }
}
```

تشير الكلمة المفتاحية base هنا إلى أنه سيتم استخدام منهج بناء الصنف الأساس الموافق للتوقيع المحدد لقد استخدمنا هنا بارامتر من نوع int واحد وبالتالي فإن منهج البناء MyBaseClass(int i) سيستخدم عندها إن القيام بذلك يعني أن منهج البناء الافتراضي MyBaseClass() لن يستدعى وبالتالي ستأخذ عملية إنشاء حالة من الصنف MyDerivedClass التسلسل الأخير الذي أوردناه مسبقا وهو ما أردنا الحصول عليه تماما.

يمكننا استخدام الكلمة base لتحديد قيم حرفية مباشرة لمناهج بناء الصنف الأساس فلربما كنا نود استخدام منهج البناء الافتراضي MyDerivedClass لاستدعاء منهج البناء غير الافتراضي للصنف MyBaseClass:

```
    public MyDerivedClass (int i,int j) : base(5)
    {
    }
```

وهذا سيعطينا تسلسل التنفيذ التالي:

- تنفيذ منهج البناء System.Object.Object() .
- تنفيذ منهج البناءة MyBaseClass.MyBaseClass(5).
- تنفيذ منهج البناء MyDerivedClass.MyDerivedClass(int i , int j).

وبالإضافة إلى الكلمة base فإن هناك كلمة مفتاحية أخرى يمكننا استخدامها هنا وهي this تقوم هذه الكلمة باستخدام منهج البناء غير الافتراضي للصنف الحالي قبل استدعاء منهج البناء الحالي وذلك كما يلي:

```
public class MyDerivedClass:MyBaseClass
{
    public MyDerivedClass () : this(4,5)
    {
    }
    public MyDerivedClass (int i)
    {
```

```
        }
        public MyDerivedClass (int i,int j) : base(i)
        {
        }
    }
```

تستخدم الكلمة this للإشارة إلى الصنف الحالي بينما تستخدم الكلمة base للإشارة إلى الصنف الأساس المورث وبناء على ذلك فإن تسلسل التنفيذ هنا سيأخذ الشكل التالي:

- تنفيذ منهج البناء System.Object.Object() .
- تنفيذ منهج البناءة MyBaseClass.MyBaseClass(int i).
- تنفيذ منهج البناء MyDerivedClass.MyDerivedClass(int i , int j).
- تنفيذ منهج البناء MyDerivedClass.MyDerivedClass().

والمحدودية الوحيدة في كل ذلك هو أنه لا يمكننا تحديد إلا منهج بناء واحد فقط باستخدام الكلمتين base أو this لكن وكما يظهر لنا في المثال السابق فإن تلك لا تعد بمشكلة في حد ذاتها باعتبار أنه ما يزال بإمكاننا استخدام أشكال مختلفة من تسلسل تنفيذ مناهج البناء.

أدوات OOP في Visual Studio.NET:

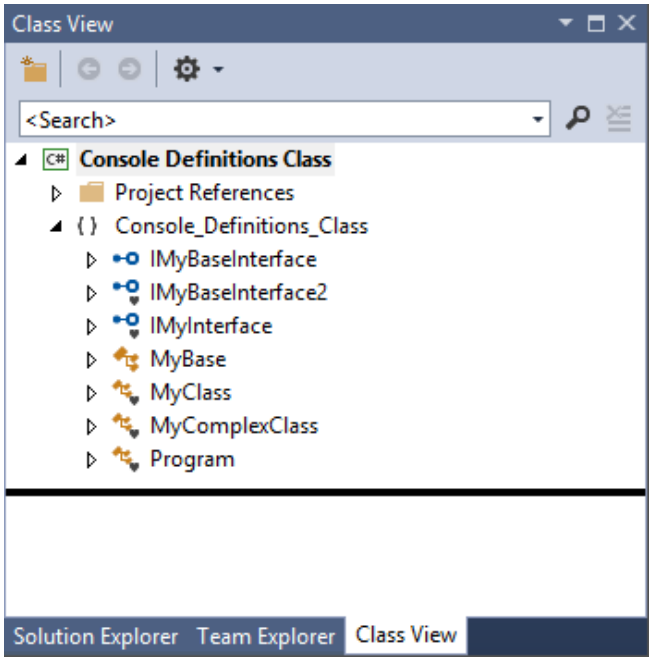
OOP Tools in Visual Studio.NET:

بما أن البرمجة كائنية التوجه تعد موضوعا جوهريا في إطار عمل .NET. فإن هناك العديد من الأدوات التي يوفرها VS والموجهة لتطوير تطبيقات OOP وسوف نتناول في هذا القسم بعضا منها.

إطار عرض الصنف Class View:

The Class View Window:

لقد تعرفنا على إطار مستعرض الحل Solution Explorer في الفصل الثاني من الجزء الأول ورأينا أن هناك إطار آخر يتشارك مع هذا الإطار في نفس النافذة ويسمى بإطار Class View يستعرض هذا الإطار شجرة أصناف تطبيقاتنا ويمكننا من رؤية مواصفات هذه الأصناف التي نستخدمها وأعضائها أيضا هناك عدد من أنماط العرض المختلفة لهذه المعلومات والنمط الافتراضي هو Sort By Alphabetically أي الترتيب حسب حروف الهجاء فبالنسبة لشجرة المثال في القسم السابق سيعرض هذا الإطار شجرة الصنف الرئيسي Console Definitions Class بالشكل التالي:



الشكل (4-2)

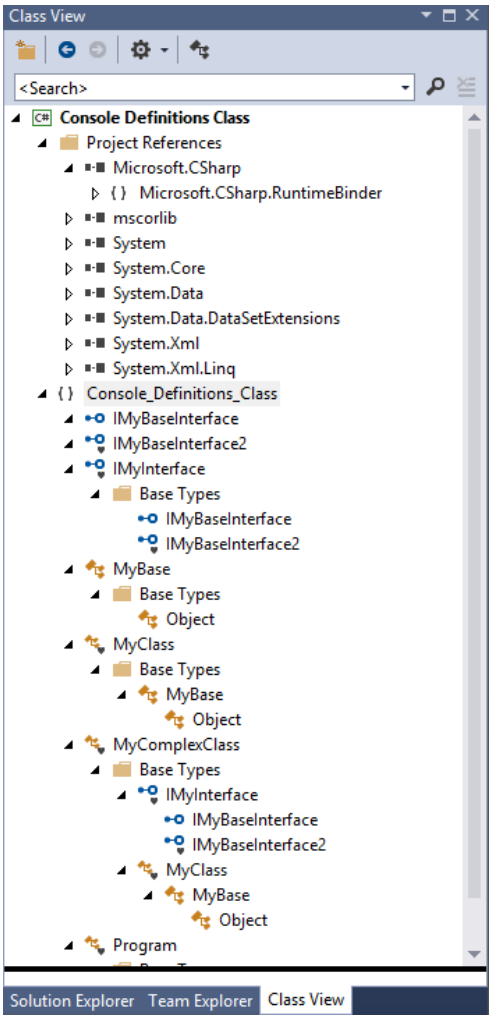
هناك عدد من الأيقونات التي نلاحظها ضمن الإطار ولكل شكل من أشكال هذه الايقونات مدلول خاص بها وهي كما يلي:

شكل الايقونة	الوصف (العربي)	الوصف (الإنكليزي)
	مشروع	Project
	فضاء أسماء	Namespace
	واجهة	Interface
	صنف	Class
	منهاج	Method
	خاصية	Property
	كائن	Object
	مقيد الوصولية	Access
	نوع البيانات	Data Type
	تعدادات	Enumeration
	عنصر من تعداد	Enumeration Item
	حدث	Event

لاحظ استخدام بعض هذه الأيقونات لوصف أنواع أخرى غير الأصناف مثل التعدادات وأنواع Struct هناك بعض العناصر التي قد يكون لها رموز أخرى متوضعة في الأسفل وتشير إلى مستوى الوصول إلى هذه العناصر حيث لن يظهر أي رمز للعناصر العامة public:

شكل الايقونة	الوصف (العربي)	الوصف (الإنكليزي)
	خاص	Private
	داخلي	Internal
	محمي	Protected

وليس هناك أية رموز تحدد العناصر الظاهرية Virtual أو المنغلقة Sealed أو حتى المجردة abstract إن جميع أنماط العرض المتوفرة في هذا الإطار تستخدم بنفس الأسلوب وتمكننا من توسعة تعاريف الأصناف باستخدام عناصر تحكم عرض شجرية إن توسعة صنفنا والواجهات وصولا إلى الصنف System.Object سيقودنا إلى إطار Class View كما في الشكل (5-2):



الشكل (5-2)

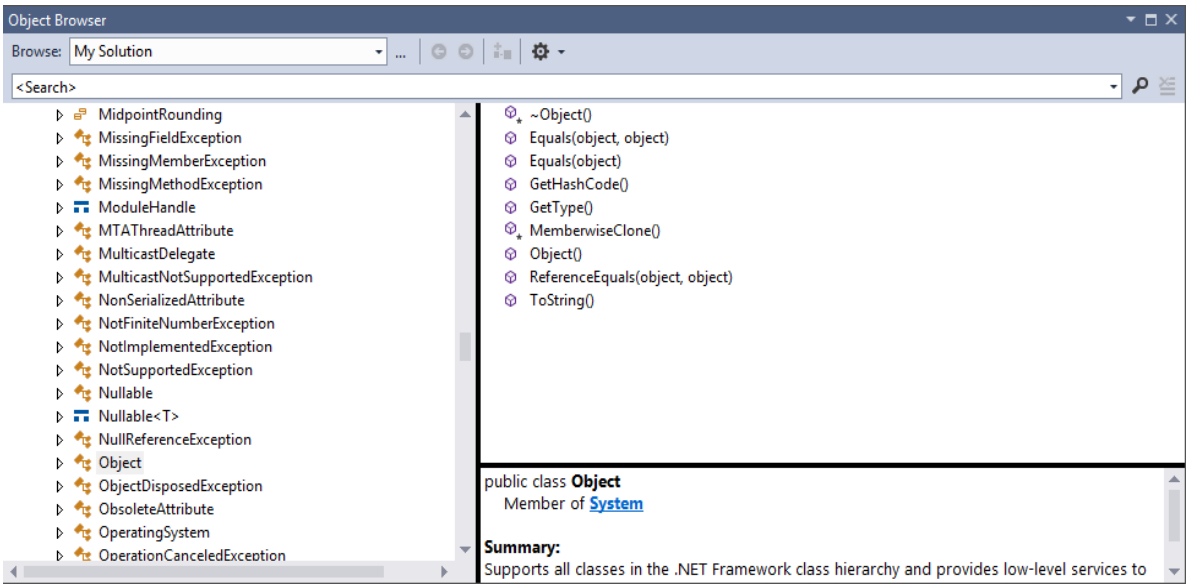
ويمكننا ان نرى جميع معلومات الأصناف والواجهات في المشروع وبالإضافة إلى إمكانية استعراض معلومات الأصناف والواجهات وأعضائها فبإمكاننا الوصول إلى الشيفرة الموافقة لهذه العناصر وهي

عادة سطر تعريف هذه العناصر بمجرد النقر المزدوج على العنصر أو بالضغط بزر الفأرة الأيمن ومن ثم اختيار Go to Definition من القائمة المنسدلة إن هذا سيحصل إذا كان هناك فعلا تعريف لهذا العنصر ضمن الشيفرة البرمجية للحل أو المشروع المفتوح حاليا وإن لم يكن هناك تعريف لهذا العنصر كالشيفرة الموجودة ضمن النوع الأساس System.Object فسننتقل إلى إطار جديد باسم Object Browser.

مستعرض الكائنات:

The Object Browser:

تمثل نافذة Object Browser إصدارة موسعة لإطار Class View حيث يوفر مستعرض الكائنات إمكانية استعراض الأصناف الأخرى الموجودة في مشروعنا إضافة إلى إمكانية عرض أصناف خارجية أيضا يمكننا الوصول إلى مستعرض الكائنات باختيار الأمر Object Browser من القائمة View في شريط القوائم المنسدلة لـ Visual Studio.NET 2013 يبين الشكل التالي نافذة Object Browser وهي تظهر ضمن الإطار الرئيسي وبالتالي يمكننا عرض هذه النافذة وإطار Class View في نفس الوقت:



الشكل (6-2)

تبين لنا هذه النافذة الأصناف وأعضاءها في منطقتين بعكس إطار Class View الذي يستعرض الأصناف وأعضاءها في مكان واحد هذا بالإضافة إلى أنه يمكننا الوصول إلى وحدات .NET المستخدمة في تطبيقنا أيضا وليس مجرد الأصناف الموجودة في مشروعنا وحسب وبالتالي يمكننا استعراض محتوى فضاء الأسماء System مثلا باستخدام هذه النافذة.

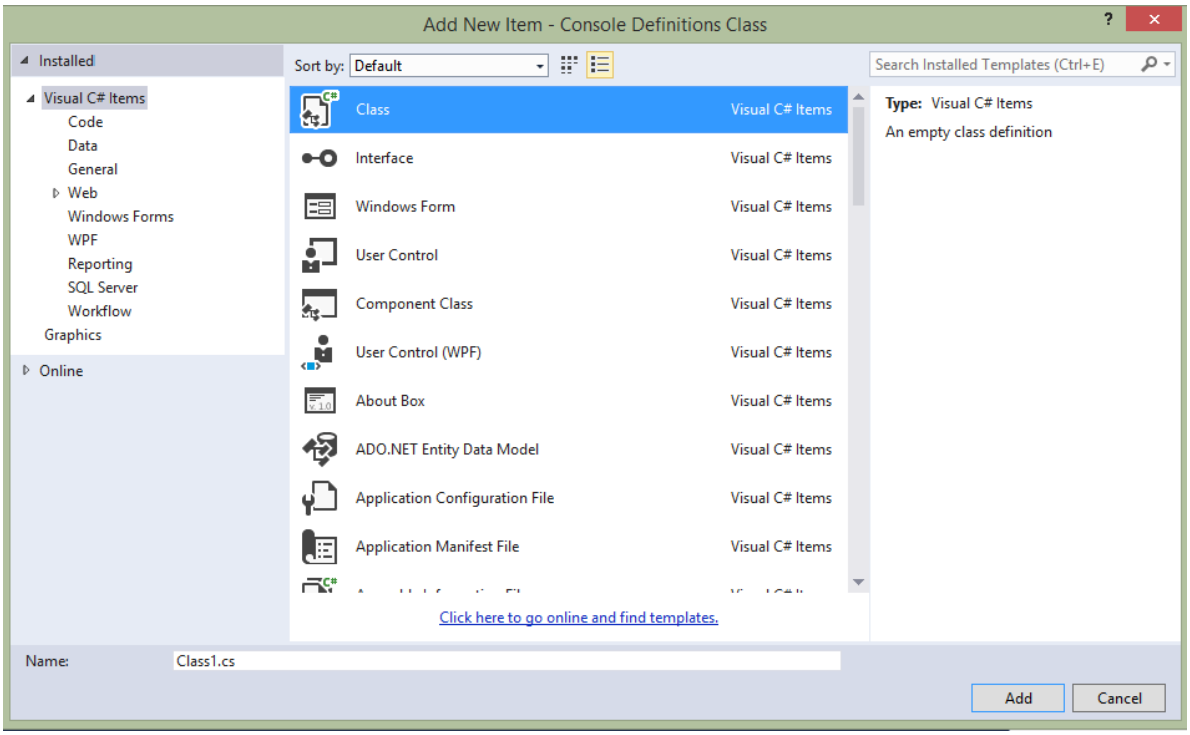
عند اختيار عنصر من إحدى القائمتين هنا سيتم استعراض معلومات عن العنصر الذي تم اختياره في أسفل يمين النافذة ويمكننا أن نجد في هذا الشكل المعلومات التي تستعرضها هذه النافذة عند اختيار الصنف Program من تطبيقنا Console Definitions Class ويمكننا أن نجد أيضا معلومات تلخيصيه عن العنصر مبوبة تحت العنوان Summary في الحقيقة يتم توليد هذه المعلومات من خلال تعليقات مكتوبة بهيئة XML ضمن الشيفرة المصدرية لهذه الأصناف سوف نتحدث عن تعليقات XML بتفصيل أكبر في الفصول القادمة.

إضافة الأصناف:

Adding Classes:

يتضمن Visual Studio.NET 2013 مجموعة أدوات تسهل تأدية المهام الشائعة وبعض هذه المهام ملائمة للاستخدام مع تقنيات OOP وأحد هذه الأدوات يمكننا من إنشاء أصناف جديدة في تطبيقنا بسرعة.

يمكننا الوصول إلى هذه الأداة باختيار الأمر Add Class من القائمة PROJECT أو باختيار الأمر Add New Item من نفس القائمة أيضا أو بالنقر بزر الفارة الأيمن على مشروعنا في إطار Solution Explorer واختيار الأمر New Item او Class من القائمة الفرعية للأمر Add من القائمة المنسدلة سيظهر عندئذ صندوق حوار كما في الشكل (2-7)



الشكل (2-7)

يمكنك عندئذ تحديد البند Class ويمكنك تحديد اسم الصنف في مربع النص في الأسفل وليكن MyNewClass وعند الانتهاء اضغط على زر Add سيتم بواسطته إنشاء صنف سيأخذ نفس الاسم الذي حددته في صندوق الحوار هذا.

لقد أضفنا في المثال السابق من هذا الفصل تعريفات الأصناف إلى الملف Program.cs بصورة يدوية إن وضع الأصناف في ملفات منفصلة يساعدنا في الغالب على تسهيل التعامل معها.

إن ما قمنا به في صندوق الحوار السابق سيؤدي إلى فتح ملف باسم MyNewClass.cs يحتوي على الشيفرة التالية:

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

namespace Console_Definitions_Class
{
    class MyNewClass
    {
    }
}
```

لاحظ أنه قد تم تعريف هذا الصنف ضمن فضاء الأسماء للمشروع نفسه أي Console Definitions والذي تم تعريف الصنف Program الذي يمثل صنف نقطة الدخول للمشروع ضمنه أيضا وبالتالي يمكننا أن نستخدم هذا الصنف من ضمن شيفرة صنفنا MyNewClass تماما كما لو كان الصنفان ضمن الملف نفسه.

توضيح:

نسمي الصنف الذي يحتوي على التابع ()Main في تطبيق Console بصنف نقطة الدخول للتطبيق.

مشاريع مكتبات الأصناف:

Class Library Projects:

وكما يمكننا أن نضع الأصناف في ملفات منفصلة ضمن تطبيقنا فإنه يمكننا أن نضعها ضمن مشاريع منفصلة تماما يسمى المشروع الذي لا يتضمن إلا عددا من الأصناف بالإضافة إلى بعض أنواع التعريفات المرتبطة الأخرى ولا يحتوي على نقطة دخول بمكتبة الأصناف Class Library.

تتم ترجمة مشاريع مكتبات الأصناف إلى مجمعات (ملفات) لها الامتداد dll. ويمكننا الاستفادة من محتويات المجمعات assemblies بإضافة مراجع لها في المشاريع التي ستستخدمها والتي يمكن أن تمثل جزءا من الحل نفسه إلا أنه ليس بالضرورة أن يكون ذلك.

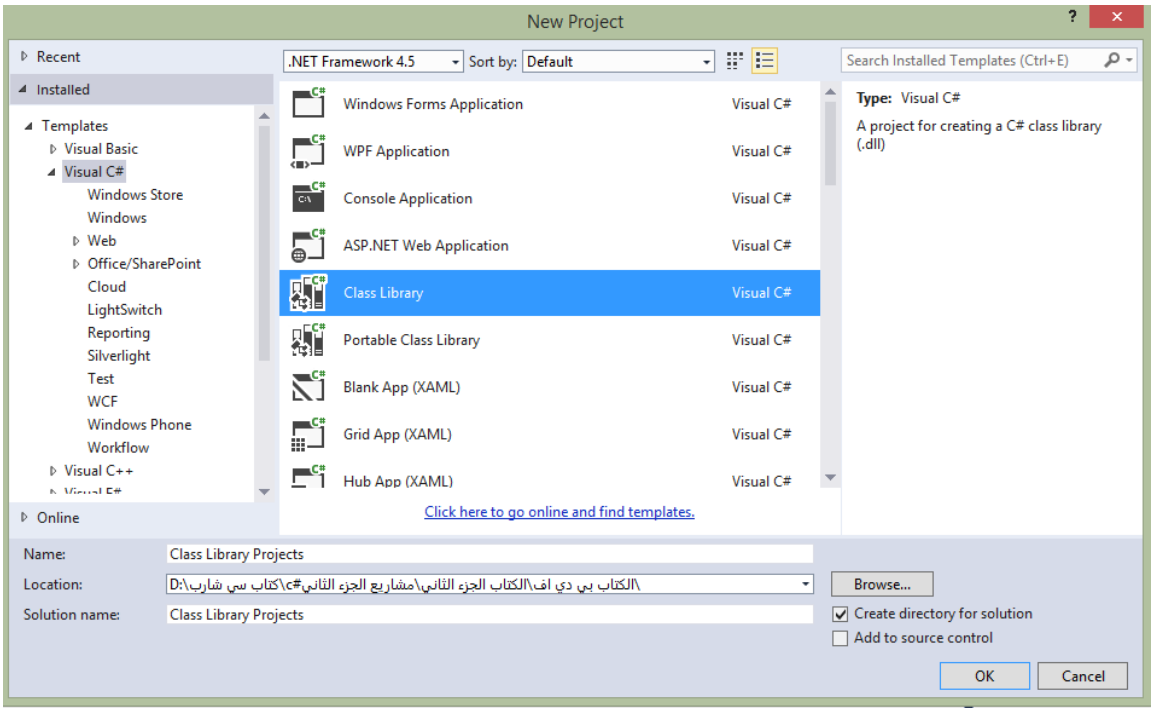
فكرة:

إن وضع الأصناف ضمن المجمعات بهذه الصورة يمثل إحدى التقنيات الرائعة المستخدمة لإعادة استخدام الشيفرة المصدرية في أكثر من تطبيق فيمكنك مثلا أن تبرمج عددا من الأصناف التي تقوم باحتساب القيمة الأكبر والأصغر والوسطى للمصفوفات وتضعها ضمن صنف واحد ومن ثم تقوم ببرمجة عدد أخر من الأصناف المختصة بالعمليات الحسابية أيضا عندئذ يمكنك أن تضع هذه الأصناف جميعها ضمن مجمعة باسم Math.dll وتستخدمها في أي تطبيق لك في .NET.

لنلق الآن نظرة على مثال بسيط نستعرض فيه كيفية إنشاء مكتبة أصناف ومشروع آخر منفصل يستخدم الأصناف الموجودة في هذه المكتبة.

تطبيق حول استخدام مكتبة الأصناف:

1- قم بإنشاء مشروع جديد من النوع Class Library وليكن اسمه Class Library Projects.



الشكل (8-2)

2- غير اسم الملف Class1.cs إلى الاسم MyExternalClass.cs يمكنك القيام بذلك بالنقر بزر الفأرة الأيمن على الملف في الإطار Solution Explorer واختيار Rename من القائمة المنسدلة.

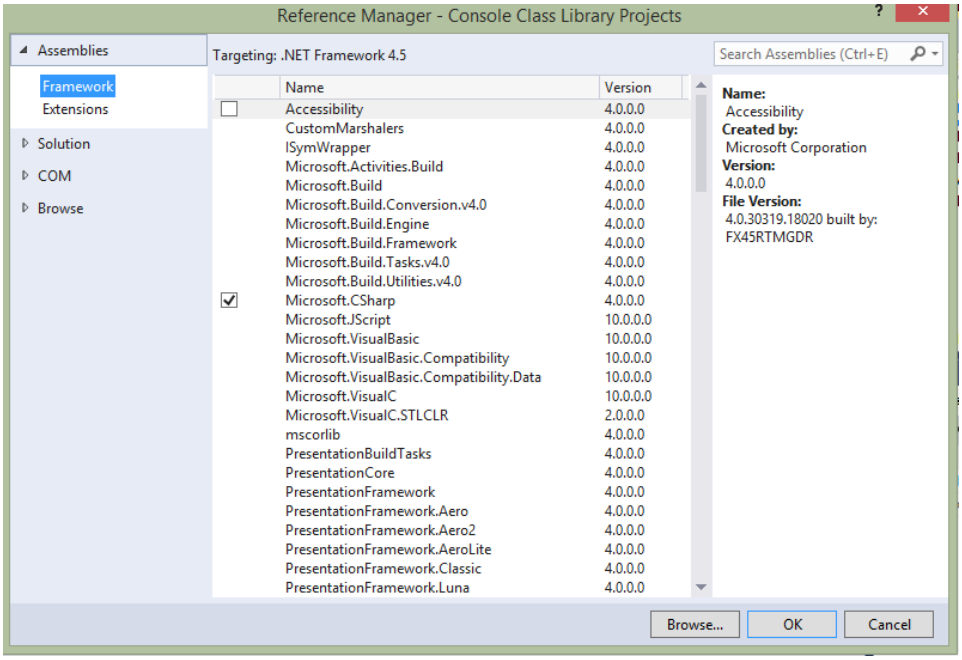
3- عدل الشيفرة في الملف MyExternalClass.cs لكي يعكس التغيير الذي أحدثناه في اسم الصنف:

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

namespace Class_Library_Projects
{
    public class MyExternalClass
    {
        public MyExternalClass()
        {
        }
    }
}
```

4- قم بإضافة صنف جديد إلى المشروع وليكن له الاسم MyInternalClass.cs.
5- عدل شيفرة الصنف الجديد وذلك لجعله صنفا داخليا:

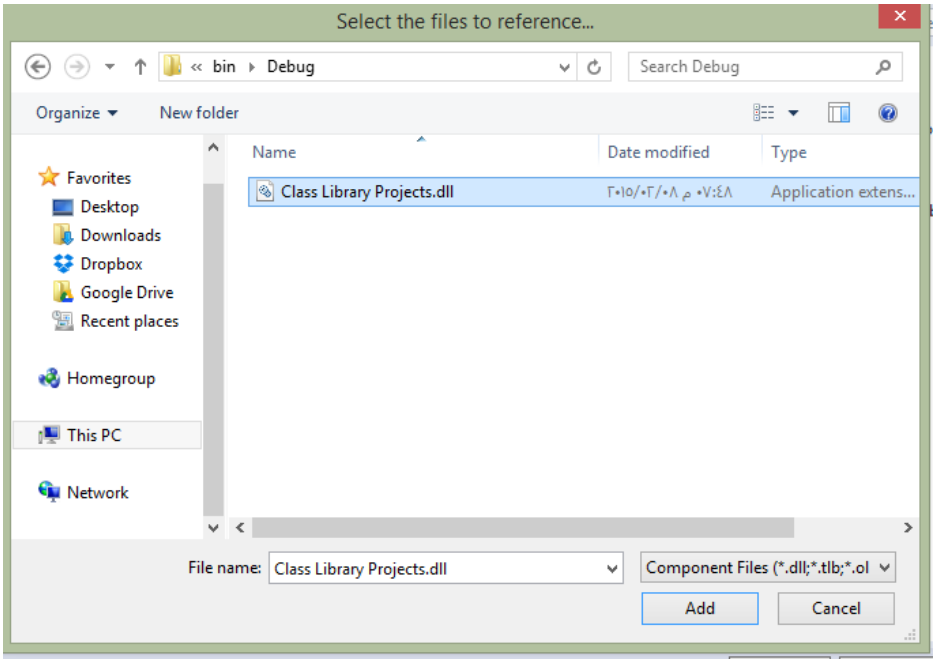
```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Class_Library_Projects
{
    internal class MyInternalClass
    {
        public MyInternalClass ()
        {
        }
    }
}
```

6- ترجم المشروع (لاحظ انه ليس لهذا المشروع نقطة دخول وبالتالي لن تستطيع تنفيذ هذا المشروع بالطريقة العادية كما كنا نقوم بذلك في السابق وبدلا من ذلك فإننا سنقوم ببناء المشروع باختيار الأمر Build Solution من القائمة BUILD).
7- من القائمة File اختر الأمر New ثم اختر الامر الفرعي Project واختر تطبيق Console غير اسم التطبيق باسم Console Class Library Projects.
8- اختر الامر Add Reference من القائمة PROJECT أو اختر الامر Reference من القائمة Add بعد النقر بزر الفارة الأيمن على إطار Solution Explorer من القائمة المنسدلة.



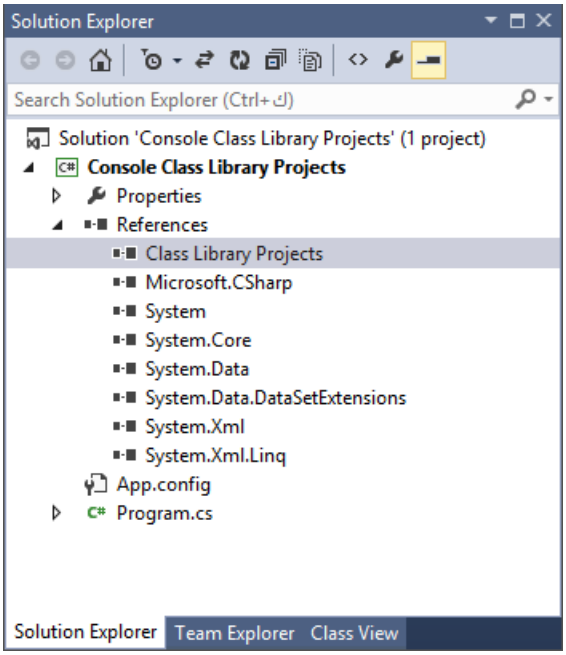
الشكل (2-9)

9- انقر على الزر Browse ثم انتقل إلى المجلد الذي حفظت ضمنه مشروع Class Library Projects ومن ثم انتقل إلى المجلد bin ثم إلى المجلد Debug ثم قم باختيار الملف Class Library Projects.dll ثم اضغط على الزر Add ثم الزر ok:



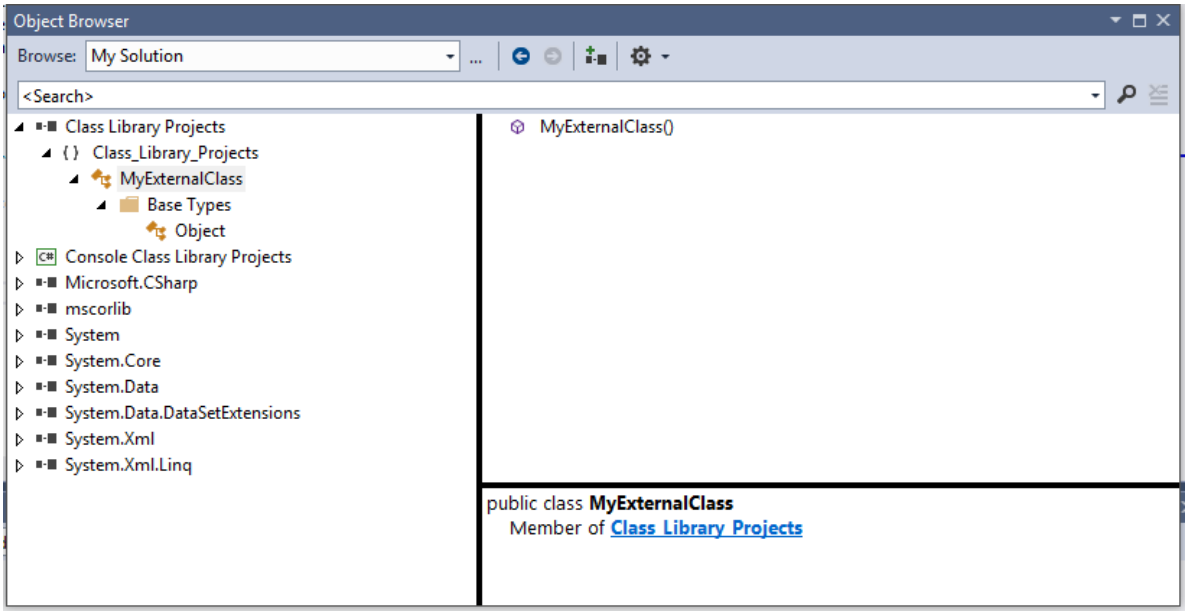
الشكل (2-10)

10- عند انتهاء العملية تأكد من ان هناك مرجعا قد تمت إضافته في إطار Solution Explorer.



الشكل (2-11)

11- افتح نافذة مستعرض الكائنات Object Browser وتفحص المرجع الجديد لرؤية الأصناف والأعضاء التي يتضمنها.

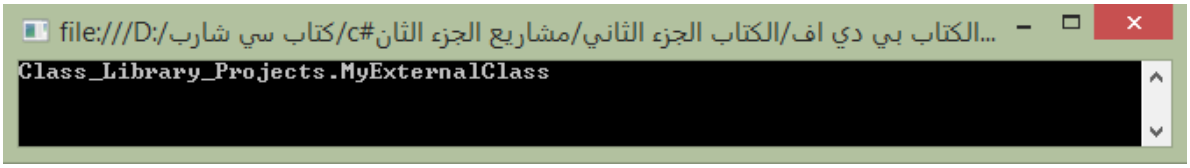


الشكل (2-12)

12- عدل شيفرة الملف Program.cs كما يلي:

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

namespace Console_Class_Library_Projects
{
    class Program
    {
        static void Main(string[] args)
        {
            MyExternalClass myObj = new MyExternalClass();
            Console.WriteLine(myObj);
            Console.ReadLine ();
        }
    }
}
```

13- نفذ التطبيق وستظهر نافذة Console كما يلي:



الشكل (2-13)

كيفية العمل:

How to Work:

لقد قمنا في هذا المثال بإنشاء تطبيقين يمثل التطبيق الأول مشروعا لمكتبة أصناف والأخر يمثل تطبيق Console يتضمن مشروع مكتبة الأصناف Class Library Projects صنفين MyExternalClass والذي له وصولية عامة والصنف MyInternalClass والذي له وصولية داخلية أما التطبيق الثاني Console Class Library Projects فهو تطبيق Console والذي يتضمن شيفرة بسيطة هدفها استخدام مكتبة الأصناف تلك.

ولكي نتمكن من استخدام الأصناف الموجودة في المشروع Class Library Projects فإن علينا ان نضيف مرجعا لها في التطبيق الذي يود استخدامها وهو التطبيق Console Class Library Projects في حالتنا هنا أي أننا سنضيف مرجعا للملف Class Library Projects.dll المولد عند بناء التطبيق ولتبسيط الأمور في هذا المثال فإننا اكتفينا بمجرد إضافة مرجع لهذا الملف تطبيق Console وعلى الرغم

من أنه يمكننا أن ننسخ الملف Class Library Projects.dll إلى نفس مجلد التطبيق Console Class Library Projects إلا أننا نود الاستمرارية في تطوير مكتبة التطبيقات تلك وهي في مكانها.

توضيح:

تتطلب عملية تحديث ملف dll لهذه المكتبة عناء كبيرا يكفي أن ننسخ الإصدارة الحديثة من الملف فوق الإصدارة القديمة وتستخدم الجديدة مباشرة! آه إن هذا يعني أن عملية تثبيت تطبيقات C# لا تتطلب سوى نسخ الملفات إلى المكان الذي نود تنفيذها منه فعلا إن الحياة تصبح أبسط بكثير مع إطار عمل .NET.

وبعد أن أضفنا مرجعا لمكتبة الأصناف تلك فإننا قمنا باستكشاف الأصناف المتوفرة ضمنها من خلال نافذة Object Browser لكن لاحظ أنه لا وجود إلا لصنف واحد فقط هو MyExternalClass مع العلم ان مكتبة الأصناف تحتوي على صنفين إذن أين الصنف الثاني؟

لا تنسى أننا عرفنا الصنف الثاني على أنه صنف داخلي internal وهو الصنف MyInternalClass ولذلك فإن هذا الصنف لن يعرض ضمن نافذة مستعرض الكائنات Object Browser وهذا يعني أن الصنف MyExternalClass هو الصنف الوحيد في مكتبة الأصناف Class Library Projects الذي يمكننا الوصول إليه من خلال تطبيقاتنا الأخرى أي من خلال شيفرة خارجية مثل تطبيق Console هذا.

يمكننا أن نستبدل شيفرة التابع Main() في تطبيق Console بالشيفرة التالية والتي تحاول استخدام الصنف الداخلي:

```
MyInternalClass myObj = new MyInternalClass();
Console.WriteLine(myObj);
Console.ReadLine ();
```

إلا ان عملية كهذه ستؤدي إلى رسالة خطأ من المترجم.

عن مسألة استخدام الأصناف الموجودة في المجمعات من خلال مجمعات أخرى تمثل المفتاح للبرمجة في C# وإطار عمل .NET. وفي الحقيقة إن هذا ما نقوم به عندما نستخدم الأصناف المعرفة في مكتبة أصناف إطار عمل .NET.

توضيح:

لاحظ أن إطار عمل .NET. هو مجموعة من مكتبات الأصناف أي مجموعة من المجمعات وأشياء أخرى تمت كتابتها مسبقا بحيث تسهل الكثير من المهام البرمجية علينا.

الواجهات مقابل الأصناف المجردة:

Interfaces vs. Abstract Classes:

لقد رأينا في هذا الفصل كيف يمكننا ان ننشئ الواجهات والأصناف المجردة كما قلنا فإننا لن نتحدث عن الأعضاء في هذا الفصل وربما قد لاحظت ان الصنف المجرد والواجهة متشابهان في عدة نقاط ومن الجدير بالذكر بنا أن نتناول جوانب كلا منها وأن نبين متى يتوجب علينا استخدام إحداهما بدلا من الأخرى ولنبدأ أولا بأوجه الشبه.

يمكن أن تحتوي الأصناف المجردة والواجهات على أعضاء يمكن توريثها لصنف أخر ولا يمكن أن ننشئ حالة من الواجهات ولا حتى الأصناف المجردة بصورة مباشرة وإنما يمكننا ذلك بواسطة التصريح عن متحولات من هذه الأنواع أي من نوع الواجهة أو الصنف المجرد وفي كلتا الحالتين فإننا لن نتمكن من استخدام أعضاء هذه الأنواع إلا من خلال تلك المتحولات على الرغم من أننا لا نستطيع الوصول المباشر على الأعضاء الأخرى للكائن المشتق.

لنتحدث الآن عن الاختلافات.

يمكن للأصناف أن ترث من صنف أساس وحيد وهذا يعني انه لا يمكن للصنف الواحد أن يرث إلا صنفا مجردا واحدا فقط بصورة مباشرة وبالعكس يمكن أن يزود الصنف بأي عدد من الواجهات إلا أن هذا لا يوضح أي فرق جوهري بينهما حتى الآن.

توضيح:

يمكن للصنف أن يرث أكثر من صنف مجرد واحد لكن ذلك يحدث بصورة غير مباشرة نتيجة للوراثة من الأصناف ذات المستويات الأعلى في هرمية الوراثة كأن يرث الصنف A صنفا مجردا ومن ثم يرث صنف آخر الصنف A.

يمكن أن يحتوي الصنف المجرد على أعضاء مجردة وغير مجردة أيضا أما أعضاء الواجهة فيجب أن يتوفر لها جميعا تزويد ضمن الصنف المجرد بهذه الواجهة فالواجهة لا تحتوي على أية كتل برمجية وإنما مجرد تعريفات للأعضاء فقط وكذلك فإن أعضاء الواجهة هي أعضاء عامة حسب تعريف الواجهة باعتبار أن وجودها للاستخدام الخارجي فقط إلا أن أعضاء الأصناف المجردة يمكن أن تكون خاصة طالما أنها ليست أعضاء مجردة أو محمية أو داخلية محمية وبالإضافة إلى ذلك فإن الواجهات لا يمكن أن تحتوي على حقول أو مناهج بناء أو تدمير أو أعضاء ستاتيكية أو حتى متحولات ثابتة.

ملاحظة:

نعيد ونذكر بأن ما نقصده بتزويد جميع أعضاء الواجهة ضمن الصنف المزود بهذه الواجهة يعني كتابة شيفرة لهذه الأعضاء ضمن الصنف تحدد سلوك هذه الأعضاء.

ملاحظة:

يجب أن تنتبه إلى أنه من الممكن أيضا أن تتضمن الأصناف أعضاء مجردة وغير مجردة.

توضيح:

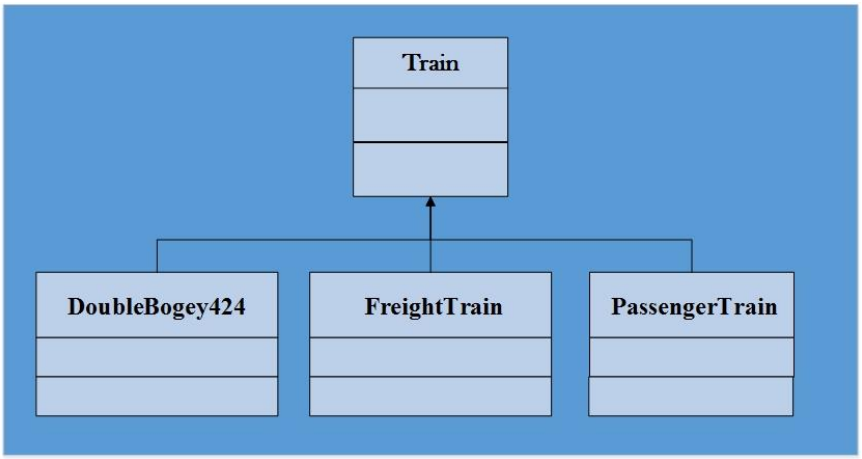
الأعضاء المجردة هي تلك الأعضاء التي لا تحتوي على شيفرة وإنما على مجرد تعريف للعضو فقط.

وبالتالي يجب أن تتضمن الأصناف التي ترث هذه الأصناف تزويدا لهذه الأعضاء ضمنها إلا إذا كان الصنف المشتق هو أيضا صنف مجرد.

أما الأعضاء غير المجردة فهي أعضاء تحتوي على كتلة شيفرة ويمكن أن تكون ظاهرية Virtual وبالتالي يمكن تجاوزها في الصنف المشتق.

وخلاصة ذلك أن هذين النوعين يستخدمان لأغراض مختلفة فالصنف المجرد يستخدم أساس للكائنات التي تشترك مع بعضها البعض بصفات مشتركة أما الواجهة فهي مصممة لكي تستخدم من قبل الأصناف التي يمكن ان تختلف عن بعضها في مستويات أساسية إلا أن هذه الأصناف تقوم ببعض العمل نفسه.

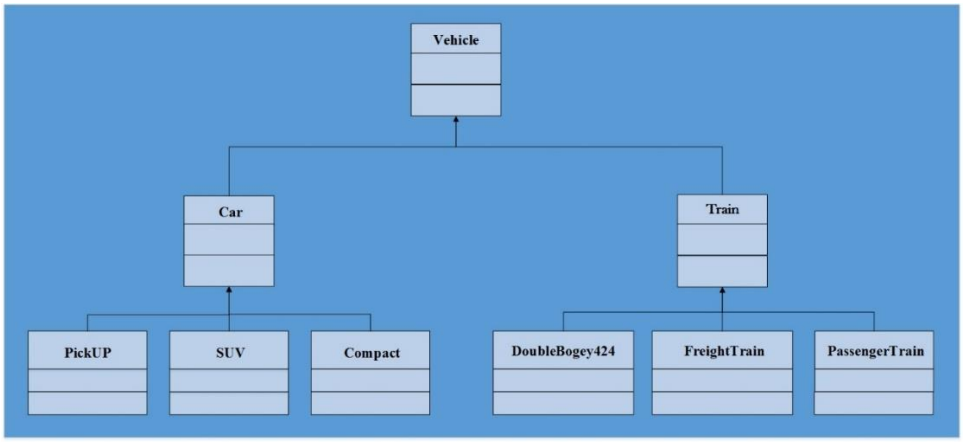
وكمثال على ذلك لنفترض أن لدينا عائلة من الكائنات تمثل مجموعة من القطارات يتضمن الصنف الأساس Train للقطارات على التعريف الأساسي للقطار مثل عرض السكة الحديدية ونوع المحرك والذي يمكن أن يكون محركا بخاريا أو كهربائيا وذلك يمثل صنفا مجردا باعتبار أنه ليس هناك ما يعرف بالقطار العام ولكي نتمكن في الحقيقة من إنشاء قطار حقيقي علينا أن نضيف خصائص محددة لهذا القطار وللقيام بذلك فإن علينا أن نشتق أصنافا مثل صنف قطار الركاب PassengerTrain أو صنف قطار الشحن FreightTrain أو قطار المناجم DoubleBogey424.



الشكل (2-14)

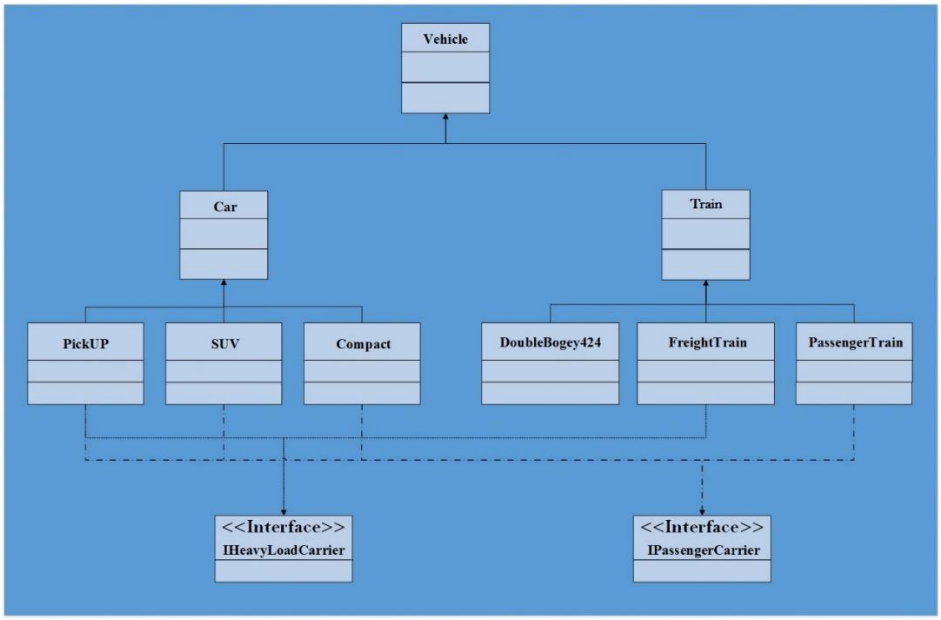
ويمكننا أن نعرف عائلة كائنات السيارات بنفس الأسلوب أيضا وذلك باستخدام صنف مجرد Car ويمكن أن نكون أصنافا مشتقة من هذا الصنف مثل Compact أو SUV أو PickUP.

ويمكننا أيضا أن نشتق صنفي السيارات Car والقطارات Train المجردين من صنف مجرد أب وليكن صنف العربات Vehicle:



الشكل (2-15)

والآن يمكن ان تتشارك بعض الأصناف الإضافية في أسفل الهرمية ببعض الخصائص بسبب أغراضها وليس تشاركها فقط لمجرد أنها تشتق من الصنف الأساس نفسه على سبيل المثال يمكن للأصناف PassengerTrain وCompact وSUV وPickup ان تتمكن جميعها من نقل الركاب وبالتالي فإنه يمكننا أن نزود جميع هذه الأصناف بواجهة نقل الركاب IPassengerCarrier ويمكننا أيضا أن نزود الصنفين FreightTrain وPickup بواجهة نقل البضائع IHeavyLoadCarrier.



الشكل (2-16)

باستخدامنا لنظام مجزأ بواسطة الكائنات بهذه الصورة قبل القيام بإسناد أية قيمة معينة لها يمكننا معرفة الأوضاع التي تحدد لنا متى نستخدم الأصناف المجردة ومتى نستخدم الواجهات إن هذا المثال الذي طرحناه هنا لا يمكننا تحقيقه باستخدام الأصناف المجردة فقط أو الواجهات فقط.

أنواع البنى Struct:

Struct Types:

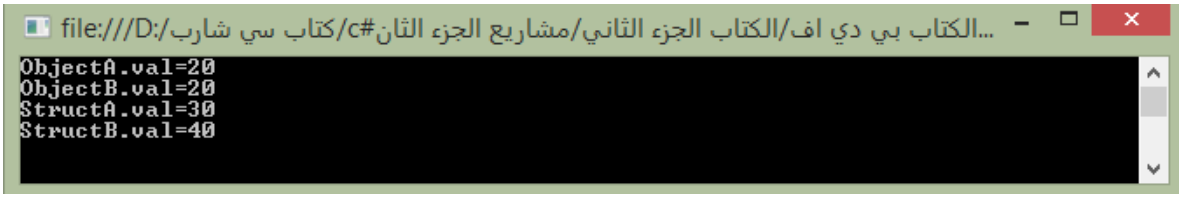
لقد نوهنا في الفصل السابق من أن أنواع البنى Struct والأصناف متشابهان بصورة كبيرة إلا أن أنواع البنى Struct هي أنواع قيمة أما الأصناف فهي أنواع مرجعية لكن ماذا يعني هذا بالنسبة إلينا؟ حسنا إن أبسط طريقة لتوضيح ذلك هو بضرب مثال:

مثال حول الأصناف مقابل بنى Struct:

1- قم بإنشاء تطبيق Console جديد باسم Console Struct Types.
2- عدل الشيفرة كما يلي:

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

namespace Console_Struct_Types
{
    class MyClass
    {
        public int val;
    }
    struct MyStruct
    {
        public int val;
    }
    class Program
    {
        static void Main(string[] args)
        {
            MyClass ObiectA = new MyClass();
            MyClass ObjectB = ObiectA;
            ObiectA.val = 10;
            ObjectB.val = 20;
            MyStruct StructA = new MyStruct();
            MyStruct StructB = StructA;
            StructA.val = 30;
            StructB.val = 40;
            Console.WriteLine("ObjectA.val={0}", ObiectA.val);
            Console.WriteLine("ObjectB.val={0}", ObjectB.val);
            Console.WriteLine("StructA.val={0}", StructA.val);
            Console.WriteLine("StructB.val={0}", StructB.val);
            Console.ReadLine();
        }
    }
}
```

3- نفذ التطبيق بالضغط على زر F5.



الشكل (2-17)

كيفية العمل:

How to Work:

يتضمن هذا التطبيق تعريفين أحدهما للنوع Struct باسم MyStruct وهو يحتوي على حقل واحد من نوع int باسم val والأخر هو صنف يسمى بـ MyClass ويحتوي على حقل مشابه للحقل المعرف ضمن البنية MyStruct.

وبالإضافة إلى تعريف هذين النوعين فإن الشيفرة السابقة تقوم بالعمليات التالية لكل نوع من هذه الأنواع:

- التصريح عن متحول النوع.
- إنشاء حالة جديدة من النوع في هذا المتحول (أي إنشاء كائن من النوع).
- التصريح عن متحول آخر من النوع.
- إسناد المتحول الأول إلى الثاني.
- إسناد قيمة للحقل val في الكائن الأول.
- إسناد قيمة للحقل val في الكائن الثاني.
- عرض قيمة val في كلا الكائنين(المتحولين).

وعلى الرغم من أننا نقوم بالعمليات نفسها على متحولات لكلا النوعين إلا أن النتائج ليست نفسها.

فعندما نعرض قيمة الحقل val فإننا نجد أن قيمة هذا الحقل هي نفسها في كلا المتحولين من نوع MyClass بينما تكون قيمة الحقل val مختلفة في المتحولين من نوع MyStruct.

إذا ماذا يحدث؟

لقد قلنا مسبقا أن الكائنات هي أنواع مرجعية وبالتالي عندما نقوم بإسناد كائن إلى متحول فإننا نقوم بإسناد هذا المتحول مؤشرا pointer إلى الكائن والمؤشر عبارة عن عنوان في الذاكرة وبناء على ذلك فإن العنوان هو موضع في الذاكرة يتواجد فيه الكائن فيها وعندما قمنا بإسناد مرجع الكائن الأول إلى متحول آخر من نوع MyClass فإن ما قمنا به في الحقيقة هو نسخ هذا العنوان وهذا يعني أن كلا المتحولين يحتويان على مؤشر للكائن نفسه.

اما أنواع Struct فهي أنواع قيمة وهذا يعني أنه بدلا من وضع مؤشر للبنية ضمن المتحول فإن هذا المتحول سيحتوي على البنية نفسها وعندما نقوم بإسناد البنية الأولى على المتحول الثاني من نوع MyStruct فإن ما نقوم به في الحقيقة هو نسخ جميع المعلومات من بنية إلى أخرى إن هذا السلوك مطابق تماما لعمليات الإسناد الطبيعية التي قمنا بها على مدار الفصول السابقة من إسناد قيم من أنواع بسيطة مثل int إلى متحولات من نفس الأنواع والنتيجة النهائية هي ان متحولي البنيتين يحتويان على بنيتين مختلفتين عن بعضهما.

إن استخدام المؤشرات بهذه الصورة غير الواضحة بالنسبة لمبرمج C# يجعل كتابة الشيفرة أسهل وأبسط ومن الممكن الوصول إلى عمليات منخفضة المستوى مثل معالجة المؤشرات في C# باستخدام ما يعرف بالشيفرة غير الآمنة unsafe code إلا أن هذا الموضوع هو موضوع متقدم ولن نتناوله في كتابنا هذا.

النسخ السطحي والنسخ العميق:

Shallow vs. Deep Copying:

إن مسألة نسخ الكائنات من متحول إلى آخر بالقيمة بدلا من المرجع هو أمر معقد أي نسخ الكائنات من متحول إلى آخر بحيث يشير كلا المتحولين إلى كائنين متساويين ولكن منفصلين وليس الإشارة إلى كائن واحد. وذلك لأن الكائن الواحد يمكن أن يتضمن على مراجع لكائنات عديدة أخرى موجودة ضمن حقول هذا الكائن مثلا.

على كل حال هناك طريقة مبسطة لنسخ الكائنات وذلك من خلال المنهج MemberwiseClone() التابع للصنف System.Object إن هذا المنهج هو منهج محمي إلا انه لا يمكننا تعريف منهج عام على كائن يستدعي هذا المنهج بسهولة تسمى عملية النسخ المستخدمة في هذا المنهج بالنسخ السطحي Shallow حيث لا يتم هنا أخذ الأعضاء ذات الأنواع المرجعية بعين الاعتبار هذا يعني أن الأعضاء المرجعية في الكائن الجديد ستشير إلى نفس الكائنات في الأعضاء الموجودة ضمن الكائن الأصلي وهو غير أمر مجد في كثير من الحالات وإذا أردنا أن ننسخ الكائن إلى كائن جديد تماما وأعضاء جديدة أيضا فإن علينا استخدام النسخ العميق Deep.

يمكننا التزود بواجهة تسمح لنا بالقيام بذلك بطريقة قياسية وهي تسمى بالواجهة IClonadble فإذا استخدمنا هذه الواجهة فإن علينا أن نزود صنفنا بمنهج واحد فقط يأخذ الاسم Clone() يعيد هذا المنهج قيمة من نوع System.Object ويمكننا ان نستخدم أية عمليات نود القيام بها للحصول على هذا الكائن وذلك بوضع الشيفرة التي ستقوم بذلك ضمن جسم هذا المنهج في صنفنا هذا يعني أنه يمكننا أن نستخدم النسخ العميق إذا أردنا ذلك لكن عملية كهذه ستتطلب إلى مجهود قليل منا.

سوف نتحدث عن ذلك بتفصيل أكبر في الفصول القادمة.

الخلاصة:

Summary:

لقد رأينا في هذا الفصل كيفية تعريف الأصناف والواجهات في C# والذي يمثل حجر الأساس لما تحدثنا عنه بشكل نظري في الفصل السابق كما تعرفنا على صيغة C# في التصريح عن الإضافات والواجهات بالإضافة إلى مقيدات الوصول التي يمكننا استخدامها والطريقة التي يمكننا وراثة الواجهات والأصناف الأخرى في أصنافنا ولقد تعرفنا أيضا على كيفية تعريف الأصناف المجردة والمنغلقة وذلك للتحكم بعملية الوراثة من تلك الأصناف هذا بالإضافة إلى تعريف مناهج البناء والتدمير.

بعد ذلك ألقينا نظرة خاطفة على الصنف System.Object ورأينا أنه يمثل الصنف الجذر الأساس بجميع الأصناف التي تعرفنا عليها يقدم هذا الصنف عددا من المناهج التي يمكننا استخدامها بعضها ظاهري يمكننا تجاوزه في أصنافنا لقد رأينا أيضا أن الصنف يمكننا من معاملة أي كائن على أنه حالة منه مما يسمح باستخدام تقنية تعددية الأشكال مع أي كائن في .NET.

لقد تناولنا بعد ذلك بعضا من الأدوات التي يقدمها Visual Studio.NET بهدف تسهيل التطوير باستخدام تقنيات البرمجة كائنية التوجه وقد تضمن ذلك إطار Class View ونافذة مستعرض الكائنات Object Browser ولقد طرحنا أيضا طريقة سريعة لإضافة أصناف جديدة إلى المشروع ولقد تعرفنا أيضا على كيفية إنشاء المجمعات التي لا يمكننا تنفيذها بصورة مباشرة إلا انه يمكننا أن نستخدمها في شيفرة مشاريع أخرى.

تحدثنا أيضا عن الأصناف المجردة والواجهات ورأينا أوجه التشابه والاختلاف بينهما وما هي الأوضاع التي يجب أن نستخدم فيها الأصناف المجردة والأوضاع التي يجب ان نستخدم فيها الواجهات.

وأخيرا فقد تناولنا موضوع الأنواع المرجعية وأنواع القيمة مجددا وتحدثنا بخصوص هذا الموضوع حول أنواع البنى Struct واختلافها عن الأصناف وقد قادنا موضوع كهذا للحديث عن النسخ السطحي والنسخ العميق للكائنات وهو موضوع سنتحدث عنه في موضع آخر من هذا الكتاب.

سوف نتناول في الفصل القادم كيفية تعريف أعضاء الأصناف مثل الخصائص والمناهج وهو ما سينقل البرمجة كائنية التوجه في C# إلى المستوى المطلوب في محاولة للبدء بإنشاء تطبيقات حقيقية.

الفصل الثالث

تعريف أعضاء الأصناف

سوف نكمل في هذا الفصل حديثنا عن تعريف الأصناف في C# وذلك بتعلم كيفية تعريف أعضاء الأصناف من حقول وخصائص ومناهج.

سوف نبدأ أولا بصيغ تعريف هذه الأعضاء وسوف نرى كيفية توليد هذه الصيغ باستخدام معالجات Visual Studio .NET 2013 المساعدة وسوف نتعلم أيضا كيفية تعديل الأعضاء بسرعة وذلك بتحرير خصائصها.

وبعد تغطية أساسيات تعريف الأعضاء سوف نتحدث عن تقنيات أكثر تقدما متعلقة بالأعضاء مثل كيفية إخفاء أعضاء الصنف الأساس واستدعاء أعضاء الصنف الأساس التي تم تجاوزها بالإضافة إلى التعريفات المعششة.

وأخيرا فإننا سنضع جميع ما تعلمناه في هذا الفصل والفصل السابق موضع الاستخدام العملي الحقيقي وذلك بإنشاء مكتبة أصناف سنعتمد عليها في الفصول القادمة من هذا الكتاب إن شاء الله.

تعريف الأعضاء:

Member Definitions:

يمكننا تعريف أعضاء الصنف بعد تعريفه مباشرة ويتضمن ذلك تعريف الحقول والمناهج والخصائص التي تنتمي لهذا الصنف ولكل عضو من أعضاء الصنف مدى الوصولية الخاص به ويمكن تحديد مدى الوصولية في تعريف أعضاء الصنف باستخدام أربعة مقيدات وصول فقط:

مقيد الوصول	الوصف
public	يمكن الوصول إلى العضو من ضمن اية شيفرة
Private	لا يمكن الوصول إلى العضو إلا من ضمن شيفرة الصنف نفسه
Internal	يمكن الوصول إلى العضو من ضمن المشروع (المجمعة) الذي يحوي على تعريفه فقط.
Protected	لا يمكن الوصول إلى العضو إلا من ضمن شيفرة الصنف نفسه أو من صنف مشتق.

ويمكن لمقيدي الوصول الأخيرين أن يستخدما معا مثل protected internal وفي تلك الحالة فإن الأعضاء المعرفة وفق مقيد الوصول هذا لا يمكن الوصول إليها إلا من خلال الأصناف المشتقة الموجودة ضمن المشروع نفسه أو ضمن المجمعة نفسها.

يمكننا التصريح عن الخصائص والمناهج والحقول باستخدام الكلمة Static وعندئذ فإن الأعضاء المعرفة وفقا لهذه الكلمة هي أعضاء ستاتيكية متعلقة بالصنف نفسه وليس بالكائنات المعرفة من هذه الأصناف.

تعريف الحقول:

Defining Fields:

تعرف الحقول كما تعرف المتحولات العادية تماما بالإضافة إلى استخدام أحد مقيدات الوصول التي شرحناها على سبيل المثال:

```
class MyClass
{
    public int MyInt;
}
```

ويمكن أن نعرف الحقول باستخدام الكلمة readonly وعندئذ فإنه لا يمكن إسناد قيمة إلى هذا الحقل إلا عند الإسناد الأولي له ويتضمن منهج بناء الصنف على سبيل المثال:

```
class MyClass
{
    public readonly  int MyInt=45;
}
```

وكما قلنا في بداية هذا الفصل فإنه يمكننا تعريف الحقول باستخدام الكلمة static على سبيل المثال:

```
class MyClass
{
    public static   int MyInt;
}
```

حيث يمكن الوصول إلى الحقول الستاتيكية عبر الصنف الذي تنتمي إليه فقط أي باستخدام MyClass.MyInt بالنسبة للمثال السابق وليس من خلال الكائنات المعرفة من هذا الصنف والمثال التالي سيوضح الفرق:

لنفرض أن لدينا الصنف Person الذي يحوي على مجموعة من الحقول المعرفة كما يلي:

```
class Person
{
    public string FirstName;
    public string LastName;
    public int Age;
    static public int counter;
}
```

فإذا قمنا بإنشاء حالة من الصنف Person باسم MyEmploed ومن ثم قمنا باستدعاء الحقول كما يلي:

```
class Program
{
    static void Main(string[] args)
    {
        Person MyEmpolyed = new Person();
        MyEmpolyed .FirstName ="Hussam";
        MyEmpolyed.LastName = "AL deen AL roz";
        MyEmpolyed.Age = 31;
        MyEmpolyed.counter = 3;
        Console .WriteLine ("{0}) Person:{1} {2} \nAge:{3}",
            Person .counter ,MyEmpolyed .FirstName
            ,MyEmpolyed .LastName ,MyEmpolyed .Age );
        Console.ReadLine();
    }
}
```

فإن مترجم C# سوف يظهر لنا خطأ يشير إلى أن الوصول إلى الحقل counter لا يتم بهذه الطريقة لأنه حقل من نوع ستاتيكي والطريقة الصحيحة هي:

```
class Program
{
    static void Main(string[] args)
    {
        Person MyEmpolyee = new Person();
        MyEmpolyed .FirstName ="Hussam";
        MyEmpolyed.LastName = "AL deen AL roz";
        MyEmpolyed.Age = 31;
        Person.counter = 3;
        Console .WriteLine ("{0}) Person:{1} {2} \nAge:{3}",
            Person .counter ,MyEmpolyed .FirstName
            ,MyEmpolyed .LastName ,MyEmpolyed .Age );
        Console.ReadLine();
    }
}
```

ملاحظة:

لاحظ أنه لا يوجد فرق لدى مترجم C# في أسبقية التعريف بين public وstatic فأيهما يسبق الاخر لا يهم.

وبالإضافة إلى ذلك يمكننا إنشاء حقول ذات قيم ثابتة وذلك باستخدام الكلمة const وكما تلاحظ فإن الحقول الثابتة هي حقول ستاتيكية حسب التعريف وبالتالي ليس هناك داع لاستخدام المقيد static (إن استخدامه سيؤدي إلى حدوث خطأ) أي ان الشيفرة التالية صحيحة:

```
namespace ConsoleApplication2
{
    class Person
    {
        public string FirstName;
```

```
        public string LastName;
        public int Age;
        public const int counter = 3;
    }

    class Program
    {
        static void Main(string[] args)
        {
            Person MyEmpolyee = new Person();
            MyEmpolyee .FirstName ="Hussam";
            MyEmpolyee.LastName = "AL deen AL roz";
            MyEmpolyee.Age = 15;
            Console .WriteLine ("{0}) Person:{1} {2} \nAge:{3}",
                Person .counter ,MyEmpolyee .FirstName
                ,MyEmpolyee .LastName ,MyEmpolyee .Age );
            Console.ReadLine();
        }
    }
}
```

تعريف المناهج:

Defining Methods:

تستخدم المناهج نفس شكل التابع بالإضافة إلى إمكانية استخدام مقيدات الوصول وإمكانية استخدام المقيد static على سبيل المثال:

```
    class MyClass
    {
        public string GetString()
        {
            return "Here is a String";
        }
    }
```

لاحظ أننا إذا استخدمنا الكلمة static في تعريف المنهج فإنه لا يمكننا الوصول إلى هذا المنهج إلا من خلال الصنف وليس من خلال حالة (كائن) من هذا الصنف.

ويمكننا أن نستخدم الكلمات التالية مع تعريف المناهج:

الكلمة	الوصف
Virtual	يمكن تجاوز هذا المنهج (أي إنشاء منهج في صنف مشتق يتجاوز شيفرة هذا المنهج)
Abstract	يجب تجاوز هذا المنهج (لا يمكننا استخدام هذه الكلمة إلا في الأصناف المجردة فقط)
Override	يمثل هذا المنهج تجاوزا لمنهج في الصنف الأساس
Extern	أي أن تعريف هذا المنهج موجود في مكان آخر

تبين الشيفرة التالية كيفية تجاوز منهج في صنف أساس من قبل منهج في صنف آخر:

```
public class MyBaseClass
 {
   public virtual void DoSomething()
   {
      // Base implementation
   }
 }

public class MyDerivedClass:MyBaseClass
{
    public override void DoSomething()
    {
        base.DoSomething();
    }
}
```

لقد قمنا هنا بإنشاء صنف باسم MyBaseClass يحتوي على منهج عام (الكلمة public) يمكن تجاوزه من قبل منهج آخر لصنف مشتق(الكلمة virtual) باسم DoSomething() ومن ثم قمنا بإنشاء صنف آخر باسم MyDerivedClass يرث الصنف MyBaseClass ويعرف منهجا باسم DoSomething() لاحظ انه يمكننا استخدام المنهج DoSomething() في الصنف المشتق دون تعريفه لأن هذا المنهج موجود ضمن الصنف المورث (صنف الاساس) إلا اننا نود لهذا الصنف القيام بأشياء أخرى لذا فإننا سنقوم بتجاوز هذا المنهج في الصنف المشتق باسم الكلمة override.

ملاحظة:

لاحظ ان مترجم #C بعد كتابتنا لكلمة override اظهر الاتمام التلقائي في Visual Studio.NET 2013 الكلمة DoSomething() وبمجرد اختيارها فإن مترجم #C سوف يكمل العبارة واضعا الكلمة void وما يليها من عبارات حيث تشير العبارة base.DoSomething() إلى أن الصنف المشتق MyDerivedClass قد تجاوز منهج الصنف الأساس MyBaseClass وكل ذلك حدث بسبب الكلمة override حيث توقع متنبئ الإكمال التلقائي بالباقي وهي ميزة جديدة موجودة ضمن Visual Studio.NET 2013.إلا أن الإبقاء على تلك العبارة سوف يعطل التجاوز كما سنرى لاحقا.

وإذا استخدمت الكلمة override يمكنك عندئذ استخدام الكلمة sealed أيضا وذلك لتحديد أنه لا يمكن وضع أية تعديلات إضافية لهذا المنهج في الأصناف المشتقة من الصنف MyDerivedClass أي أنه لا يمكن تجاوز هذا المنهج من قبل الأصناف المشتقة على سبيل المثال:

```
public class MyDerivedClass : MyBaseClass
{
    public sealed override void DoSomething()
    {
        base.DoSomething();
    }
}
```

إن هذا يشير إلى أن أي صنف يرث الصنف MyDerivedClass لا يمكنه تجاوز المنهج DoSomething() وتستخدم الكلمة extern لتوفير تزويد للمنهج خارج المشروع وذلك موضوع متقدم ولن نتحدث عنه في هذا الكتاب.

تعريف الخصائص:

Defining Properties:

تعرف الخصائص بنفس الأسلوب الذي تعرف وفقه الحقول إلا أن هناك أشياء أخرى ضمن الخصائص فالخصائص يمكنها القيام بأمور أكبر من مجرد وسيلة لحفظ حالة أو قيمة معينة كما تقوم الحقول بذلك حيث تتضمن الخصائص على كتلتين برمجيتين يتم تنفيذهما عند قراءة قيمة الخاصية أو عند إسناد قيمة للخاصية.

تعرف هاتين الكتلتان باستخدام الكلمتين get وset ومن الممكن تجاهل إحدى هاتين الكتلتين ضمن تعريف الخاصية وفي حالة كهذه فإنه يمكننا أن ننشئ خاصية للقراءة فقط عند إهمال الكتلة set او للكتابة فقط عند إهمال الكتلة get لكن يجب أن نضع واحدا من هاتين الكتلتين على الأقل في تعريف الخاصية لكن كما نرى فإنه من غير المنطقي وجود خاصية كهذه لذا فإن إنشاء خاصية للكتابة فقط أمر غير مفيد عمليا إلا انه ممكن.

تتألف البنية الأساسية للخاصية من مقيد وصول public أو private أو غيرها متبوعة بنوع الخاصية واسمها ومن ثم إحدى أو كلا الكتلتين get وset:

```
<accessModifier> <propertyType> <propertyName>
{
   Get
   {
      // code run in read
    }
    Set
    {
      // code run in write
     }
```

```
}
```

على سبيل المثال:

```
class Person
{
    public int MyAge
    {
        get
        {
            // property get code
        }
        set
        {
            // property set code
        }
    }
}
```

لاحظ ان السطر الأول من تعريف الخاصية مشابه جدا لتعريف الحقول والفرق الواضح هنا هو عدم وجود الفاصلة المنقوطة ";" عند نهاية السطر وبدلا من ذلك فإن لدينا كتلة برمجية تتضمن على كتلتين برمجيتين معشعشتين ضمنها وهما get وset.

إذن كيف سنستخدم هاتين الكتلتين؟

حسنا أولا يجب أن تحتوي الكتلة get على قيمة معادة من نفس نوع الخاصية أي أنه يجب أن تتضمن الكتلة get على تعليمة return.

وكقاعدة عامة فإنه يجب دائما ان ترتبط الخاصية بحقل خاص private معرف ضمن الصنف وعندئذ تصبح مهمة الخاصية التحكم في الوصول إلى هذا الحقل على سبيل المثال:

```
class Person
{
    private int Age;
    public int MyAge
    {
        get
        {
            return Age;
        }
        set
        {
            // property set code
        }
    }
}
```

لاحظ أن الشيفرة خارج الصنف لا يمكنها الوصول إلى الحقل Age ابدا وذلك لأنه معرف وفق مقيد الوصول الخاص private ولكي نتمكن من قراءة قيمة الحقل فإن علينا أن نستخدم الخاصية MyAge والتي تمثل مغلفا لهذا الحقل.

ثانيا تقوم الكتلة set بإسناد قيمة لحقل الخاصية لكن كيف يمكننا الوصول إلى القيمة المسندة من خارج الصنف من خلال الكتلة set؟ في الحقيقة إن أية قيمة يتم إسنادها إلى الخاصية التي تحتوي على كتلة set ستخزن ضمن بارامتر وهمي باسم value وهذا يعني أنه يمكننا الوصول إلى القيمة التي تم إسنادها إلى الخاصية من خلال الكلمة value.

حسنا لقد تعلمنا كيفية قراءة قيمة الخاصية من خلال قراءة الحقل والآن إليك كيفية تعديل الشيفرة السابقة لكي نتمكن من إسناد قيمة إلى الحقل من خلال خاصية ذلك الحقل:

```
class Person
{
    private int Age;
    public int MyAge
    {
        get
        {
            return Age;
        }
        set
        {
            Age = value;
        }
    }
}
```

يجب ان تكون قيمة value من نفس نوع الخاصية والذي يجب أن يكون من نفس نوع الحقل الخاص المرتبط بهذه الخاصية أيضا وبالتالي ليس هناك داع للقلق حول استخدام التشكيل cast لحفظ القيمة ضمن الحقل.

إن الخاصية تقوم بأكثر من مجرد غلاف للوصول إلى الحقل والقوة الحقيقية للخصائص تأتي عندما نبذل تحكما أكبر على الأحداث على سبيل المثال يمكننا أن نكتب كتلة set بالصورة التالية:

```
        set
        {
            if (!(value > 100 || value < 1))
                Age = value;
        }
```

ستعدل قيمة الحقل Age هنا فقط إذا كانت القيمة المسندة إلى الخاصية بين 0 و100 وفي حالات كهذه من المهم جدا أن نحدد السلوك الذي يجب إتباعه عند إدخال قيم خارج هذا المجال أي قيمة خاطئة ويكون لدينا أربعة خيارات عادة:

- عدم القيام بأي شيء (كما في الشيفرة السابقة)
- إسناد القيمة الافتراضية للحقل (إذا كانت هناك قيمة احترافية)
- الاستمرار كما لو أنه لم يحدث شيء خاطئ ولكن تسجيل ما حدث للتحليل المستقبلي.
- رمي اعتراض.

وبصورة عامة فإن الخيارين الأخيرين هما الخياران المفضلان واختيار أحدهما يعتمد على الكيفية التي سيستخدم وفقها الصنف وما مدى التحكم الذي يمكننا أن نقدمه لمستخدمي الصنف فرمي اعتراض يقدم لمستخدمي الصنف تحكما أكبر ويسمح بمعرفة ما يحدث والاستجابة لذلك بصورة مناسبة ويمكننا أن نستخدم اعتراض System عادي لذلك على سبيل المثال:

```
set
{
    if (!(value > 100 || value < 1))
        Age = value;
    else
        throw (new ArgumentOutOfRangeException("MyAge", value,
            "MyAge must be assigned a value between 0 and 100"));
}
```

ويمكننا معالجة اعتراض كهذا من خلال استخدام البنية try..catch..finally في الشيفرة التي تستخدم الخاصية.

أما تسجيل البيانات في الغالب ضمن ملف نصي فيمكن أن يكون مفيدا في حالات الشيفرة الإنتاجية والتي لا يجب أن تحدث المشاكل فيها فهي تسمح للمطورين بتفحص أداء التطبيق وإمكانية تصحيح الشيفرة الحالية عند الضرورة.

يمكن أن تكون الخصائص ظاهرية virtual ويمكن أن تكون قابلة للتجاوز override ويمكن ان تكون مجردة أيضا abstract وهو أمر غير ممكن مع الحقول.

تطبيق حول استخدام الحقول والمناهج والخصائص:

1- قم بإنشاء تطبيق Console جديد باسم Console Properties Methods Fields.
2- أضف صنفا جديدا باسم MyClass باستخدام أحد الطرق التي شرحناها في الفصل السابق.
3- عدل شيفرة الصنف MyClass.cs كما يلي:

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

namespace Console_Properties_Methods_Fields
{
    public class MyClass
    {
        public readonly string Name;
        private int intVal;
        public int val
        {
            get
            {
                return intVal;
```

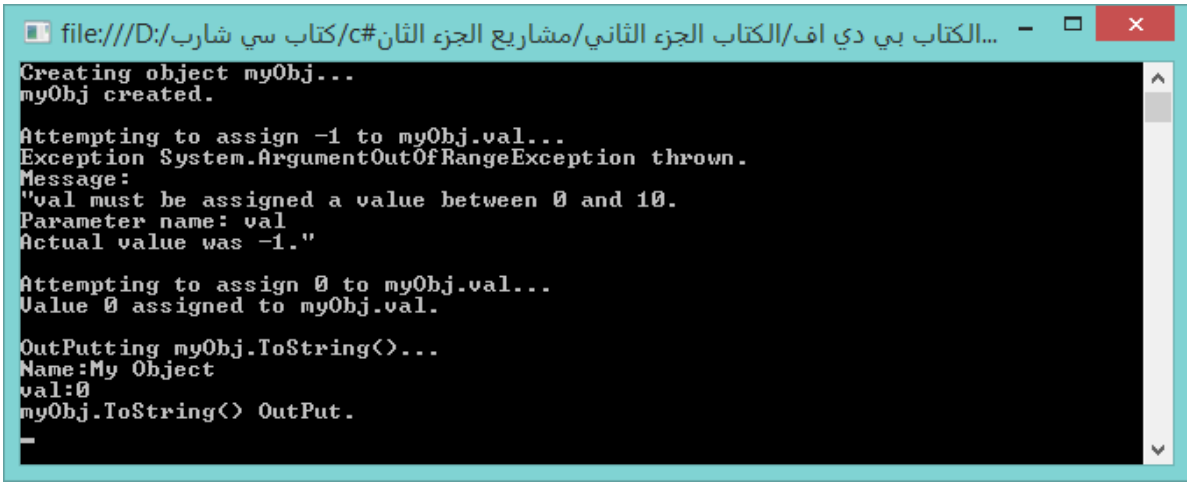
```
            }
            set
            {
                if (value >= 0 && value <= 10)
                    intVal = value;
                else
                    throw (new ArgumentOutOfRangeException("val", value,
                        "val must be assigned a value between 0 and 10."));
            }
        }
        public override string ToString()
        {
            // return base.ToString();
            return "Name:" + Name + "\nval:" + val;
        }
        private MyClass ():this("Default Name")
        {
        }
        public MyClass (string newName)
        {
            Name = newName;
            intVal = 0;
        }
    }
}
```

4- عدل شيفرة الملف Program.cs كما يلي:

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

namespace Console_Properties_Methods_Fields
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Creating object myObj...");
            MyClass myObj = new MyClass("My Object");
            Console.WriteLine("myObj created.");
            for (int i=-1;i<=0;i++)
            {
                try
                {
                    Console.WriteLine("\nAttempting to assign {0} to myObj.val...",
                        i);
                    myObj.val = i;
                    Console.WriteLine("Value {0} assigned to myObj.val.",
                        myObj.val);
                }
                catch (Exception e)
```

```
                {
                    Console.WriteLine("Exception {0} thrown.",
                        e.GetType().FullName);
                    Console.WriteLine("Message:\n\"{0}\"", e.Message);
                }
            }
            Console.WriteLine("\nOutPutting myObj.ToString()...");
            Console.WriteLine(myObj);
            Console.WriteLine("myObj.ToString() OutPut.");
            Console.ReadLine();
        }
    }
}
```

5- نفذ الشيفرة بالضغط على زر F5.



الشكل (1-3)

كيفية العمل:

How to Work:

تقوم شيفرة المنهج Main() بإنشاء كائن من الصنف MyClass المعرف ضمن الملف MyClass.cs ويجب أن يتم إنشاء حالة من هذا الصنف باستخدام منهج بناء غير افتراضي وذلك لأن منهج البناء الافتراضي معرف وفق المقيد private:

```
private MyClass ():this("Default Name")
{
}
```

توضيح:

لاحظ أنني استخدمت العبارة this ("Default Name") وذلك للتأكد من أن الحقل Name سيأخذ قيمة إذا تم استدعاء منهج البناء هذا بطريقة أو بأخرى وهو ما يمكن حدوثه عند توريث هذا الصنف

إلى صنف آخر يجب أن يحتوي الحقل Name على قيمة دوما لأن عدم القيام بذلك يمكن أن يمثل مصدرا لحدوث الأخطاء لاحقا.

لذا فقد استخدمت منهج بناء غير افتراضي يقوم بإسناد قيمة إلى حقل القراءة فقط Name بما أن هذا الحقل معرف بالمقيد readonly فإنه لا يمكننا إسناد قيمة إليه إلا من خلال الإسناد الأولي في سطر التصريح عنه أو من خلال مناهج بناء الصنف الذي يتبع له فقط ولقد تم أيضا تعريف حقل خاص باسم intVal.

بعد ذلك سيقوم المنهج Main() بمحاولتين لإسناد قيمة إلى الخاصية val للكائن myObj وهو كائن من الصنف MyClass ولقد استخدمنا لهذا الغرض حلقة for تبدأ من القيمة 1- إلى الصفر أي القيمتان 1- و 0 فقط! ولقد استخدمنا بنية try..catch لاكتشاف ومعالجة الاعتراضات الناتجة من عملية الإسناد تلك فعند إسناد القيمة 1- للخاصية سيتم رمي اعتراض من النوع System. ArgumentOutOfRangeException وسيتم اصطياد هذا الاعتراض من خلال كتلة catch حيث ستطبع معلومات هذا الاعتراض على نافذة Console أما الدورة الثانية للحلقة (عندما i=0) فسيتم إسناد القيمة 0 إلى الخاصية val بنجاح أي سيأخذ الحقل intVal القيمة 0 من خلال تلك الخاصية.

وأخيرا فإننا سنستخدم المنهج ToString() للكائن myObj لإخراج نص يمثل محتويات الكائن لاحظ اننا قمنا بتجاوز هذا المنهج وهو منهج موروث من الصنف الجذر System.Object في الصنف MyClass وذلك كي يقوم بإعطائنا تمثيلا نصيا محددا بحسب ما نريد وليس مجرد إعادة سلسلة نصية تمثل اسم الكائن وهو السلوك الافتراضي لهذا المنهج إن لم نقم بتجاوزه واستخدمناه كما هو في الصنف System.Object.

```
public override string ToString()
{
    // return base.ToString();
    return "Name:" + Name + "\nval:" + val;
}
```

يجب ان نصرح عن هذا المنهج بالكلمة override وذلك لأنه سيمثل تجاوزا للمنهج ToString() الظاهري الموجود في الصنف الأساس System.Object تستخدم الشيفرة هنا الخاصية val بصورة مباشرة بدلا من استخدام الحقل الخاص intVal إلا أنه يمكننا أن نستخدم الحقل intVal إذا أردنا ذلك وليس هناك أي سبب وجيه يجعلنا نفضل استخدام حقل الخاصية او الخاصية نفسها طالما ان ذلك يتم ضمن الصنف نفسه.

ملاحظة:

إن Visual Studio 2013 أثناء التصريح عن المنهج ToString() باستخدام الكلمة override سوف يقوم بوضع شيفرة تشير إلى أنك قد تجاوزت هذا المنهج في الصنف الأساس System.Object وهذا الأسلوب الجديد في Visual Studio 2013 كي يتأكد حقيقة من أن المبرمج يريد تجاوز المنهج

في الصنف الأساس فإذا أبقيت على تلك الشيفرة فلن يحدث التجاوز و إن محوتها أو جعلتها كتعليق فسيتم التجاوز كما تلاحظ في الشيفرة السابقة جرب ذلك.

توضيح:

في الحقيقة هناك تأثير ضئيل جدا على الأداء عند استخدام الخصائص في الصنف بدلا من الحقول مباشرة إلا ان استخدام الخصائص له بعض الفوائد وذلك في الحالة التي سيتم تنفيذ شيفرة ما موجودة ضمن كتلتي الخاصية عند اسناد أو قراءة قيمها.

معالجات VS 2013 المساعدة في إنشاء أعضاء الأصناف:

Visual Studio 2013 Member Wizards:

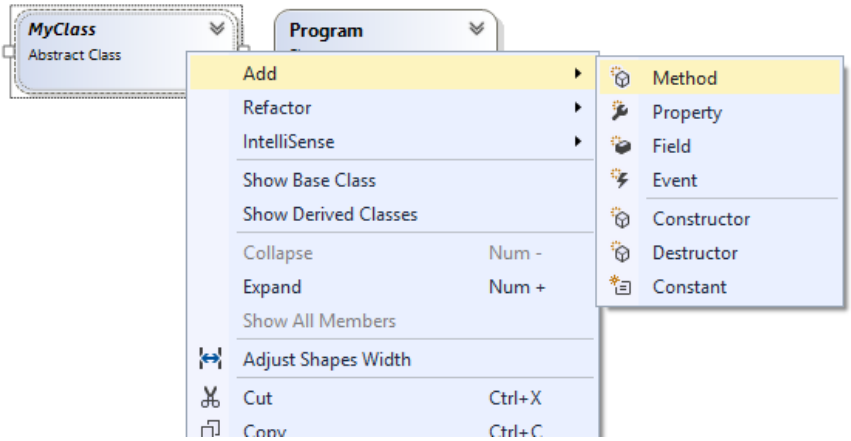
لقد رأينا في الفصل السابق كيفية استخدام أسلوب مختصر لإنشاء الأصناف في Visual Studio 2013 وهناك عدد من هذه الأدوات تسمى بالمعالجات المساعدة Wizards والتي يمكننا استخدامها من اجل محتوى الأصناف أيضا سنتحدث في هذا القسم عن المعالجات المساعدة التي تمكننا من إضافة الخصائص والمناهج والحقول بسهولة إلى اصنافنا.

كما أشرنا إلى نافذة Class Diagram والتي نستطيع الحصول عليها بالضغط بزر الفأرة الأيمن فوق نافذة Solution Explorer ومن ثم من الأمر View نختار الخيار View Class Diagram ربما تتساءل عن سبب ذكرنا لهذه النافذة في الحقيقة إن استخدام المعالجات المساعدة للحقول والمناهج والخصائص لا يتم من خلال نافذة Solution Explorer كما هو الحال مع المعالج المساعد للأصناف إنما يتم من خلال نافذة Class Diagram.

معالج إضافة منهج:

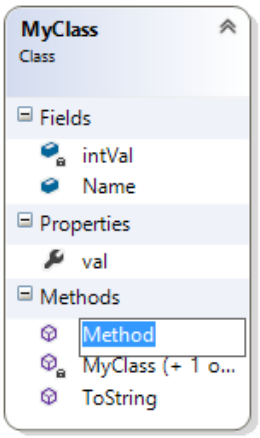
The Add Method Wizard:

بعد فتح نافذة Class Diagram يمكننا اختيار أحد الأصناف الموجودة في مشروعنا ثم الضغط عليها بزر الفأرة الأيمن حيث تظهر قائمة منسدلة من الأمر Add تظهر قائمة فرعية تحوي على العديد من الخيارات نختار منها الخيار Method الشكل (2-3) يظهر قائمة الخيارات الفرعية للأمر Add.



الشكل (3-2)

يمكننا المعالج المساعد للمناهج من إضافة منهج إلى الصنف بسهولة ويسر ويمكننا تحقيق ذلك من خلال تغير شكل صندوق الصنف في نافذة Class Diagram الذي يتغير عند اختيار الأمر Method من القائمة الفرعية حيث يظهر الشكل (3-3).



الشكل (3-3)

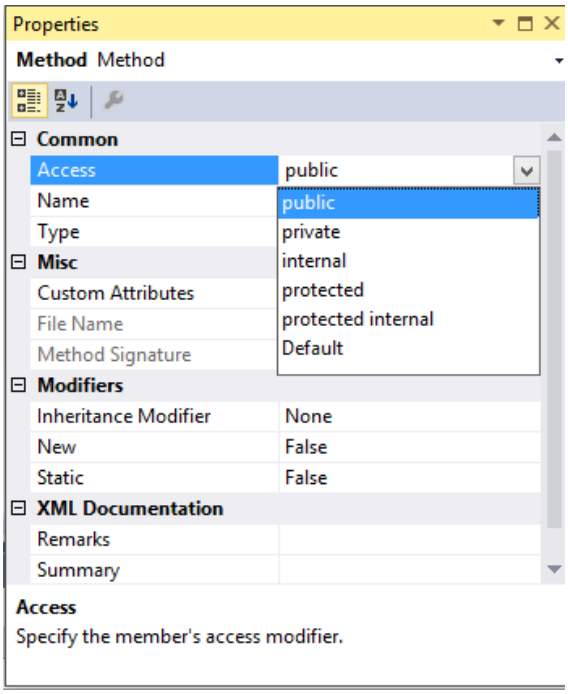
وباختيار اسم للمنهج وضغط الزر Enter نكون بذلك قد أضفنا منهجا جديدا لصنفنا MyClass لكن السؤال هنا ما هي صفات هذا المنهج هل هو عام أم خاص أم ..

لنذهب إلى الملف MyClass.cs سنجد ضمن شيفرة هذا الصنف قد أضيفت شيفرة منهج جديد باسم Method كما يلي:

```
public void Method()
{
    throw new System.NotImplementedException();
}
```

كما ترى من الشيفرة فإن هذا المنهج عام Public لكن هل يمكننا تغيير مدى وصولية هذا المنهج. بالتأكيد لكن كيف. لا تقلق عد إلى النافذة Class Diagram ثم قم باختيار الكلمة Method من الشجرة Methods وانظر للزاوية اليمنى السفلى نعم انها نافذة الخصائص Properties حيث ستظهر كما في الشكل (4-3).

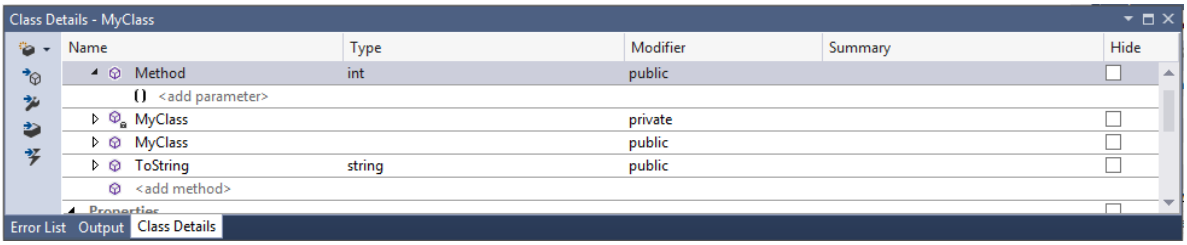
توفر نافذة الخصائص إمكانية الوصول إلى جميع المزايا التي يمكننا استخدامها لإنشاء المناهج فيمكننا هنا تحديد مستوى الوصولية من خلال مربع القائمة المنبثقة Access وتحديد نوع القيم المعادة من خلال القائمة المنبثقة type وكذلك اسم المنهج من خلال مربع النص Name كما يمكننا إضافة مقيدات وصول أخرى من خلال مربعات الاختيارات أسفل التبويب Modifiers وبالإضافة لذلك يمكننا تحديد البارامترات باستخدام النافذة Class Details الشكل (3-5) وذلك باختيار المنهج Method نلاحظ بعد الضغط على السهم على يسار كلمة Method ظهور للأسفل منه قوسين وإلى جانبه بخط رمادي كلمة <add parameter>.



الشكل (4-3)

عند الضغط مكان النص الرمادي يظهر محرر نصوص من خلاله نستطيع تحديد اسم البارامتر وبتحديد أسفل الكلمة Type نستطيع تحديد نوع البارامتر وأسفل الكلمة Modifier نستطيع تحديد فيما إذا كان البارامتر من نوع ref أو out أو مصفوفة بارامترات params كما أنه أسفل الكلمة Summary نستطيع كتابة تعليقات مصدرية بلغة XML سنتحدث عن هذا لاحقا لاحظ أيضا أنه يمكننا أن نضع تعليقات مصدرية بلغة XML للمنهج الذي أضفناه من نافذة الخصائص تحت التبويب XML Documentation لاحظ في النافذة Class Details وجود مربع اختيار أسفل الكلمة Hide إن تفعيل مربع الاختيار سيؤدي

إلى إخفاء العضو من صندوق MyClass في نافذة Class Diagram ويظهر رمز ضمن صندوق MyClass يشير إلى أن هناك أعضاء مخفية جرب ذلك.



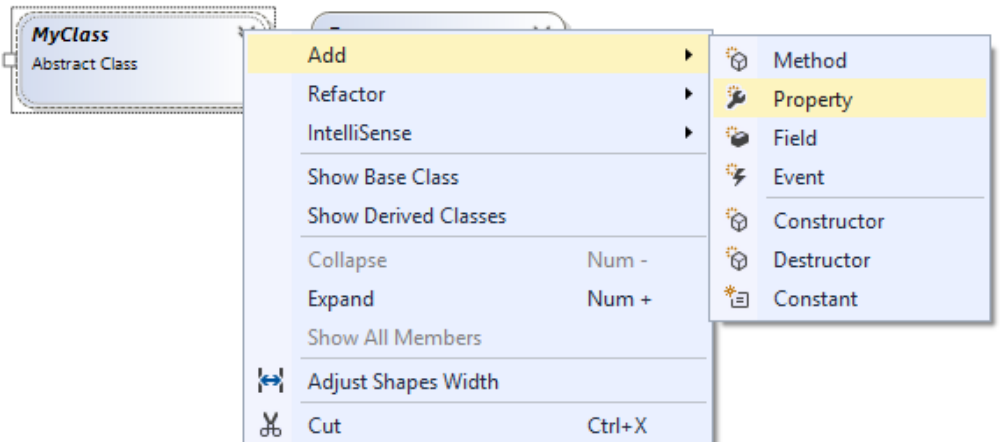
الشكل (3-5)

وبالطبع فإن معالج إضافة المناهج لا يوفر لنا شيفرة جسم المنهج وإنما يوفر البنية الأساسية للمنهج ويقلل من أخطاء كتابة تعريف المنهج.

معالج إضافة خاصية:

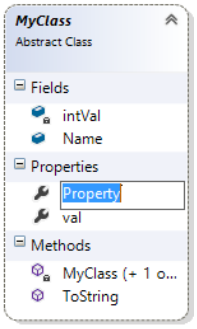
The Add Property Wizard:

بعد فتح نافذة Class Diagram يمكننا اختيار أحد الأصناف الموجودة في مشروعنا ثم الضغط عليها بزر الفأرة الأيمن حيث تظهر قائمة منسدلة من الأمر Add تظهر قائمة فرعية تحوي على العديد من الخيارات نختار منها الخيار Property الشكل (3-6) يظهر قائمة الخيارات الفرعية للأمر Add.



الشكل (3-6)

يمكننا المعالج المساعد للخصائص من إضافة خاصية إلى الصنف بسهولة ويسر ويمكننا تحقيق ذلك من خلال تغير شكل صندوق الصنف في نافذة Class Diagram الذي يتغير عند اختيار الأمر Property من القائمة الفرعية حيث يظهر الشكل (3-7).



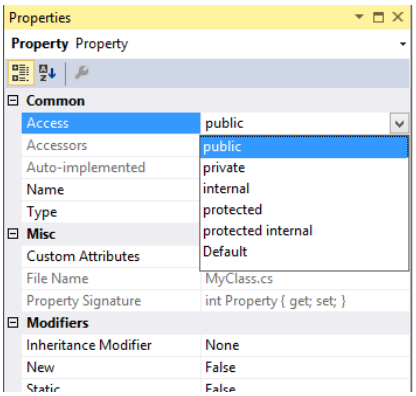
الشكل (3-7)

وباختيار اسم للخاصية وضغط الزر Enter نكون بذلك قد أضفنا خاصية جديدة لصنفنا MyClass لكن السؤال هنا ما هي صفات هذا الخاصية هل هو عام أم خاص أم ..

لنذهب إلى الملف MyClass.cs سنجد ضمن شيفرة هذا الصنف قد أضيفت شيفرة خاصية جديدة باسم Property كما يلي:

```
public int Property
{
    get
    {
        throw new System.NotImplementedException();
    }
    set
    {
    }
}
```

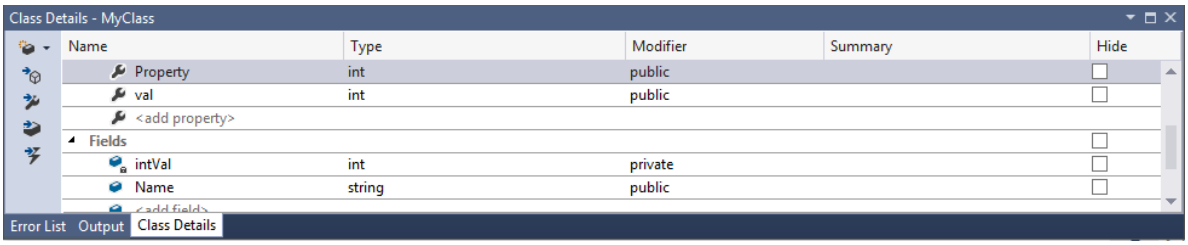
كما ترى من الشيفرة فإن هذه الخاصية تحمل نوع وصولية عام Public لكن هل يمكننا تغيير مدى وصولية هذه الخاصية. بالتأكيد لكن كيف. لا تقلق عد إلى النافذة Class Diagram ثم قم باختيار الكلمة Property من الشجرة Properties وانظر للزاوية اليمنى السفلى نعم انها نافذة الخصائص Properties حيث ستظهر كما في الشكل (3-8).



الشكل (3-8)

توفر نافذة الخصائص إمكانية الوصول إلى جميع المزايا التي يمكننا استخدامها لإنشاء الخصائص فيمكننا هنا تحديد مستوى الوصولية من خلال مربع القائمة المنبثقة Access وتحديد نوع القيم المعادة من خلال القائمة المنبثقة type وكذلك اسم الخاصية من خلال مربع النص Name كما يمكننا إضافة مقيدات وصول أخرى من خلال مربعات الاختيارات أسفل التبويب Modifiers لاحظ أيضا أنه يمكننا أن نضع تعليقات مصدرية بلغة XML للخاصية الذي أضفناه من نافذة الخصائص تحت التبويب XML Documentation.

وبالإضافة لذلك يمكننا تحديد كل ما تحدثنا عنه الآن من خلال النافذة Class Details الشكل (3-9).



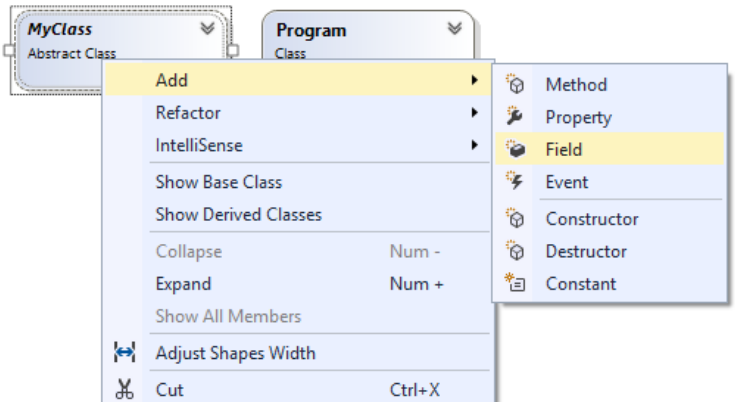
الشكل (3-9)

وبالطبع فإن معالج إضافة الخصائص لا يوفر لنا شيفرة جسم الخاصية وإنما يوفر البنية الأساسية للخاصية ويقلل من أخطاء كتابة تعريف الخاصية.

معالج إضافة حقل:

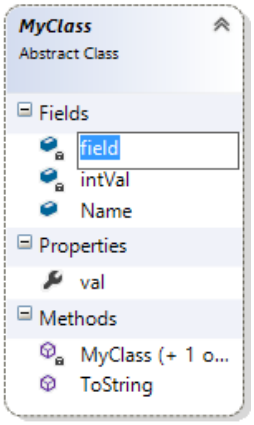
The Add Field Wizard:

بعد فتح نافذة Class Diagram يمكننا اختيار أحد الأصناف الموجودة في مشروعنا ثم الضغط عليها بزر الفأرة الأيمن حيث تظهر قائمة منسدلة من الأمر Add تظهر قائمة فرعية تحوي على العديد من الخيارات نختار منها الخيار Field الشكل (3-10) يظهر قائمة الخيارات الفرعية للأمر Add.



الشكل (3-10)

يمكننا المعالج المساعد للحقول من إضافة حقل إلى الصنف بسهولة ويسر ويمكننا تحقيق ذلك من خلال تغير شكل صندوق الصنف في نافذة Class Diagram الذي يتغير عند اختيار الأمر Field من القائمة الفرعية حيث يظهر الشكل (3-11).



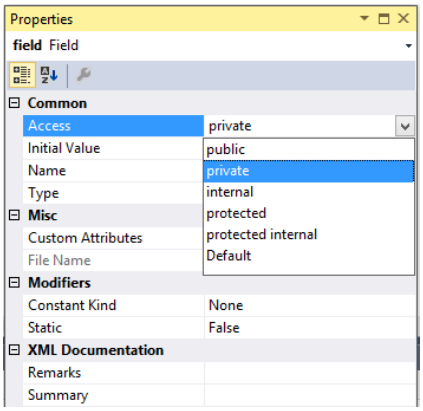
الشكل (3-11)

وباختيار اسم للحقل وضغط الزر Enter نكون بذلك قد أضفنا حقل جديدة لصنفنا MyClass لكن السؤال هنا ما هي صفات هذا الحقل هل هو عام أم خاص أم ..

لنذهب إلى الملف MyClass.cs سنجد ضمن شيفرة هذا الصنف قد أضيفت شيفرة حقل جديدة باسم Field كما يلي:

```
private int field;
```

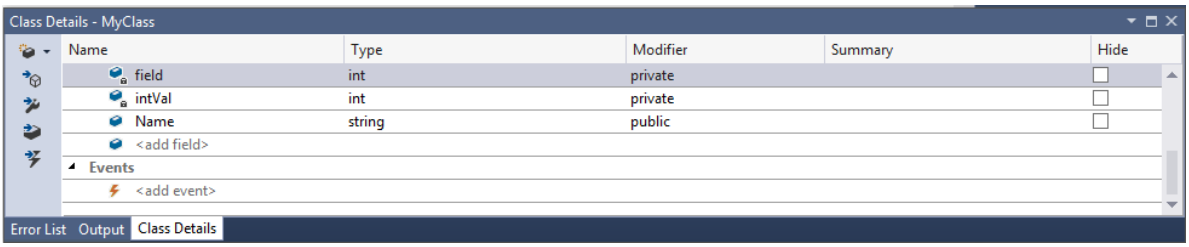
كما ترى من الشيفرة فإن هذه الحقل يحمل نوع وصولية خاص Private لكن هل يمكننا تغيير مدى وصولية هذا الحقل. بالتأكيد لكن كيف. لا تقلق عد إلى النافذة Class Diagram ثم قم باختيار الكلمة Field من الشجرة Fields وانظر للزاوية اليمنى السفلى نعم انها نافذة الخصائص Properties حيث ستظهر كما في الشكل (3-12).



الشكل (3-12)

توفر نافذة الخصائص إمكانية الوصول إلى جميع المزايا التي يمكننا استخدامها لإنشاء الحقول فيمكننا هنا تحديد مستوى الوصولية من خلال مربع القائمة المنبثقة Access وتحديد نوع القيم المعادة من خلال القائمة المنبثقة type وكذلك اسم الخاصية من خلال مربع النص Name كما يمكننا إضافة مقيدات وصول أخرى من خلال مربعات الاختيارات أسفل التبويب Modifiers لاحظ أيضا أنه يمكننا أن نضع تعليقات مصدرية بلغة XML للحقل الذي أضفناه من نافذة الخصائص تحت التبويب XML Documentation.

وبالإضافة لذلك يمكننا تحديد كل ما تحدثنا عنه الآن من خلال النافذة Class Details الشكل (13-3).

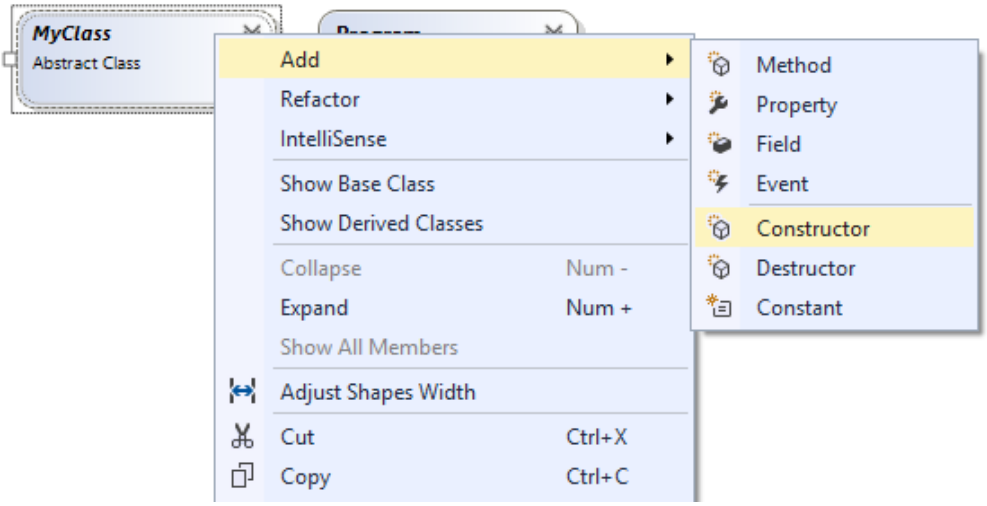


الشكل (13-3)

معالج إضافة مناهج البناء:

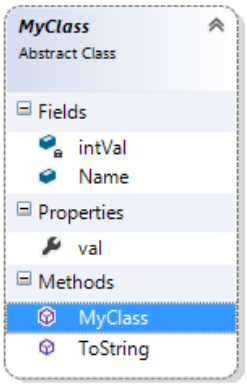
The Add constructor Wizard:

بعد فتح نافذة Class Diagram يمكننا اختيار أحد الأصناف الموجودة في مشروعنا ثم الضغط عليها بزر الفأرة الأيمن حيث تظهر قائمة منسدلة من الأمر Add تظهر قائمة فرعية تحوي على العديد من الخيارات نختار منها الخيار Constructor الشكل (14-3) يظهر قائمة الخيارات الفرعية للأمر Add.



الشكل (14-3)

يمكننا المعالج المساعد لمناهج البناء من إضافة منهج بناء إلى الصنف بسهولة ويسر ويمكننا تحقيق ذلك من خلال تغير شكل صندوق الصنف في نافذة Class Diagram الذي يتغير عند اختيار الأمر Constructor من القائمة الفرعية حيث يظهر الشكل (3-15).

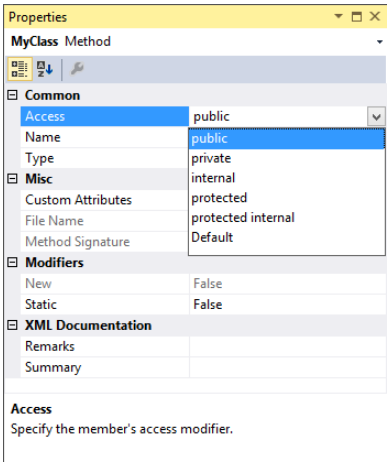


الشكل (3-15)

لنذهب إلى الملف MyClass.cs سنجد ضمن شيفرة هذا الصنف قد أضيفت شيفرة منهج بناء افتراضي للصنف MyClass جديدة باسم MyClass كما يلي:

```
public MyClass()
{
    throw new System.NotImplementedException();
}
```

كما ترى من الشيفرة فإن منهج البناء هذا يحمل نوع وصولية خاص public لكن هل يمكننا تغيير مدى وصولية منهج البناء هذا. بالتأكيد لكن كيف. لا تقلق عد إلى النافذة Class Diagram ثم قم باختيار الكلمة MyClass من الشجرة Methods وانظر للزاوية اليمنى السفلى نعم انها نافذة الخصائص Properties حيث ستظهر كما في الشكل (3-16).

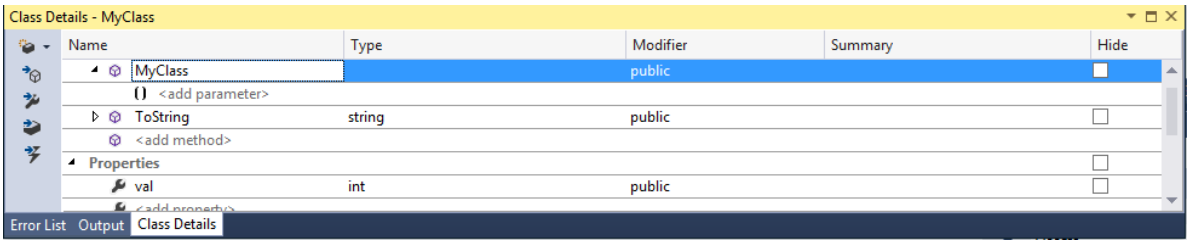


الشكل (3-16)

توفر نافذة الخصائص إمكانية الوصول إلى جميع المزايا التي يمكننا استخدامها لإنشاء مناهج البناء فيمكننا هنا تحديد مستوى الوصولية من خلال مربع القائمة المنبثقة Access لاحظ أيضا أنه يمكننا أن نضع تعليقات مصدرية بلغة XML لمنهج البناء الذي أضفناه من نافذة الخصائص تحت التبويب XML Documentation.

وبالإضافة لذلك يمكننا تحديد كل ما تحدثنا عنه الآن من خلال النافذة Class Details الشكل (3-17).

كما تمكننا النافذة Class Details من بناء مناهج بناء غير افتراضية عن طريق إضافة بارامترات كما تعلمنا ذلك سابقا في معالج إنشاء المناهج.

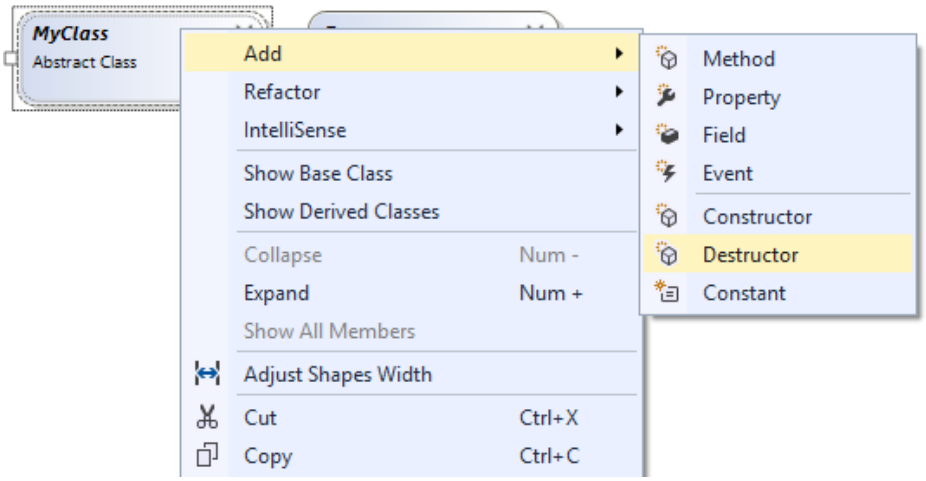


الشكل (3-17)

معالج إضافة مناهج التدمير:

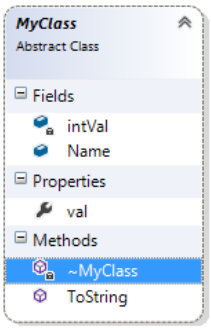
The Add constructor Wizard:

بعد فتح نافذة Class Diagram يمكننا اختيار أحد الأصناف الموجودة في مشروعنا ثم الضغط عليها بزر الفأرة الأيمن حيث تظهر قائمة منسدلة من الأمر Add تظهر قائمة فرعية تحوي على العديد من الخيارات نختار منها الخيار Destructor الشكل (3-18) يظهر قائمة الخيارات الفرعية للأمر Add.



الشكل (3-18)

يمكننا المعالج المساعد لمناهج التدمير من إضافة منهج تدمير إلى الصنف بسهولة ويسر ويمكننا تحقيق ذلك من خلال تغير شكل صندوق الصنف في نافذة Class Diagram الذي يتغير عند اختيار الأمر Destructor من القائمة الفرعية حيث يظهر الشكل (3-19).



الشكل (3-19)

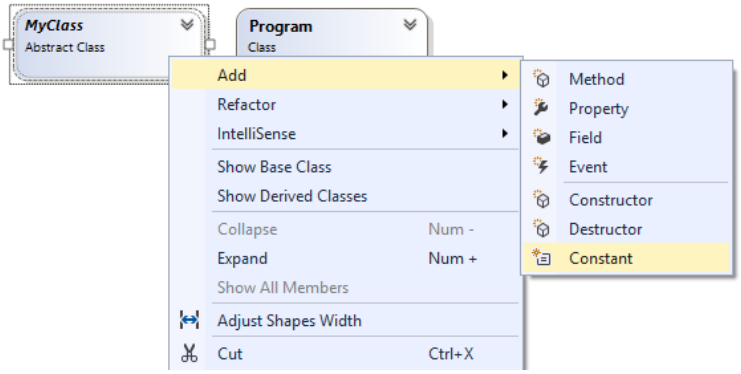
لنذهب إلى الملف MyClass.cs سنجد ضمن شيفرة هذا الصنف قد أضيفت شيفرة منهج تدمير للصنف MyClass جديدة باسم MyClass~ كما يلي:

```
~MyClass()
{
    throw new System.NotImplementedException();
}
```

معالج إضافة الثوابت:

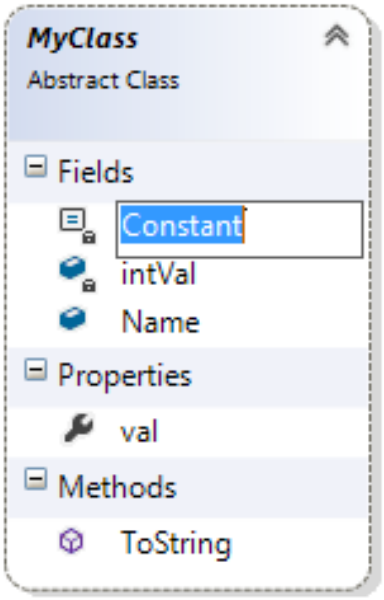
The Add Constant Wizard:

بعد فتح نافذة Class Diagram يمكننا اختيار أحد الأصناف الموجودة في مشروعنا ثم الضغط عليها بزر الفأرة الأيمن حيث تظهر قائمة منسدلة من الأمر Add تظهر قائمة فرعية تحوي على العديد من الخيارات نختار منها الخيار Constant الشكل (3-20) يظهر قائمة الخيارات الفرعية للأمر Add.



الشكل (3-20)

يمكننا المعالج المساعد للثوابت من إضافة ثوابت إلى الصنف بسهولة ويسر ويمكننا تحقيق ذلك من خلال تغير شكل صندوق الصنف في نافذة Class Diagram الذي يتغير عند اختيار الأمر Constant من القائمة الفرعية حيث يظهر الشكل (3-21).



الشكل (3-21)

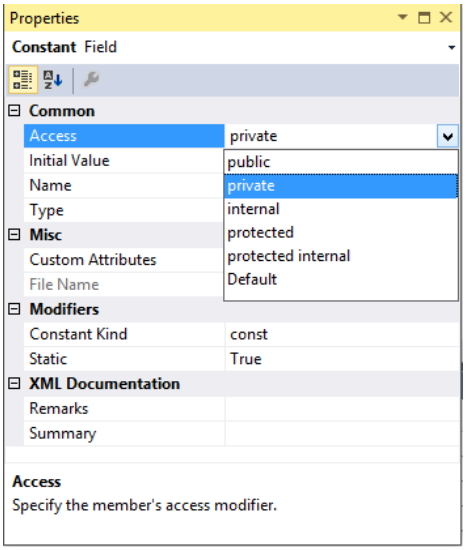
لنذهب إلى الملف MyClass.cs سنجد ضمن شيفرة هذا الصنف قد أضيفت شيفرة ثابت جديد للصنف MyClass باسم Constant كما يلي:

```
private const int Constant = 1;
```

كما ترى من الشيفرة فإن هذا الثابت يحمل نوع وصولية خاص private وهو من النوع int ويحمل قيمة افتراضية تساوي 1 لكن هل يمكننا تغيير مدى وصولية هذا الحقل ونوع البيانات وقيمته.

بالتأكيد لكن كيف. لا تقلق عد إلى النافذة Class Diagram ثم قم باختيار الكلمة Constant من الشجرة Fields وانظر للزاوية اليمنى السفلى نعم انها نافذة الخصائص Properties حيث ستظهر كما في الشكل (3-22).

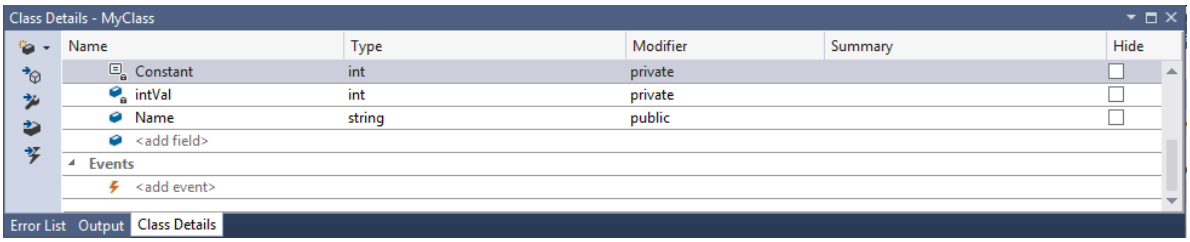
توفر نافذة الخصائص إمكانية الوصول إلى جميع المزايا التي يمكننا استخدامها لإنشاء الثوابت فيمكننا هنا تحديد مستوى الوصولية من خلال مربع القائمة المنبثقة Access لاحظ أيضا أنه يمكننا تغيير اسم الثابت من خلال مربع النص Name كما يمكننا تغيير قيمته من خلال مربع النص initial vale وتغيير نوع البيانات من مربع النص Type كما يمكننا تغيير مدى الوصولية من تبويب Modifiers من مربع القائمة المنبثقة Constant Kind كما يمكننا أن نضع تعليقات مصدرية بلغة XML لهذا الثابت من نافذة الخصائص تحت التبويب XML Documentation.



الشكل (3-22)

وبالإضافة لذلك يمكننا تحديد كل ما تحدثنا عنه الآن من خلال النافذة Class Details الشكل (3-23).

لاحظ في النافذة Class Details وجود مربع اختيار أسفل الكلمة Hide وقد تدثنا عن هذا الخيار سابقا

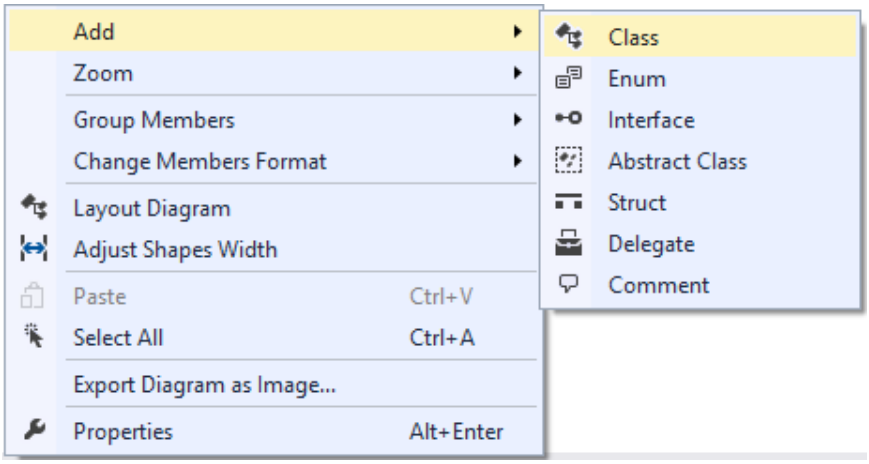


الشكل (3-23)

وبذلك يتبقى لدينا المعالج الأخير من هذه القائمة وهو معالج الأحداث Event سوف نتحدث عنه أثناء حديثنا عن الأحداث في لغة C#.

ملاحظة:

تمكننا نافذة Class Diagram من استخدام معالجات مساعدة أخرى قم بالنقر بزر الفأرة الأيمن في المساحة البيضاء من نافذة Class Diagram ثم قم باختيار الأمر Add فنلاحظ ظهور قائمة فرعية تحوي على مجموعة من الأوامر التي تستدعي المعالجات المساعدة الشكل (3-24) لإنشاء الأصناف Class والتعدادات Enum والواجهات Interface والأصناف المجردة Abstract Class والمفوضات Delegate والبنى Struct التعليقات المصدرية Comment لا تقلق سوف نتعرف عليها جميعها.

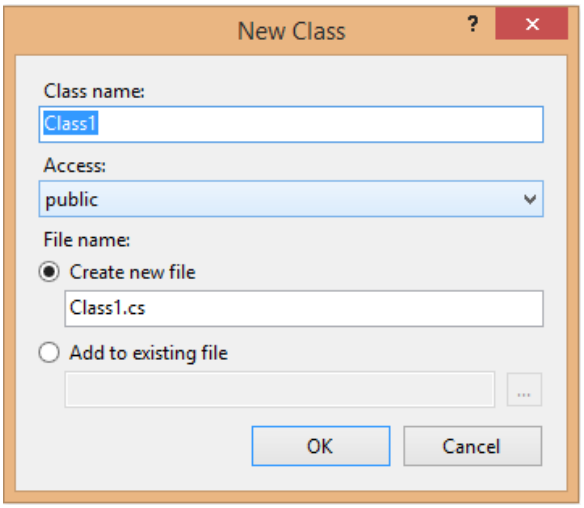


الشكل (3-24)

معالج إضافة الأصناف:

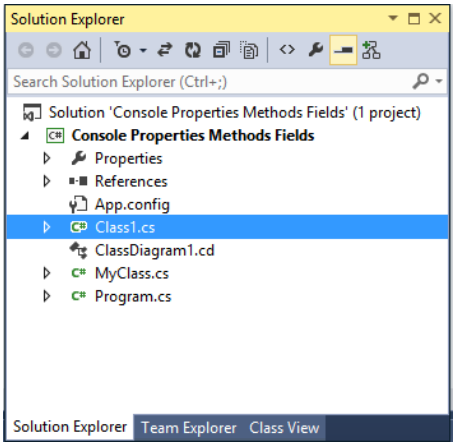
The Add Class Wizard:

قم باختيار الخيار Class من القائمة الفرعية لأمر Add فيظهر صندوق حوار كما في الشكل (3-25).



الشكل (3-25)

يمكننا صندوق الحوار من اختيار اسم الصنف من صندوق النص Class name كما يمكننا من تحديد مدى وصولية الصنف هذا من صندوق الاختيار Access لاحظ أن صندوق الحوار هذا له خياران إما إنشاء ملف جديد Create new file أو إضافة لملف موجود Add to exiting file فإذا اخترنا خيار إنشاء ملف سوف يضاف ملف جديد باسم Class1.cs إلى نافذة Solution Explorer الشكل (3-26).



الشكل (3-26)

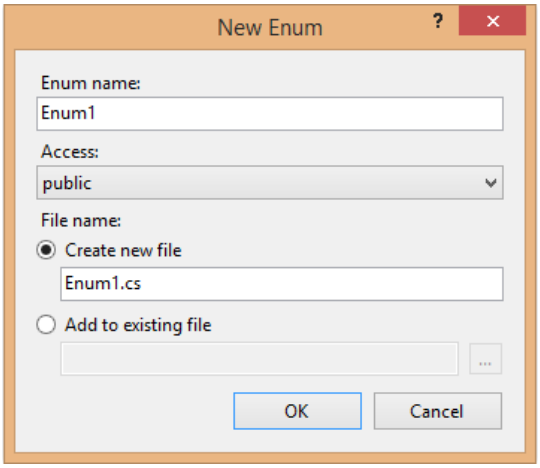
باختيار صندوق Class1 من نافذة Class Diagram يمكننا تحرير خصائص صنفنا الجديد من نافذة properties حيث نتمكن من تبويب Modifiers من الوصول إلى مدى الوصولية Inheritance Modifiers حيث يمكننا تحديد فيما إذا كان الصنف مغلقا أو مجردا أو صنفا ستاتيكيا كما تعلمنا سابقا تمكننا نافذة الخصائص من تحرير العديد من الأمور لقد أصبحت قادرا على التعامل معها جميعها.

وكما تعلمنا سابقا يمكننا اختيار الصنف من نافذة Class Diagram بالضغط بز الفأرة الأيمن يمكننا إضافة المزيد من الكائنات لهذا الصنف.

معالج إضافة التعدادات:

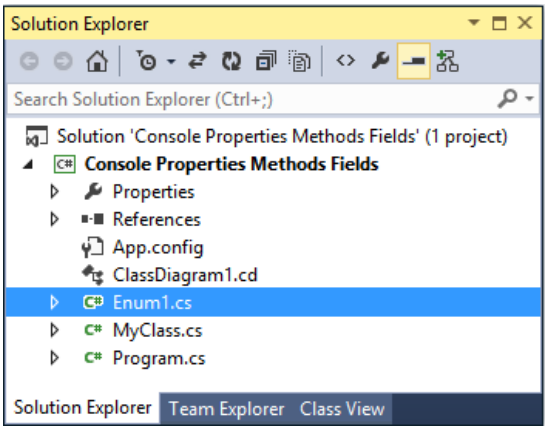
The Add Enum Wizard:

قم باختيار الخيار Enum من القائمة الفرعية لأمر Add فيظهر صندوق حوار كما في الشكل (3-27).



الشكل (3-27)

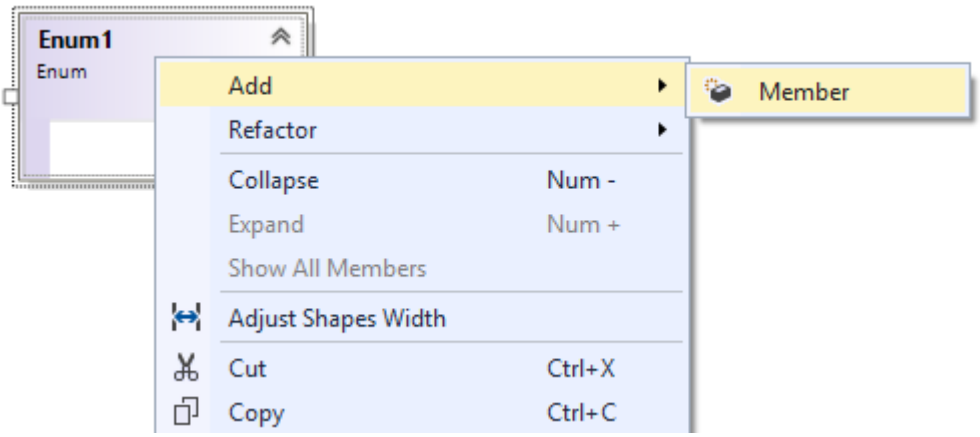
يمكننا صندوق الحوار هذا من اختيار اسم التعداد من صندوق النص Enum name كما يمكننا من تحديد مدى وصولية التعداد هذا من صندوق الاختيار Access لاحظ أن صندوق الحوار هذا له خياران إما إنشاء ملف جديد Create new file أو إضافة ملف موجود Add to exiting file فإذا اخترنا خيار إنشاء ملف سيضاف ملف جديد باسم Enum1.cs إلى نافذة Solution Explorer الشكل (3-28).



الشكل (3-28)

باختيار صندوق Enum1 من نافذة Class Diagram يمكننا تحرير خصائص تعدادنا الجديد من نافذة properties كما تعلمنا سابقا تمكننا نافذة الخصائص من تحرير العديد من الأمور لقد أصبحت قادرا على التعامل معها جميعها.

وكما تعلمنا سابقا يمكننا اختيار التعداد من نافذة Class Diagram بالضغط بز الفأرة الأيمن حيث يمكننا إضافة أعضاء هذا العداد الشكل (3-29).



الشكل (3-29)

وباختيار الخيار Member يمكننا إضافة عضو جديد للتعداد حيث يتغير شكل صندوق التعداد Enum1 كما في الشكل (3-30) حيث يمكننا تحديد اسم العضو الجديد في التعداد وتمكننا نافذة الخصائص

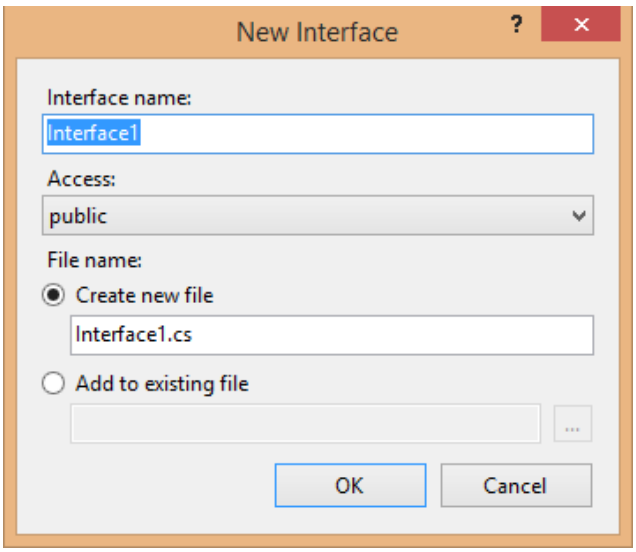
properties ونافذة Class Details من تحرير اسم العضو أيضا كما تمكننا من تحديد قيمة value لهذا العضو راجع تعريف التعدادات.

الشكل (3-30)

معالج إضافة الواجهات:

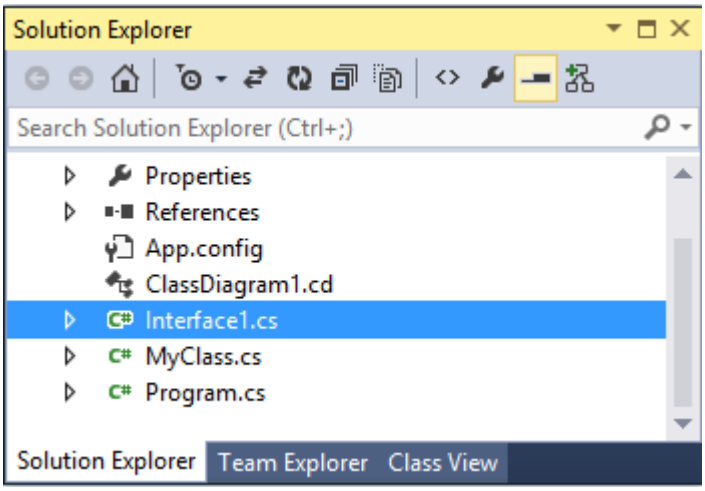
The Add Interface Wizard:

قم باختيار الخيار Interface من القائمة الفرعية للأمر Add فيظهر صندوق حوار كما في الشكل (3-31).



الشكل (3-31)

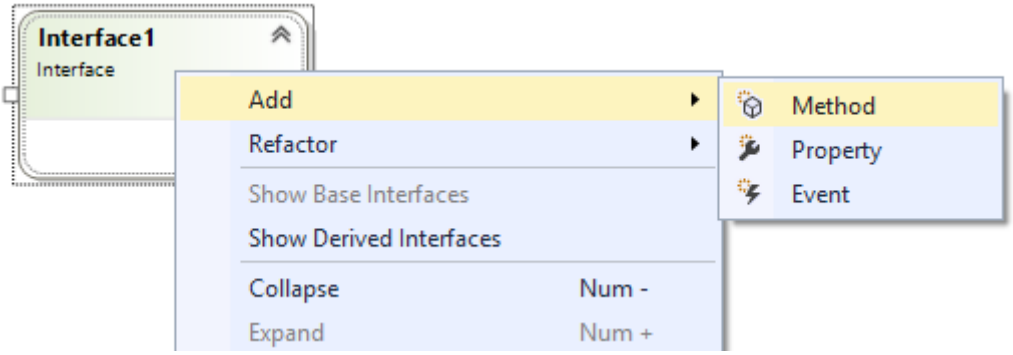
يمكننا صندوق الحوار هذا من اختيار اسم الواجهة من صندوق النص Interface name كما يمكننا من تحديد مدى وصولية الواجهة هذه من صندوق الاختيار Access لاحظ أن صندوق الحوار هذا له خياران إما إنشاء ملف جديد Create new file أو إضافة ملف موجود Add to exiting file فإذا اخترنا خيار إنشاء ملف سيضاف ملف جديد باسم Interface1.cs إلى نافذة Solution Explorer الشكل (3-32).



الشكل (3-32)

باختيار صندوق Interface1 من نافذة Class Diagram يمكننا تحرير خصائص الواجهة الجديدة من نافذة properties كما تعلمنا سابقا تمكننا نافذة الخصائص من تحرير العديد من الأمور لقد أصبحت قادرا على التعامل معها جميعها.

وكما تعلمنا سابقا يمكننا اختيار الواجهة من نافذة Class Diagram بالضغط بز الفأرة الأيمن حيث يمكننا إضافة أعضاء هذه الواجهة الشكل (3-33).



الشكل (3-33)

والآن سوف اترك لك التعرف على المعالجات المساعدة لإضافة الأصناف المجردة Abstract Class والمفوضات Delegate والبنى Struct أما المعالج المساعد لإضافة التعليقات المصدرية Comment سوف نتحدث عنه في المستقبل.

ملاحظة:

لاحظ ان جميع هذه المعالجات تمككنا من إضافة المكونات الجديدة ضمن ملفات مفصلة ضمن جسم المشروع أو الحل وهذا أسلوب جديد في Visual Studio 2013.

لكن السؤال هنا هل هذه الملفات تتبع لمجال أسماء مختلف الجواب سيأتيك إذا ما ضغط على أحد هذه الملفات ضمن نافذة Solution Explorer لتظهر نافذة الشيفرة لهذا الملف وليكن ملف Struct1.cs فيظهر الكود التالي:

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

namespace Console_Properties_Methods_Fields
{
    public struct Struct1
    {
    }
}
```

نلاحظ من خلال الشيفرة أنها تتبع لمجال أسماء مشروعنا أي وكننا كتبناها أسفل مجال الأسماء لمشروعنا.

مواضيع أخرى متعلقة بأعضاء الأصناف:

Additional Class Member Topics:

والآن وبعد أن تناولنا أساسيات تعريف الأعضاء فقد حان الوقت لتناول مواضيع أكثر تقدما وسوف نتحدث في هذا القسم عن:

- إخفاء مناهج الصنف الأساس.
- استدعاء مناهج الصنف الأساس المخفية أو التي تم تجاوزها.
- تعريفات الأنواع المعششة.

إخفاء مناهج الصنف الأساس:

Hiding Base Class Methods:

عندما نقوم بوراثة عضو غير مجرد من صنف أساس فإننا نرث أيضا شيفرة هذا العضو إذا كان منهجا أو خاصية طبعا وإذا ورثنا عضوا ظاهريا يمكننا أن نتجاوز شيفرة هذا العضو بتعريف نفس العضو ولكن باستخدام الكلمة override وبغض النظر عما إذا كان العضو ظاهريا أم لا فإن بإمكاننا حجب شيفرة الأعضاء المورثة إن ذلك مفيد أحيانا على سبيل المثال يمكن لهذا العضو أن يحتوي على شيفرة عامة لا تتوافق مع الصنف المشتق مثلا.

ويمكننا القيام بذلك كما في المثال التالي:

```
public class MyBaseClass
{
    public void DoSomething()
    {
        // Base implementation
    }
}
public class MyDerivedClass:MyBaseClass
{
    public void DoSomething()
    {
        // derived class implementation, hides base implementation
    }
}
```

لقد قمنا هنا بتوريث الصنف MyBaseClass للصنف MyDerivedClass لاحظ أن الصنف الأساس يحتوي على المنهج DoSomething() ويمكننا حجب شيفرة هذا المنهج في الصنف MyDerivedClass بمجرد إنشاء المنهج نفسه ضمن هذا الصنف.

على الرغم من أن هذه الشيفرة تعمل بصورة جيدة إلا أنها ستولد تحذيرا يبين أننا قمنا بحجب عضو في الصنف الأساس وسيعطينا هذا التحذير فرصة تصحيح الأمور إذا كنا قد قمنا بعملية الحجب تلك عن طريق الخطأ وهذا هو مضمون رسالة التحذير.

1 Warning ConsoleApplication3.MyDerivedClass.DoSomething()' hides ' inherited member 'ConsoleApplication3.MyBaseClass.DoSomething()'. Use the new keyword if hiding was intended

على كل حال هناك طريقة صريحة يمكننا استخدامها لحجب أعضاء الصنف الأساس وذلك باستخدام الكلمة new كما أشارت إلى ذلك رسالة التحذير كما في الشيفرة التالية:

```
public class MyBaseClass
{
    public void DoSomething()
    {
        // Base implementation
    }
}
public class MyDerivedClass : MyBaseClass
{
    new public void DoSomething()
    {
        // Derived class implementation, hides base implementation
    }
}
```

ستعمل هذه الشيفرة تماما كما في الشيفرة السابقة إلا أننا هنا سنتجنب رسالة التحذير.

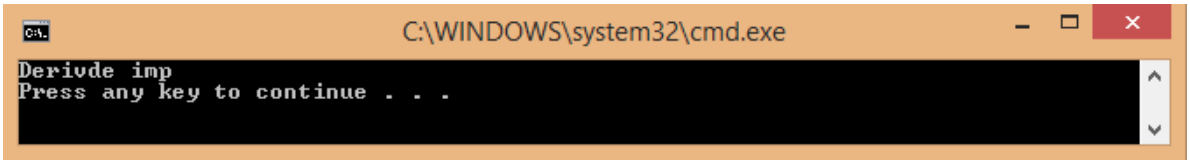
وعند هذه النقطة من الجدير بنا أن ننوه إلى الفرق بين حجب وتجاوز أعضاء الصنف الأساس ولنأخذ الشيفرة التالية مثلا.

```
public class MyBaseClass
{
    public virtual  void DoSomething()
    {
        Console.WriteLine("Base imp");
    }
}
public class MyDerivedClass : MyBaseClass
{
    public override  void DoSomething()
    {
        Console.WriteLine("Derived imp");
    }
}
```

لقد قمنا هنا بتجاوز المنهج DoSomething() في الصنف الأساس واستخدمنا تزويدا آخر له ضمن الصنف المشتق MyDerivedClass وفي هذه الحالة فإن أي كائن من الصنف MyDerivedClass سيستخدم التزويد الجديد للمنهج وحتى إن قمنا باستدعاء هذا المنهج من خلال الصنف الأساس لكائن من الصنف المشتق (أي باستخدام تعددية الأشكال):

```
public class MyBaseClass
{
    public virtual  void DoSomething()
    {
        Console.WriteLine("Base imp");
    }
}
public class MyDerivedClass : MyBaseClass
{
    public override  void DoSomething()
    {
        Console.WriteLine("Derivde imp");
    }
}

class Program
{
    static void Main(string[] args)
    {
        MyDerivedClass myObj = new MyDerivedClass();
        MyBaseClass myBaseObj;
        myBaseObj = myObj;
        myBaseObj.DoSomething();
    }
```

إن تنفيذ هذه الشيفرة سوف يعطينا الخرج التالي:

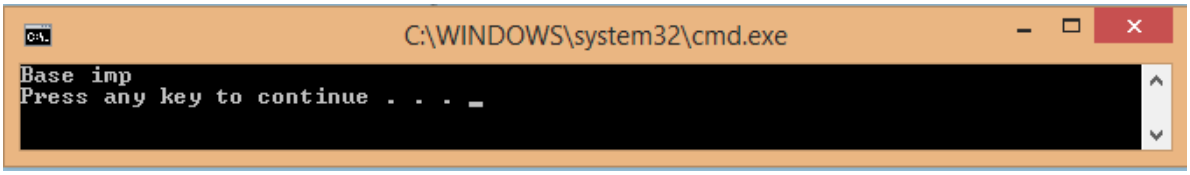


الشكل (3-34)

أما إذا قمنا بحجب المنهج DoSomething() في الصنف الأساس:

```
public class MyBaseClass
{
    public virtual  void DoSomething()
    {
        Console.WriteLine("Base imp");
    }
}
public class MyDerivedClass : MyBaseClass
{
    new public void DoSomething()
    {
        Console.WriteLine("Derived imp");
    }
}

class Program
{
    static void Main(string[] args)
    {
        MyDerivedClass myObj = new MyDerivedClass();
        MyBaseClass myBaseObj;
        myBaseObj = myObj;
        myBaseObj.DoSomething();
    }
```

فإن النتيجة ستكون كما في الشكل (3-35).



الشكل (3-35)

ملاحظة:

لاحظ أنه ليس من الضروري أن يكون العضو ظاهريا أي تم تعريفه باستخدام الكلمة Virtual وذلك كي نتمكن من حجبه في الصنف المشتق عليك أن تدرك الفرق بين حجب العضو وتجاوزه.

لاحظ هنا على الرغم من أن شيفرة المنهج في الصنف الأساس محجوبة إلا أنه مازال بإمكاننا الوصول إليها وذلك طبعا من خلال الصنف الأساس وباستخدام تقنية تعددية الأشكال.

استدعاء مناهج الصنف الأساس المحجوبة والتي تم تجاوزها:

Calling Overridden or Hidden Base Class Methods:

سواء تم تجاوز او حجب عضو في الصنف الأساس من الصنف المشتق فإن بإمكاننا الوصول إلى ذلك العضو من داخل الصنف المشتق وهناك عدة أوضاع نحتاج فيها إلى ذلك على سبيل المثال:

- ✓ عندما نود حجب العضو العام الموروث عن مستخدمي صنف مشتق ما مع إمكانية الوصول إلى وظائف هذا العضو ضمن الصنف.
- ✓ عندما نود إضافة تزويد للعضو الظاهري الموروث بدلا من مجرد استبداله بتزويد جديد أي تجاوزه ولتحقيق ذلك يمكننا استخدام الكلمة base والتي تشير إلى الصنف الأساس دائما على سبيل المثال:

```
public class MyBaseClass
{
    public virtual void DoSomething()
    {
        // Base implementation
    }
}
public class MyDerivedClass : MyBaseClass
{
    public override void DoSomething()
    {
        // Derived class implementation extends base class
        // implementation
        base.DoSomething();
        // More derived class implementatio
    }
}
```

تقوم هذه الشيفرة بتنفيذ منهج DoSomething() الموجود ضمن الصنف MyBaseClass والذي يمثل الصنف الأساس للصنف MyDerivedClass وذلك من ضمن المنهج DoSomething() الموجود ضمن الصنف MyDerviedClass.

وباعتبار أن الكلمة base تعمل وفقا لحالات الكائنات فإنه لا يمكننا استخدامها مع عضو ستاتيكي.

الكلمة this:

This keyword:

وبالإضافة إلى الكلمة base التي يمكننا استخدامها للوصول إلى أعضاء الصنف الأساس فإن هناك الكلمة this والتي يمكننا استخدامها للوصول إلى أعضاء الصنف الحالي وهي كالكلمة base حيث لا تستخدم إلا مع كائنات الصنف إي عندما نستخدم هذه الكلمة فإننا نشير إلى حالة من الصنف وليس إلى الصنف بصورة عامة وبالتالي فإنه لا يمكننا الوصول إلى الأعضاء الستاتيكية بواسطتها.

إن أكثر الاستخدامات أهمية لهذه الكلمة عندما نمرر مرجعا للكائن الحالي من خلالها كما يلي:

```
public void DoSomething()
{
    MyTargetClass myObj = new MyTargetClass();
    myObj.DoSomethingWith(this);
}
```

لقد قمنا هنا بإنشاء كائن باسم myObj من النوع MyTargetClass ومن ثم قمنا باستدعاء المنهج myObj.DoSomethingWith() لهذا الكائن بحيث يأخذ بارامتر واحدا حيث يمثل هذا البارامتر كائنا من نفس النوع الذي يحتوي على المنهج DoSomething وبالتالي يمكن لنوع البارامتر أن يكون من نفس نوع الصنف الحالي أو يمكن أن يكون من نفس نوع الصنف الأساس للصنف الحالي أو من نوع واجهة يستخدمها الصنف ويمكن أن يكون طبعا من النوع System.Object.

تعريفات الأنواع المعششة:

Nested Type Definitions:

وبالإضافة إلى تعريف الأنواع كالأصناف في فضاءات الأسماء فإنه يمكننا أن نعرف الأصناف ضمن أصناف أخرى وإذا أردنا القيام بذلك عندئذ سنتمكن من استخدام مجال تام من مقيدات الوصول لتعريف الأصناف وأعضائها وذلك بدلا من مجرد استخدام المقيد public والمقيد internal ويمكننا ان نستخدم الكلمة new أيضا لإخفاء تعريف النوع الموروث من صنف أساس.

على سبيل المثال تقوم الشيفرة التالية بتعريف الصنف MyClass وتعريف صنفا معششا ضمنه باسم MyNestedClass:

```
public class MyBaseClass
{
    public class MyNestedClass
    {
        public int nestedClassField;
    }
}
```

وبالتالي إذا أردنا أن ننشئ كائنا من النوع MyNestedClass من خارج الصنف MyClass فإن علينا أن نبين اسم النوع التام كما يلي:

```
MyBaseClass.myNestedClass myObj = new MyBaseClass.myNestedClass();
```

لاحظ أننا لن نتمكن من إنشاء كائن من الصنف المعشعش إذا كان معرفا على انه صنف خاص private أو استخدام مقيد وصول غير متوافق مع الشيفرة في النقطة التي تم فيها إنشاء حالة من الكائن.

إن السبب الرئيسي لوجود ميزة كهذه يعود إلى تعريف أصناف خاصة أي أنها private بالنسبة للصنف الذي يحتويها وبالتالي لا يمكن لأي شيفرة في فضاء الأسماء الوصول إلى تلك الأصناف عدا الصنف الذي يحتويها فقط.

التزود بالواجهات:

Interface Implementation:

سوف نتناول في هذا القسم كيفية تعريف الواجهات وتزويد الأصناف بها لقد رأينا في الفصل السابق أن الواجهات تعرف بصورة مشابهة لتعريف الأصناف وذلك باستخدام شيفرة كما يلي مثلا:

```
interface IMyInterface
{
    // interface members
}
```

وتعرف أعضاء الواجهة تماما كأعضاء الأصناف إلا أن هناك عددا من الاختلافات المهمة بين أعضاء الأصناف وأعضاء الواجهة:

- لا يمكن استخدام مقيدات الوصول أي الكلمات (puplic,private,protected,internal) فجميع أعضاء الواجهة هي أعضاء عامة بصورة مطلقة.
- لا يمكن لأعضاء الواجهة أن تتضمن على شيفرة برمجية وإنما مجرد تعريف فقط.
- لا يمكن تعريف حقول ضمن الواجهة.
- لا يمكن تعريف أعضاء الواجهة باستخدام الكلمات (static,virtual,abstract,sealed).
- إن تعريف أنواع ضمن الواجهات غير مسموح به.

بينما يمكننا أن نعرف الأعضاء باستخدام الكلمة new وذلك إذا أردنا أن نحجب الأعضاء الموروثة من الواجهات الأساس على سبيل المثال:

```
interface IMyBaseInterface
{
    void DoSomething();
}
interface IMyDerivedInterface:IMyBaseInterface
{
    new void DoSomething();
}
```

لقد قمنا هنا بحجب المنهج DoSomething() من الواجهة المشتقة واستخدام المنهج DoSomething المعرف ضمن الواجهة الأساس.

ويمكننا تعريف الخصائص ضمن الواجهات أيضا وبتعريفها فإننا نحدد ما إذا كانت الخاصية تتضمن الكتلة set أو الكتلة get أو كلتاهما على سبيل المثال:

```
interface IMyInterface
{
    int MyInt
    {
        get;
        set;
    }
}
```

لقد عرفنا هنا خاصية MyInt ضمن الواجهة IMyInterface وهي خاصية قراءة/كتابة ويمكننا أن نحذف السطر get أو set فيما إذا كنا نود إنشاء خاصية للقراءة فقط أو الكتابة فقط.

بما انه لا يمكننا كتابة شيفرة ضمن أعضاء الواجهة فإنه لا يمكننا تحديد كيفية حفظ قيمة الخاصية هنا فالواجهات لا يمكنها أن تحتوي على حقول.

وأخيرا فإن الواجهات يمكن أن تعرف كالأصناف كأعضاء لأصناف لكن ليس كأعضاء لواجهات أخرى وذلك باعتبار ان الواجهات لا يمكن أن تحتوي على تعريفات لأنواع.

تزويد الأصناف بالواجهات:

Implementing Interface in Class:

يجب على الصنف المزود بواجهة أن يحتوي على تزويد لجميع أعضاء تلك الواجهة بحيث يتطابق توقيع (تعريف) هذه الأعضاء في الصنف مع توقيعها (تعريفها) في الواجهة بالإضافة إلى تطابق وجود الكتلتين get وset للخصائص في الصنف كما هي في الواجهة ويجب أن تكون تلك الأعضاء المزودة عامة ومن الممكن ان نزود أعضاء الواجهة باستخدام الكلمات virtual أو abstract لكن لا يمكن استخدام الكلمات static أو const على سبيل المثال:

```
public interface IMyInterface
{
    void DoSomething();
    void DoSomethingElse();
}
public class MyClass:IMyInterface
{
    public void IMyInterface.DoSomething ()
    {
    }
    public void DoSomethingElse()
    {
    }
}
```

ويمكن ان تزود أعضاء الواجهة على أصناف أساس على سبيل المثال:

```
public interface IMyInterface
{
    void DoSomething();
    void DoSomethingElse();
}
public class MyClass:IMyInterface
{
    public void DoSomething ()
    {
    }
}
public class MyDerivedClass:MyClass ,IMyInterface
{
    public void DoSomethingElse()
    {
    }
}
```

وعندما يرث صنف ما صنف أساس مزود بواجهة ما فإن الصنف المشتق سيدعم هذه الواجهة بصورة مطلقة على سبيل المثال:

```
public interface IMyInterface
{
    void DoSomething();
    void DoSomethingElse();
}
public class MyBaseClass:IMyInterface
{
    public virtual void DoSomething ()
    {
    }
    public virtual void DoSomethingElse()
    {
    }
}
public class MyDerivedClass:MyBaseClass
{
    public override void DoSomethingElse()
    {
    }
}
```

وكما ترى هنا فإنه من المفيد جدا أن نعرف الأعضاء التي تم تزويدها من واجهة في الصنف الأساس كأعضاء ظاهرية وذلك كي تتمكن الأصناف المشتقة من استبدال شيفرة هذه الأعضاء بشيفرة أخرى مناسبة للصنف المشتق وإذا قمنا بإخفاء عضو الصنف الأساس بالكلمة new بدلا من تجاوزه بهذه الطريقة فإن المنهج IMyInterface.DoSomething() سيشير دوما إلى إصدارة الصنف الأساس من هذا العضو.

تزويد أعضاء الواجهة بشكل صريح:

Explicit Interface Member Implementation:

يمكن تزويد أعضاء الواجهة بشكل صريح (explicit members) بواسطة الصنف وإذا قمنا بذلك فعندئذ يمكن الوصول إلى هذا العضو من خلال هذه الواجهة فقط وليس من خلال الصنف أما الأعضاء المطلقة (implicit members) وهي الأعضاء التي رأيناها في شيفرة القسم السابق قد يمكننا الوصول إليها إما عبر الصنف أو عبر الواجهة.

على سبيل المثال إذا تم تزويد المنهج DoSomething() في الواجهة IMyInterface ضمن الصنف MyClass بشكل مطلق كما قمنا بلك مسبقا عندئذ فإن الشيفرة التالية صحيحة:

```
MyClass myObj = new MyClass();
myObj.DoSomething();
```

تماما كالشيفرة التالية:

```
MyClass myObj = new MyClass();
IMyInterface myInt=myObj ;
myInt.DoSomething();
```

لكن إذا زود الصنف MyDerivedClass بالمنهج DoSomething() بشكل صريح فإننا لن نتمكن من الوصول إلى هذا المنهج من ضمن هذا الصنف طبعا إلا عند ذكر اسم الواجهة أيضا والشيفرة التالية تقوم بتزويد المنهج DoSomething() بشكل صريح والمنهج DoSomethingElse() بشكل مطلق:

```
public class MyClass:IMyInterface
{
    void IMyInterface.DoSomething()
    {
    }
    public void DoSomethingElse()
    {
    }
}
```

مثال تطبيقي:

Example Application:

سوف نقوم في هذا القسم من الفصل بوضع ما تعلمناه عن البرمجة كائنية التوجه OOP حتى الآن موضع التطبيق العملي وسوف نقوم هنا ببناء وحدة أصناف سنعتمد عليها في الفصول المقبلة تتألف وحدة أصناف هذه من صنفين رئيسيين:

- صنف ورقة اللعب Card والذي يمثل ورقة لعب قياسية لها شكل ورقم.

- صنف مجموعة ورق اللعب Deck والذي يمثل الـ 52 ورقة لعب المتوفرة في مجموعة ورق اللعب النظامية حيث يمكن الوصول إلى كل ورقة على حدة بحسب موقع الورقة من المجموعة بالإضافة إلى إمكانية إعادة ترتيب أوراق اللعب بصورة عشوائية.

سوف نقوم أيضا بتطوير برنامج زبون بسيط لاختبار وحدة الصنف تلك ولكن لن نستخدم هذه الوحدة في إنشاء تطبيق لعبة ورق كامل! (على الأقل حتى الآن)

توضيح:

نطلق مصطلح برنامج زبون (client application) على التطبيق الذي يقوم باستضافة أو استخدام تطبيق برمجي آخر ويمكن لهذا التطبيق البرمجي الآخر أن يمثل تطبيقا غير تنفيذي وبالتالي نعتمد على تطبيق الزبون في اختبار هذا التطبيق وفي الغالب نستخدم تطبيقات زبون لتجربة واستخدام مكتبات الأصناف.

لاحظ أن بإمكاننا معاملة جميع المشاريع التي نقوم بها منذ بداية هذا الكتاب على أنها تطبيقات زبون لمكتبات أصناف إطار عمل .NET.

التخطيط للتطبيق:

Planning the Application:

ستتضمن مكتبة الأصناف لهذا التطبيق (والتي سنسميها بـ CardLibrary) على هذين الصنفين ولكن قبل أن نبدأ بإنشاء التطبيق علينا ان نخطط لبنية التطبيق والوظائف التي ستقوم بها اصنافه.

فكرة:

في الحقيقة أن الخطوة الأولى في إنشاء أي تطبيق هي التخطيط لما ستقوم به وكيف يجب أن تسير الأمور وماذا يجب ان يحقق كل جزء من تطبيقك إن هذا النوع من التخطيط سيساعدك على فهم الأمور بشكل جيد ويجنبك البرمجة الاعتباطية.

صنف ورقة اللعب:

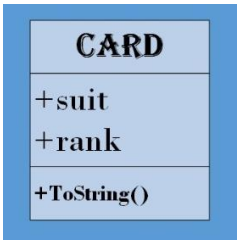
The Card Class:

بغض النظر عن صنف ورقة اللعب لنفكر في ورقة اللعب فقط مم تتكون ورقة اللعب؟ في الحقيقة إن أية ورقة لعب تمثل بواحدة من أربعة أشكال (Heart,Spade,Diamond,Club) ومن أحد ثلاثة عشرة رتبة تتراوح من الآس إلى الملك وبالتالي لكي نحدد ورقة لعب ما فإن علينا أن نحدد شكل ورتبة الورقة.

لنعد إلى صنف ورقة اللعب في الحقيقة إن الصنف Card يمثل حاو لحقلين للقراءة فقط: حقل الشكل suit وحقل الرتبة rank إن السبب الذي يجعلنا نستخدم حقولا للقراءة فقط هنا هو أنه من غير المنطقي أن تكون لدينا ورقة لعب فارغة بالإضافة إلى أنه من غير الطبيعي أن تتغير ورقة اللعب بعد إنشاءها ولتبسيط الأمور سنجعل منهج البناء الافتراضي لهذا الصنف منهجا خاصا وسنستخدم منهج بناء آخر يجب ان يأخذ بارامترين: الأول يمثل شكل الورقة والثاني يمثل رتبتها.

وبالإضافة على ذلك فإن الصنف Class سيتجاوز المنهج ToString() الذي يوفره الصنف الأب System.Object وذلك لأننا نود ان يعطي هذا المنهج نصا يدل على ورقة اللعب الحالية بدلا من مجرد نص عادي يشير إلى اسم الكائن (وهو السلوك الافتراضي لهذا المنهج ولتبسيط الأمور أكثر فإننا سنستخدم التعدادات لحصر القيم التي يمكن ان يأخذها الحقلان suit و rank.

يمكننا تمثيل صنف ورقة اللعب Card بالمخطط التالي:



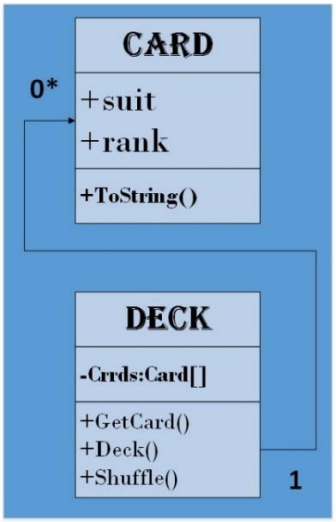
الشكل (3-36)

صنف مجموعة أوراق اللعب:

The Deck Class:

يجب أن يعالج هذا الصنف 52 كائنا من نوع صنف ورقة اللعب Card وسوف نستخدم لذلك مصفوفة من نوع Card لن نتمكن من الوصول إلى هذه المصفوفة بصورة مباشرة وإنما سنوفر منهجا باسم GetCard() للوصول إلى كل ورقة لعب في المصفوفة على حدة فهو مشابه لسحب ورقة لعب من مجموعة أوراق اللعب حيث سيعيد هذا المنهج كائنا من نوع Card وفقا لموقع ورقة اللعب تلك من المصفوفة.

سيتضمن هذا الصنف منهجا لخلط وإعادة ترتيب أوراق اللعب وسنسميه Shuffle() وبناء على ذلك يمكننا رسم تخطيط مكتبة الأصناف CardLibrary كما في الشكل التالي:



الشكل (3-37)

برمجة مكتبة أصناف ورق اللعب:

Writing the Class Library

ستفترض في هذا المثال أن بيئة Visual Studio 2013 أصبحت مألوفة بالنسبة إليك يجب أن تكون كذلك لذا فإننا لن نرتب ما سنقوم به ضمن خطوات بشكل صريح.

سنضع صنفينا Card و Deck بالإضافة إلى تعدادين في مشروع مكتبة أصناف Class Library باسم CardLibrary سيتضمن هذا المشروع ملفين بالامتداد cs. أحدهما Card.cs والذي يتضمن صنف ورقة اللعب Card بالإضافة إلى التعدادين Suit و Rank والآخر Deck.cs والذي يتضمن صنف مجموعة أوراق اللعب Deak.

الملف Card.cs:

Card.cs:

سوف نقوم هنا بتجزئة شيفرة Card.cs وذلك لشرح كل خطوة فيها وسنبدأ بالطبع مع تصريح فضاء الأسماء وتعليمة using.

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

namespace CardLibrary
```

{

بعد ذلك سنقوم بتعريف تعداد ليمثل أشكال ورقة اللعب Suit:

```
public enum Suit
{
    Clup,
    Diamond,
    Heart,
    Spade
}
```

ثم سنعرف تعداد آخر لتمثيل رتب الأوراق باسم Rank حيث سنبدأ بالقيمة 1 أي الآس:

```
public enum Rank
{
    Ace=1,
    Deuce,
    Three,
    Four,
    Five,
    Six,
    Seven,
    Eight,
    Nine,
    Ten,
    Jack,
    Queen,
    King
}
```

لاحظ أن هذين التعدادين عامان وذلك لأننا سنستخدمهما خارج مكتبة الأصناف من خلال تطبيق الزبون.

بعد ذلك سننتقل إلى القسم الرئيسي لهذا الملف وهو تعريف الصنف Card حيث سنقوم بتعديل الوصولية إلى عام:

```
public class Card
{
}
```

في القسم الأول من شيفرة الصنف Card سنعرف حقلين للقراءة فقط أحدهما من نوع التعداد Suit والآخر من نوع التعداد Rank:

```
    public readonly Suit suit;
    public readonly Rank rank;
```

ومن ثم سنقوم بتجاوز المنهج ToString() حيث سنستخدم الحقلين السابقين لعرض نص يبين ورقة اللعب الحالية:

```
    public override string ToString()
    {
        //return base.ToString();
```

```
            return "The"+rank +" of"+suit+"s";
        }
```

وبوضعنا لأسماء الحقول بهذه الطريقة سيؤدي ذلك إلى إعادة اسم ورقة اللعب وفقا لشكلها ولرتبتها على هيئة نص مقروء بالنسبة إلينا.

بعد ذلك سنعرف منهج البناء الافتراضي للصنف وسنجعله منهجا خاصا:

```
        private Card ()
        {
        }
```

سنعرف الآن منهج بناء عام لهذا الصنف سنستخدمه لإنشاء كائنات الصنف Card ويتقبل هذا المنهج بارامترين الأول يمثل شكل الورقة والثاني يمثل رتبتها:

```
        public Card (Suit newSuit,Rank newRank)
        {
            suit =newSuit ;
            rank =newRank ;
        }
```

وبهذا نكون قد انتهينا من كتابة صنف ورقة اللعب Card والشكل النهائي للملف Card.cs هو كما يلي:

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

namespace CardLibrary
{
    public enum Suit
    {
        Clup,
        Diamond,
        Heart,
        Spade
    }
    public enum Rank
    {
        Ace = 1,
        Deuce,
        Three,
        Four,
        Five,
        Six,
        Seven,
        Eight,
        Nine,
        Ten,
        Jack,
        Queen,
        King
```

```
    }
    public class Card
    {
        public readonly Suit suit;
        public readonly Rank rank;
        public override string ToString()
        {
            //return base.ToString();
            return "The "+rank +" of "+suit+"s";
        }
        private Card ()
        {
        }
        public Card (Suit newSuit,Rank newRank)
        {
            suit = newSuit;
            rank = newRank;
        }
    }
}
```

الملف Deck.cs:

Deck.cs:

يبدأ هذا الملف بنفس ما يبدأ به الملف Card.cs كما يلي:

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

namespace CardLibrary
{
```

ليست لدينا هنا أية أنواع لتعريفها سوى الصنف Deck لذا سنقوم بتعريف هذا الصنف كما يلي:

```
    public class Deck
    {
    }
```

وأول عضو سنعرفه ضمن هذا الصنف هو مصفوفة خاصة تتضمن عناصر من نوع Card أي أن هذه المصفوفة تحتوي على كائنات من نوع Card وسنسمي هذه المصفوفة بـ Cards:

```
        private Card[] Cards;
```

بعد ذلك سنقوم بإنشاء منهج بناء للصنف وبما انه ليست هناك أية شروط معينة يجب القيام بها قبل إنشاء مجموعة أوراق اللعب فإننا سنستخدم منهج البناء الافتراضي لذلك سيقوم هذا المنهج بوضع 52 كائنا من الصنف Card ضمن المصفوفة Cards ومن ثم سنقوم بإنشاء حلقة للمرور على جميع الأوراق المتوفرة

في مجموعة أوراق اللعب ووضعها ضمن المصفوفة لكي نقوم بذلك علينا أن نشكل حلقتين واحدة ضمن الأخرى حيث ستمر الحلقة الأولى على أشكال أوراق اللعب الأربعة ولكل شكل من هذه الأشكال سيتم المرور على الرتب الثلاثة عشر الممكنة وهو ما مجموعه 52 دورة لكلا الحلقتين والنتيجة هي وضع أوراق اللعب ضمن المصفوفة Cards بصورة مرتبة كما لو أننا قمنا بشراء مجموعة أوراق لعب جديدة!:

```
public Deck ()
{
    Cards = new Card[52];
    for (int suitVal=0;suitVal <4;suitVal ++)
    {
        for (int rankVal=1;rankVal <14;rankVal ++)
        {
            Cards[suitVal * 13 + rankVal - 1] = new Card((Suit)suitVal,
                (Rank)rankVal);
        }
    }
}
```

ملاحظة:

لاحظ أننا استخدمنا التشكيل cast في السطر الأخير وهذا واضح لأن متحولي الحلقتين suitVal وrankVal من النوع int وبالتالي علينا أن نشكلهما للنوع Suit وRank على الترتيب.

والآن سوف نكتب المنهج GetCard() والذي يأخذ بارامترا من النوع int يمثل دليل ورقة اللعب في المصفوفة ويعيد كائنا من النوع Card وذلك إذا كانت قيمة البارامتر بين 0 و 51 وإلا فسيتم رمي اعتراض بنفس الطريقة التي تعلمناها مسبقا:

```
public Card GetCard(int cardNum)
{
    if (cardNum >= 0 && cardNum <= 51)
        return Cards[cardNum];
    else
        throw (new System.ArgumentOutOfRangeException("cardNum",
           cardNum, "Value must be between 0 and 51."));
}
```

وأخيرا سنقوم بإنشاء منهج لخلط أوراق اللعب وإعادة ترتيبها بصورة عشوائية وسنسمي هذا المنهج بالاسم Shuffle() سيقوم هذا المنهج بإنشاء مصفوفة مؤقتة نقوم فيها بنسخ أوراق اللعب من المصفوفة cards الحالية إلى المصفوفة المؤقتة بصورة عشوائية إن الجسم الرئيسي لهذا التابع عبارة عن حلقة تبدأ من 0 إلى 51 وعند كل دورة لهذه الحلقة سيتم توليد رقم عشوائي بين 0 و 51 باستخدام كائن من النوع System.Random يستخدم لهذا الغرض ومتى حصلنا على القيمة العشوائية فإننا سنستخدمها كدليل لورقة اللعب (كائن Card) ضمن مصفوفتنا المؤقتة حيث سنضع ضمن هذا الدليل ورقة لعب من المصفوفة Cards الأصلية.

توضيح:

يمثل الصنف System.Random أحد أصناف إطار عمل .NET. ومتى قمنا بإنشاء كائن من هذا الصنف فإنه يمكننا توليد قيمة بين الصفر وX وذلك باستخدام المنهج Next(X).

ولكي نتمكن من تتبع أوراق اللعب التي تم وضعها ضمن المصفوفة المؤقتة فإننا سنستخدم مصفوفة عناصرها من نوع Bool وسنعطى كل عنصر من هذه المصفوفة القيمة true لكل ورقة لعب تم نسخها إلى المصفوفة المؤقتة حيث سيتطابق دليل القيمة في المصفوفة من النوع bool مع دليل ورقة اللعب التي تم سحبها من المصفوفة Cards وعند توليد القيم العشوائية سنقوم بتفحص ما إذا كانت القيمة المولدة تمثل دليلا لعنصر في المصفوفة المنطقية يأخذ القيمة true أو false فإذا كانت القيمة true فهذا يعني أن هذا العنصر قد تم نسخه إلى المصفوفة المؤقتة وإلا فإنه سيتم نسخه إليها.

قد تجد أن هذه الشيفرة ليست فعالة بالدرجة الكافية فقد يتم توليد عدد كبير من الأرقام العشوائية قبل الوصول إلى ورقة لعب لم يتم نسخها بعد إلى المصفوفة المؤقتة ولكننا لن نلحظ ذلك ابدا لأن شيفرة C# تنفذ بسرعة كبيرة بحيث لن نشعر بأي تأخر زمني لذلك.

وشيفرة المنهج Shuffle() كما يلي:

```
public void Shuffle()
{
    Card[] newDeck = new Card[52];
    bool[] assigned = new bool[52];
    for (int i=0;i<52;i++)
    {
        int destCard = 0;
        bool foundCard = false;
        Random sourceGen = new Random();
        while (foundCard ==false )
        {
            destCard = sourceGen.Next(52);
            if (assigned[destCard] == false)
                foundCard = true;
        }
        assigned [destCard ]=true ;
        newDeck[destCard] = Cards[i];
    }
    Cards = newDeck;
}
```

وبذلك نكون قد أنهينا هذا الصنف والشكل النهائي للملف Deck.cs كما يلي:

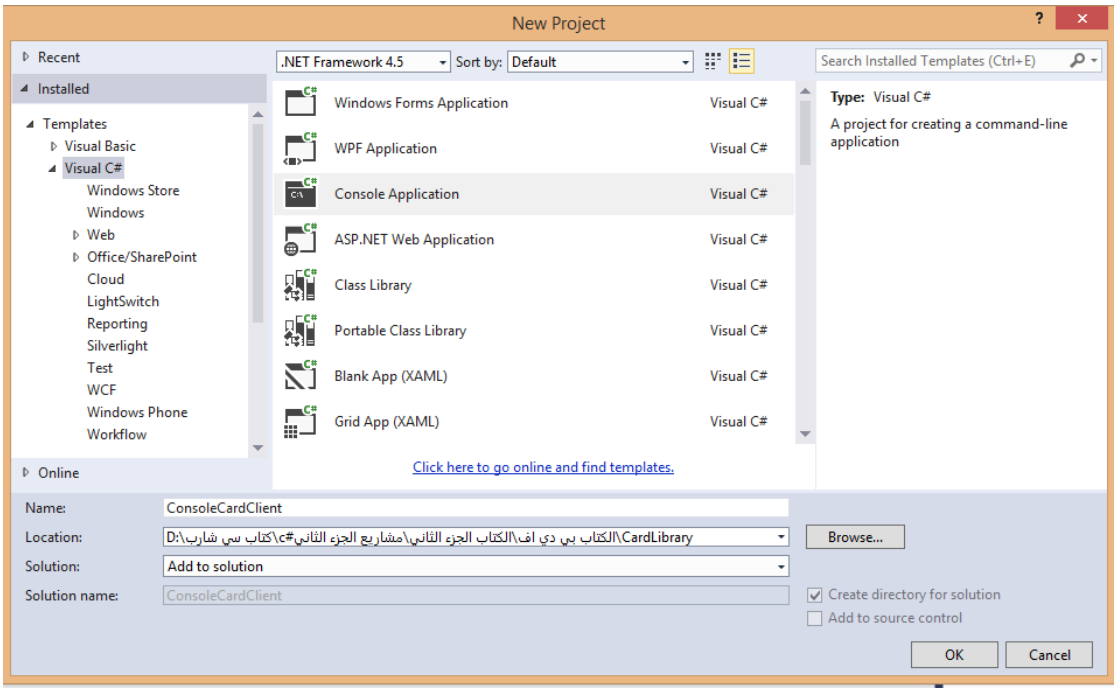
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CardLibrary
```

```
{
    public class Deck
    {
        private Card[] Cards;
        public Deck ()
        {
            Cards = new Card[52];
            for (int suitVal=0;suitVal <4;suitVal ++)
            {
                for (int rankVal=1;rankVal <14;rankVal ++)
                {
                    Cards[suitVal * 13 + rankVal - 1] = new Card((Suit)suitVal,
                        (Rank)rankVal);
                }
            }
        }
        public Card GetCard(int cardNum)
        {
            if (cardNum >= 0 && cardNum <= 51)
                return Cards[cardNum];
            else
                throw (new System.ArgumentOutOfRangeException("cardNum",
                   cardNum, "Value must be between 0 and 51."));
        }
        public void Shuffle()
        {
            Card[] newDeck = new Card[52];
            bool[] assigned = new bool[52];
            for (int i=0;i<52;i++)
            {
                int destCard = 0;
                bool foundCard = false;
                Random sourceGen = new Random();
                while (foundCard ==false )
                {
                    destCard = sourceGen.Next(52);
                    if (assigned[destCard] == false)
                        foundCard = true;
                }
                assigned [destCard ]=true ;
                newDeck[destCard] = Cards[i];
            }
            Cards = newDeck;
        }
    }
}
```

تطبيق زبون لمكتبة أصناف أوراق اللعب:

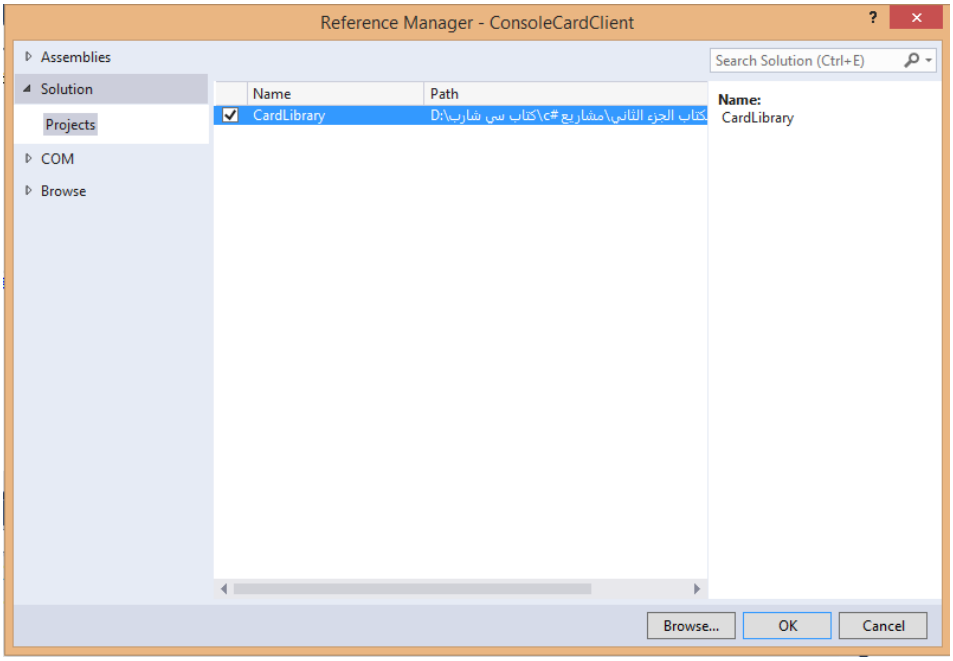
A Client Application for the Class Library:

سنقوم بإنشاء تطبيق Console بسيط ليمثل تطبيق زبون لمكتبة أصناف أوراق اللعب سنضيف مشروع هذا التطبيق إلى الحل Solution الذي يتضمن مشروع مكتبة الأصناف السابقة وللقيام بذلك تأكد من اختيار الخيار Add To Solution من القائمة المنسدلة Solution عند إنشاء مشروع جديد حيث سنسميه ConsoleCardClient:



الشكل (3-38)

ولكي نستخدم مكتبة الأصناف في تطبيق Console الجديد هذا علينا أن ننشئ مرجعا لمكتبة الأصناف ضمن التطبيق ويتم ذلك بعد إنشاء التطبيق الجديد من خلال الضغط بزر الفأرة الأيمن على ConsoleCardClient في نافذة مستعرض الحل Solution Explorer ومن ثم اختيار الأمر Reference من القائمة الفرعية Add للقائمة المنسدلة عندئذ يظهر صندوق حوار Reference Manager من البند Solution المتوضع على يسار صندوق الحوار نختار project ثم نقوم بتفعيل صندوق التحقق امام CardLibrary ثم نضغط الزر ok.



الشكل (3-39)

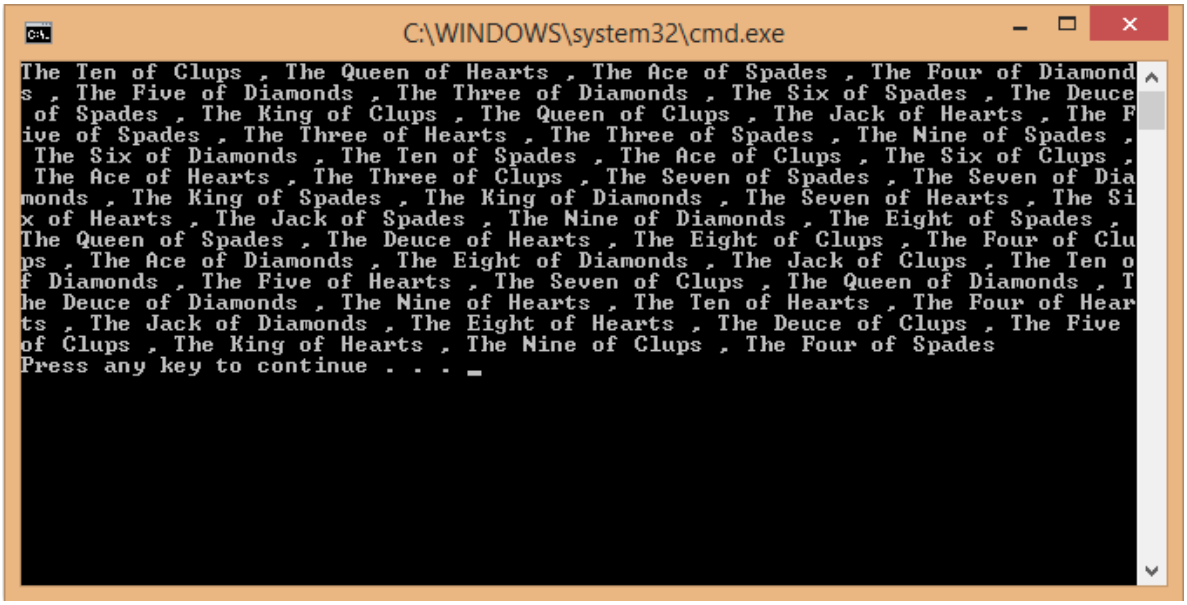
عندئذ سيتم إضافة المرجع إلى التطبيق. بما أن المشروع هذا هو المشروع الثاني من الحل الحالي فإن علينا أن نحدد أنه يمثل مشروع بدء التنفيذ Startup Project للحل مما يعني أنه سيتم تنفيذ هذا المشروع عند الضغط على زر التنفيذ وللقيام بذلك يكفي أن نضغط بزر الفأرة الأيمن على المشروع ومن ثم نختار الأمر Set as StartUp Project من القائمة المنسدلة في إطار Solution Explorer.

بعد ذلك سنقوم بإضافة الشيفرة التالية التي ستستخدم صنفينا اللذين أنشأناهما مسبقا إن هذه الشيفرة لا تتطلب أي شيء خاص والشيفرة التالية تقوم بذلك:

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

namespace ConsoleCardClient
{
    class Program
    {
        static void Main(string[] args)
        {
            Deck myDeck = new Deck();
            myDeck.Shuffle();
            for (int i=0;i<52;i++)
            {
                Card tempCard = myDeck.GetCard(i);
                Console.Write(tempCard.ToString());
```

```
                if (i != 51)
                    Console.Write(" , ");
                else
                    Console.WriteLine();
            }

        }
    }
}
```

والنتيجة هي كما يلي:



الشكل (3-40)

حيث تم طباعة مجموعة أوراق اللعب في نافذة Console بصورة غير مرتبة.

سوف نستمر في تطوير واستخدام مكتبة الأصناف هذه في الفصول اللاحقة.

الخلاصة:

Summary:

لقد انتهينا في هذا الفصل من أساسيات تعريف الأصناف ومازال هناك الكثير للمناقشة حول الأصناف في الفصول القادمة ولكن كما نلاحظ فإن ما تعلمناه حتى الآن يمكننا من إنشاء تطبيقات معقدة بالصورة التي نتطلبها.

لقد تعلمنا كيفية تعريف الحقول والمناهج والخصائص بالإضافة إلى مناقشة مستويات الوصول العديدة وكيفية استخدام مقيدات الوصول مع هذه الأعضاء تحدثنا بعد ذلك عن أدوات ومعالجات Visual Studio 2013 المساعدة التي تمكننا من إنشاء وتعريف الأعضاء ضمن الأصناف والواجهات بسرعة ويسر.

وبعد أن انتهينا من المواضيع الأساسية انتقلنا إلى مواضيع أكثر تقدما مثل سلوك وراثة الأعضاء ورأينا كيفية حجب الأعضاء الموروثة غير المرغوب فيها في الأصناف المشتقة باستخدام الكلمة new وكيفية الوصول إلى أعضاء الصنف الأساس بدلا من استبدالها بتزويد جديد وذلك باستخدام الكلمة base ولقد تعرفنا أيضا على تعريفات الأصناف المعششة.

ثم انتقلنا من الأصناف إلى الواجهات وكيفية تعريفها وتزويد الأصناف بها بالإضافة على مفاهيم التزويد المطلق والتزويد الصريح للواجهات.

وأخيرا فقد قمنا بتطوير واستخدام مكتبة أصناف تمثل مجموعة أوراق اللعب وسوف نحدث ونستخدم هذه المكتبة في الفصول اللاحقة أيضا.

الفصل الرابع

المزيد عن الأصناف

لقد تناولنا حتى الآن كافة أساسيات البرمجة كائنية التوجه OOP في لغة البرمجة C# إلا أن هناك بعض التقنيات المتقدمة والتي يجب ان نتعلمها قبل الاستعداد للبدء ببناء تطبيقات في C# سوف نتناول في هذا الفصل التقنيات التالية:

- **المجموعات(Collections):** يمكن أن تحوي الكائنات على مصفوفات لكائنات أخرى بحيث تتضمن مصفوفات الكائنات تلك وظائفا للتحكم بالوصول إلى هذه الكائنات وتسمى هذه الآلية بالمجموعات.
- **التحميل الزائد للعوامل (Operator overloading):** ويعرف بأنه إعداد معين للأصناف يمكننا من استخدام عوامل مثل عامل الجمع + أو الطرح – مع كائنات الصنف.
- **النسخ العميق (Deep Copying):** أي نسخ كائن إلى كائن آخر جديد بصورة كاملة (دون وجود أية مرجعية للبيانات المخزنة ضمن الكائن المصدر في الكائن المنسوخ).
- **الاعتراضات المخصصة (Custom exceptions):** حيث سنتعلم كيفية إنشاء اعتراضاتنا الخاصة بنا بتوفير معلومات إضافية للشيفرة التي ستصطاد الاعتراض.

المجموعات:

Collections:

لقد رأينا في الفصل الخامس من الجزء الأول للكتاب كيفية استخدام المصفوفات لإنشاء متحول يمكنه استيعاب عدد من المتحولات التي لها نوع محدد إن للمصفوفات نقاط سلبية عديدة وأهم تلك السلبيات هي أن لها حجما ثابتا فمتى قمنا بإنشاء المصفوفة لن نتمكن بعدها من تغيير حجمها إن هذا يعني أن الشيفرة التي ستعالج المصفوفات ستصبح معقدة لدرجة كبيرة. تمكننا تقنية OOP من إنشاء أصناف تقوم بمهمة إدارة العناصر ضمن المصفوفة وبذلك تصبح الشيفرة التي تستخدم لائحة العناصر أو المصفوفة تلك بسيطة وغير معقدة.

كما تعرفنا في الفصل الخامس على اللوائح List وقلنا إنها تشبه المصفوفات إلا أنها ديناميكية أي أنها ليست بحاجة للتهيئة مسبقا في الحقيقة تعرف المصفوفات واللوائح التي تعرفنا عليها سابقا بأنها أصناف

Generic Collections أي انها موجودة ضمن فضاء الأسماء System.Collections.Generic أما المجموعات التي سنتحدث عنها الان فهي توصف بأنها non-Generic كما أنها تنتمي لفضاء الأسماء System.Collections وتتميز هذه المجموعات بأنها تملك خصائص المصفوفات واللوائح معا.

في C# تستخدم أصناف المجموعة بصورة عامة لمعالجة لائحة من الكائنات ويمكن أن تحتوي على وظائف إضافية بالمقارنة مع المصفوفات الاعتيادية ويتم الوصول إلى هذه الوظائف من خلال التزود بواجهات من فضاء الأسماء System.Collections يتضمن فضاء الأسماء بعض الأمور المثيرة الأخرى أيضا.

يمكننا الوصول إلى عناصر المجموعة من خلال دليل العنصر كما في المصفوفات بالإضافة إلى إمكانية الوصول إلى العناصر من خلال أشياء أخرى فوظائف الوصول إلى العناصر ضمن المجموعات ليست محددة باستخدام أصناف المجموعة الأساسية كما في النوع System.Array وإنما يمكننا إنشاء أصناف مجموعة مخصصة لنا وبالتالي يمكننا إنشاء مجموعات تخص كائنات من نوع محدد بحيث تتضمن على وظائف خاصة للائحة هذه الكائنات.

يحتوي فضاء الأسماء System.Collections على عدد من الواجهات والأصناف والبنى والتي تعرف وظائف المجموعة الأساسية.

والجدول التالي يبين أهم الأصناف التي يحتويها فضاء الأسماء System.Collections:

الفئة	الوصف
ArrayList	يرث هذا الصنف الواجهة IList التي تمكن من إنشاء مصفوفة متغيرة الحجم (ديناميكية) بالإضافة إلى إمكانيات الحذف والإضافة.
BitArray	هذا الصنف يمكننا من التحكم بمصفوفة من البتات عن طريق إرجاعها لقيم منطقية حيث تمثل القيمة 1 بالقيمة المنطقية True و القيم 0 بالقيمة المنطقية False
Comparer	يقوم هذا الصنف بالمقارنة بين كائنين.
CollectionBase	وهو صنف مجرد أساسي يزودنا بمناهج قوية للمجموعات.
Hashtable	يعرف هذا الصنف بأنه مجموعة من المفاتيح المرتبطة يشار أليها بدليلها.
Queue	هذا الصنف يعرف باسم الطابور ويعمل على مبدأ أول الداخلين هو اول الخارجين
SortedList	هذا الصنف يقوم بترتيب عناصر المجموعة بناء على المفاتيح والأدلة.
Stack	هذا الصنف يشبه صنف الطابور إلا أن العنصر الأخير في الدخول هو العنصر الأول في الخروج.

والجدول التالي يبين اهم الواجهات التي يحتويها فضاء الأسماء System.Collections:

الواجهة	الوصف
ICollection	وتمكننا من إيجاد عدد العناصر في المجموعة وإمكانية نسخ العناصر إلى مصفوفة عادية.

IComparer	تمكننا من المقارنة بين كائنين.
IDictionary	تمكننا من الوصول إلى عناصر اللائحة عبر قيمة مفتاحية محددة بدلا من مجرد استخدام دليل العنصر ضمن اللائحة كما في المصفوفات التقليدية
IEnumerable	توفر هذه إمكانية المرور خلال العناصر في المجموعات.
IEnumerator	تدعم تكرار بسيط للمجموعة
IList	وتوفر لائحة من العناصر تتضمن إمكانية للوصول إلى هذه العناصر بالإضافة إلى إمكانية مرتبطة بلوائح العناصر.
IDictionaryEnumerator	يعدد عناصر المجموعة عبر قيمة مفتاحية.

يزود الصنف System.Collections بالواجهات IList وICollection وIEnumerable وهذا واضح لأن جميع الوظائف التي تقدمها هذه الواجهات الثلاثة متوفرة في المصفوفات التقليدية إلا أنها لا تدعم بعضا من المزايا المتقدمة للواجهة IList بالإضافة إلى أنها تمثل لائحة العناصر بحجم ثابت.

استخدام المجموعات:

Using Collections:

يحتوي فضاء الأسماء System.Collections على الصنف System.Collections.ArrayList وهو مزود بالواجهات IList وICollection وIEnumerable إلا انه يحتوي على وظائف إضافية بالمقارنة مع الصنف System.Array فهذا الصنف يمكننا من إنشاء مصفوفات بأحجام متغيرة إن المثال أفضل طريقة لفهم الأمور لذا إليك التطبيق التالي الذي يبين لنا كيفية استخدام المجموعات.

تطبيق حول المصفوفات مقابل المجموعات الأكثر تقدما:

1- قم بإنشاء تطبيق Console جديد باسم ConsoleCollections.
2- أضف صنفا جديدا باسم Animal إلى المشروع في ملف مستقل Animal.cs وذلك باستخدام المعالج المساعد.
3- عدل الشيفرة في الملف Animal.cs كما يلي:

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

namespace ConsoleCollections
{
    public abstract class Animal
```

```
    {
        protected string name;
        public string Name
        {
            get
            {
                return name;
            }
            set
            {
                name = value;
            }
        }
        public Animal ()
        {
            name = "The animal with no name";
        }
        public Animal (string newName)
        {
            name = newName;
        }
        public void Feed()
        {
            Console.WriteLine("{0} has been fed.", name);
        }
    }
    public class Cow:Animal
    {
        public void Milk()
        {
            Console.WriteLine("{0} has been milked.", name);
        }
        public Cow (string newName):base(newName )
        {
        }
    }
    public class Chicken:Animal
    {
        public void LayEgg()
        {
            Console.WriteLine("{0} has laid an egg.", name);
        }
        public Chicken (string newName):base(newName )
        {
        }
    }
}
```

4- عدل الشيفرة في الملف Program.cs كما يلي:

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

```
using System.Threading.Tasks;
using System.Collections;

namespace ConsoleCollections
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Creat an Array type collection of" +
                "Animal objects and use it:");
            Animal[] animalArray = new Animal[2];
            Cow myCow = new Cow("Deirdre");
            animalArray[0] = myCow;
            animalArray[1] = new Chicken("Ken");
            foreach (Animal myAnimal in animalArray )
            {
                Console.WriteLine("New {0} object added to Array" +
                    "collection,Name={1}",myAnimal .ToString (),
                    myAnimal .Name );
            }
            Console.WriteLine("Array collection contains {0} objects>",
                animalArray.Length);
            animalArray[0].Feed();
            ((Chicken)animalArray[1]).LayEgg();
            Console.WriteLine();
            Console.WriteLine("Create an ArrayList type collection"+
                "of Animal object and use it:");
            ArrayList animalArrayList = new ArrayList();
            Cow myCow1 = new Cow("Hayley");
            animalArrayList.Add(myCow1);
            animalArrayList.Add(new Chicken("Roy"));
            foreach (Animal myAnimal in animalArrayList )
            {
                Console.WriteLine("New {0} object added to ArrayList" +
                    "collection,Name={1}",
                    myAnimal.ToString(), myAnimal.Name);
            }
            Console.WriteLine("ArrayList collection contains {0}" +
                "objects.", animalArrayList.Count);
            ((Animal)animalArrayList[0]).Feed();
            ((Chicken)animalArrayList[1]).LayEgg();
            Console.WriteLine();
            Console.WriteLine("Additional manipulation of ArrayList:");
            animalArrayList.RemoveAt(0);
            ((Animal)animalArrayList[0]).Feed();
            animalArrayList.AddRange(animalArray);
            ((Chicken)animalArrayList[2]).LayEgg();
            Console.WriteLine("The animal called {0} is at index {1}.",
                myCow .Name ,animalArrayList .IndexOf (myCow ));
            myCow.Name = "Janice";
            Console.WriteLine("The animal is now called {0}.",
                ((Animal)animalArrayList[1]).Name);
            Console.ReadLine();
```

```
        }
    }
}
```

5- نفذ التطبيق بالضغط على مفتاح F5.

الشكل (4-1)

كيفية العمل:

How To Work:

لقد قمنا في هذا المثال بإنشاء مجموعتين من الكائنات الأولى باستخدام الصنف System.Array أي مصفوفة تقليدية والثانية باستخدام الصنف System.Collections.ArrayList حيث تحتوي كلتا المجموعتين على لائحة من كائنات Animal أي كائنات من النوع Animal والذي تم تعريفه ضمن الملف Animal.cs إن الصنف Animal هو صنف مجرد وبالتالي لا يمكننا إنشاء كائنات منه مباشرة ولكن باستخدام تعددية الأشكال يمكن أن يكون لدينا عناصر ضمن المجموعتين من النوع Cow و Chickenباعتبارها أصنافا مشتقة من الصنف Animal.

ومتى قمنا بإنشاء هاتين المجموعتين في المنهج Main() موجود ضمن الملف Program.cs ستتم معالجة هاتين المجموعتين لاستعراض صفاتهما وإمكانياتهما إن معظم العمليات المستخدمة هنا يمكن تطبيقها على كلا المجموعتين أي مجموعة Array ومجموعة ArrayList على الرغم من أن لكل منهما صيغ مختلفة للقيام بذلك إلا أن هناك بعض العمليات التي يمكن استخدامها مع المجموعات من نوع ArrayList فقط.

لنتناول العمليات المتشابهة في كلتا المجموعتين لنقارن الشيفرة والنتائج لنوعي المجموعتين.

أولا إنشاء المجموعة: فبالنسبة للمصفوفات التقليدية علينا تهيئة المصفوفة بحجم ثابت لكي نتمكن من استخدامها ولقد قمنا بذلك باستخدام مصفوفة باسم animalArray باستخدام الصيغة القياسية التي تعلمناها في الفصل الخامس:

```
Animal[] animalArray = new Animal[2];
```

أما بالنسبة للمجموعات من الصنف ArrayList فإنها لا تحتاج إلى تحديد حجم لها أثناء تهيئتها لذا يمكننا إنشاء المجموعة والتي تسمى في شيفرتنا بـ animalArrayList بسهولة كما يلي:

```
ArrayList animalArrayList = new ArrayList();
```

وهناك منهجا بناء آخران يمكننا استخدامهما عند إنشاء مجموعات من النوع ArrayList يقوم المنهج الأول بنسخ مجموعة موجودة مسبقا إلى المجموعة الجديدة وذلك بتحديد المجموعة المراد نسخها كبارامتر في منهج البناء وأما منهج البناء الآخر فهو يقوم بتحديد سعة المجموعة وذلك أيضا عبر تمرير بارامتر يمثل ذلك والسعة عبارة عن قيمة من النوع int حيث تحدد العدد الأول للعناصر التي يمكن أن تتضمنها المجموعة في الحقيقة إن هذه القيمة ليست بالسعة الحقيقة حيث سيتم مضاعفة هذه القيمة تلقائيا متى تجاوز عدد العناصر في المجموعة هذه القيمة.

إن تهيئة المصفوفات ذات الأنواع المرجعية مثل المصفوفات التي تحتوي كائنات من نوع Animal أو كائنات مشتقة من النوع Animal لا تعني تهيئة العناصر التي ستحتويها ولكي نتمكن من استخدام مدخل (المدخل هو مكان وضع العنصر في المصفوفة) في المصفوفة علينا أن نهيئ هذا المدخل أيضا مما يعني أن علينا إسناد كائنات مهيئة مسبقا إليه وذلك كي يمثل عنصرا فعالا في المصفوفة:

```
Cow myCow = new Cow("Deirdre");
animalArray[0] = myCow;
animalArray[1] = new Chicken("Ken");
```

لقد قمنا بوضع الكائنات في المصفوفة بطريقتين هنا الأولى بإسناد كائن موجود مسبقا-وهو myCow والثانية بإسناد كائن من الصنف وهو الصنف المشتق Chicken في حالتنا هنا مباشرة إن الفرق هنا هو أن الحالة الأولى تترك لنا مرجعا للكائن الذي أسندناه في المصفوفة لاستخدامه لاحقا في الشيفرة.

أما بالنسبة لمجموعات ArrayList فليست هناك أية عناصر موجودة مسبقا ولا حتى عناصر فارغة أي كائنات Null وبالتالي لا توجد هناك مداخل كما في المصفوفات وهذا يعني أنه لا يمكننا إسناد كائنات جديدة إلى مواقع محددة ضمن المجموعات باستخدام الأدلة كما نقوم بذلك مع المصفوفات وبدلا من ذلك فإننا نستخدم المنهج Add() لإضافة عناصر جديدة كما يلي:

```
Cow myCow1 = new Cow("Hayley");
animalArrayList.Add(myCow1);
animalArrayList.Add(new Chicken("Roy"));
```

وبعيدا عن الاختلاف في صيغة العناصر بين مجموعات ArrayList والمصفوفات فإنه يمكننا هنا أيضا إضافة كائنات جديدة أو موجودة مسبقا إليها كما في المصفوفات.

ومتى قمنا بإضافة العناصر بهذه الطريقة يمكننا عندئذ تجاوز هذه العناصر واستبدالها باستخدام صيغة مطابقة لما نستخدمه مع المصفوفات:

```
animalArrayList[0]=new Cow("Alma");
```

لكننا لم نقم بذلك في شيفرة تطبيقنا هذا.

لقد رأينا في الفصل الخامس من الجزء الأول للكتاب كيفية استخدام الحلقة Foreach للمرور عبر عناصر المصفوفة إن ذلك ممكن بالنسبة للمصفوفة هنا باعتبار أن الصنف System.Array مزود بالواجهة IEnumerable والمنهج الوحيد في هذه الواجهة وهو GetEnumerator() يمكننا من المرور خلال العناصر في المصفوفة سوف نتناول ذلك بتفصيل أكبر لاحقا. لاحظ اننا قمنا في شيفرتنا هنا بكتابة معلومات عن كل كائن Animal في المصفوفة باستخدام هذه الحلقة:

```
foreach (Animal myAnimal in animalArray )
{
    Console.WriteLine("New {0} object added to Array" +
        "collection,Name={1}",myAnimal .ToString (),
        myAnimal .Name );
}
```

والصنف ArrayList مزود بالواجهة IEnumerable وبالتالي يمكننا أن نستخدم حلقة foreach والصيغة المستخدمة هي نفسها مع المصفوفات التقليدية:

```
foreach (Animal myAnimal in animalArrayList )
{
    Console.WriteLine("New {0} object added to ArrayList" +
        "collection,Name={1}",
        myAnimal.ToString(), myAnimal.Name);
}
```

سنستخدم بعد ذلك الخاصية Length للمصفوفة لطباعة العناصر ضمنها على الشاشة:

```
Console.WriteLine("Array collection contains {0} objects>",
    animalArray.Length);
```

ويمكننا تحقيق الأمر نفسه مع مجموعة ArrayList فيما عدا أننا سنستخدم الخاصية Count بدلا من Length مع المصفوفات والتي تمثل جزء من الواجهة ICollection:

```
Console.WriteLine("ArrayList collection contains {0}" +
    "objects.", animalArrayList.Count);
```

إن المجموعات سواء كانت مصفوفات بسيطة أو مجموعات معقدة لن تصبح مفيدة إلا إذا تمكنا من الوصول إلى العناصر التي تحتويها إن المصفوفات مصنفة بصورة تامة Strongly typed وهذا يعني أنها تسمح بالوصول المباشر إلى الأنواع التي تحتويها وهذا يعني أنه يمكننا الوصول إلى أعضاء عناصرها بصورة مباشرة:

```
animalArray[0].Feed();
```

إن نوع المصفوفة هنا هو الصنف المجرد Animal وبالتالي فإنه لا يمكننا استدعاء المناهج المزودة من الأصناف المشتقة مباشرة لذا فإن علينا استخدام التشكيل Cast:

```
((Chicken)animalArray[1]).LayEgg();
```

إن مجموعة ArrayList هي مجموعة من كائنات System.Object ولقد قمنا بإسناد كائنات Animal إلى هذه المجموعة عبر تعددية الأشكال وهذا يعني أن علينا استخدام التشكيل cast مع جميع العناصر:

```
((Animal)animalArrayList[0]).Feed();
((Chicken)animalArrayList[1]).LayEgg();
```

لقد استعرضنا في شيفرتنا حتى الآن الإمكانيات التي يقدمها نوعين من المجموعات: مجموعات System.Array والتي تمثل في جوهرها المصفوفات التقليدية ومجموعات System.ArrayList والشيفرة المتبقية من هذا التطبيق تستعرض بعضا من المزايا المتوفرة في مجموعات ArrayList والتي تتفوق فيها على المصفوفات التقليدية.

أولا يمكننا إزالة العناصر من مجموعة ArrayList باستخدام المنهج ()Remove أو ()RemoveAt واللذين يمثلان جزءا من الواجهة IList المزود به الصنف ArrayList يقوم هذان المنهجان بإزالة العناصر من المجموعة بالاعتماد على مرجع العنصر أو وفقا لدليله من المجموعة ولقد استخدمنا في مثالنا هذا طريقة إزالة العنصر بواسطة دليله حيث قمنا بإزالة كائن Cow والذي له الخاصية Name التي تحمل القيمة Hayley:

```
animalArrayList.RemoveAt(0);
```

ويمكننا ان نحذف العنصر نفسه وذلك كما يلي:

```
animalArrayList.Remove(myCow1);
```

لاحظ أن لدينا مرجعا محليا لهذا الكائن حيث أننا أضفنا هذا العنصر إلى المجموعة عبر كائن موجود مسبقا بدلا من إنشاء كائن جديد للقيام بذلك.

وسواء استخدمنا أي من الطريقتين فإن العنصر المتبقي في المجموعة هو كائن Chicken والذي قمنا باستدعاء أحد أعضاءه وهو المنهج ()Feed كما يلي:

```
((Animal)animalArrayList[0]).Feed();
```

لاحظ أن العناصر في مجموعة ArrayList قد لا تبقى في نفس موقعها الأصلي عند إزالة عنصر منها فعلى سبيل المثال عند إزالة العنصر ذو الدليل 0 من المجموعة سيؤدي إلى إزاحة العناصر الأخرى المتبقية موقعا واحدا من اللائحة وذلك لملء الموقع الفارغ الناتج عن العنصر المحذوف وهذا يعلل سبب وصولنا إلى كائن Chicken في سطر الشيفرة السابق باستخدام الدليل 0 على الرغم من أن الدليل الأصلي لهذا الكائن هو 1 وذلك قبل أن نحذف العنصر ذو الدليل 0 وبالتالي سيتم رمي اعتراض إذا قمنا بكتابة شيفرة كالتالية بدلا من السطر السابق:

```
((Animal)animalArrayList[1]).Feed();
```

توضيح:

إن هذا يعني انه من غير المنطقي في كثير من الأحيان الاعتماد على الأدلة في مجموعات ArrayList للوصول إلى العناصر فإذا كانت شيفرتك تقوم بعمليات حذف للعناصر من المجموعة فإن هذا سيؤدي حتما إلى تغيير مواقع العناصر في المجموعة والذي سيؤدي بالطبع إلى تغير أدلتها إلا إذا كنت ستحذف العنصر الأخير دوما من المجموعة عندئذ لن تكون هناك أية عمليات إزاحة للعناصر.

وتمكننا مجموعات ArrayList من إضافة عناصر عديدة دفعة واحدة إلى المجموعة وذلك باستخدام المنهج AddRange() يتقبل هذا المنهج أي كائن مزود بالواجهة ICollection وبالتالي فإن المصفوفة التقليدية animalArray التي أنشأناها مسبقا يمكن أن تمثل بارامترا لهذا المنهج:

```
animalArrayList.AddRange(animalArray);
```

ولكي نتمكن من أن هذا المنهج قام بإضافة جميع العناصر الموجودة ضمن المصفوفة animalArray إلى المجموعة animalArrayList فإننا سنحاول الوصول إلى العنصر الثالث في المجموعة والذي يقودنا إلى العنصر الثاني من المصفوفة animalArray:

```
((Chicken)animalArrayList[2]).LayEgg();
```

ملاحظة:

لاحظ اننا لم نقم هنا بنسخ الكائن من المصفوفة animalArray إلى المجموعة animalArrayList وإنما ما يحدث هنا عبارة عن إنشاء مرجع للإشارة إلى نفس كائن Chicken الذي تشير له المصفوفة هذا يعني أن العنصر الثالث من المجموعة animalArrayList يشير إلى نفس الكائن الذي يشير له العنصر الثاني من المصفوفة animalArray.

لا يمثل المنهج AddRange() جزء من أي واجهات يدعمها الصنف ArrayList وإنما هذا المنهج خاص للصنف ArrayList فقط ويستعرض حقيقة إمكانية تحديد سلوك مخصص لأصناف مجموعاتنا أكثر مما تقدمه الواجهات لنا يحتوي هذا الصنف في الواقع على مناهج أخرى مثير للاهتمام مثل المنهج InsertRange() وذلك لإضافة مصفوفة من الكائنات إلى أي موضع من المجموعة وهناك مناهج لمهام أخرى كفرز العناصر وإعادة ترتيبها.

وأخيرا فإننا نعود إلى حقيقة أنه يمكن أن يكون لدينا مراجع عديدة إلى الكائن نفسه فباستخدام المنهج IndexOf() والذي يمثل جزءا من الواجهة IList لا يمكننا أن نرى أن الكائن myCow وهو كائن أصلي تمت إضافته إلى المصفوفة animalArray أصبح يمثل جزءا من المجموعة animalArrayList وحسب وإنما يمكننا أيضا إيجاد دليل الكائن في المجموعة:

```
Console.WriteLine("The animal called {0} is at index {1}.",
    myCow .Name ,animalArrayList .IndexOf (myCow ));
```

وكامتداد لذلك فإن السطرين التاليين للسطر السابق بقومان بتغيير اسم الكائن عبر مرجعه ومن ثم عرض الاسم الجديد من خلال مرجع الكائن في المجموعة:

```
myCow.Name = "Janice";
Console.WriteLine("The animal is now called {0}.",
    ((Animal)animalArrayList[1]).Name);
```

تعريف المجموعات:

Defining Collections:

لقد رأينا للتو ما يمكننا القيام به باستخدام أصناف مجموعات تحوي مزايا متقدمة لقد حان الوقت الآن لكي نرى كيف يمكننا إنشاء مجموعاتنا الخاصة بنا عن أحد الطرق للقيام بذلك هو بتزويد صنف مجموعتنا بالمناهج الضرورية لمعالجة المجموعة وبكتابة شيفرة هذه المناهج يدويا إلا أن هذه الطريقة مرهقة لنا وقد تكون معقدة جدا مع بعض المناهج ويمكننا كحل بديل لذلك اشتقاق مجموعتنا من صنف مثل الصنف System.Collections.CollectionBase والذي يمثل صنفا مجردا يوفر التزويدات التي نحتاجها لمعالجة المجموعات وذلك هو الخيار الأفضل طبعا.

إن الصنف CollectionBase مزود بالواجهات IEnumerable وICollection وIList إلا أنه يقدم الشيفرة الأساسية المطلوبة لأعضاء هذه الواجهات مثل المنهج Clear() والمنهج RemoveAt() للواجهة IList والخاصية Count للواجهة ICollection وبالتالي فإننا نحتاج إلى التزود بجميع الأشياء الأخرى بأنفسنا إذا أردنا توفير وظائف أخرى لمجموعتنا.

لكي نبسط الأمور عليك يوفر الصنف CollectionBase خاصيتين محميتين تمكننا من الوصول إلى الكائنات المخزنة ضمن المجموعة يمكننا أن نستخدم الخاصية List والتي تمكننا من الوصول إلى العناصر من خلال الواجهة IList أو الخاصية InnerList والتي تمثل ArrayList المستخدم لحفظ العناصر.

على سبيل المثال يمكننا تعريف الأشياء الأساسية لصنف مجموعة لحفظ كائنات من نوع Animal ضمنها كما يلي سوف نتناول شيفرة هذه التعريفات لاحقا:

```
public class Animals:CollectionBase
{
    public void Add(Animal newAnimal)
    {
        List.Add(newAnimal);
    }
    public void Remove(Animal oldAnimal)
    {
        List.Remove(oldAnimal);
    }
    public Animals ()
    {
    }
```

```
}
```

لقد قمنا هنا بكتابة شيفرة المنهجين Add() وRemove() بصورة صريحة لتقبل الكائنات من نوع Animals وهما يستخدمان المنهجين Add() وRemove() الأصليين للواجهة IList ويعمل المنهجان السابقان مع الكائنات من نوع Animals فقط أو مع أنواع مشتقة من النوع Animals وذلك بعكس الصنف ArrayList الذي رأيناه مسبقا والذي يمكن أن يتقبل أي كائن من أي نوع.

يمكننا الصنف الأساس CollectionBase من استخدام الحلقة foreach أيضا مع مجموعتنا ويمكننا على سبيل المثال كتابة شيفرة كما يلي:

```
Console.WriteLine("Using custom collection class Animals:");
Animals animalCollection = new Animals();
animalCollection.Add(new Cow("Sarah"));
foreach (Animal myAnimal in animalCollection )
{
    Console.WriteLine("New {0} object added to custom collection" +
        "Name={1}", myAnimal.ToString(), myAnimal.Name);
}
```

ولكن لا يمكننا على سبيل المثال كتابة شيفرة كما يلي:

```
animalCollection[0].Feed();
```

والسبب في ذلك أنه لا يمكننا الوصول إلى العناصر عبر أدلتها ولكي نتمكن من ذلك يجب أن نستخدم ما يعرف بالمفهرسات (indexers).

المفهرسات:

Indexers:

يمثل المفهرس خاصية من نوع خاص يمكننا إضافتها إلى الصنف لتوفير وصول إلى عناصر المجموعة بشكل مشابه لأسلوب الوصول إلى العناصر في المصفوفات في الحقيقة تمكننا هذه الخاصية من الوصول إلى العناصر بصورة أكثر تعقيدا وذلك باعتبار أنه يمكننا استخدام بارامترات معقدة ضمن قوسي دليل العنصر في المجموعة بالطريقة التي نريد وكما لاحظنا مع المصفوفات فإن استخدام قيمة رقمية للوصول إلى العنصر في المجموعة هو أبسط أنواع المفهرسات.

يمكننا إضافة مفهرس إلى المجموعة Animals لكائنات Animal كما يلي:

```
public class Animals : CollectionBase
{
    public Animal this[int animalIndex]
    {
        get
        {
            return (Animal)List[animalIndex];
        }
```

```
            set
            {
                List[animalIndex] = value;
            }
        }
    }
```

استخدمنا هنا الكلمة this مع بارامتر ضمن القوسين "[]" وفيما عدا ذلك فهي مثل أية خاصية أخرى إن هذه الصيغة منطقية وذلك لأننا سنصل إلى المفهرس باستخدام اسم الكائن متبوعا ببارامتر الدليل ضمن القوسين "[]" عبى سبيل المثال MyAnimals[0].

تستخدم هذه الشيفرة مفهرسا على الخاصية List بواسطة الواجهة IList التي توفر لنا الوصول إلى مجموعات ArrayList في الصنف CollectionBase التي تحتفظ بعناصر مجموعتنا:

```
return (Animal)List [animalIndex];
```

التشكيل الصريح مطلوب هنا باعتبار أن الخاصية IList.List تعيد قيمة من نوع System.Object إن ما يجب ان ننوه عنه هنا هو تعريف نوع للمفهرس فهذا هو النوع الذي سنحصل عليه عند الوصول إلى عنصر ما باستخدام هذا المفهرس هذا يعني أنه يمكننا كتابة شيفرة كما يلي:

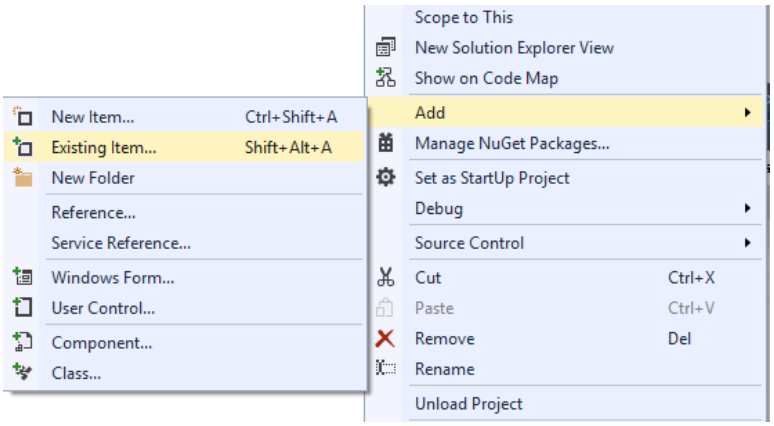
```
animalCollection[0].Feed();
```

بدلا من:

```
((Animal)animalCollection[0]).Feed();
```

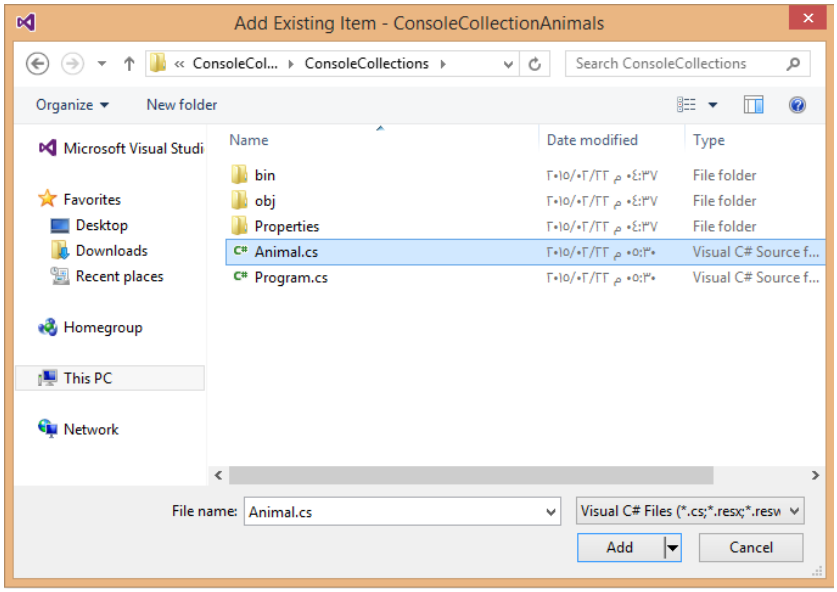
تطبيق حول استخدام المجموعة Animals:

1- قم بإنشاء تطبيق Console جديد باسم ConsoleCollectionAnimals.
2- انقر بزر الفارة الأيمن على اسم المشروع في إطار Solution Explorer ومن ثم اختر Add ثم Add Existing Item من القائمة المنسدلة.



الشكل (2-4)

3- اختر الملف Animal.cs من مجلد مشروع ConsoleCollections ثم انقر على زرAdd:



الشكل (3-4)

4- عدل فضاء تعريف الأسماء في الملف Animal.cs على التالي:

```
namespace ConsoleCollectionAnimals
{
```

5- أضف صنفا جديدا باستخدام معالج إضافة الأصناف باسم Animals ومن ثم احفظ هذا الصنف ضمن الملف Animals.cs.

6- عدل شيفرة الصنف Animals.cs كما يلي:

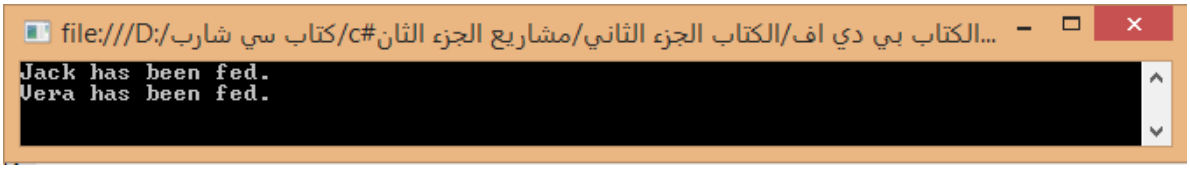
```
    public class Animals : CollectionBase
    {
        public void Add(Animal newAnimal)
        {
            List.Add(newAnimal);
        }
        public void Remove(Animal oldAnimal)
        {
            List.Remove(oldAnimal);
        }
        public Animals()
        {
        }
            public Animal this[int animalIndex]
            {
                get
                {
                    return (Animal)List[animalIndex];
                }
```

```
            set
            {
                List[animalIndex] = value;
            }
        }
}
```

7- عدل شيفرة الصنف Program.cs كما يلي:

```
class Program
{
    static void Main(string[] args)
    {
        Animals animalcollection = new Animals();
        animalcollection.Add(new Cow("Jack"));
        animalcollection.Add(new Chicken("Vera"));
        foreach (Animal myAnimal in animalcollection )
        {
            myAnimal.Feed();
        }
        Console.ReadLine();
    }
}
```

8- نفذ الشيفرة:



الشكل (4-4)

كيفية العمل:

How To Work:

يستخدم هذا المثال الشيفرة التي تعرضنا لها في القسم الأخير وذلك لإنشاء مجموعة من النوع Animals والتي لا يمكن أن تحوي إلا كائنات من النوع Animal او من أنواع مشتقة من هذا النوع يقوم المنهج Main() هنا بإنشاء حالة من مجموعة Animals في المتحول animalCollection ومن ثم يقوم بإضافة عنصرين (كائن من الصنف cow وآخر من الصنف Chicken) وهي أنواع مشتقة من الصنف Animals ويستخدم بعد ذلك حلقة foreach لاستدعاء المنهج Feed() الذي يرثه كائني cow و Chicken من الصنف Animal.

المجموعات المزودة بالمفاتيح والواجهة IDictionary:

Keyed Collections and IDictionary:

يمكننا استخدام واجهة أخرى غير الواجهة IList التي سبق واستخدمناها ألا وهي الواجهة IDictionary لقد رأينا أن الواجهة IList تمكننا من فهرسة العناصر ضمن المجموعة بواسطة قيمة رقمية فريدة أطلقنا عليها اسم الدليل index لكن ماذا لو أن مواقع (أدلة) العناصر ضمن المجموعة معرضة للتغيير بشكل مستمر عندئذ لن نتمكن من الوصول إلى العنصر ضمن المجموعة من خلال دليله بسهولة لهذا السبب وجدت الواجهة IDictionary والتي تمكننا من الوصول إلى العناصر ضمن المجموعة من خلال قيم مفتاحية فريدة تعطى لكل عنصر فيها كسلسلة نصية مثلا.

يمكننا تحقيق ذلك من خلال المفهرسات أيضا إلا أن البارامتر المستخدم ضمن المفهرس يمثل المفتاح المرتبط بالعنصر الموجود ضمن المجموعة وذلك بدلا من مجرد استخدام دليل ذلك العنصر بارامتر من نوع int.

وكما في المجموعات المفهرسة فإن هناك صنف أساس يمكننا استخدامه لتبسيط التزود بالواجهة IDictionary ألا وهو الصنف DictionaryBase إن هذا الصنف مزود أيضا بالواجهتين IEnumerable وICollection واللتان توفران الإمكانيات الأساسية لمعالجة أي مجموعة.

الصنف DictionaryBase مشابه للصنف CollectionBase حيث أنه مزود ببعض أعضاء الواجهات التي يدعمها فكما في الصنف CollectionBase فهو يتضمن المنهج Clear() والخاصية Count ولكن لا يستخدم المنهج RemoveAt() إن هذا بسبب أن المنهج RemoveAt() موجود ضمن الواجهة IList فقط ولا تحتوي الواجهة IDictionary عليه ولكن بالمقابل فإن الواجهة IDictionary تحتوي على المنهج Remove() والذي يمثل أحد المناهج التي يجب أن نكتب شيفرتها ضمن أصناف المجموعات المخصصة أصناف المجموعات التي ننشئها نحن والتي تعتمد على الصنف DictionaryBase.

تبين الشيفرة التالية إصدارا بديلا عن الصنف Animals في القسم السابق حيث قمنا باشتقاق الصنف Animals من الصنف الأساس DictionaryBase ولقد ضمنا المنهجين Add() وRemove() بالإضافة إلى مفهرس مفتاحي:

```
public class Animals:DictionaryBase
{
    public void Add1(string newID,Animal newAnimal)
    {
        Dictionary.Add(newID, newAnimal);
    }
    public void Remove1(string animalID)
    {
        Dictionary.Remove(animalID);
    }
    public Animals ()
    {
```

```
        }
        public Animal this[string animalID]
        {
            get
            {
                return (Animal)Dictionary[animalID];
            }
            set
            {
                Dictionary[animalID] = value;
            }
        }
    }
```

بمقارنة الصنف Animals الحالي مع الصنف Animals السابق نلاحظ ان الأول يرث الصنف الأساس CollectionBase بينما يرث الأخير الصنف الأساس DictionaryBase وعلى الرغم من ان المناهج والخصائص هي نفسها إلا ان هناك بعض الاختلافات بين أعضاء الصنف Animals الأول وأعضاء الصنف Animals هذا:

- يأخذ منهج إضافة عنصر إلى المجموعة Add1() بارامترين الأول يمثل مفتاح العنصر والثاني يمثل العنصر المراد إضافته إلى المجموعة يتضمن الصنف الأساس DictionaryBase العضو Dictionary إن هذا العضو هو عضو للواجهة IDictionary المزود بها الصنف الأساس DictionaryBase.
- أما منهج إزالة عنصر من المجموعة Remove() فهو يأخذ بارامترا واحدا يمثل مفتاح العنصر بدلا من استخدام موقع العنصر (دليله) من المجموعة حيث سيتم إزالة العنصر ذو القيمة المفتاحية المحددة ضمن بارامتر هذا التابع.
- العضو الأخير هو المفهرس والذي يستخدم أيضا بارامترا نصيا من نوع String يمثل مفتاح العنصر بدلا من استخدام دليل العنصر في المجموعة ولاحظ أن علينا استخدام التشكيل casting هنا لأن المنهج Dictionary يعيد قيمة من نوع Object.

هناك فرق آخر بين المجموعات التي تعتمد على الصنف DictionaryBase والمجموعات التي تعتمد على الصنف CollectionBase وهو في طريقة استخدام الحلقة foreach إن استخدام foreach مع الأصناف المشتقة من الصنف DictionaryBase يعيد لنا كائنات من نوع البنية DictionaryEntry وهو نوع آخر معرف ضمن فضاء الأسماء System.Collections وللحصول على كائنات Animal نفسها يجب أن نستخدم الخاصية Value في هذه البنية ويمكننا أيضا استخدام العضو key في هذه البنية للحصول على قيمة مفتاح العنصر.

بالعودة إلى شيفرة المرور عبر عناصر مجموعة CollectionBase:

```
            foreach (Animal myAnimal in animalcollection )
            {
                myAnimal.Feed();
            }
```

لكي نتمكن من كتابة شيفرة مشابهة لتلك ولكن مع مجموعات DictionaryBase علينا كتابة شيفرة كما يلي:

```
foreach (DictionaryEntry myEntry in animalcollection)
{
    ((Animal)myEntry.Value).Feed();
}
```

من الممكن في الحقيقة تجاوز هذا السلوك والوصول إلى عناصر Animal بصورة مباشرة ضمن حلقات foreach إلا أن هذا الموضوع معقد ولن نتحدث في تفاصيل ذلك في هذا الكتاب.

التحديث الأول لمكتبة أصناف أوراق اللعب:

Upgrading CardLibrary part1:

لقد قمنا في نهاية الفصل السابق بإنشاء مشروع لمكتبة أصناف ورق اللعب باسم CardLibrary ولقد احتوى على الصنف Card الذي يمثل ورقة لعب واحدة بالإضافة إلى الصنف Deck والذي يمثل مجموعة أوراق اللعب كاملة أي أن الصنف Deck يمثل مجموعة من كائنات Card لقد أنشأنا لهذا الغرض مصفوفة بسيطة تؤدي المهمة بصورة لا بأس بها.

سوف نقوم في هذا الفصل بإضافة صنف جديد إلى هذه المكتبة وسوف نعيد تسمية مكتبة الأصناف بالاسم CardLibrary1 الصنف الجديد هو Cards والذي يمثل مجموعة مخصصة لكائنات Card وبواسطة هذه المجموعة فإننا سنحصل على المميزات التي تحدثنا عنها مسبقا في هذا الفصل الشيفرة الكاملة للملف Cards.cs هي كالتالي.

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

namespace CardLibrary1
{
    public class Cards:CollectionBase
    {
        public void Add1(Card newCard)
        {
            List.Add(newCard);
        }
        public void Remove1(Card oldCard)
        {
            List.Remove(oldCard);
        }
        public Cards ()
        {
        }
```

```
        public Card this[int cardIndex]
        {
            get
            {
                return (Card)List[cardIndex];
            }
            set
            {
                List[cardIndex] = value;
            }
        }
        public bool Contains1(Card card)
        {
            return InnerList.Contains(card);
        }
    }
}
```

توضيح:

نلاحظ إضافة منهج جديد باسم Contains1 يعيد قيمة منطقية من نوع bool يأخذ هذا المنهج بارامترا من النوع Card ويقوم بالتحقق فيما إذا كانت المجموعة تحتوي على هذا الكائن أم لا أي أنه يمكننا استخدام هذا المنهج للتحقق فيما إذا كانت مجموعة أوراق اللعب تحتوي على ورقة لعب معينة أم لا.

بعد ذلك علينا تعديل الملف Deck.cs لكي يتمكن من استخدام المجموعة الجديدة تلك بدلا من استخدام المصفوفة:

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

namespace CardLibrary1
{
    public class Deck
    {
        private Cards cards = new Cards();
        public Deck ()
        {
           // Cards = new Card[52];
            for (int suitVal=0;suitVal <4;suitVal ++)
            {
                for (int rankVal=1;rankVal <14;rankVal ++)
                {
                    cards.Add1(new Card((Suit)suitVal, (Rank)rankVal));
                }
            }
        }
        public Card GetCard(int cardNum)
        {
            if (cardNum >= 0 && cardNum <= 51)
```

```
            return cards[cardNum];
        else
            throw (new System.ArgumentOutOfRangeException("cardNum",
               cardNum, "Value must be between 0 and 51."));
    }
    public void Shuffle()
    {
        //Card[] newDeck = new Card[52];
        Cards newDeck = new Cards();
        bool[] assigned = new bool[52];
        for (int i=0;i<52;i++)
        {
            int sourceCard = 0;
            bool foundCard = false;
            Random sourceGen = new Random();
            while (foundCard ==false )
            {
                sourceCard = sourceGen.Next(52);
                if (assigned[sourceCard] == false)
                    foundCard = true;
            }
            assigned [sourceCard ]=true ;
            newDeck.Add1(cards[sourceCard]);
        }
        cards  = newDeck;
    }
  }
}
```

ملاحظة:

لاحظ أن علينا تعديل تصريحات فضاءات الأسماء عند نسخ ملفات المصدر من CdrdLibrary إلى CardLibrary1 وذلك للإشارة إلى فضاء الأسماء CardLibrary1 إن هذا مطبق أيضا على الملف Card.cs والتطبيق الزبون ConsoleCardClient أيضا.

ليست هناك تعديلات تستوجب مناقشتها في هذه الشيفرة فمعظم التعديلات التي أجريناها في الشيفرة أعلاه بمنطق خلط الأوراق عشوائيا وحقيقة إضافة أوراق اللعب إلى مجموعة Cards الجديدة التي تحمل الاسم newDeck من دليل عشوائي في أوراق اللعب وذلك بدلا من إضافتها إلى دليل عشوائي في المصفوفة newCards من موقع تتابعي في المجموعة cards.

يمكننا استخدام برنامج الزبون ConsoleCardClient الذي استخدمناه لتجربة مكتبة الأصناف CardLibrary مع مكتبة الأصناف الجديدة حيث سنحصل على النتائج نفسها وذلك باعتبار أن تواقيع(تصريحات) المناهج للصنف Deck لم تتغير ولكن يمكن لتطبيق الزبون الآن الاستفادة من صنف مجموعة Cards بدلا من الاعتماد على مصفوفة لكائنات من النوع Card.

التحميل الزائد للعوامل:

Operator Overloading:

الموضوع التالي الذي سنتناوله الآن يتحدث عن التحميل الزائد للعوامل Operator overloading تمكننا تقنية التحميل الزائد للعوامل من استخدام العوامل القياسية مثل + و> مع الأصناف التي نقوم بإنشائها بأنفسنا لقد سمي هذا السلوك بالتحميل الزائد وذلك لأننا نقوم بإعطاء معان خاصة لتلك العوامل غير المعاني العادية التي تحملها هذه العوامل وذلك عند استخدامها مع أنواع بارامترات محددة إن هذا مشابه إلى حد ما لسلوك وضع تواقيع أنواع بارامترات جديدة مختلفة للمناهج.

إن استخدام التحميل الزائد للعوامل مفيد جدا باعتبار أنه يمكننا القيام بأية معالجة نود القيام بها ضمن التزويد المخصص لهذا العامل المحمل بصورة زائدة وهو ما قد يتجاوز منطق العامل الأساسي كمثال العامل+ الذي يعني إضافة الحد الأول إلى الحد الثاني سوف نرى قريبا مثالا جيدا لذلك عندما نقوم بتحديث آخر لمكتبة أصناف ورق اللعب CardLibrary سوف نقوم بتزويد الأصناف بتزويدات خاصة لعوامل المقارنة بحيث نتمكن من استخدام هذه العوامل لمقارنة ورقتي لعب فيما إذا كانت إحداها تغلب الأخرى.

تتم مقارنة أوراق اللعب بالاعتماد على اشكال الأوراق وهي متمثلة بأربعة مجموعات فقط كما نعلم القلب ♥والديناري ♦والسباتي ♣والبستوني ♠والأمر هنا ليس مجرد مقارنة أرقام (رتب) أوراق اللعب وحسب فإن كان شكل ورقة اللعب الثانية مختلف عن شكل ورقة اللعب الأولى فإن ورقة اللعب الأولى ستربح بغض النظر عن رتبة ورقة اللعب الثانية إن كيفية تحديد ترتيب الورقة التي تم طرحها قبل الثانية يعتمد على ترتيب الحدود في العامل وفي حالات لألعاب أخرى يمكن ان يكون لدينا شكل رابح دائما Trump كالطرنيب في لعبة الطرنيب وفي حالة كهذه فإن الورقة التي تنتمي إلى الشكل الرابح هي الورقة الرابحة دوما بغض النظر عن شكل الورقة الأخرى أو ترتيب طرح الورقتين أثناء اللعب هذا يعني أن احتساب نتيجة المقارنة Card1>Card2 فيما إذا كانت مساوية لـ true أي أن ورقة اللعب Card1 ستغلب ورقة اللعب Card2 إذا تم طرح ورقة اللعب Card1 أولا لا يعني بالضرورة ان نتيجة المقارنة Card2>Card1 ستأخذ القيمة false فإن لم تكن ورقتي اللعب Card1 وCard2 كلتاهما تنتميان إلى الشكل الرابح دوما الطرنيب وإنما إلى شكلين مختلفين ففي هذه الحالة ستكون نتيجة المقارنتين السابقتين مساوية للقيمة true.

كبداية لنلق نظرة على الصيغة الأساسية للتجميل الزائد للعوامل.

يمكننا تحميل العوامل بصورة زائدة وذلك بإضافة أعضاء أنواع العوامل إلى الصنف وهي يجب أن تكون أعضاء ستاتيكية Static دوما قد يكون لبعض العوامل استخدامات متعددة مثل العامل (-) الذي يمكن استخدامه كعامل أحادي أو ثنائي ولذلك فإن علينا تحديد عدد الحدود التي سيتعامل معها العامل بالإضافة إلى تحديد أنواع الحدود أيضا وبصورة عامة فإن الحدود التي سنتعامل معها ستكون من نوع واحد دائما وهو الصنف الذي تم تعريف العامل فيه مع العلم أن بإمكاننا تعريف العوامل لكي تتعامل مع حدود تنتمي إلى أنواع مختلفة كما سنرى كيفية ذلك لاحقا.

وكمثال لنفترض الصنف البسيط AddClass1 المعرف كما يلي:

```
public class AddClass1
{
    public int val;
}
```

يمثل هذا الصنف صنفا مغلفا لقيمة من نوع int والآن إذا استخدمنا شيفرة كما يلي وفقا للصنف السابق فإن المترجم سيعطي رسالة خطأ:

```
AddClass1 op1=new AddClass1 ();
op1 .val =5;
AddClass1 op2 = new AddClass1();
op2.val = 7;
AddClass1 op3 = op1 + op2;
```

يمكنك أن تلاحظ من رسالة الخطأ الناتجة أن الخطأ حاصل من أن العامل + لا يمكن تطبيقه على حدود من النوع AddClass1 وذلك باعتبار أنه ليس هناك تعريف لهذه العوامل لاستخدامها وفقا لذلك.

1 Error Operator '+' cannot be applied to operands of type 'ConsoleApplication1.AddClass1' and 'ConsoleApplication1.AddClass1'

إن شيفرة كالشيفرة التالية ستعمل بصورة نظامية مع أنها لن تعطينا النتيجة التي نبتغيها:

```
AddClass1 op1=new AddClass1 ();
op1 .val =5;
AddClass1 op2 = new AddClass1();
op2.val = 7;
bool op3 = op1 == op2;
```

بعد تنفيذ شيفرة كهذه سيأخذ المتحول op3 القيمة false لقد تم هنا مقارنة المتحول op1 مع المتحول op2 وذلك باستخدام العامل المنطقي == وذلك لمعرفة فيما إذا كان كلا المتحولين يشيران إلى الكائن نفسه أم لا وليس للتحقق من أن قيمتهما متساوية وهذا يعني أن نتيجة المقارنة ستبقى false حتى إن كانت قيمة op1.val مساوية لقيمة op2.val طالما ان المتحولين لا يشيران إلى الكائن نفسه.

ولكي نتمكن من تحميل العامل + بصورة زائدة فإننا سنستخدم شيفرة كما يلي:

```
public class AddClass1
{
    public int val;
    public static AddClass1 operator+(AddClass1 op1,AddClass1 op2)
    {
        AddClass1 returnVal = new AddClass1();
        returnVal.val  = op1.val + op2.val;
        return returnVal;
    }
}
```

وكما ترى هنا فإن التصريح عن التحميل الزائد للعامل مشابه لشيفرة التصريح عن منهج ستاتيكي فيما عدا أننا نستخدم الكلمة المفتاحية operator ورمز العامل بدلا من اسم المنهج.

يمكننا الآن استخدام العامل + مع كائنات من النوع AddClass1 كما لو كنا نستخدمه مع أية قيم رقمية كمثال:

```
AddClass1 op3 = op1 + op2;
```

التحميل الزائد للعوامل المنطقية يتم بنفس الصورة أما التحميل الزائد للعوامل الأحادية فهو يتم بنفس الصورة أيضا فيما عدا أننا نستخدم هنا بارامترا واحدا بدلا من اثنين:

```
public class AddClass1
{
    public int val;
    public static AddClass1 operator+(AddClass1 op1,AddClass1 op2)
    {
        AddClass1 returnVal = new AddClass1();
        returnVal.val  = op1.val + op2.val;
        return returnVal;
    }
    public static AddClass1 operator -(AddClass1 op1)
    {
        AddClass1 returnVal = new AddClass1();
        returnVal.val = -op1.val;
        return returnVal;
    }
}
```

نلاحظ اننا عرفنا هنا عاملين بصورة زائدة حيث يطبق كلا العاملين على النوع نفسه وهو نوع الصنف المعرف ضمنه ويعيدان قيمة من نفس نوع الصنف أيضا لنأخذ على سبيل المثال شيفرة الأصناف التالية:

```
public class AddClass1
{
    public int val;
    public static AddClass3 operator+(AddClass1 op1,AddClass2 op2)
    {
        AddClass3 returnVal = new AddClass3();
        returnVal.val  = op1.val + op2.val;
        return returnVal;
    }
}
public class AddClass2
{
    public int val;
}
public class AddClass3
{
    public int val;
}
```

والآن يمكننا استخدام الشيفرة السابقة كما يلي:

```
AddClass1 op1=new AddClass1 ();
op1 .val =5;
AddClass2 op2 = new AddClass2();
op2.val = 5;
AddClass3 op3 = op1 + op2;
```

نلاحظ هنا أننا استخدمنا العامل + مع حدين ينتميان لنوعين مختلفين وهذا ممكن طالما اننا أضفنا تعريفا للعامل + ضمن الصنف AddClass3 ويعالج ذلك لكن لاحظ أنه لو قمنا بتعريف هذا العامل بنفس الصورة ضمن الصنف AddClass2 وقمنا بتنفيذ الشيفرة السابقة فإنها لن تعمل وذلك لأننا في حالة لبس باعتبار ان هناك تعريفان للعامل نفسه وبالتالي علينا ان ننتبه إلى مسألة كهذه وذلك بأن لا نضيف العامل نفسه وبنفس التوقيع إلى أكثر من صنف واحد.

لاحظ أيضا عند خلط الأنواع كما في العامل السابق علينا ان نراعي مسألة ترتيب ورود الحدود بحيث يوافق نفس ترتيب البارامترات في شيفرة التصريح عن العامل فإذا حاولنا أن نضع الحدود بترتيب مخالف لترتيب ورودها ضمن توقيع العامل فإن ذلك سيتسبب في حدوث خطأ وبناء على ذلك فإنه لا يمكننا كتابة شيفرة كما يلي:

```
AddClass3 op3 = op2 + op1;
```

إلا إذا قمنا بإضافة تحميل زائد آخر للعامل بحيث نعكس فيه ترتيب البارامترات طبعا:

```
public static AddClass3 operator +(AddClass2 op1, AddClass1 op2)
{
    AddClass3 returnVal = new AddClass3();
    returnVal.val = op1.val + op2.val;
    return returnVal;
}
```

والآن إليك لائحة بالعوامل التي يمكنك تحميلها بشكل زائد:

- العوامل الأحادية: true, false,++,--,~,-,+.
- العوامل الثنائية:+,-,*,/,%,&,|,^,<<,>>.
- عوامل المقارنة:==,!=,<,>,<=,>=.

ملاحظة:

إذا قمنا بتحميل العاملين true وfalse بصورة زائدة يمكننا استخدام الأصناف ضمن التعابير المنطقية ككتابة if(op1){} مثلا.

لا يمكننا استخدام التحميل الزائد مع عوامل الإسناد مثل العامل =+ إلا ان هذا النوع من العوامل لا يمثل إلا حالة خاصة من عوامل ثنائية مثل العامل =+ والعامل + والعامل =- والعامل – وبالتالي فإننا عندما نقوم بتحميل العامل + بصورة زائدة فإن العامل =+ سيعمل بصورة موافقة للعامل الأساسي وهو العامل+ إن عامل الإسناد الطبيعي = يدخل في ذلك أيضا فمن المنطقي جدا أن نحمل هذا العامل بشكل زائد باعتبار

ان لهذا العامل استخدام أساسي على كل حال إن هذا العامل مرتبط بعوامل تحويل الأنواع المعرفة من قبل المستخدم وهو ما سنتحدث عنه في القسم التالي.

لاحظ انه لا يمكننا استخدام التحميل الزائد مع العاملين المنطقيين && و|| إلا أننا نعلم أيضا أن هذين العاملين يمثلان حالة خاصة للعاملين & و| على التوالي ولذا فإن التحميل الزائد للعامل & يعني اننا قمنا أيضا بالتحميل الزائد للعامل && بالمثل.

هناك بعض العوامل التي يجب ان تحمل بصورة زائدة على هيئة ازواج مثل العاملان > و < فلا يمكننا ان نستخدم التحميل الزائد مع احد العوامل إلا إذا استخدمناه أيضا مع العامل الآخر أي أنه لا يمكننا تحميل العامل > بشكل زائد ما لم نحمل العامل < بشكل زائد أيضا على سبيل المثال:

```
public class AddClass1
{
    public int val;
    public static bool operator >=(AddClass2 OP1,AddClass1 OP2)
    {
        return (OP1.val >= OP2.val);
    }
    public static bool operator <=(AddClass2 OP1, AddClass1 OP2)
    {
        return (OP1.val >= OP2.val);
    }
    public static bool operator < (AddClass2 op1,AddClass1 op2)
    {
        return (op1.val < op2.val);
    }
    public static bool operator > (AddClass2 op1, AddClass1 op2)
    {
        return (op1.val < op2.val);
    }
}
public class AddClass2
{
    public int val;
}
```

إن الامر نفسه مطبق على العاملين == و=! لكن من الأفضل هنا ان نقوم بتجاوز المنهجين Object.Equals() و Object.GetHashCode() باعتبار ان بإمكاننا استخدام هذين المنهجين لمقارنة الكائنات فبتجاوز هذين المنهجين فإننا نكون واثقين من اننا سنحصل على نتيجة المقارنة نفسها مهما كانت الطريقة المستخدمة للمقارنة لاحظ ان ذلك ليس أمرا واجبا إلا أنه من الأفضل استخدامه بهدف الكمالية لا أكثر ولتحقيق ذلك علينا إضافة تزويد جديد للمنهجين كما يلي:

```
    public static bool operator ==(AddClass2 OP1, AddClass1 OP2)
    {
        return (OP1.val == OP2.val);
    }
    public static bool operator !=(AddClass2 OP1, AddClass1 OP2)
    {
```

```
        return !(OP1.val == OP2.val);
    }
    public override bool Equals(object op1)
    {
        //return base.Equals(obj);
        return val == ((AddClass1)op1).val;
    }
    public override int GetHashCode()
    {
        //return base.GetHashCode();
        return val;
    }
```

لاحظ هنا ان المنهج Equals() يستخدم بارامترا من النوع Object ونحن يجب أن نستخدم هذا التوقيع او علينا استخدام التحميل الزائد لهذا المنهج بدلا من تجاوزه وسنتمكن من استخدام التزويد الافتراضي من ضمن هذا الصنف وفي تلك الحالة يجب استخدام التشكيل cast للحصول على النتيجة المطلوبة.

يستخدم المنهج GetHashCode للحصول على قيمة int فريدة للكائن وذلك بالاعتماد على حالته يمكننا هنا استخدام القيمة val باعتبار أنها من النوع int أيضا.

عوامل التحويل:

Conversion Operators:

لقد رأينا في القسم السابق كيفية التحميل الزائد للعوامل الرياضية وهذا يعني إمكانية استخدام هذه العوامل مع أنواع معطيات مخالفة للمنطق العام الذي تستخدم لأجله سنتعرف هنا أيضا على ميزة أخرى للغة C# والتي تمكننا من تعريف عمليات التحويل بين الأنواع سواء المطلقة منها أو الصريحة إن هذا مهم في كثير من المواضع التي نحتاج فيها إلى تحويل أنواع غير مرتبطة مع بعضها البعض فعلى سبيل المثال إذا لم تكن هناك أية علاقة وراثية بين هذه الأنواع وليست هناك أية واجهات مشتركة بينها.

لنفترض اننا نود تعريف عملية تحويل مطلقة implicit بين النوع convClass1 وconvClass2 إن هذا يعني انه يمكننا كتابة شيفرة كما يلي:

```
convClass1 op1 = new convClass1();
convClass2 op2 = op1;
```

توضيح:

تذكر أن ما نقصده بالأنواع يشمل الأصناف والواجهات والبنى والتعدادات بالإضافة إلى الأنواع الأساسية التي يوفرها إطار عمل .NET بصورة عامة.

وبصورة بديلة لنفترض اننا نود تعريف عملية تحويل صريحة بين النوعين السابقين وهذا يعني انه يمكننا كتابة شيفرة كما يلي:

```
convClass1 op1 = new convClass1();
convClass2 op2 = (convClass2)op1;
```

وكمثال لنأخذ الشيفرة التالية:

```
public class convClass1
{
    public int val;
    public static implicit operator convClass2 (convClass1 op1)
    {
       convClass2 retrunVal=new convClass2 ();
        retrunVal .val =op1 .val ;
        return retrunVal ;
    }
}
public class convClass2
{
    public double val;
    public static explicit operator convClass1(convClass2 op1)
    {
        convClass1 retrnVal = new convClass1();
        checked
        {
            retrnVal.val = (int)op1.val;
        }
        return retrnVal;
    }
}
```

في البداية تتلخص هذه الشيفرة بتعريف صنفين convClass1 وconvClass2 بالإضافة إلى تعريف علاقتي تحويل بين النوعين الأولى لتحويل قيم convClass1 إلى convClass2 والثانية لتحويل قيم convClass2 إلى convClass1.

نلاحظ هنا أن النوع او الصنف قيمة convClass1 يتضمن من نوع int والنوع convClass2 يتضمن قيمة من النوع double وبما ان مجال قيم النوع int محتوى ضمن مجال قيم النوع double فهذا يعني أن تحويل القيم من النوع int إلى double لن يسبب أية مشاكل مهما كانت قيمة int لذا فإن تحويلا كهذا هو تحويل مطلق implicit ولقد قمنا ضمن الصنف convClass2 بتعريف تحويل مطلق بين النوعين convClass1 وconvClass2 لكن لاحظ ان التحويل العكسي بين النوعين int وdouble لا يمثل تحويلا مطلقا وهذا لأن مجال قيم النوع double يتعدى مجال قيم النوع int لذا قمنا بتعريف تحويل صريح explicit.

لاحظ أيضا أننا استخدمنا الكلمتين المفتاحيتين implicit وexplicit في سطر تصريح التحويل وبناء على الشيفرة السابقة فإن كتابة شيفرة كما يلي هو أمر مقبول تماما:

```
convClass1 op1 = new convClass1();
op1.val = 3;
convClass2 op2 = op1;
```

لكن يتطلب التحويل في الاتجاه المعاكس عملية تحويل صريحة تستوجب استخدام التشكيل casting كما يلي:

```
convClass2 op1 = new convClass2();
op1.val = 3e15;
convClass1 op2 = (convClass1)op1;
```

لاحظ أننا استخدمنا في تعريف التحويل الصريح الكلمة المفتاحية checked وهذا يعني أننا سنحصل على اعتراض بعد تنفيذ الشيفرة السابقة لأن قيمة الحقل val للمتحول op1 أكبر بكثير من أن تتسع ضمن الحقل val1 للمتحول op2 وهذا منطقي لأن القيمة 3e15 تقع خارج مجال قيم النوع int.

التحديث الثاني لمكتبة أصناف أوراق اللعب:

Upgrading CardLibrary part2:

والآن سنقوم بتحديث مشروع مكتبة أصناف ورق اللعب مجددا وذلك بإضافة ميزة التحميل الزائد لصنف ورقة اللعب Card علينا أن نضيف حقولا جديدة للصنف Card وذلك لكي نسمح بتحديد الشكل الرابح دوما الطرنيب بالإضافة إلى خيار يحدد فيما إذا كان الآس أكبر من جميع الرتب الأخرى سنجعل هذه الحقول حقولا ستاتيكية وبالتالي عند وضع قيمة لهذين الحقلين فإن هذه القيمة ستسري على جميع الكائنات من نوع Card.

```
public class Card
{
    public static bool useTrumps = false;
    public static Suit trump = Suit.Clup;
    public static bool isAceHigh = true;
    public readonly Suit suit;
    public readonly Rank rank;
    public override string ToString()
    {
        //return base.ToString();
        return "The "+rank +" of "+suit+"s";
    }
    private Card ()
    {
    }
    public Card (Suit newSuit,Rank newRank)
    {
        suit = newSuit;
        rank = newRank;
    }
}
```

وبما اننا قمنا بذلك فمن الجدير أن نضع مناهج البناء للصنف Deck وذلك لكي نتمكن من تهيئة مجموعات أوراق اللعب بصفات مختلفة.

```
using System;
```

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

namespace CardLibrary1
{
    public class Deck
    {
        private Cards cards = new Cards();
        public Deck ()
        {
           // Cards = new Card[52];
            for (int suitVal=0;suitVal <4;suitVal ++)
            {
                for (int rankVal=1;rankVal <14;rankVal ++)
                {
                    cards.Add1(new Card((Suit)suitVal, (Rank)rankVal));
                }
            }
        }
        public Deck (bool isAceHigh):this()
        {
            Card.isAceHigh = isAceHigh;
        }
        public Deck (bool useTrumps,Suit trump):this()
        {
            Card.useTrumps = useTrumps;
            Card.trump = trump;
        }
        public Deck (bool isAceHigh,bool useTrumps,Suit trump):this()
        {
            Card.isAceHigh = isAceHigh;
            Card.useTrumps = useTrumps;
            Card.trump = trump;
        }
        public Card GetCard(int cardNum)
        {
            if (cardNum >= 0 && cardNum <= 51)
                return cards [cardNum];
            else
                throw (new System.ArgumentOutOfRangeException("cardNum",
                   cardNum, "Value must be between 0 and 51."));
        }
        public void Shuffle()
        {
            //Card[] newDeck = new Card[52];
            Cards newDeck = new Cards();
            bool[] assigned = new bool[52];
            for (int i=0;i<52;i++)
            {
                int sourceCard = 0;
                bool foundCard = false;
                Random sourceGen = new Random();
```

```
                while (foundCard ==false )
                {
                    sourceCard = sourceGen.Next(52);
                    if (assigned[sourceCard] == false)
                        foundCard = true;
                }
                assigned [sourceCard ]=true ;
                newDeck.Add1(cards[sourceCard]);
            }
            cards  = newDeck;
        }
    }
}
```

لقد عرفنا مناهج البناء الثلاثة السابقة باستخدام الكلمة this() الذي تعرفنا عليه في الفصل الثاني من هذا الجزء وبالتالي ففي جميع الأحوال سيتم استدعاء منهج البناء الافتراضي الذي يقوم بتهيئة مجموعة أوراق اللعب قبل استدعاء المنهج غير الافتراضي.

بعد ذلك سنقوم بإضافة بعض التحميلات الزائدة للعوامل بالإضافة إلى تجاوز المنهجين Equals() وGetHashTable() للصنف الأب Object في الصنف Card:

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

namespace CardLibrary1
{
    public enum Suit
    {
        Clup,
        Diamond,
        Heart,
        Spade
    }
    public enum Rank
    {
        Ace = 1,
        Deuce,
        Three,
        Four,
        Five,
        Six,
        Seven,
        Eight,
        Nine,
        Ten,
        Jack,
        Queen,
        King
    }
```

```
public class Card
{
    public static bool useTrumps = false;
    public static Suit trump = Suit.Clup;
    public static bool isAceHigh = true;
    public readonly Suit suit;
    public readonly Rank rank;
    public override string ToString()
    {
        //return base.ToString();
        return "The "+rank +" of "+suit+"s";
    }
    private Card ()
    {
    }
    public Card (Suit newSuit,Rank newRank)
    {
        suit = newSuit;
        rank = newRank;
    }
    public static bool operator==(Card card1,Card card2)
    {
        return (card1.suit == card2.suit) && (card1.rank == card2.rank);
    }
    public static bool operator !=(Card card1,Card card2)
    {
        return !(card1 == card2);
    }
    public override bool Equals(object card)
    {
        //return base.Equals(obj);
        return this == (Card)card;
    }
    public override int GetHashCode()
    {
        //return base.GetHashCode();
        return 12 * (int)rank + (int)suit;
    }
    public static bool operator >(Card card1,Card card2)
    {
        if (card1 .suit ==card2 .suit )
        {
            if (isAceHigh)
                return (card1.rank > card2.rank) ^ (card2.rank == Rank.Ace);
            else
                return (card1.rank > card2.rank);
        }
        else
        {
            if (useTrumps && (card2.suit == Card.trump))
                return false;
            else
                return true;
        }
```

```
        }
        public static bool operator < (Card card1,Card card2)
        {
            return !(card1 >= card2);
        }
        public static bool operator >=(Card card1,Card card2)
        {
            if (card1 .suit ==card2 .suit )
            {
                if (isAceHigh)
                    return (card1.rank >= card2.rank) ^ (card2.rank == Rank.Ace);
                else
                    return (card1.rank >= card2.rank);
            }
            else
            {
                if (useTrumps && (card2.suit == Card.trump))
                    return false;
                else
                    return true;
            }
        }
        public static bool operator <=(Card card1,Card card2)
        {
            return !(card1 > card2);
        }
    }
}
```

ليس هناك ما يحتاج للتوضيح في هذه الشيفرة فيما عدا الاستخدام المعقد بعض الشيء للعامل المنطقي XOR لقد استخدم هذا العامل مع التحميل الزائد للعاملين > و< وإذا تتبعنا القيم الممكنة لسطر الشيفرة الذي يستخدم هذا العامل سنلاحظ كيفية عمله وضرورته هنا.

نحن نقوم بمقارنة ورقتي لعب هنا:card1 وcard2 حيث يفترض لورقة اللعب card1 أن تكون الورقة المطروحة أولا أي التي تم لعبها أولا وكما ناقشنا ذلك مسبقا فإن هذا العامل يصبح ضروريا جدا عند استخدام أوراق اللعب الرابحة دوما وهي أوراق الطرنيب باعتبار أن ورقة الطرنيب تربح دوما أوراق اللعب من غير فئة الطرنيب حتى إن كانت رتبة ورقة الطرنيب هي الأدنى وبالطبع إذا كانت كلتا الورقتين من الفئة أو الشكل نفسها حتى وإن كانت من فئة الطرنيب فإن ورقة اللعب الرابحة هي ورقة اللعب ذات الرتبة الأكبر لذا فإن التعليمة الشرطية if الأولى تقوم بالتحقق من ذلك:

```
        public static bool operator >(Card card1,Card card2)
        {
            if (card1 .suit ==card2 .suit )
            {
```

والآن إذا كانت قيمة العلم isAceHigh تساوي true فهذا يعني أن الآس هو أعلى رتبة في الفئة وبالتالي لا يمكننا مجرد مقارنة رتبتي الورقتين مباشرة حيث أن قيمة الآس هي 1 ضمن التعداد Rank وقيمته أقل من جميع الرتب الأخرى هنا يأتي دور العامل ^ المنطقي فنحن نقوم هنا بعمليتي مقارنة الأولى على

قيمة الرتبة والثانية لمعرفة فيما إذا كانت الرتبة هي الآس او لا إن الاحتمالات المتوقعة في هذه الحالة هي:

- إذا كانت رتبة الورقة card1 أكبر من رتبة الورقة card2 ولم تكن رتبة الورقة card2 هي الآس فإن الورقة card1 هي الأعلى.
- إذا كانت رتبة الورقة card1 أكبر من رتبة الورقة card2 ولم تكن رتبة الورقة card2 هي الآس فإن الورقة card2 هي الأعلى.
- إذا كانت رتبة الورقة card1 أصغر من رتبة الورقة card2 ولم تكن رتبة الورقة card1 هي الآس فإن الورقة card2 هي الأعلى.
- إذا كانت رتبة الورقة card1 أصغر من رتبة الورقة card2 ولم تكن رتبة الورقة card1 هي الآس فإن الورقة card1 هي الأعلى.

يمكننا تمثيل ذلك بالجدول التالي:

card1>card2	card2.rank=Rank.Ace	نتيجة المقارنة
true	**false**	**true**
true	**true**	**false**
false	**false**	**false**
false	**true**	**true**

وباسترجاع كيفية العامل المنطقي XOR أو ^:

op1	op2	op1^op2
true	false	true
true	true	false
false	false	false
false	true	true

لاحظ أن المنطق هنا متطابق تماما وبالتالي فإن استخدام العامل ^ سيعطينا النتيجة التي نبتغيها:

```
if (isAceHigh)
    return (card1.rank > card2.rank) ^ (card2.rank == Rank.Ace);
else
```

ملاحظة:

إن استخدام العامل ^ في حالات كهذه يعد أمرا ذكيا والقليل من المبرمجين الذين يمكنهم التفكير بهذا الأسلوب فمعظم المبرمجين يلجئون إلى تعشيش تعابير if للحصول على النتيجة نفسها التي يمكن الحصول عليها في سطر واحد.

إن كانت قيمة isAceHigh تساوي false فإن هذا يعني أن الآس لا يمثل الرتبة الأعلى وهو في موقعه من التعداد Rank وبالتالي فإن عملية المقارنة البسيطة بين رتبتي ورقتي اللعب كافية:

```
            else
                return (card1.rank > card2.rank);
        }
```

الجزء المتبقي من هذه الشيفرة متعلق بالحالة التي تكون فيها فئة ورقتي اللعب مختلفة عندئذ سنستخدم المتحول الستاتيكي useTrump والذي يشير فيما إذا كانت هناك فئة رابحة أي فئة طرنيب ضمن مجموعة أوراق اللعب أم لا فإن كانت قيمة العلم useTrump تساوي true وكانت ورقة اللعب من فئة الطرنيب فإننا نستطيع ان نقول إن ورقة اللعب card1 ليست من فئة الطرنيب فلكي نصل إلى هذه النقطة من الشيفرة يجب ان تكون كلتا الورقتين من فئتين مختلفتين لذا فإن ورقة اللعب card2 هي الورقة الأعلى في هذه الحالة:

```
        else
        {
            if (useTrumps && (card2.suit == Card.trump))
                return false;
```

ولكن إذا لم تكن ورقة اللعب card2 من فئة الطرنيب أو إذا لم تستخدم مجموعة أوراق اللعب سياسة الطرنيب بتاتا أي إذا كانت قيمة useTrumps تساوي false فإن ورقة اللعب card1 هي الأكبر الرابحة باعتبار أنها الورقة المطروحة أولا:

```
            else
                return true;
        }
    }
```

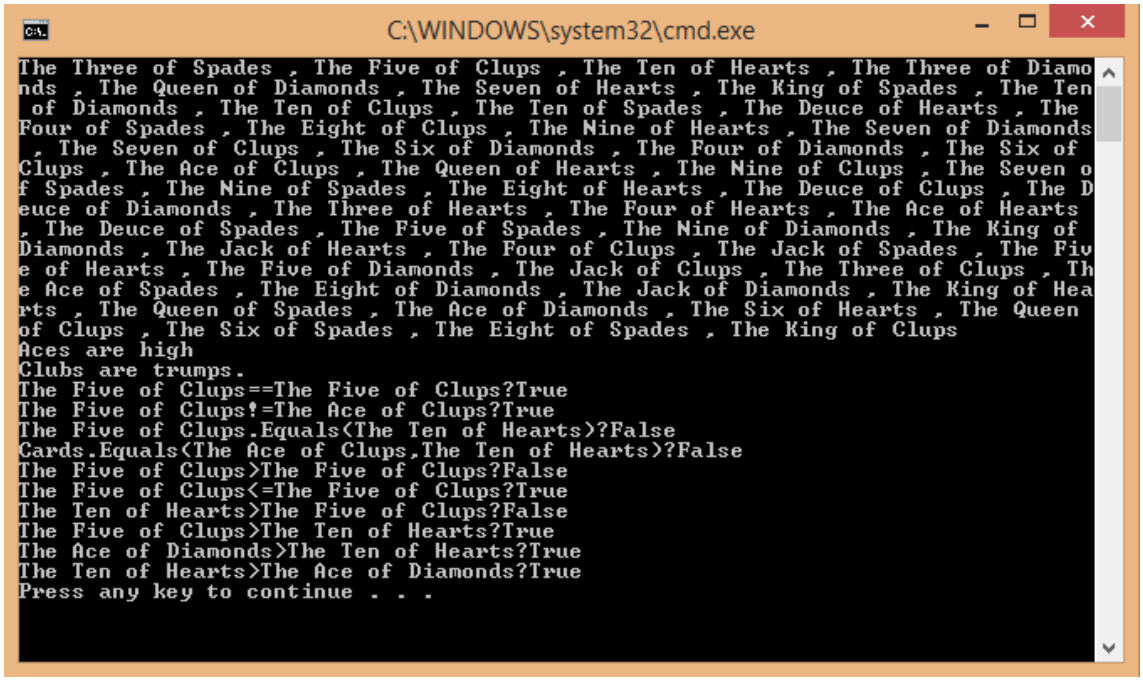
هناك عامل آخر سيستخدم نفس الأسلوب الذي اتبعناه مع هذا العامل وهو العامل =< وأما باقي العوامل فهي عوامل بسيطة جدا لذا لن نتحدث عنها بالتفصيل.

تبين الشيفرة التالية مثالا لاستخدام هذه العوامل ضع هذه الشيفرة ضمن التابع Main() لمشروع الزبون الذي استخدمناه مسبقا مع أمثلة مكتبة الأصناف CardLiprary.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using CardLibrary1;

namespace ConsoleCardClient
{
    class Program
    {
        static void Main(string[] args)
        {
            Deck myDeck = new Deck();
```

```
            myDeck.Shuffle();
            for (int i=0;i<52;i++)
            {
                Card tempCard = myDeck.GetCard(i);
                Console.Write(tempCard.ToString());
                if (i != 51)
                    Console.Write(" , ");
                else
                    Console.WriteLine();
            }
            Card.isAceHigh = true;
            Console.WriteLine("Aces are high");
            Card.useTrumps = true;
            Card.trump = Suit.Clup;
            Console.WriteLine("Clubs are trumps.");
            Card card1, card2, card3, card4, card5;
            card1 = new Card(Suit.Clup, Rank.Five);
            card2 = new Card(Suit.Clup, Rank.Five);
            card3 = new Card(Suit.Clup, Rank.Ace);
            card4 = new Card(Suit.Heart, Rank.Ten);
            card5 = new Card(Suit.Diamond, Rank.Ace);
            Console.WriteLine("{0}=={1}?{2}",
                card1.ToString(), card2.ToString(), card1 == card2);
            Console.WriteLine("{0}!={1}?{2}",
                card1.ToString(), card3.ToString(), card1 != card3);
            Console.WriteLine("{0}.Equals({1})?{2}",
                card1.ToString(), card4.ToString(), card1.Equals(card4));
            Console.WriteLine("Cards.Equals({0},{1})?{2}",
                card3.ToString(), card4.ToString(), Card.Equals(card3, card4));
            Console.WriteLine("{0}>{1}?{2}",
                card1.ToString(), card2.ToString(), card1 > card2);
            Console.WriteLine("{0}<={1}?{2}",
                card1.ToString(), card2.ToString(), card1 <= card2);
            Console.WriteLine("{0}>{1}?{2}",
                card4.ToString(), card1.ToString(), card4 > card1);
            Console.WriteLine("{0}>{1}?{2}",
                card1.ToString(), card4.ToString(), card1 > card4);
            Console.WriteLine("{0}>{1}?{2}",
                card5.ToString(), card4.ToString(), card5 > card4);
            Console.WriteLine("{0}>{1}?{2}",
                card4.ToString(), card5.ToString(), card4 > card5);
        }
    }
}
```

والنتيجة ستظهر كما في الشكل التالي:



الشكل (4-5)

التحويلات المتقدمة:

Advanced Conversions:

لقد تعلمنا حتى الآن كيفية تعريف عوامل التحويل وبالتالي أصبح بالإمكان مناقشة بعض مزايا لغة C# المتعلقة بتحويل الأنواع سوف نتناول المواضيع التالية:

- التغليف وعدم التغليف ويقصد بهما التحويل بين أنواع القيمة وأنواع المرجع.
- العامل is والذي يستخدم لتفحص المتحول فيما إذا كان من نوع محدد أو أنه متوافق مع هذا النوع.
- العامل as والذي يستخدم لتحويل المتحول إلى نوع محدد كأسلوب مختلف عن استخدام التشكيل casting.

التغليف وعدم التغليف:

Boxing and Unboxing:

لقد تحدثنا في الفصل الأول من هذا الجزء حول الاختلاف بين أنواع القيمة وأنواع المرجع ولقد عدنا للحديث عن هذا الموضوع في الفصل الثاني أيضا وذلك بمقارنة بنى Struct التي تمثل أنواع قيمة مع الأصناف والتي تمثل أنواع مرجع يعرف التغليف boxing على أنه عملية تحويل نوع القيمة إلى نوع

المرجع System.Object أو إلى نوع واجهة مزودة عبر نوع القيمة أما عدم التغليف unboxing فيمثل العملية المعاكسة للتغليف.

على سبيل المثال لنفترض أن لدينا البنية Struct التالية:

```
struct myStruct
{
    public int val;
}
```

يمكننا تغليف المتحول من هذا النوع إلى متحول من النوع object كما يلي:

```
myStruct valType1 = new myStruct();
valType1.val = 5;
object refType = valType1;
```

لقد قمنا هنا بإنشاء متحول جديد valType1 من النوع myStruct وأسندنا إلى العضو val ضمن هذه البنية قيمة ما ومن ثم قمنا بتغليف هذا المتحول ضمن متحول من النوع object باسم refType يتضمن الكائن الذي تم إنشائه بتغليف متحول بهذه الصورة على مرجع لنسخة من متحول نوع القيمة يتضمن الكائن الذي تم إنشائه وليس مرجعا لمتحول نوع القيمة الأصلي ويمكننا التحقق من ذلك بتعديل محتوى البنية الأصلي ومن ثم إزالة تغليف البنية في المتحول من نوع object إلى متحول جديد وتفحص محتويات متحول القيمة الجديد:

```
myStruct valType1 = new myStruct();
valType1.val = 5;
object refType = valType1;
valType1.val = 6;
myStruct valType2 = (myStruct)refType;
Console.WriteLine("valType2.val={0}", valType2.val);
```

ستطبع هذه الشيفرة السطر التالي:

valType2.val=5

لكن عند إسناد نوع مرجعي إلى كائن فإننا نحصل على نتيجة مختلفة يمكننا تمثيل ذلك بتعديل قيمة myStruct ضمن الصنف بتجاهل حقيقة أن اسم هذا الصنف لم يعد مناسبا بعد الآن:

```
class  myStruct
{
    public int val;
}
```

والآن وبدون أي تعديل آخر على الشيفرة السابقة وتنفيذها سنلاحظ الخرج التالي:

valType2.val=6

يمكننا تغليف أنواع القيمة ضمن واجهة وذلك طالما أنها مزودة بهذه الواجهة على سبيل المثال لنفترض أن البنية myStruct مزودة بالواجهة IMyInterface كما يلي:

```
interface IMyInterface
{
}
struct myStruct:IMyInterface
{
    public int val;
}
```

يمكننا عندئذ تغليف البنية ضمن النوع IMyInterface كما يلي:

```
myStruct valType1 = new myStruct();
IMyInterface refType = valType1;
```

ويمكننا إزالة التغليف أيضا باستخدام صيغة التشكيل العادية:

```
myStruct valType2 = (myStruct)refType;
```

وكما ترى من هذه الأمثلة إن عملية التغليف لا تتطلب شيفرات خاصة لتطبيقها أما عملية إزالة التغليف لقيمة فهي تتطلب تحويلا صريحا لها والذي يتطلب استخدام التشكيل.

ملاحظة:

يمكننا أن نعتبر التغليف عملية تحويل مطلقة أما إزالة التغليف فهي عملية تحويل صريحة.

يمكن ان تتساءل عن سبب استخدامنا للتغليف أساسا في الحقيقة هناك سببان يجعلان استخدام التغليف أمرا مهما ومفيدا جدا الأول أنه يسمح لنا باستخدام أنواع القيمة ضمن المجموعات مثل النوع ArrayList حيث لا يتم تخزين إلا العناصر من النوع object أو الأنواع المشتقة منه أي جميع الكائنات والسبب الثاني متمثل في الآلية التي تسمح لنا باستدعاء مناهج الكائن على أنواع القيمة وكملاحظ أخيرة من الجدير التنويه إلى أن إزالة تغليف القيمة أمر ضروري قبل الوصول إلى محتوياتها.

العامل is:

The is Operator:

يسمح العامل is بتفحص فيما إذا كان متحول مجهول ربما متحول ممرر ككائن على شكل بارامتر إلى منهج ما يمكن تحويله إلى نوع محدد فإذا كانت نتيجة هذا العامل هي true فهذا يعني أن عملية التحويل إلى هذا النوع ممكنة يمكننا استخدام هذا العامل قبل استدعاء المناهج على الكائنات وذلك لتفحص ما إذا كان الكائن من نوع يدعمه المنهج.

توضيح:

لا يتفحص العامل is ما إذا كان النوعان المتشابهان أم لا وإنما يتفحص فيما إذا كان النوعان متوافقين مع بعضهما البعض.

يأخذ العامل is الصيغة التالية:

<operand> is <type>

والنتيجة المتوقعة لهذا العامل هي كالتالي:

- إذا كان <type> يمثل نوع صنف فإن النتيجة هي true إذا كان <operand> من هذا النوع أو أنه يرث من هذا النوع أو إن كان بالإمكان تغليف نوعه ضمن النوع <type>.
- إذا كان النوع <type> يمثل نوع واجهة فإن النتيجة هي true إذا كان <operand> من هذا النوع أو إذا كان نوعه مزودا بهذه الواجهة.
- إذا كان النوع <type> يمثل نوع قيمة فإن النتيجة هي true إذا كان <operand> من هذا النوع أو إذا كان من النوع الذي يمكننا إزالة تغليفه إلى نوع القيمة <type>.

سوف نتناول ذلك في المثال التالي لنرى كيفية استخدام هذا العامل بشكل عملي:

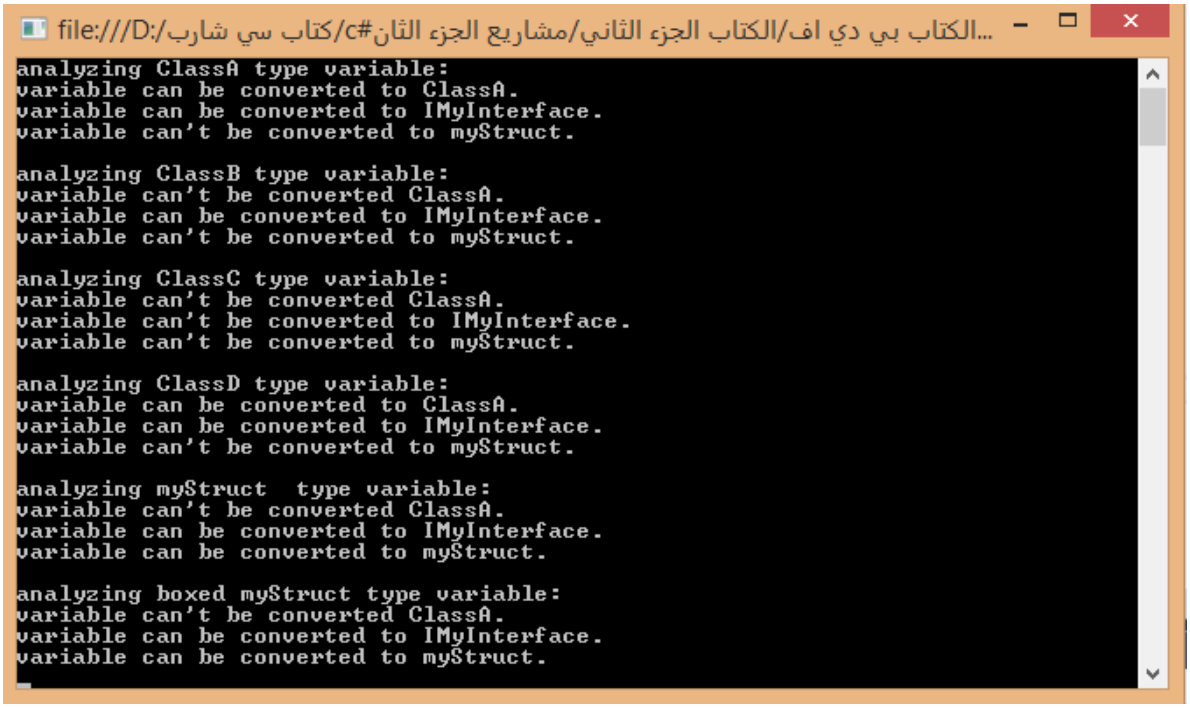
تطبيق حول استخدام العامل is:

1- قم بإنشاء تطبيق Console جديد باسم ConsoleIS.
2- عدل شيفرة الملف Program.cs كما يلي:

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

namespace ConsoleIS
{
    class Checker
    {
        public void Check(object param1)
        {
            if (param1 is ClassA)
                Console.WriteLine("variable can be converted to" +
                    " ClassA.");
            else
                Console.WriteLine("variable can't be converted" +
                    " ClassA.");
            if (param1 is IMyInterface)
                Console.WriteLine("variable can be converted to" +
                    " IMyInterface.");
            else
                Console.WriteLine("variable can't be converted to" +
                    " IMyInterface.");
            if (param1 is myStruct)
                Console.WriteLine("variable can be converted to" +
                    " myStruct.");
            else
                Console.WriteLine("variable can't be converted to" +
```

```
                    " myStruct.");

        }
    }
    interface IMyInterface
    {
    }
    class ClassA:IMyInterface
    {
    }
    class ClassB:IMyInterface
    {
    }
    class ClassC
    {
    }
    class ClassD:ClassA
    {
    }
    struct myStruct:IMyInterface
    {
    }
    class Program
    {
        static void Main(string[] args)
        {
            Checker check = new Checker();
            ClassA try1 = new ClassA();
            ClassB try2 = new ClassB();
            ClassC try3 = new ClassC();
            ClassD try4 = new ClassD();
            myStruct try5 = new myStruct();
            object try6 = try5;
            Console.WriteLine("analyzing ClassA type variable:");
            check.Check(try1);
            Console.WriteLine("\nanalyzing ClassB type variable:");
            check.Check(try2);
            Console.WriteLine("\nanalyzing ClassC type variable:");
            check.Check(try3);
            Console.WriteLine("\nanalyzing ClassD type variable:");
            check.Check(try4);
            Console.WriteLine("\nanalyzing myStruct type variable:");
            check.Check(try5);
            Console.WriteLine("\nanalyzing boxed myStruct type variable:");
            check.Check(try6);
            Console.ReadLine();
        }
    }
}
```

3- نفذ الشيفرة بالضغط على زر F5.



الشكل (6-4)

كيفية العمل:

How to Work:

يستعرض هذا المثال النتائج العديدة الممكنة عند استخدام العامل is لقد تم تعريف ثلاثة أصناف وواجهة وبنية وقد تم استخدامها كبارامترات في منهج يستخدم العامل is لرؤية ما إذا كنا نستطيع تحويلها إلى النوع ClassA أو الواجهة أو البنية.

إن النوعين ClassA وClassD اللذين يرثان من الصنف ClassA هما النوعان الوحيدان مع النوع ClassA فالأنواع التي لا ترث من هذا الصنف ليست متوافقة مع هذا الصنف.

أما الأنواع ClassA وClassB وmyStruct فجميعها مزود بالواجهة IMyInterface لذا فإن جميعها متوافق مع النوع IMyInterface النوع ClassD يرث من النوع ClassA وبالتالي فهو متوافق مع هذه الواجهة أيضا أما النوع ClassC فهو غير متوافق مع الواجهة IMyInterface.

وأخيرا فإن المتحولات من النوع myStruct وجميع المتحولات المغلفة من هذا النوع متوافقة مع البنية MyStruct وذلك باعتبار أنه لا يمكننا تحويل أنواع المرجع إلى أنواع قيمة إلا أنواع المرجع الناتجة من تغليف أنواع القيمة.

العامل as:

The AS Operator:

يقوم العامل as بتحويل النوع إلى نوع مرجع محدد وذلك باستخدام الصيغة التالية:

<operand> as <type>

وإليك الحالات الممكنة لهذا العامل:

- إذا كان <operand> من النوع <type>.
- إذا أمكن تحويل <operand> بشكل مطلق على النوع <type>.
- إذا أمكن تغليف <operand> إلى النوع <type>.

أما إن أمكن استخدام التحويل الصريح بين <operand> و<type> فإن النتيجة هي null على سبيل المثال بالعودة إلى المثال الأخير هناك تحويل صريح ممكن بين النوع ClassA والنوع ClassD الذي يرث من الصنف ClassA وبالتالي فإنه وفقا للشيفرة التالية:

```
 ClassA obj1 = new ClassA();
ClassD obj2 = obj1 as ClassD;
```

سيأخذ المتحول obj2 القيمة null.

أما إن تضمن متحول من النوع ClassA كائنا من النوع ClassD فإن العامل سيعمل وفقا للشيفرة:

```
ClassD obj1 = new ClassD();
ClassA  obj2 = obj1;
ClassD obj3 = obj2 as ClassD;
```

فهنا سيمثل المتحول obj3 مرجعا على الكائن نفسه الذي يشير إليه obj1 وليس القيمة null إن هذا يجعل استخدام العامل as مفيدا جدا باعتبار أنه سينتج عن الشيفرة التالية رمي لاعتراض:

```
ClassA obj1 = new ClassA();
ClassD obj2 = (ClassD)obj1;
```

بينما الشيفرة الموافقة التي تستخدم العامل as ستعيد القيمة null للمتحول obj2 إن هذا يعني أن شيفرة كالسابقة شائعة جدا في تطبيقات C# وكمثال لنعد إلى الصنفين اللذين أنشأناها مسبقا في هذا الفصل وهما الصنف Animal والصنف Cow الذي يرث من الصنف Animal:

```
public void MilkCow(Animal myAnimal)
{
    Cow myCow = myAnimal as Cow;
    if (myCow !=null)
    {
        myCow.Milk();
    }
    else
```

```
        {
            Console.WriteLine("{0} is n't a cow,and so can't be milked",
                myAnimal.Name());
        }
    }
```

إن هذا أبسط بكثير من التعامل مع الاعتراضات الناتجة عن عدم تحقق عملية التحويل الصريحة بين النوع Animal والنوع Cow.

النسخ العميق:

Deep Copying:

لقد رأينا في الفصل الثاني من هذا الجزء كيفية القيام بالنسخ السطحي shallow copying باستخدام المنهج المحمي System.Object.MemberwiseClone() وباستخدام منهج كالتالي:

```
public class cloner
{
    public int val;
    public cloner (int newVal)
    {
        val = newVal;
    }
    public object GetCopy()
    {
        return MemberwiseClone();
    }
}
```

لنفترض ان لدينا حقولا من أنواع مرجعية بدلا من أنواع قيمة (كالكائنات مثلا):

```
public class Content
{
    public int val;
}
public class Cloner
{
    public Content MyContent = new Content();
    public Cloner (int newVal)
    {
        MyContent.val = newVal;
    }
    public object GetCopy()
    {
        return MemberwiseClone();
    }
}
```

وفي حالة كهذه سيعيد المنهج GetCopy() حقلا يشير إلى نفس الكائن الذي يشير له الكائن الأصلي والشيفرة التالية تستعرض ذلك باستخدام هذا الصنف:

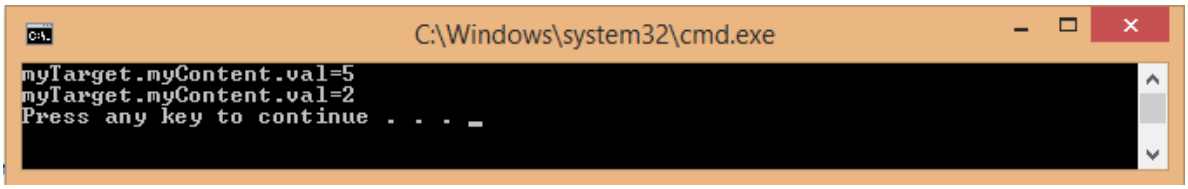
```
Cloner mySource = new Cloner(5);
Cloner myTarget = (Cloner)mySource.GetCopy();
Console.WriteLine("myTarget.myContent.val={0}",
    myTarget.MyContent.val);
mySource.MyContent.val = 2;
Console.WriteLine("myTarget.myContent.val={0}",
    myTarget.MyContent.val);
```

لقد قمنا هنا بإنشاء متحول جديد mySource من النوع Cloner وذلك باستخدام منهج البناء غير الافتراضي لقد قمنا هنا بوضع قيمة للحقل MyContent.val لهذا المتحول وهي 5 وفي السطر الثاني قمنا بإنشاء متحول آخر myTarget من نوع Cloner باستخدام المنهج GetCopy() على المتحول mySource والذي يقوم بإعادة نسخة سطحية للكائن mySource ويقوم السطر الثالث بطباعة قيمة الحقل MyContent.val لنسخة المتحول الجديدة.

ملاحظة:

لاحظ أننا استخدمنا التشكيل في السطر الثاني فكما تلاحظ ضمن شيفرة الصنف Cloner فإن المنهج GetCopy() يعيد قيمة من النوع Object.

أما في السطر الرابع فقد قمنا بإسناد قيمة جديدة للحقل mySource.MyContent.val وسينتج عن ذلك تغير لقيمة الحقل نفسه في المتحول myTarget وذلك لأن كلا المتحولين يشيران إلى الكائن MyContent نفسه والخرج سيصبح بالصورة التالية:



الشكل (7-4)

إذن كي نتمكن من إنشاء نسخة منفصلة تماما عن النسخة الأصلية علينا أن نلجأ لاستخدام النسخ العميق وللقيام بذلك لن نحتاج سوى لتعديل المنهج GetCopy() المستخدم في الشيفرة السابقة ولكن من المفضل استخدام طريقة إطار عمل .NET القياسية للقيام بذلك وتتلخص هذه الطريقة بالتزود بالواجهة ICloneable والتي تحتوي على منهج واحد فقط هو Clone() لا بأخذ هذا المنهج أية بارامترات ويعيد قيمة من نوع object تماما كالمنهج GetCopy().

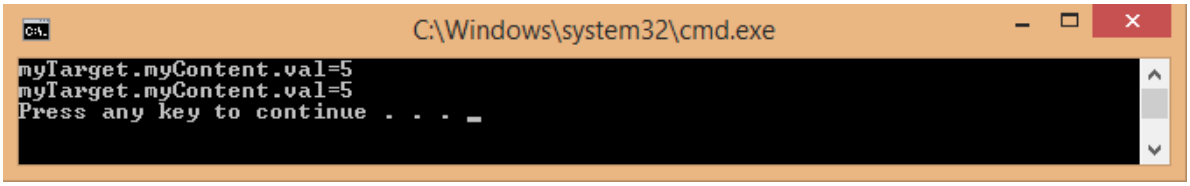
والآن لكي نتمكن من استخدام النسخ العميق مع الشيفرة السابقة سنعدلها كما يلي:

```
public class Content
{
    public int val;
}
public class Cloner:ICloneable
{
```

```
        public Content MyContent = new Content();
        public Cloner (int newVal)
        {
            MyContent.val = newVal;
        }
        public object Clone()
        {
            Cloner clonedCloner = new Cloner(MyContent.val);
            return clonedCloner;
        }
    }
```

لقد قمنا هنا بإنشاء كائن Cloner جديد باستخدام قيمة الحقل MyContent.val المتضمنة في كائن Cloner الأصلي إن هذا الحقل حقل قيمة وليس كائن وبالتالي لا حاجة لاستخدام نسخ أعمق من ذلك.

والآن إذا استخدمنا شيفرة مشابهة للشيفرة السابقة ولكن باستبدال استدعاء المنهج GetCopy() بالمنهج Clone() سيعطينا ذلك النتيجة التالية:



الشكل (4-8)

والآن نلاحظ ان نسخة الكائن الجديدة مستقلة تماما عن الكائن الأصلي.

لاحظ أن شيفرة النسخ العميق تختلف من صنف لآخر وذلك تبعا لأنواع المرجع التي يحتويها كل صنف وفي حالة كهذه نحن بحاجة إلى استدعاء تعاودي (عميق) للمنهج Clone() خصوصا ضمن أنظمة الكائنات الأكثر تعقيدا على سبيل المثال إذا تطلب الحقل MyContent للصنف Cloner نسخا عميقا أيضا فإن علينا القيام بما يلي:

```
public class Content
{
    public int val;
}
public class Cloner:ICloneable
{
    public Content MyContent = new Content();
    public Cloner ()
    {

    }
    public Cloner (int newVal)
    {
        MyContent.val = newVal;
    }
    public object Clone()
    {
```

```
            Cloner clonedCloner = new Cloner();
            clonedCloner.MyContent = MyContent;
            return clonedCloner;
        }
    }
```

التحديث الثالث لمكتبة أصناف أوراق اللعب:

Upgrading CardLibrary part3:

سنقوم الآن بوضع ما تعلمناه حول النسخ في إطار التنفيذ العملي وذلك لإتاحة النسخ العميق لكائنات الأصناف Card وCards وDeck وذلك باستخدام الواجهة ICloneable ويمكن ان يكون ذلك مفيدا في بعض ألعاب الورق.

إن تطبيق ميزة النسخ العميق للصنف Card سهل جدا باعتبار ان جميع الحقول ضمن الصنف هي حقول قيمة وبالتالي يمكننا استخدام النسخ السطحي فقط سنعدل شيفرة الصنف كما يلي:

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

namespace CardLibrary1
{
    public enum Suit
    {
        Clup,
        Diamond,
        Heart,
        Spade
    }
    public enum Rank
    {
        Ace = 1,
        Deuce,
        Three,
        Four,
        Five,
        Six,
        Seven,
        Eight,
        Nine,
        Ten,
        Jack,
        Queen,
        King
    }
    public class Card:ICloneable
    {
```

```
public object Clone()
{
    return MemberwiseClone();
}
public static bool useTrumps = false;
public static Suit trump = Suit.Clup;
public static bool isAceHigh = true;
public readonly Suit suit;
public readonly Rank rank;
public override string ToString()
{
    //return base.ToString();
    return "The "+rank +" of "+suit+"s";
}
private Card ()
{
}
public Card (Suit newSuit,Rank newRank)
{
    suit = newSuit;
    rank = newRank;
}
public static bool operator==(Card card1,Card card2)
{
    return (card1.suit == card2.suit) && (card1.rank == card2.rank);
}
public static bool operator !=(Card card1,Card card2)
{
    return !(card1 == card2);
}
public override bool Equals(object card)
{
    //return base.Equals(obj);
    return this == (Card)card;
}
public override int GetHashCode()
{
    //return base.GetHashCode();
    return 12 * (int)rank + (int)suit;
}
public static bool operator >(Card card1,Card card2)
{
    if (card1 .suit ==card2 .suit )
    {
        if (isAceHigh)
            return (card1.rank > card2.rank) ^ (card2.rank == Rank.Ace);
        else
            return (card1.rank > card2.rank);
    }
    else
    {
        if (useTrumps && (card2.suit == Card.trump))
            return false;
        else
```

```
                return true;
            }
        }
        public static bool operator < (Card card1,Card card2)
        {
            return !(card1 >= card2);
        }
        public static bool operator >=(Card card1,Card card2)
        {
            if (card1 .suit ==card2 .suit )
            {
                if (isAceHigh)
                    return (card1.rank >= card2.rank) ^ (card2.rank == Rank.Ace);
                else
                    return (card1.rank >= card2.rank);
            }
            else
            {
                if (useTrumps && (card2.suit == Card.trump))
                    return false;
                else
                    return true;
            }
        }
        public static bool operator <=(Card card1,Card card2)
        {
            return !(card1 >card2 );
        }
    }
}
```

لاحظ أن التزود بالواجهة ICloneable هنا يمثل نسخا سطحيا فليست هناك أية قواعد لتحديد ما يجب أن يحدث ضمن المنهج Clone() وهذا كاف هنا.

بعد ذلك سنقوم بتزويد صنف المجموعة Cards بالواجهة ICloneable أيضا. إن النسخ العميق هنا معقد بعض الشيء باعتبار أن علينا نسخ كل كائن Card في المجموعة الأصلية إلى المجموعة الناتجة عن النسخ وبالتالي فإننا سنستخدم النسخ العميق بالصورة التالية:

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

namespace CardLibrary1
{
    public class Cards:CollectionBase,ICloneable
    {
        public object Clone()
        {
            Cards newCards = new Cards();
```

```
            foreach (Card sourceCard in List )
            {
                newCards.Add1((Card)sourceCard.Clone());
            }
            return newCards;
        }
        public void Add1(Card newCard)
        {
            List.Add(newCard);
        }
        public void Remove1(Card oldCard)
        {
            List.Remove(oldCard);
        }
        public Cards ()
        {
        }
        public Card this[int cardIndex]
        {
            get
            {
                return (Card)List[cardIndex];
            }
            set
            {
                List[cardIndex] = value;
            }
        }
        public bool Contains1(Card card)
        {
            return InnerList.Contains(card);
        }
    }
}
```

وأخيرا علينا أن نزود الصنف Deck بالواجهة ICloneable أيضا هناك مشكلة صغيرة هنا فالصنف Deck لا يملك أية وسيلة لتعديل أوراق اللعب التي يحتويها فيما عدا أنه يمكنه إعادة ترتيب أوراق اللعب فقط فليست هناك طريقة لتعديل حالة مجموعة أوراق اللعب Deck لكي تأخذ أوراقها ترتيبا معينا مثلا لتجاوز هذه المشكلة سنقوم بتعريف منهج بناء خاص للصنف Deck يسمح بتمرير مجموعة Cards محددة إلى كائن Deck عند إنشائه وبالتالي الشيفرة التي سيتم إضافتها إلى الصنف Deck هي كما يلي:

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

namespace CardLibrary1
{
    public class Deck:ICloneable
    {
        public object Clone()
```

```
{
    Deck newDeck = new Deck((Cards)cards.Clone());
    return newDeck;
}
private Deck (Cards newCards)
{
    cards = newCards;
}
private Cards cards = new Cards();
public Deck ()
{
   // Cards = new Card[52];
    for (int suitVal=0;suitVal <4;suitVal ++)
    {
        for (int rankVal=1;rankVal <14;rankVal ++)
        {
            cards.Add1(new Card((Suit)suitVal, (Rank)rankVal));
        }
    }
}
public Deck (bool isAceHigh):this()
{
    Card.isAceHigh = isAceHigh;
}
public Deck (bool useTrumps,Suit trump):this()
{
    Card.useTrumps = useTrumps;
    Card.trump = trump;
}
public Deck (bool isAceHigh,bool useTrumps,Suit trump):this()
{
    Card.isAceHigh = isAceHigh;
    Card.useTrumps = useTrumps;
    Card.trump = trump;
}
public Card GetCard(int cardNum)
{
    if (cardNum >= 0 && cardNum <= 51)
        return cards [cardNum];
    else
        throw (new System.ArgumentOutOfRangeException("cardNum",
           cardNum, "Value must be between 0 and 51."));
}
public void Shuffle()
{
    //Card[] newDeck = new Card[52];
    Cards newDeck = new Cards();
    bool[] assigned = new bool[52];
    for (int i=0;i<52;i++)
    {
        int sourceCard = 0;
        bool foundCard = false;
        Random sourceGen = new Random();
        while (foundCard ==false )
```

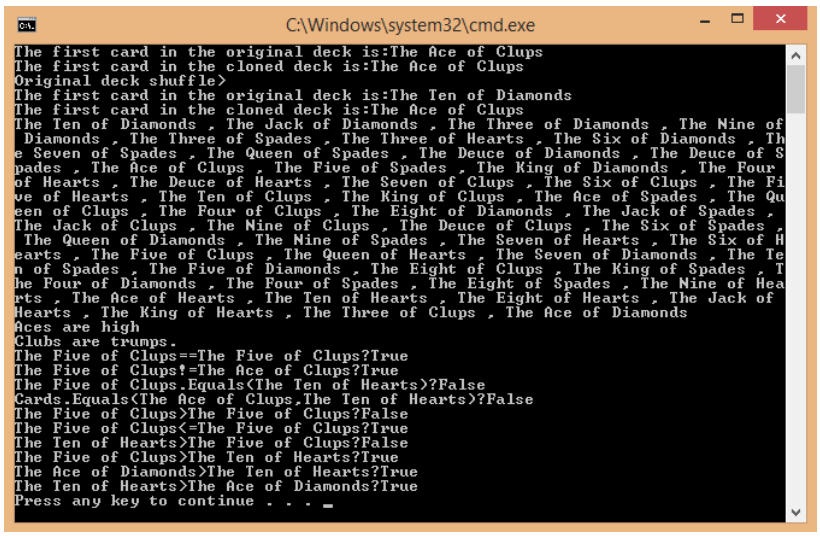
```
                {
                    sourceCard = sourceGen.Next(52);
                    if (assigned[sourceCard] == false)
                        foundCard = true;
                }
                assigned [sourceCard ]=true ;
                newDeck.Add1(cards[sourceCard]);
            }
            cards  = newDeck;
        }
    }
}
```

ومجددا يمكننا تجربة عملية النسخ باستخدام تطبيق زبون بسيط كما في السابق حيث سنضع الشيفرة التالية ضمن المنهج Main() لمشروع تطبيق الزبون:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using CardLibrary1;

namespace ConsoleCardClient
{
    class Program
    {
        static void Main(string[] args)
        {
            Deck deck1 = new Deck();
            Deck deck2 = (Deck)deck1.Clone();
            Console.WriteLine("The first card in the original deck is:{0}",
                deck1.GetCard(0));
            Console.WriteLine("The first card in the cloned deck is:{0}",
                deck2.GetCard(0));
            deck1.Shuffle();
            Console.WriteLine("Original deck shuffle>");
            Console.WriteLine("The first card in the original deck is:{0}",
                deck1.GetCard(0));
            Console.WriteLine("The first card in the cloned deck is:{0}",
                deck2.GetCard(0));
            Deck myDeck = new Deck();
            myDeck.Shuffle();
            for (int i=0;i<52;i++)
            {
                Card tempCard = myDeck.GetCard(i);
                Console.Write(tempCard.ToString());
                if (i != 51)
                    Console.Write(" , ");
                else
                    Console.WriteLine();
            }
            Card.isAceHigh = true;
            Console.WriteLine("Aces are high");
```

```
            Card.useTrumps = true;
            Card.trump = Suit.Clup;
            Console.WriteLine("Clubs are trumps.");
            Card card1, card2, card3, card4, card5;
            card1 = new Card(Suit.Clup, Rank.Five);
            card2 = new Card(Suit.Clup, Rank.Five);
            card3 = new Card(Suit.Clup, Rank.Ace);
            card4 = new Card(Suit.Heart, Rank.Ten);
            card5 = new Card(Suit.Diamond, Rank.Ace);
            Console.WriteLine("{0}=={1}?{2}",
                card1.ToString(), card2.ToString(), card1 == card2);
            Console.WriteLine("{0}!={1}?{2}",
                card1.ToString(), card3.ToString(), card1 != card3);
            Console.WriteLine("{0}.Equals({1})?{2}",
                card1.ToString(), card4.ToString(), card1.Equals(card4));
            Console.WriteLine("Cards.Equals({0},{1})?{2}",
                card3.ToString(), card4.ToString(), Card.Equals(card3, card4));
            Console.WriteLine("{0}>{1}?{2}",
                card1.ToString(), card2.ToString(), card1 > card2);
            Console.WriteLine("{0}<={1}?{2}",
                card1.ToString(), card2.ToString(), card1 <= card2);
            Console.WriteLine("{0}>{1}?{2}",
                card4.ToString(), card1.ToString(), card4 > card1);
            Console.WriteLine("{0}>{1}?{2}",
                card1.ToString(), card4.ToString(), card1 > card4);
            Console.WriteLine("{0}>{1}?{2}",
                card5.ToString(), card4.ToString(), card5 > card4);
            Console.WriteLine("{0}>{1}?{2}",
                card4.ToString(), card5.ToString(), card4 > card5);
        }
    }
}
```

والخرج سيصبح كما يلي:



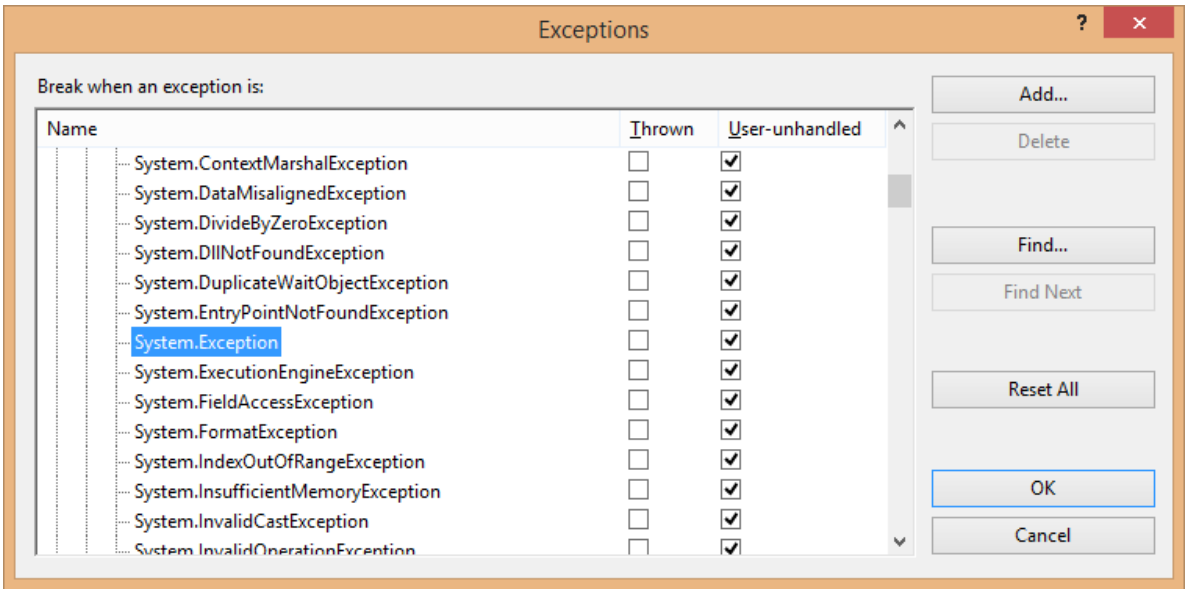
الشكل (9-4)

الاعتراضات المخصصة:

Custom Exceptions:

لقد رأينا في الفصل السابع من الجزء الأول من هذا الكتاب كيفية التعامل مع الاعتراضات وتعلمنا كيفية استخدام الكتل try..catch..finally ولقد رأينا في هذا الكتاب عددا من اعتراضات .NET. وتتضمن ذلك الصنف الأساس للاعتراضات System.Exception لاحظ أنه من المفيد أحيانا إنشاء أصناف اعتراضاتك المخصصة من هذا الصنف الأساس واستخدامها ضمن تطبيقك بدلا من استخدام الاعتراضات القياسية إن هذا يسمح بتخصيص أكبر للمعلومات الناتجة عن الاعتراضات عند اصطيادها فعلى سبيل المثال يمكننا إضافة خاصية لصنف الاعتراض بحيث يحجب الوصول إلى بعض المعلومات التحتية للاعتراضات مما قد يتيح للاعتراض مستقبلا القيام بالتعديلات اللازمة أو مجرد عرض معلومات الاعتراض الحاصل.

ومتى عرفنا صنف الاعتراض يمكننا إضافته إلى لائحة الأصناف المصنفة من قبل Visual Studio 2013 باستخدام الأمر Exception من القائمة Debug يمكنك من خلال صندوق الحوار هذا التحكم في كيفية استجابة Visual Studio 2013 عند رمي الاعتراضات غير المعالجة يسمح صندوق الحوار هذا بتمكين Visual Studio 2013 من التوقف عن تنفيذ التطبيق وفتح نافذة التنقيح عند حصول الاعتراض أو الاستمرار بالتنفيذ بداية عند رمي اعتراض من هذا النوع ومن ثم إذا لم يكن الاعتراض معالجا.



الشكل (4-10)

التحديث الرابع لمكتبة أصناف أوراق اللعب:

Upgrading CardLibrary part4:

يمكننا الاستفادة من الاعتراضات المخصصة ضمن مكتبة أصناف أوراق اللعب لذا سنقوم بتحديث المشروع CardLibrary نلاحظ أن المنهج Deck.GetCard() يقوم برمي اعتراض .NET. قياسي وسنقوم بتعديل ذلك لاستخدام اعتراض مخصص.

أولا نحن بحاجة إلى تعريف الاعتراض لذا سنعرف صنفا جديدا ضمن الملف Exceptions.cs:

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

namespace CardLibrary1
{
    public class CardOutOfRangeException:Exception
    {
        private Cards deckContents;
        public Cards DeckContents
        {
            get
            {
                return deckContents;
            }
        }
        public CardOutOfRangeException(Cards sourceDeckContents):
        base ("There are only 52 cards in the deck.")
        {
            deckContents = sourceDeckContents;
        }
    }
    class Exceptions
    {
    }
}
```

يجب ان يتم إنشاء حالة من هذا الصنف ضمن الصنف Cards إن ذلك يسمح بالوصول إلى كائن Cards من خلال الخاصية DeckCotents ومن ثم توفير رسالة الخطأ المناسبة لمنهج بناء الصنف Exception الأساس وبالتالي يصبح متوفرا من خلال الخاصية Message للصنف.

بعد ذلك سنقوم بإضافة الشيفرة التي تقوم برمي هذا الاعتراض في الملف Deck.cs أي استبدال الاعتراض التقليدي السابق.

```
        public Card GetCard(int cardNum)
        {
```

```
            if (cardNum >= 0 && cardNum <= 51)
                return cards[cardNum];
            else
                throw new CardOutOfRangeException((Cards)cards.Clone());
        }
```

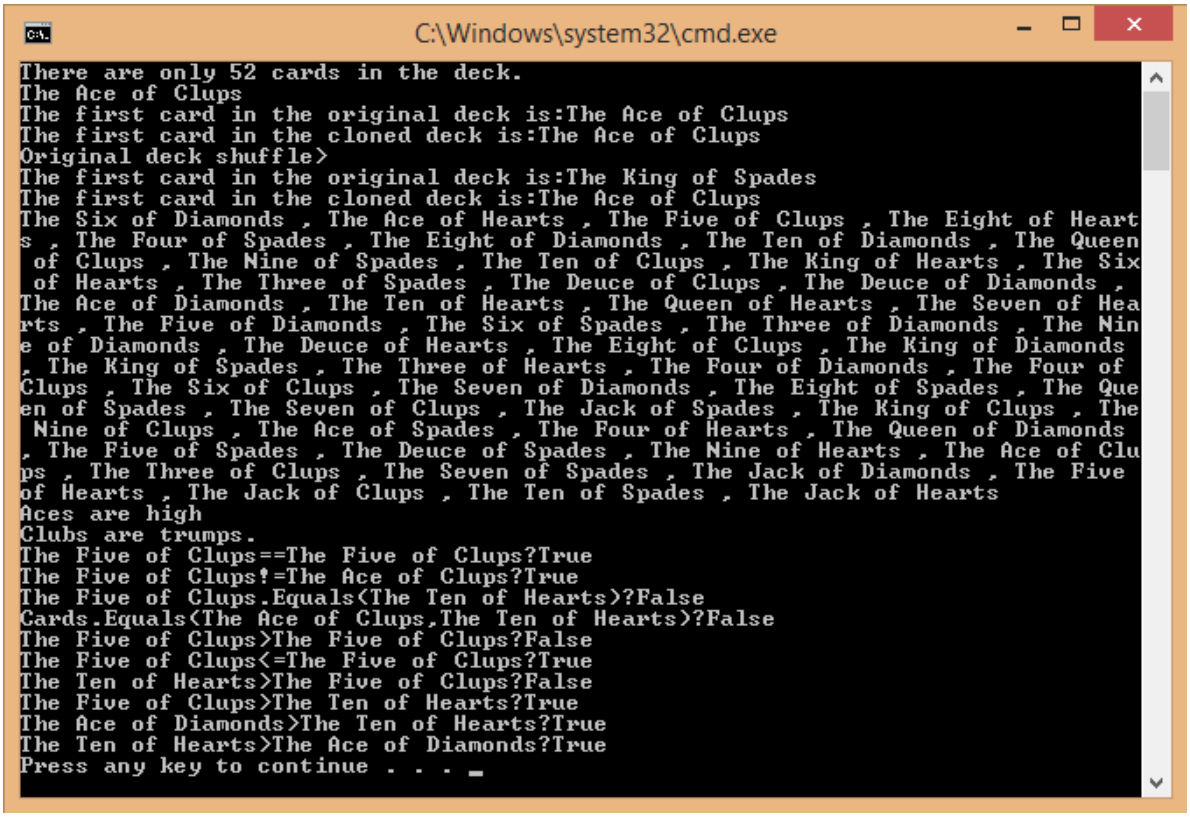
سيتم تهيئة الخاصية DeckContents عند النسخ العميق للمحتويات الحالية لكائن Deck وذلك بهيئة كائن من نوع Cards إن هذا يعني أنه يمكننا ان نرى المحتويات في النقطة التي حصل عندها الخطأ لذا فإن التعديلات اللاحقة بمحتوى مجموعة أوراق اللعب لن تضيع هذه المعلومات.

لتفحص ذلك يمكننا استخدام شيفرة تطبيق الزبون التالية:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using CardLibrary1;

namespace ConsoleCardClient
{
    class Program
    {
        static void Main(string[] args)
        {
            Deck deck1 = new Deck();
            try
            {
                Card myCard = deck1.GetCard(60);
            }
            catch (CardOutOfRangeException e)
            {
                Console.WriteLine(e.Message);
                Console.WriteLine(e.DeckContents[0]);
            }
            Deck deck2 = (Deck)deck1.Clone();
            Console.WriteLine("The first card in the original deck is:{0}",
                deck1.GetCard(0));
            Console.WriteLine("The first card in the cloned deck is:{0}",
                deck2.GetCard(0));
            deck1.Shuffle();
            Console.WriteLine("Original deck shuffle>");
            Console.WriteLine("The first card in the original deck is:{0}",
                deck1.GetCard(0));
            Console.WriteLine("The first card in the cloned deck is:{0}",
                deck2.GetCard(0));
            Deck myDeck = new Deck();
            myDeck.Shuffle();
            for (int i=0;i<52;i++)
            {
                Card tempCard = myDeck.GetCard(i);
                Console.Write(tempCard.ToString());
                if (i != 51)
```

```
                Console.Write(" , ");
            else
                Console.WriteLine();
        }
        Card.isAceHigh = true;
        Console.WriteLine("Aces are high");
        Card.useTrumps = true;
        Card.trump = Suit.Clup;
        Console.WriteLine("Clubs are trumps.");
        Card card1, card2, card3, card4, card5;
        card1 = new Card(Suit.Clup, Rank.Five);
        card2 = new Card(Suit.Clup, Rank.Five);
        card3 = new Card(Suit.Clup, Rank.Ace);
        card4 = new Card(Suit.Heart, Rank.Ten);
        card5 = new Card(Suit.Diamond, Rank.Ace);
        Console.WriteLine("{0}=={1}?{2}",
            card1.ToString(), card2.ToString(), card1 == card2);
        Console.WriteLine("{0}!={1}?{2}",
            card1.ToString(), card3.ToString(), card1 != card3);
        Console.WriteLine("{0}.Equals({1})?{2}",
            card1.ToString(), card4.ToString(), card1.Equals(card4));
        Console.WriteLine("Cards.Equals({0},{1})?{2}",
            card3.ToString(), card4.ToString(), Card.Equals(card3, card4));
        Console.WriteLine("{0}>{1}?{2}",
            card1.ToString(), card2.ToString(), card1 > card2);
        Console.WriteLine("{0}<={1}?{2}",
            card1.ToString(), card2.ToString(), card1 <= card2);
        Console.WriteLine("{0}>{1}?{2}",
            card4.ToString(), card1.ToString(), card4 > card1);
        Console.WriteLine("{0}>{1}?{2}",
            card1.ToString(), card4.ToString(), card1 > card4);
        Console.WriteLine("{0}>{1}?{2}",
            card5.ToString(), card4.ToString(), card5 > card4);
        Console.WriteLine("{0}>{1}?{2}",
            card4.ToString(), card5.ToString(), card4 > card5);
    }
  }
}
```

والنتيجة بالشكل التالي:


C:\Windows\system32\cmd.exe
There are only 52 cards in the deck.
The Ace of Clups
The first card in the original deck is:The Ace of Clups
The first card in the cloned deck is:The Ace of Clups
Original deck shuffle>
The first card in the original deck is:The King of Spades
The first card in the cloned deck is:The Ace of Clups
The Six of Diamonds , The Ace of Hearts , The Five of Clups , The Eight of Hearts , The Four of Spades , The Eight of Diamonds , The Ten of Diamonds , The Queen of Clups , The Nine of Spades , The Ten of Clups , The King of Hearts , The Six of Hearts , The Three of Spades , The Deuce of Clups , The Deuce of Diamonds , The Ace of Diamonds , The Ten of Hearts , The Queen of Hearts , The Seven of Hearts , The Five of Diamonds , The Six of Spades , The Three of Diamonds , The Nine of Diamonds , The Deuce of Hearts , The Eight of Clups , The King of Diamonds , The King of Spades , The Three of Hearts , The Four of Diamonds , The Four of Clups , The Six of Clups , The Seven of Diamonds , The Eight of Spades , The Queen of Spades , The Seven of Clups , The Jack of Spades , The King of Clups , The Nine of Clups , The Ace of Spades , The Four of Hearts , The Queen of Diamonds , The Five of Spades , The Deuce of Spades , The Nine of Hearts , The Ace of Clups , The Three of Clups , The Seven of Spades , The Jack of Diamonds , The Five of Hearts , The Jack of Clups , The Ten of Spades , The Jack of Hearts
Aces are high
Clubs are trumps.
The Five of Clups==The Five of Clups?True
The Five of Clups!=The Ace of Clups?True
The Five of Clups.Equals(The Ten of Hearts)?False
Cards.Equals(The Ace of Clups,The Ten of Hearts)?False
The Five of Clups>The Five of Clups?False
The Five of Clups<=The Five of Clups?True
The Ten of Hearts>The Five of Clups?False
The Five of Clups>The Ten of Hearts?True
The Ace of Diamonds>The Ten of Hearts?True
The Ten of Hearts>The Ace of Diamonds?True
Press any key to continue . . .

الشكل (4-11)

الخلاصة:

Summary:

لقد تناولنا في هذا الفصل العديد من التقنيات التي يمكننا استخدامها لصقل تطبيقاتنا المكتوبة بلغة C# والتي تتعلق بأساليب البرمجة كائنية التوجه OOP وعلى الرغم من أن التقنيات المشروحة تحتاج لبعض المجهود لتطبيقها إلا انها ستسهل التعامل مع أصنافك والذي يؤدي إلى تبسيط مهمة كتابة شيفرة التطبيقات ككل.

لكل موضوع في هذا الفصل استخدامات عديدة ستجد لاحقا أنك بحاجة في الغالب لاستخدام أحد أشكال المجموعات في أي تطبيق هذا بالإضافة إلى إنشاء مجموعات مخصصة تجعل عملية التعامل مع عناصرها وأدارتها أسهل وأقوى كما تعرفنا أيضا على المفهرسات للوصول المباشر لعناصر المجموعة.

وتحدثنا أيضا عن التحميل الزائد للعوامل ورأينا أنه يسمح لنا بتعريف عوامل C# لاستخدامها مع أنواع مختلفة عن الأنواع التي تتعامل معها بصورة افتراضية وتحدثنا عن النسخ العميق والذي يمثل أساسا مهما لتجنب أحد أكثر الأخطاء شيوعا في البرمجة كائنية التوجه وأخيرا تعرفنا على إنشاء كائنات الاعتراضات المخصصة وتمرير معلومات مفصلة لمعالج الاعتراض.

الفصل الخامس

الأحداث

هذا هو الفصل الأخير في قسم البرمجة كائنية التوجه لهذا الكتاب وبه نكون قد أكملنا حديثنا عن OOP وذلك بتناولنا لأحد أهم التقنيات الشائعة الاستخدام في OOP.NET ألا وهي الأحداث Events.

سنبدأ كالمعتاد بالسؤال ماهي الأحداث وبعد ذلك سنرى بعض الأحداث البسيطة وكيفية التعامل معها ومتى انتهينا من ذلك سننتقل إلى مواضيع أكثر تقدما حيث سنتعلم كيف ننشئ أحداثنا الخاصة.

وفي القسم الثاني من هذا الفصل سنقوم بتشطيب مكتبة أصناف اللعب CardLibrary بإضافة حدث إليها وإضافة إلى ذلك وباعتبار أن هذا الفصل هو المحطة الأخيرة التي سننتقل بعدها إلى مواضيع أكثر تقدما فإننا سنقوم بإنشاء لعبة ورق بسيطة تستخدم مكتبة الأصناف CardLibrary.

ما هو الحدث؟

What is an Event?

تتشابه الأحداث مع الاعتراضات من ناحية عملها فلاعتراضات ترمي thrown من الكائنات ويمكن أن تحتوي على شيفرة برمجية معينة تنفذ عند رمي الاعتراض أي حدوثه الأحداث تماما كالاعتراضات من ناحية آلية العمل فهي ترمى (لا ترمى الأحداث في الواقع وإنما ترفع أو تقدح) وعند حدوث الحدث فإن شيفرة معينة يتم تنفيذها ولكن هناك عدد من الاختلافات الهامة بينهما والاخلاف الأكثر أهمية هو أنه ليست للأحداث بنية الكتل try..catch لكي تتم معالجتها وبدلا من ذلك فإن علينا أن نسجل او نقر (subscribe) الأحداث وتسجيل الأحداث يعني تزويدها بشيفرة معينة يتم تنفيذها عند رفع الحدث وهو ما يسمى بمعالج الحدث event handler.

يمكن ان يكون للحدث الواحد العديد من المعالجات المسجلة له والتي سيتم استدعاؤها متى رفع الحدث يمكن أن يتضمن ذلك معالجات أحداث كجزء من صنف لكائن برفع الحدث إلا أنه من النادر أن نجد معالجات أحداث في أصناف أخرى غير الصنف الذي رفع عنده الحدث.

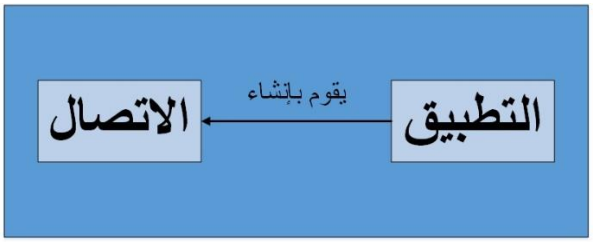
والحقيقة البسيطة التي يجب أن تعلمها أن معالجات الأحداث ليست أكثر من توابع أو مناهج عادية والتقييد الوحيد لتابع معالج الحدث هو أنه يجب أن يطابق التوقيع اللازم للحدث (يشتمل ذلك على القيمة المعادة والبارامترات) يمثل هذا التوقيع جزءا من تعريف الحدث ويتم تحديده من خلال المفوضات delegates.

ملاحظة:

تكمن الاستفادة من المفوضات delegates في استخدامها مع الأحداث ذاك هو السبب الذي اضطرني للتنويه عنها في الفصل السادس من الجزء الأول لهذا الكتاب لذا فأنا أتمنى أن تعيد قراءة هذا القسم لإنعاش ذاكرتك من المفوضات وكيفية الاستفادة منها.

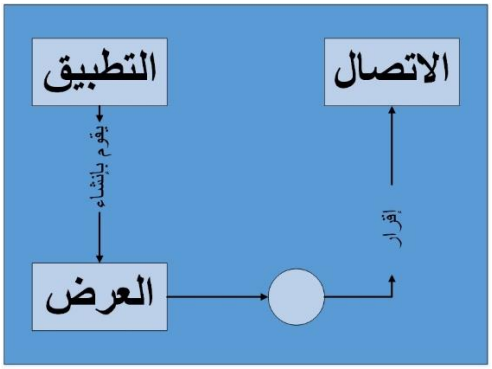
والآن إليك آلية عمل الأحداث:

أولا سيقوم التطبيق بإنشاء كائن يستطيع أن يرفع حدثا ما وكمثال لنفترض أن تطبيقنا يمثل تطبيق مراسلة وأن الكائن الذي يقوم بإنشائه يمثل كائن الاتصال مع المستخدم البعيد يمكن لكائن الاتصال هذا أن يرفع حدثا وذلك عندما تصل رسالة من المستخدم البعيد من خلال هذا الاتصال.



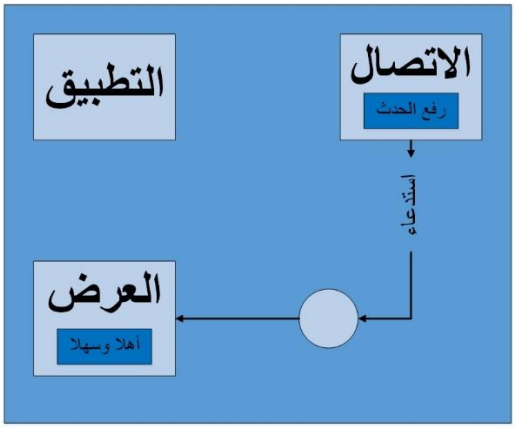
الشكل (5-1)

بعد ذلك سيقوم التطبيق بتسجيل الحدث سيقوم تطبيق المراسلة بذلك من خلال تعريف تابع يمكن استخدامه مع المفوض المحدد من قبل الحدث ويقوم بتمرير مرجع لهذا التابع إلى الحدث يمكن أن يكون تابع معالج الحدث هذا منهجا لأي كائن ولنفترض أن كائنا ما يمثل شاشة العرض التي ستعرض الرسائل عند وصولها.



الشكل (5-2)

عند رفع الحدث سيلحظ المسجل ذلك وعند وصول الرسالة عبر كائن الاتصال سيتم استدعاء منهج معالج الحدث الموجود في كائن العرض ويمكن للكائن الذي رفع الحدث أن يمرر أية معلومات من خلال البارامترات وذلك لجعل الأحداث متعددة الاستعمالات عندئذ يمكن استخدام هذه المعلومات التي أصبحت تمثل بارامترات لمعالج الحدث لعرضها من خلال كائن العرض.



الشكل (3-5)

توضيح:

ليست الأحداث بالمنطق الذي يصعب فهمه هناك عدد غير متناهي من الأمثلة التي يمكننا وصف الأحداث بها لنأخذ على سبيل المثال واجهة القيادة في السيارة نلاحظ أنها تحتوي على عدد من المؤشرات مثل مؤشر السرعة ومؤشر تسرب الزيت فعند تسرب الزيت سيتم رفع حدث تسريب الزيت الذي سيؤدي إلى إشعال منبه تسريب الزيت في واجهة القيادة ويمكن أن يمثل هذا المنبه بصوت متقطع بإضاءة LED مثلا إن تسريب الزيت هو الحدث وعند رفع هذا الحدث سيتم تنفيذ معالج هذا الحدث ومعالج الحدث هنا هو الدارة الالكترونية التي ستقوم بإضاءة الـ LED أو إصدار الصوت المنبه.

استخدام الأحداث:

Using Events:

سنتعلم في هذا القسم كيفية كتابة الأحداث ومعالجتها بعد ذلك سننتقل للحديث عن كيفية تعريف وإنشاء أحداثنا المخصصة.

معالجة الاحداث:

Handling Events:

لكي نعالج حدثا ما يجب أن نسجل الحدث وذلك بتوفير تابع معالجة الحدث الذي يتطابق توقيعه مع توقيع مفوض محدد لاستخدامه مع الحدث لنلق نظرة على مثال يستخدم كائن المؤقت البسيط لرفع الأحداث والذي سيؤدي إلى استدعاء معالج الحدث.

بعبارة أخرى لكي نستطيع أن نستخدم الحدث يجب أن نربطه بمعالج حدث له توقيع موافق لتوقيع مفوض الحدث والصيغة العامة لعملية الربط هي كما يلي:

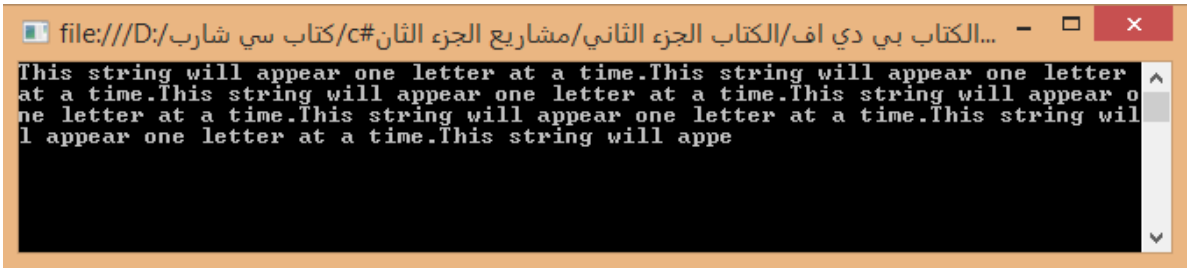
eventName += new delegateName(eventHandlerFunction)

تطبيق حول معالجة الأحداث:

1- قم بإنشاء تطبيق Console جديد باسم ConsoleHandlingEvents.
2- عدل الشيفرة في الملف Program.cs كما يلي:

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

namespace ConsoleHandlingEvents
{
    class Program
    {
        static int counter = 0;
        static string displayString =
            "This string will appear one letter at a time.";
        static void writeChar(object source, ElapsedEventArgs e)
        {
            Console.Write(displayString[counter++ %
                displayString.Length]);
        }
        static void Main(string[] args)
        {
           Timer myTimer=new Timer (100);
            myTimer .Elapsed +=new ElapsedEventHandler (writeChar );
            myTimer .Start ();
            Console .ReadLine ();
        }
    }
}
```

3- نفذ التطبيق بالضغط على زر F5 ولاحظ ماذا يكتب على الشاشة للخروج من التطبيق اضغط Enter.



الشكل (4-5)

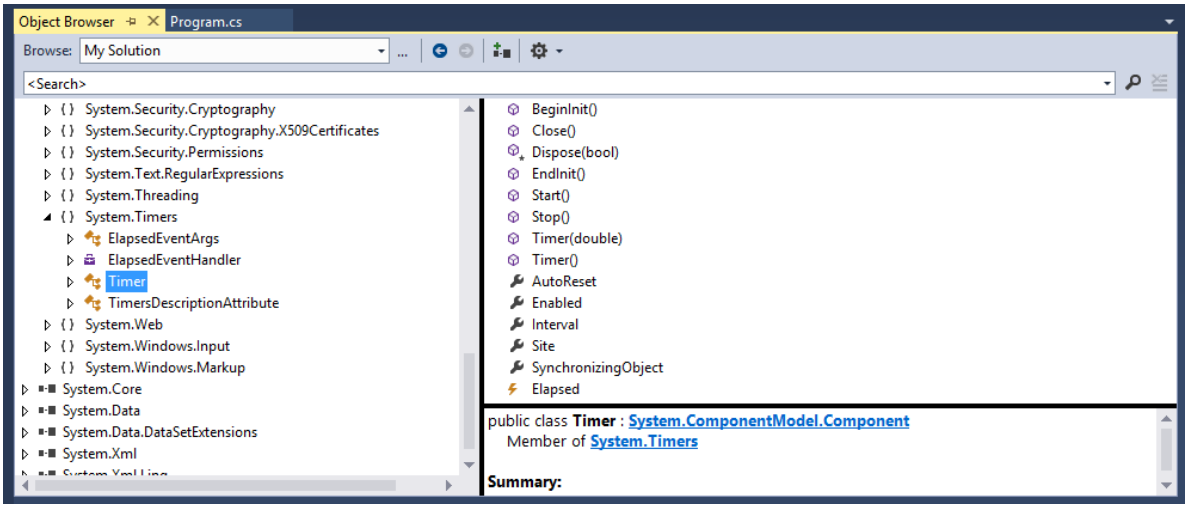
كيفية العمل:

How To Work:

إن الكائن المستخدم لرفع الأحداث يمثل كائنا من الصنف System.Timers.Time يتم تهيئة هذا الكائن بفترة زمنية محددة بالميلي ثانية وعند بدء عمل كائن Timer باستخدام المنهج Start سيتم رفع سيل من الأحداث التي تفصل عن بعضها البعض بالفترة الزمنية المحددة لقد تم تهيئة كائن المؤقت في التابع Main() بالقيمة 100 وهذا يعني أنه سيتم رفع الأحداث عشر مرات في الثانية متى استدعينا المنهج Start():

```
static void Main(string[] args)
{
   Timer myTimer=new Timer (100);
```

يمتلك كائن المؤقت Timer حدثا باسم Elapsed ويمكنك أن تجد هذا الحدث بالإضافة إلى حدث آخر اسمه Tick ضمن نافذة مستعرض الكائنات Object Browser:



الشكل (5-5)

أما توقيع معالج الحدث لهذا الحدث فهو من نوع المفوض System.Timers.ElapsedEventHandler والذي يمثل واحدا من المفوضات القياسية المعرفة في إطار عمل .NET. يستخدم هذا المفوض مع التوابع التي تطابق التوقيع التالي:

Void functionName(object source,ElapsedEventArgs e);

يقوم كائن المؤقت Timer بإرسال مرجع لنفسه في البارامتر الأول وكائنا من الصنف ElapsedEventArgs في بارامتره الثاني من الأفضل تجنب الحديث عن هذين البارامترين الآن وسنتحدث عنهما لاحقا.

ونلاحظ في شيفرتنا أن هناك منهجا يتطابق مع هذا التوقيع:

```
static void writeChar(object source, ElapsedEventArgs e)
{
    Console.Write(displayString[counter++ %
        displayString.Length]);
}
```

يستخدم هذا المنهج حقلين ستاتيكيين للصنف program هما counter وdisplayString وذلك لعرض رمز وحيد على الشاشة وفي كل مرة يستدعى هذا المنهج فيها سيتم طباعة رمز مختلف.

المهمة التالية هي ربط هذا المنهج مع الحدث وللقيام بذلك فإننا سنستخدم العامل =+ لإضافة معالج هذا الحدث على هيئة حالة لمفوض جديد تمت تهيئته بواسطة منهج معالج الحدث:

```
static void Main(string[] args)
{
   Timer myTimer=new Timer (100);
    myTimer .Elapsed +=new ElapsedEventHandler (writeChar );
```

يقوم هذا الأمر الذي يستخدم صيغة غريبة نوعا ما خاصة بالمفوضات بإضافة معالج حدث إلى لائحة التوابع التي سيتم استدعائها عند رفع الحدث Elapsed يمكننا إضافة أي عدد من المعالجات إلى هذه اللائحة طالما أن هذه المعالجات تتوافق مع المعايير المطلوبة للحدث أي تطابق توقيع الحدث حيث سيتم استدعاء هذه المعالجات كلا بدوره عند رفع الحدث.

إن ما تبقى من شيفرة الأمر Main() هو منهج البدء بتشغيل المؤقت:

```
    myTimer .Start ();
```

وبما اننا لا نود من التطبيق أن يتوقف قبل معالجة أية أحداث فإننا سنبقى التطبيق منفذا والطريقة الأبسط لتحقق ذلك في تطبيقات Console تقتضي بطلب إدخال من المستخدم باعتبار أن عملية المعالجة لن تنتهي قبل أن يدخل المستخدم أي شيء أو أن يضغط على مفتاح enter:

```
    Console .ReadLine ();
}
```

وعلى الرغم من أن المعالجة في التابع Main() متقطعة إلا أن المعالجة ضمن كائن المؤقت Timer مستمرة فعندما يرفع هذا الكائن الأحداث سيتم استدعاء منهجنا WriteChar() والذي سينفذ بتزامن مع تعليمة Conaole.ReadLine().

تعريف الأحداث:

Defining Events:

والآن سنتعرف على كيفية إنشاء وتعريف الأحداث وبالتالي استخدام أحداثنا الخاصة. سوف نحاول تطبيق ذلك على مثال برنامج المراسلة الذي تحدثنا عنه في بداية هذا الفصل وسنقوم بإنشاء كائن Connection يقوم برفع الأحداث المعالجة عبر كائن Display.

تطبيق حول تعريف الأحداث:

1- قم بإنشاء تطبيق Console جديد باسم ConsoleDefiningEvents.
2- أضف صنفا جديدا باسم Connection واحفظه ضمن الملف Connection.cs:

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

namespace ConsoleDefiningEvents
{
    public delegate void MessageHandler(string messageText);
    public class Connection
    {
        public event MessageHandler MessageArrived;
        private Timer pollTimer;
        public Connection ()
        {
            pollTimer = new Timer(100);
            pollTimer .Elapsed +=new
            ElapsedEventHandler (CheckForMessage );
        }
        public void Connect()
        {
            pollTimer.Start();
        }
        public void Disconnect()
        {
            pollTimer.Stop();
        }
        private void CheckForMessage(object  source,
            ElapsedEventArgs e)
        {
            Console.WriteLine("Checking for new messages.");
            Random random = new Random();
            if ((random .Next (9)==0)&&(MessageArrived !=null))
            {
                MessageArrived ("Hello Mum!");
```

```
            }
        }
    }
}
```

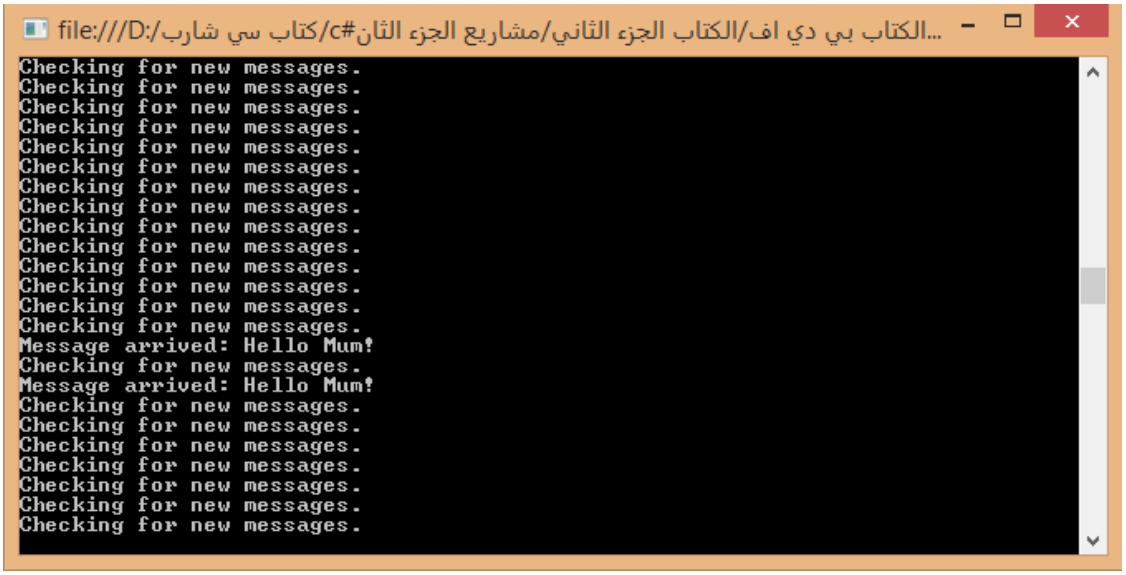
3- أضف صنفا جديدا باسم Display واحفظه ضمن الملف Display.cs:

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

namespace ConsoleDefiningEvents
{
    public class Display
    {
        public void DisplayMessage(string message)
        {
            Console.WriteLine("Message arrived: {0}", message);
        }
    }
}
```

4- عدل الشيفرة في الصنف Program.cs كما يلي:

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

namespace ConsoleDefiningEvents
{
    class Program
    {
        static void Main(string[] args)
        {
            Connection myConnection = new Connection();
            Display myDisplay = new Display();
            myConnection.MessageArrived +=
                new MessageHandler(myDisplay.DisplayMessage);
            myConnection.Connect();
            Console.ReadLine();
        }
    }
}
```

5- نفذ التطبيق بالضغط على زر F5.



الشكل (6-5)

كيفية العمل:

How to Work:

يقوم الصنف Connection بجل العمل في هذا التطبيق فالكائنات من هذا الصنف تستخدم كائن المؤقت Timer بصورة مشابهة لاستخدامه في التطبيق الأول حيث تتم تهيئة كائن Timer في منهج البناء للصنف ويأخذ حالة تمكين أو عدم التمكين من خلال المنهجين Connect() وDisconnect:

```
public Connection ()
{
    pollTimer = new Timer(100);
    pollTimer .Elapsed +=new
    ElapsedEventHandler (CheckForMessage );
}
public void Connect()
{
    pollTimer.Start();
}
public void Disconnect()
{
    pollTimer.Stop();
}
```

لاحظ أيضا أننا قمنا بتسجيل معالج للحدث Elapsed ضمن منهج البناء تماما كما قمنا بذلك في التطبيق السابق أما منهج الحدث CheckMessage() فسوف يرفع بمعدل عشر مرات في الثانية ولكن قبل أن نتناول شيفرة هذا المنهج سنلقي نظرة على تعريف الحدث.

قبل أن نعرف حدثا علينا أن نعرف نوع مفوض لاستخدامه مع الحدث أي أن نوع المفوض هو الذي يحدد توقيع معالج الحدث الذي يطابق الحدث ولكي نعرف مفوضا فإننا سنستخدم صيغة تعريف المفوض وذلك بتعريفه كمفوض عام داخل فضاء الأسماء ConsoleDefiningEvents وذلك لتمكين الوصول لهذا النوع من خلال الشيفرة الموجودة خارج الصنف.

```
namespace ConsoleDefiningEvents
{
    public delegate void MessageHandler(string messageText);
```

لقد أطلقنا على مفوض الحدث هنا الاسم MessageHandler وهو مفوض من نوع Void أي لا يعيد قيمة وله بارامتر واحد من نوع String سنستخدم هذا البارامتر لتمرير الرسالة المستلمة من كائن Connection إلى كائن Display.

ومتى تم تعريف المفوض أو كان هناك مفوض مناسب موجود مسبقا فإنه يمكننا عندئذ تعريف الحدث وذلك كعضو في الصنف Connection:

```
    public class Connection
    {
        public event MessageHandler MessageArrived;
```

وكما تلاحظ فإن صيغة تعريف الحدث بسيطة فقد قمنا بتسجيل الحدث وفقا للمفوض MessageHandler وباسم MessageArrived والكلمة المفتاحية event تحدد أن التعريف هو تعريف الحدث.

ومتى صرحنا عن الحدث بهذه الطريقة فإنه يمكننا رفع هذا الحدث بمجرد استدعاءه باسمه كما لو كنا نستدعي منهجا عاديا ووفقا للتوقيع المحدد بواسطة المفوض على سبيل المثال يمكننا رفع الحدث كما يلي:

```
MessageArrived ("This is a message");
```

وإذا كان المفوض المعرف لهذا الحدث لا يحوي أية بارامترات فإننا سنرفع الحدث بمجرد كتابة:

```
MessageArrived ();
```

وبصورة بديلة فإنه يمكننا تعريف أكثر من بارامتر ضمن مفوض الحدث والذي سيتطلب منا تحديد هذه البارامترات لرفع الحدث.

الجزء الذي يرفع الحدث MessageArrived في هذا التطبيق هو المنهج CheckForMessage() وشيفرته كما يلي:

```
private void CheckForMessage(object  source,
    ElapsedEventArgs e)
{
    Console.WriteLine("Checking for new messages.");
    Random random = new Random();
    if ((random .Next (9)==0)&&(MessageArrived !=null))
    {
        MessageArrived ("Hello Mum!");
    }
```

```
    }
```

لقد استخدمنا هنا كائنا من نوع Random لتوليد رقم عشوائي بين 0 و9 ومن ثم سيتم رفع الحدث عندما يكون الرقم هو 0 والذي يمكن أن يحدث بنسبة 10% إن هذا يحاكي اقتراع الاتصال لرؤية قيما إذا وصلت رسالة ام لا.

لاحظ اننا استخدمنا منطقا إضافيا هنا فنحن سنرفع الحدث إذا كانت قيمة التعبير MessageArrived =!null ومعنى هذا التعبير "هل للحدث أية مسجلات؟" فإن لم يكن هناك أية مفوضات مسجلة لهذا الحدث فإن MessageArrived سيعيد القيمة null وبالتالي لن تكون هناك أية إشارة لرفع الحدث.

يسمى الصنف الذي سيستجيب للحدث بالصنف Display ويتضمن منهجا واحدا DisplayMessage() معرفا كما يلي:

```
public class Display
{
    public void DisplayMessage(string message)
    {
        Console.WriteLine("Message arrived: {0}", message);
    }
}
```

يطابق توقيع هذا المنهج توقيع المفوض في MessageHandler وبالتالي يمكننا استخدام هذا المنهج كاستجابة للحدث MessageArrived.

كل ما تبقى لدينا الآن هو شيفرة المنهج Main() والذي يقوم بتهيئة حالة من الصنف Connection والصنف Display ومن ثم يقوم بربط الحدث MessageArrived للصنف Connection مع المنهج DisplayMessage للصنف وفقا للمفوض MessageHandler ومن ثم يقوم باستدعاء المنهج Connect() لبدء عمل المؤقت.

```
    static void Main(string[] args)
    {
        Connection myConnection = new Connection();
        Display myDisplay = new Display();
        myConnection.MessageArrived +=
            new MessageHandler(myDisplay.DisplayMessage);
        myConnection.Connect();
        Console.ReadLine();
    }
```

ومجددا فقد قمنا باستدعاء المنهج Console.ReadLine() وذلك لإيقاف تنفيذ المنهج Main() مؤقتا كي نتمكن من رفع الاحداث قبل إنهاء التطبيق.

معالجات الأحداث متعدد الأغراض:

Multi-Purpose Event Handlers:

يتضمن توقيع الحدث Timer.Elapsed بارامترين وهما من النوع الذي سنجده غالبا في معالجات الأحداث:

- البارامتر source من النوع object وهو مرجع للكائن الذي رفع الحدث.
- البارامتر e من النوع ElapsedEventArgs ويتضمن البارامترات المرسلة من قبل الحدث.

إن سبب وجود بارامتر من النوع object هو حاجتنا في أغلب الأحيان إلى استخدام معالج حدث وحيد لأكثر من حدث بحيث يرفع هذا الحدث من أكثر من كائن وبالتالي ستنفذ شيفرة محددة ضمن معالج الحدث وفقا للكائن الذي رفع الحدث.

لشرح ذلك بطريقة أفضل سنقوم بتطوير التطبيق السابق.

تطبيق حول استخدام معالج الحدث متعدد الأغراض:

1- قم بإنشاء تطبيق Console جديد باسم ConsoleMulti-PurposeEventHandlers.
2- انسخ شيفرة الأصناف Display.cs وConnection.cs وProgram.cs من المشروع ConsoleDefiningEvents إلى المشروع الحالي ولا تنسى أن تعدل فضاء الأسماء إلى ConsoleMulti-PurposeEventHandlers.
3- أضف صنفا جديدا باسم MessageArrivedEventArgs.

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

namespace ConsoleMulti_PurposeEventHandlers
{
    public  class MessageArrivedEventArgs:EventArgs
    {
        private string message;
        public string Message
        {
            get
            {
                return message;
            }
        }
        public MessageArrivedEventArgs ()
        {
            message = "No message sent.";
        }
        public MessageArrivedEventArgs (string newMessage)
        {
            message = newMessage;
        }
    }
}
```

4- عدل شيفرة الملف Connection.cs كما يلي:

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

namespace ConsoleMulti_PurposeEventHandlers
{
    public delegate void MessageHandler(Connection source,
    MessageArrivedEventArgs e);
    public class Connection
    {
        public event MessageHandler MessageArrived;
        private string name;
        public string Name
        {
            get
            {
                return name;
            }
            set
            {
                name = value;
            }
        }
        private Timer pollTimer;
        public Connection()
        {
            pollTimer = new Timer(100);
            pollTimer.Elapsed += new
            ElapsedEventHandler(CheckForMessage);
        }
        public void Connect()
        {
            pollTimer.Start();
        }
        public void Disconnect()
        {
            pollTimer.Stop();
        }
        private void CheckForMessage(object source,
            ElapsedEventArgs e)
        {
            Console.WriteLine("Checking for new messages.");
            Random random = new Random();
            if ((random.Next(9) == 0) && (MessageArrived != null))
            {
                MessageArrived(this ,
                    new MessageArrivedEventArgs ("Hello Mum!."));
            }
        }
```

```
    }
}
```

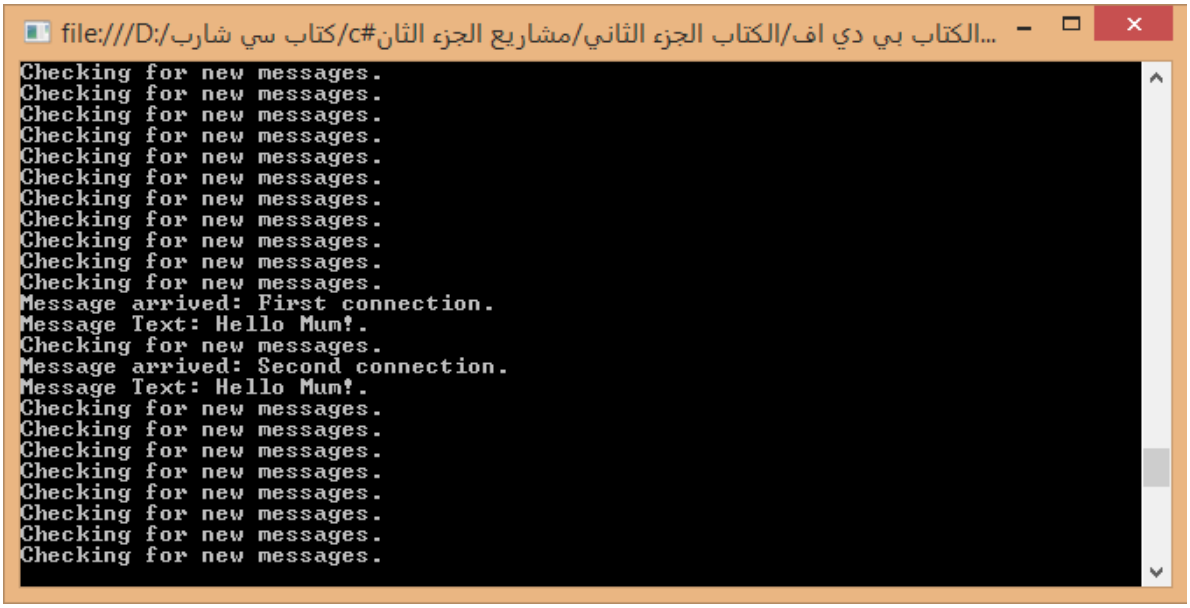
5- عدل الملف Display.cs كما يلي:

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

namespace ConsoleMulti_PurposeEventHandlers
{
    public class Display
    {
        public void DisplayMessage(Connection source,
            MessageArrivedEventArgs e)
        {
            Console.WriteLine("Message arrived: {0}", source .Name );
            Console.WriteLine("Message Text: {0}", e.Message);
        }
    }
}
```

6- عدل شيفرة الملف Program.cs كما يلي:

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

namespace ConsoleMulti_PurposeEventHandlers
{
    class Program
    {
        static void Main(string[] args)
        {
            Connection myConnection1 = new Connection();
            myConnection1.Name = "First connection.";
            Connection myConnection2 = new Connection();
            myConnection2.Name = "Second connection.";
            Display myDisplay = new Display();
            myConnection1.MessageArrived +=
                new MessageHandler(myDisplay.DisplayMessage);
            myConnection2.MessageArrived +=
                new MessageHandler(myDisplay.DisplayMessage);
            myConnection1.Connect();
            myConnection2.Connect();
            Console.ReadLine();
        }
    }
}
```

7- نفذ التطبيق:



الشكل (6-6)

كيفية العمل:

How to Work:

يمكننا تخصيص استجابة معالج الحدث تبعا للكائنات التي رفعت الحدث وذلك بإرسال مرجع لهذا الكائن الواحد من بارامترات معالج الحدث فهذا المرجع يمكننا من الوصول إلى الكائن الأصلي بما يتضمن من خصائص ومناهج.

وبإرسال البارامترات الموجودة في الصنف الذي يرث من الصنف System.EventArgs كما يقوم الصنف ElapsedEventArgs فإنه يمكننا تزويد معالج الحدث بأية معلومات إضافية ضرورية على هيئة بارامترات مثل البارامتر Message ضمن الصنف MessageArrivedEventArgs.

وبالإضافة إلى ذلك فإن هذه البارامترات ستستفيد من تعددية الأشكال حيث يمكننا تعريف معالج حدث للحدث MessageArrived كما يلي مثلا:

```
public void DisplayMessage(object source,EventArgs e)
{
    Console.WriteLine("Message arrived: {0}",
        ((Connection )source ).Name  );
    Console.WriteLine("Message Text: {0}",
        ((MessageArrivedEventArgs )e).Message );
}
```

وتعديل تعريف المفوض في الملف Connection.cs كما يلي:

```
public delegate void MessageHandler(object source,EventArgs e);
```

سينفذ التطبيق تماما كالسابق ولكن بهذه الطريقة أصبح المنهج DisplayMessage() أكثر مرونة فمعالج الحدث هذا يمكن أن يستخدم لأحداث أخرى مثل الحدث Timer.Elapsed مثلا سنحتاج عندها إلى تعديل شيفرة المنهج وبارامتراته لكي نتمكن من استقبال الحدين.

القيم المعادة من معالجات الأحداث:

Return Values and Event Handlers:

إن جميع معالجات الأحداث التي رأيناها إلى الآن لا تعيد قيما أي أنها من النوع Void من الممكن أن تعيد الأحداث قيما معينة لحدث ما إلا أن هذا قد يقود إلى مشاكل ويعود سبب ذلك إلى أن الحدث الواحد يمكن أن يؤدي إلى استدعاء عدة معالجات أحداث فإن أعادت جميع معالجات الأحداث قيمة ما فإننا لن نتمكن من تحديد القيمة الحقيقية التي أعيدت من معالج حدث معين.

يتعامل النظام مع هذه المشكلة بالسماح لنا بوصول إلى القيمة الأخيرة المعادة من معالج الحدث وتلك هي القيمة المعادة من آخر معالج حدث مسجل للحدث.

قد تحتاج لسبب ما إلى إعادة قيمة من معالج الحدث إلا أن هذا نادر جدا لذا فإنني أنصحك باستخدام معالجات الأحداث التي لا تعيد أية قيمة أي التي تستخدم الكلمة void وهذا يشمل أيضا تجنب استخدام بارامترات الخرج out مع معالجات الأحداث.

توسعة واستخدام مكتبة أصناف أوراق اللعب:

Expanding and Using CardLibrary:

والآن وبعد ان تعرفنا على كيفية تعريف واستخدام الأحداث سنقوم بإضافة أحداث مكتبة أصناف أوراق اللعب التي طورناها في الفصل السابق والتي كانت باسم CardLibrary وكما في السابق سوف نقوم بإنشاء مشروع جديد باسم CardLibrary2 وتذكر تغيير فضاء الأسماء من CardLibrary إلى CardLibrary2 سنقوم بإضافة واحد إلى مكتبة الأصناف وسيتم رفع هذا الحدث عند الحصول على كائن Card الأخير في كائن Deck باستخدام المنهج GetCard() وسيسمى هذا الحدث بـ LastCardDrawn بمعنى أوضح سيتم رفع هذا الحدث عندما نسحب أو نلعب الورقة الأخيرة في مجموعة أوراق اللعب سيسمح هذا الحدث للمسجلات بإعادة خلط مجموعة أوراق اللعب بشكل تلقائي أما المفوض المستخدم مع هذا الحدث وهو باسم LastCardDrawnHandler فيجب أن يزود بمرجع لكائن Deck بحيث يمكن الوصول إلى المنهج Shuffle() من أي معالج حدث وذلك كما يلي:

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Text;
using System.Threading.Tasks;

namespace CardLibrary2
{
    public delegate void LastCardDrawnHandler(Deck currentDeck);
    public class Deck:ICloneable
    {
        public object Clone()
        {
            Deck newDeck = new Deck((Cards)cards.Clone());
            return newDeck;
        }
        private Deck (Cards newCards)
        {
            cards = newCards;
        }
        private Cards cards = new Cards();
        public Deck ()
        {
           // Cards = new Card[52];
            for (int suitVal=0;suitVal <4;suitVal ++)
            {
                for (int rankVal=1;rankVal <14;rankVal ++)
                {
                    cards.Add1(new Card((Suit)suitVal, (Rank)rankVal));
                }
            }
        }
        public Deck (bool isAceHigh):this()
        {
            Card.isAceHigh = isAceHigh;
        }
        public Deck (bool useTrumps,Suit trump):this()
        {
            Card.useTrumps = useTrumps;
            Card.trump = trump;
        }
        public Deck (bool isAceHigh,bool useTrumps,Suit trump):this()
        {
            Card.isAceHigh = isAceHigh;
            Card.useTrumps = useTrumps;
            Card.trump = trump;
        }
        public event LastCardDrawnHandler LastCardDrawn;
        public Card GetCard(int cardNum)
        {
            if (cardNum >= 0 && cardNum <= 51)
            {
                if ((cardNum ==51)&&(LastCardDrawn !=null ))
                    LastCardDrawn(this );
                return cards[cardNum];
            }
```

```
            else
                throw new CardOutOfRangeException((Cards)cards.Clone());
        }
        public void Shuffle()
        {
            //Card[] newDeck = new Card[52];
            Cards newDeck = new Cards();
            bool[] assigned = new bool[52];
            for (int i=0;i<52;i++)
            {
                int sourceCard = 0;
                bool foundCard = false;
                Random sourceGen = new Random();
                while (foundCard ==false )
                {
                    sourceCard = sourceGen.Next(52);
                    if (assigned[sourceCard] == false)
                        foundCard = true;
                }
                assigned [sourceCard ]=true ;
                newDeck.Add1(cards[sourceCard]);
            }
            cards  = newDeck;
        }
    }
}
```

تلك هي الشيفرة الضرورية لإضافة حدث للصنف Deck والآن سنرى كيف سنستخدم هذا الحدث.

تطبيق زبون للعبة ورق اللعب باستخدام مكتبة الأصناف CardLibrary2:

A Card Game Clint for CardLibrary2:

بعد أن استغرقنا كل هذا الوقت في إنشاء مكتبة الأصناف CardLibrary2 سيكون من المخجل عدم استخدامها وقبل أن ننهي هذا القسم المتعلق بالبرمجة كائنية التوجه في C# وإطار عمل .NET. علينا إن نلهو قليلا بكتابة أساسيات تطبيق لعبة ورق تستخدم أصناف أوراق اللعب التي أصبحت مألوفة بالنسبة لنا.

وكما في الفصول السابقة فإننا سنضيف تطبيق Console زبون إلى الحل CardLibrary2 ومن ثم سنضيف إلى هذا التطبيق مرجعا للمشروع CardLibrary2 وسنجعل هذا التطبيق مشروع نقطة البدء سيسمى هذا التطبيق بـ ConsoleCardClient2 كبداية سنقوم بإنشاء صنف جديد باسم Player في ملف جديد باسم Player.cs وذلك ضمن المشروع CardLibrary2 سيحتوي هذا الصنف على حقل خاص من نوع Cards باسم hand وحقل سلسلة نصية خاص باسم name وخاصيتان للقراءة فقط الخاصية PlayHand ستمثل هاتين الخاصيتين واجهة للحقلين name وhand.

سنقوم بإخفاء منهج البناء الافتراضي وذلك باستخدام محدد الوصول Private وسنقوم بإنشاء منهج بناء غير افتراضي يتقبل قيمة أولية للخاصية Name في كائنات Player.

وأخيرا سنقوم بتوفير منهج يعيد قيمة منطقية باسم HasWon() سيعيد هذا المنهج القيمة True إذا كانت جميع الأوراق في يد اللاعب من الفئة نفسها وهو شرط للفوز بسيط إلا أننا لن نخوض في ألعاب أكثر تعقيدا.

إن شيفرة الملف Player.cs هي كما يلي:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using CardLibrary2;

namespace CardLibrary2
{
    public class Player
    {
        private Cards hand;
        private string name;
        public string Name
        {
            get
            {
                return name;
            }
        }
        public Cards PlayHand
        {
            get
            {
                return hand;
            }
        }
        private Player ()
        {

        }
        public Player (string newName)
        {
            name = newName;
            hand = new Cards();
        }
        public bool HasWon()
        {
            bool won = true;
            Suit match = hand[0].suit;
            for (int i=1;i<hand .Count ;i++)
            {
                won &= hand[i].suit == match;
            }
            return won;
        }
    }
```

}

بعد ذلك سنعرف صنفا يقوم بمعالجة لعبة الورق وسنسميه بالصنف Game قم بحفظ هذا الصنف ضمن الملف Game.cs ضمن المشروع CardLibrary2.

لهذا الصنف أربعة حقول خاصة:

- الحقل playDeck وهو من النوع Deck ويحتوي على مجموعة أوراق اللعب المستخدمة.
- الحقل currentCard ويمثل قيمة من النوع int تستخدم كمؤشر للورقة التالية في مجموعة أوراق اللعب التي سيتم سحبها.
- الحقل players ويمثل مصفوفة كائنات player تمثل اللاعبين في اللعبة.
- الحقل discardedCards ويمثل مجموعة Cards للأوراق التي تم رميها عزلها ولم تتم إعادتها إلى مجموعة أوراق اللعب.

يقوم منهج البناء الافتراضي في الصنف بتهيئة وإعادة فرز مجموعة أوراق اللعب كائن Deck المخزن ضمن المتحول playDeck ثم يقوم بإعداد المتحول currentCard كمؤشر للورقة الأولى في المجموعة بالقيمة 0 ومن ثم يقوم بتسجيل معالج الحدث PlayDeck.LastCardDrawn وربطه بالمنهج Reshuffle() يقوم هذا المنهج بإعادة خلط أوراق اللعب بطريقة عشوائية بالإضافة إلى الأوراق التي تم رميها الموجودة ضمن المجموعة discardedCards ويقوم بوضع القيمة 0 ضمن currentCard وذلك استعدادا للبدء بقراءة أوراق اللعب من المجموعة الجديدة.

يتضمن الصنف Game منهجين مساعدين أيضا setPlayers() الذي يستخدم لإعداد اللاعبين في اللعبة على شكل مصفوفة من كائنات players والمنهج DealHands() وذلك لإنشاء أوراق اللعب في اليد أي الأوراق التي يحملها كل لاعب بحيث يحمل كل لاعب سبع أوراق أما عدد اللاعبين فيتراوح بين لاعبين إلى سبعة لاعبين كحد أقصى.

وأخيرا فإن هناك المنهج playGame() والذي يتضمن منطق اللعبة وسنتحدث عن هذا المنهج لاحقا وإليك شيفرة الملق Game.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using CardLibrary2;

namespace CardLibrary2
{
    public class Game
    {
        private int currentCard;
        private Deck playDeck;
        private Player[] players;
        private Cards discardedCards;
```

```
        public Game ()
        {
            currentCard = 0;
            playDeck = new Deck(true);
            playDeck.LastCardDrawn += new
            LastCardDrawnHandler(Reshuffle);
            playDeck .Shuffle ();
        }
        private void Reshuffle(Deck currentDeck)
        {
            currentDeck.Shuffle();
            discardedCards = new Cards();
            currentCard = 0;
        }
        public void SetPlayers(Player [] newPlayers)
        {
            if (newPlayers.Length > 7)
                throw new ArgumentException("A maximum of 7 players" +
            "may play this game.");
            if (newPlayers.Length < 2)
                throw new ArgumentException("A minimum of 2 players" +
            "may play this game.");
            players = newPlayers;
        }
        private void DealHands()
        {
            for (int p=0;p<players .Length ;p++)
            {
                for (int c=0;c<7;c++)
                {
                    players[p].PlayHand.Add1(playDeck.GetCard(currentCard++));
                }
            }
        }
        public int PlayGame()
        {
            //code to follow
        }
    }
}
```

يشتمل الصنف Program.cs على التابع Main() والذي سيقوم بتهيئة وتنفيذ اللعبة يقوم هذا المنهج بما يلي:

- عرض مقدمة اللعبة.
- حث المستخدم على إدخال عدد الاعبين.
- إعداد مصفوفة كائنات players وفقا لعدد اللاعبين الذي تم إدخاله.
- حث المستخدم على إدخال أسماء اللاعبين وذلك لتخزين هذه الأسماء ضمن كائنات player في المصفوفة.
- إنشاء كائن اللعبة Game وإنشاء اللاعبين باستخدام المنهج SetPlayers().

❖ استخدام القيمة المعادة من المنهج playGame() لعرض اللاعب الرابح تمثل هذه القيمة دليل اللاعب الفائز في مصفوفة كائنات players.

إن شيفرة هذا التابع هي كما يلي:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using CardLibrary2;

namespace ConsoleCardClient2
{
    class Program
    {
        static void Main(string[] args)
        {
            //Display introduction.
            Console.ForegroundColor = ConsoleColor.Cyan;
            Console.WriteLine("karliCards:a new and exciting card game.");
            Console.WriteLine("to win you must have 7 cards of the same" +
                "suit in your hand.");
            Console.WriteLine();
            // prompet for number of player.
            bool inputOK = false;
            int choice = -1;
            do
            {
                Console.ForegroundColor = ConsoleColor.Red;
                Console.WriteLine("how many players (2-7)?");
                string input = Console.ReadLine();
                try
                {
                    //Attempt to convert input a vallid number of player.
                    choice = Convert.ToInt32(input);
                    if ((choice >= 2) && (choice <= 7))
                        inputOK = true;
                }
                catch
                {
                    //ignore failed conversions just continue prompting
                }
            }
            while (inputOK == false);
            Player[] players = new Player[choice];
            for (int p = 0; p < players.Length; p++)
            {
                Console.ForegroundColor = ConsoleColor.Green;
                Console.WriteLine("player {0},enter your name:", p + 1);
                string playerName = Console.ReadLine();
                players[p] = new Player(playerName);
            }
```

```
            Game newGame = new Game();
            newGame.SetPlayers(players);
            int whoWon = newGame.PlayGame();
            Console.WriteLine("{0} has won the game!", players[whoWon].Name);
        }
    }
}
```

والآن حان دور المنهج PlayGame() والذي يمثل الجسم الرئيسي للتطبيق لن نتحدث هنا عن التفاصيل لهذا المنهج ففي الحقيقة إن هذه الشيفرة ليست معقدة بالدرجة التي تتصورها وستتمكن من فهمها بسهولة.

تبدأ اللعبة بعد عرض أوراق كل لاعب وحيث يقوم كل لاعب بقلب ورقة من اوراقه السبعة على الطاولة عندئذ يمكن للاعب آخر اختيار هذه الورقة المقلوبة او سحب ورقة جديدة من مجموعة الأوراق وبعد أن يسحب اللاعب ورقة ينبغي عليه أن يرمي ورقة بالمقابل وذلك إما باستبدال الورقة الموجودة على الطاولة بواحدة أخرى من الأوراق التي بحوزته إذا اختار تلك الورقة المقلوبة على الطاولة أو بوضع الورقة التي لا يريدها فوق الورقة الموجودة على الطاولة أي إضافة ورقة سابقة على الطاولة إلى المجموعة discardedCards.

إن إحدى النقاط الرئيسية في هذه الشيفرة تكمن في طريقة معالجة كائنات Card والمبدأ هنا يعتمد على تعريف أوراق اللعب بأنواع مرجعية بدلا من أنواع قيمة وهو أمر يجب أن يكون واضحا بالنسبة إليك فنلاحظ ان كائن Card يمكن ان يكون موجودا في أماكن عديدة في الوقت نفسه فسيكون هناك مؤشر إلى ورقة اللعب الموجودة في مجموعة أوراق اللعب (كائن Deck) بالإضافة إلى مرجع للورقة نفسها في يد اللاعب (ضمن حقول hand لكائنات Player) وضمن الأوراق التي تم رميها ضمن المجموعة discardedCards ويمكن لان تكون الورقة الموجودة على الطاولة حاليا (كائن PlayCard أيضا) إن استخدام نوع مرجعي لكل ورقة لعب سيسهل من عملية تتبع الأوراق فالورقة التي سيتم سحبها من مجموعة أوراق اللعب سيتم قبولها فقط إذا لم تكن في يد أي لاعب ولم تكن على الطاولة ولم تكن أيضا ضمن مجموعة الأوراق التي تم رميها.

وشيفرة المنهج PlayGame() هي كما يلي:

```
public int PlayGame()
{
    if (players == null)
        return -1;
    DealHands();
    bool GameWon = false;
    int currentPlayer;
    Card playCard = playDeck.GetCard(currentCard++);
    do
    {
        for (currentPlayer = 0; currentPlayer < players.Length
            ; currentPlayer++)
        {
            Console.ForegroundColor = ConsoleColor.Yellow;
            Console.WriteLine("{0} 's turn",
                players[currentPlayer].Name);
```

```
Console.WriteLine("current hand:");
foreach (Card card in players[currentPlayer].PlayHand)
{
    Console.WriteLine(card);
}
Console.WriteLine("card in play:{0}", playCard);
bool inputOk = false;
do
{
    Console.ForegroundColor = ConsoleColor.Blue;
    Console.WriteLine("press T to take card in play or D" +
        " to draw:");
    string input = Console.ReadLine();
    if (input.ToLower() == "t")
    {
        Console.WriteLine("Drawn: {0}", playCard);
        players[currentPlayer].PlayHand.Add1(playCard);
        inputOk = true;
    }
    if (input.ToLower() == "d")
    {
        Card newCard;
        bool cardIsInPlayerHand;
        do
        {
            newCard = playDeck.GetCard(currentCard++);
            cardIsInPlayerHand = false;
            foreach (Player testPlayer in players)
            {
                cardIsInPlayerHand = testPlayer.PlayHand.
                    Contains1(newCard);
            }
        }
        while (cardIsInPlayerHand);
        Console.WriteLine("Drawn: {0}", newCard);
        inputOk = true;
    }
}
while (inputOk == false);
Console.WriteLine("New hand:");
for (int i = 0; i < players[currentPlayer].
    PlayHand.Count; i++)
{
    Console.WriteLine("{0}: {1}", i + 1,
        players[currentPlayer].PlayHand[i]);
}
inputOk = false;
int choice = -1;
do
{
    Console.WriteLine("choose card to discard:");
    string input = Console.ReadLine();
    try
    {
```

```
                    choice = Convert.ToInt32(input);
                    if ((choice > 0) && (choice <= 8))
                        inputOk = true;
                }
                catch
                {

                }
            }
            while (inputOk == false);
            playCard = players[currentPlayer].PlayHand[choice - 1];
            players[currentPlayer].PlayHand.RemoveAt(choice - 1);
            Console.WriteLine("Discarding:{0}", playCard);
            Console.WriteLine();
            GameWon = players[currentPlayer].HasWon();
            if (GameWon == true)
                break;
        }
    }
    while (GameWon == false);
    return currentPlayer;
}
```

والآن اضغط على زر F5 واستمتع باللعبة وتأكد من استيعابك لمنطقها وكيفية تطبيقه بصورة برمجية في المنهج السابق.

الخلاصة:

Summary:

لقد تعلمنا في هذا الفصل أحد أهم المواضيع المرتبطة بالبرمجة كائنية التوجه في لغة C# ألا وهو الأحداث وعلى الرغم من أن هذا الموضوع يحتاج إلى تفصيل أكبر مما قدمناه في هذا الفصل إلا أن الشيفرة التي طرحناها هنا تمكنك من استيعاب معظم ما يدور حول الأحداث وكيفية معالجتها سوف تجد الأحداث كثيرا في الفصول القادمة وستجد أنها تمثل جزءا لا يمكنك التخلي عنه أثناء تصميم التطبيقات في إطار عمل .NET.

وبالإضافة إلى أساسيات الأحداث وكيفية إنشاء معالجات الأحداث وتسجيلها لتنفيذها عند رفع حدث ما فقد تناولنا تطبيقا كاملا يبين كيفية استخدام الأحداث وذلك باستخدام مكتبة أصناف مجموعة أوراق اللعب واستخدمنا معظم ما تعلمناه منذ بداية الكتاب وحتى الآن ضمن لعبة الورق ومكتبة الأصناف .CardLibrary