

الواجهات، التفويض، والمواصفات

ونحن على مشارف الانتهاء من تعلم أساسيات لغة البرمجة Visual Basic .NET، تبقى لدينا مجموعة من المواضيع المتفرقة والتي يتحتم على ذكرها من منطلق الشمولية للإلمام بأساسيات لغة البرمجة Visual Basic .NET.

سأختم معك الجزء الأول من هذا الكتاب بالحديث عن مواضيع متعددة كالواجهات Interfaces وطريقة تطبيق مبدأ تعدد الواجهات Polymorphism، كما سأطرق أيضا إلى الإجراءات المفوضة Delegates والتي تعطيك مرونة كبيرة في استدعاء الإجراءات، واختتم الفصل بالتلميح إلى موضوع المواصفات Attributes لننتهي بذلك مرحلة تعلم أساسيات لغة البرمجة Visual Basic .NET.

الواجهات

في البداية دعني أوضح لك ما المقصود بكلمة واجهة في لغات OOP. الواجهة Interface هي مجموعة من الطرق والخصائص والأحداث التي تصف فئة معينة، فالفئة التالية:

```
Class Person
    Event Die()

    Public Property Name() As String
    ...
    ...
End Property

Sub Move()
    ...
    ...
End Sub
End Class
```

تحتوي على واجهة متمثلة في حدث باسم Die، خاصية باسم Name، وطريقة باسم Move(). لاحظ ان كلمة الواجهة تطلق على أسماء الأعضاء فقط وليس وظائفها (أي ليس سلوكها أو الشيفرات التي تحتويها).

من المبادئ الأساسية في لغات OOP هو مبدأ تعدد الواجهات Polymorphism، وهو احتواء الفئة الواحدة على أكثر من واجهة تصفها، ليصل الأمر إلى ان نتشارك مجموعة فئات في واجهات موحدة فنقول: أسماء متشابهة لكن إنجازات مختلفة Same names but different implementation.

العديد من الفوائد التي تجنيها من تطبيق واستخدام مبدأ تعدد الواجهات Polymorphism، لعل أبرزها اختصار جمل الشرط في برنامجك، فتخيل ان لدينا فئة تمثل مستند نصي TextFile، وفئة أخرى تمثل مستند منسق RTFFile، وفئة أخرى تمثل مستند صفحة ويب HTMLFile، وفئة رابعة تمثل مستند تنسيق البيانات XMLFile، وفئة أخيرة تمثل صورة PictureFile، ستجد ان برمجة وإنجاز كل فئة من هذه الفئات سيختلف تماماً عن الفئة الأخرى، ولكن الواجهات التي تحتويها متشابهة ومشاركة، فيمكن تعريف خاصية تمثل اسم الملف FileName أو حجم الملف FileSize، وطريقة تؤدي إلى عملية الحفظ Save() أو الحذف Delete()، ويمكن تطبيق جميع هذه الواجهات على الفئات المختلفة.

فمثلاً لو قمنا بتطوير إجراء يقوم بحفظ الملف النصي فسنجعل الوسيطة تستقبل كائن من النوع TextFile:

```
Sub SaveTextFile(ByVal FileObject As TextFile)
    ...
    FileObject.Save
    ...
End Sub
```

ونفس الشيء سنفعله مع الانواع الاخرى من الملفات:

```
Sub SaveRTFFile(ByVal FileObject As RTFFile)
    ...
    FileObject.Save
    ...
End Sub

Sub SaveHTMLFile(ByVal FileObject As HTMLFile)
    ...
    FileObject.Save
    ...
End Sub
```

ليس هذا فقط، بل أنه عند استدعاء إجراء الحفظ ستجد أنه يتوجب عليك كتابة الجمل الشرطية في كل مرة نود فيها فعل ذلك للتحقق من نوع الملف، وعلى ضوء المقارنة تقوم باستدعاء فئة الملف المعنية لتقوم بالحفظ، كما أن عليك أيضاً الإعلان عن جميع الكائنات التي تمثل الفئات المختلفة لإرسالها إلى الإجراء المختص:

```
Dim TextFileObject As New TextFile
Dim RTFFileObject As New RTFFile
Dim HTMLFileObject As New HTMLFile

...

Select Case FileType
    Case 1
        TextFileObject.FileName = "xxxx"
        SaveTextFile ( TextFileObject )
        ...
    Case 2
        RTFFileObject.FileName = "xxxx"
        RTFRTFFile ( RTFFileObject )
        ...
    Case 3
        HTMLFileObject.FileName = "xxxx"
        HTMLTextFile ( HTMLFileObject )
        ...
    ...
End Sub
```

والآن لنفترض انه ظهرت الحاجة لإضافة فئة جديدة لنوع معين من التسبيقات BinaryFile مثلاً، فإن ذلك يفرض عليك ان تضيف جميع الاستدعاءات والإجراءات والإعلانات والشروط التي ستتعامل مع هذه الفئة في كل أنحاء البرنامج، وتقوم بتكرار كل ما فعلته في الفئات السابقة. ولكن مع تعدد الواجهات فإن الأمر مختلف تماماً، فكل ما سنفعله هنا هو القيام بتعريف واجهة نسميها مثلاً IFile ونعرف إجراء يستقبل كل أنواع الكائنات شريطة احتوائه على الواجهة IFile:

```
Sub SaveFile(ByVal FileObject As IFile)
    ...
    FileObject.Save ()
    ...
End Sub
```

لا يقتصر الإجراء السابق على استقبال الأنواع التي ذكرناها سابقا (TextFile، RTFFile، HTMLFile، ... الخ) بل يمكنه ان يستقبل كل الأنواع الأخرى والتي قد تحتوي يوما من الأيام على الواجهة IFile.

بناء واجهة

بعد ان أوضحت لك الفكرة من الواجهات، لنبدأ ببناء واجهة تحمل الاسم IFile، تحتوي على جميع الأعضاء التي نود من الفئات الأخرى استخدامها. لتعريف واجهة في Visual Basic .NET، استخدم التركيب Interface ... End Interface:

```
Interface IFile
    ' خصائص
    Property FileName() As String
    Property FileSize() As Integer
    ...

    ' طرق
    Sub Save()
    Sub Save(ByVal TargetFile As String)
    Sub Delete()
    ...
End Interface
```

كما ترى في التركيب السابق، الواجهة لا تحتوي على أية شيفرات برمجية فهي مجرد واصفة لأسماء ووسيطات الأعضاء فقط، فلا تفكر في وضع عبارات كـ End Function، End Sub، أو End Property.

ملاحظة

جرى العرف سابقا عند مبرمجي OOP باستخدام الحرف I قبل اسم الواجهة، وهو نفس الأسلوب الذي تقترحه مستندات .NET. مع استخدام PascalCase لكتابة اسم الواجهة.

بالنسبة لقابلية الرؤية Visibility للواجهات، فهي افتراضيا Friend ويمكنك استخدام محددات الوصول الأخرى كما تفعل مع الفئات تماما:

```
' Friend
Interface IFile
...
End Interface

' Public
Public Interface IView
...
End Interface

Module Module1
    ' Private
    Private Interface ITool
        ...
    End Interface

    Sub Main()
        ...
    End Sub
End Module
```

مع ذلك، لا يمكنك استخدام محددات الوصول مع أعضاء الواجهة، فجميع أعضاء الواجهات قابلة
رويتها Public دائما وأبدا:

```
Interface ITest
    ' لا يمكن استخدام محددات الوصول
    ' مع أعضاء الواجهة
    Public Sub MyMethod ()
    Private Sub YourMethod ()
        ...
    End Interface
```

المزيد أيضا، يمكن للواجهات ان تكون متداخلة Nested:

```
Interface IView
    Interface IWindow
        ...
    End Interface

    Interface IWEB
        ...
    End Interface
...
End Interface
```

اخيراً، الطرق، الخصائص، والأحداث هي التي يمكن كتابتها في الواجهات، أما الحقول فغير ممكنة:

```
Interface IMyInterface
    X As Integer ' بوندنا ولكن لاسف
...
End Interface
```

تضمين الواجهة

بعد تعريفك للواجهة، يمكنك تضمينها في أي فئة باستخدام الكلمة المحجوزة `Implement`:

```
Class TextFile
    Implements IFile ' تضمين الواجهة
...
...
End Class
```

الشرط الأساسي لتضمين الواجهة في الفئة الحالية هو تعريف جميع أعضاء الواجهة وإلا ستظهر لك رسالة خطأ:

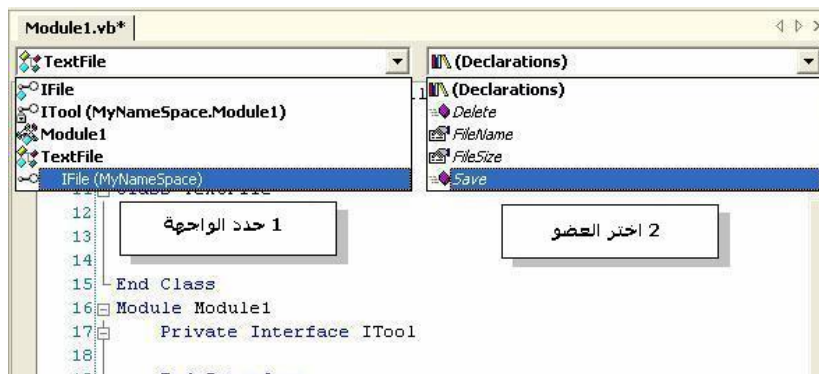
```
Class TextFile
    Implements IFile

    ' FileName الخاصية
    Property FileName() As String Implements IFile.FileName
        Get
            ...
        End Get
        Set(ByVal Value As String)
            ...
        End Set
    End Property

    ' Delete() الطريقة
    Sub Delete() Implements IFile.Delete
        ...
    End Sub

    ' عليك اضافة جميع الأعضاء الاخرى
    ...
    ...
End Class
```

وبدلاً من تعريف أعضاء الواجهات يدوياً، يمكنك الاستعانة بمحرر الشيفرات لعمل ذلك نيابة عنك، حدد الواجهة من القائمة العلوية اليسرى ومن ثم اختر العضو المراد تعريفه من القائمة العلوية اليمنى (شكل 5-1).



شكل 5-1: الاستعانة بمحرر الشيفرة لكتابة أعضاء الواجهة.

عند تعريف أعضاء الواجهة، فلا بد من توافق وسيطات الطرق، الأحداث، والخصائص مع نفس الوسيطات الموجودة في الواجهة، ولابد من توافق نوع القيم التي تعود بها الدوال Functions. المزيد أيضاً، قابلية القراءة والكتابة (اقصد استخدام الكلمات `ReadOnly` و `WriteOnly`) للخصائص، لابد وان تكون مطابقة لتلك الموجودة في الواجهة. فالتضمينات الموجودة في الفئة `TestClass` بالصفحة التالية خاطئة 100%:

```
Interface IMyInterface
    ' خاصية للقراءة فقط
    ReadOnly Property BirthDate() As Date

    ' دالة تعود بقيمة حرفية
    Function MyMethod() As String
End Interface
```

```

Class TestClass
    Implements IMyInterface
        ' قابلية القراءة والكتابة غير متوافقة
        Public WriteOnly Property BirthDate() As Date Implements _
            IMyInterface.BirthDate

            Set(ByVal Value As Date)
            ...
            ...
        End Set
    End Property

    ' القيمة التي تعود بها الدالة غير متوافقة
    Public Function MyMethod() As Integer Implements _
        IMyInterface.MyMethod

        ...
        ...
    End Function
End Class

```

بالنسبة للواجهات المتداخلة، فيمكنك تضمينها كما تفعل مع الواجهات العادية، ولكن لا تنسى كتابة اسم الواجهة الحاضنة لها أولاً:

```

Interface IView
    Interface IWindow
        ...
    End Interface

    Interface IWEB
        ...
    End Interface
    ...
End Interface

Class TestClass
    Implements IView.IWindow ' لا تنسى كتابة اسم الواجهة الحاضنة أولاً
    ...
    ...
End Class

```

أما إن استخدمت الواجهة الحاضنة:

```

Class TestClass2
    Implements IView
    ...
End Class

```

فالمؤكد ان الفئة السابقة لن تحتوي إلا على الواجهة IView فقط، ولا تعتقد للحظة ان الواجهات المحصورة بها (IWeb و IWindow) قد تم تضمينها هي الأخرى. أخيراً، يمكن للفئة الواحدة ان تحتوي على أكثر من واجهة:

```
Class ManyInterfaces
    Implements IFile
    Implements IView
    Implements IView.IWindow
    Implements IView.IWEB
...
End Class
```

عند تعارض اسماء الأعضاء في أكثر من واجهتين، فلا بد من تغيير اسمها لحظة كتابتها في الفئة، فمثلاً لو اكتشفت أن الطريقة MyMethod() موجودة في الواجهتين السابقتين IView و IFile، فيمكنك تمييزهما بوضع رقم:

```
Class ManyInterfaces
    Implements IFile
    Implements IView
...
Sub MyMethod() Implements IFile.MyMethod
...
End Sub

' لابد من تغيير اسم الطريقة '
Sub MyMethod2() Implements IView.MyMethod
...
End Sub
...
End Class
```

من المثال السابق، يتضح لنا ان مترجم .NET Visual Basic لا يهتم باسم العضو بقدر ما يهتم بالاسم الذي يلي الكلمة المحجوزة Implements لحظة تعريفه، لنستنتج من ذلك انه يمكننا تغيير اسم الطرق السابقة:

```

Class ManyInterfaces
    Implements IFile
    Implements IView
    ...
    ...

    Sub MethodFromFile () Implements IFile.MyMethod
        ...
    End Sub

    Sub MethodFromView () Implements IView.MyMethod
        ...
    End Sub
    ...
    ...
End Class

```

الوصول إلى الواجهة

بافتراض ان فئات الملفات السابقة (TextFile، RTFFile، HTMLFile... الخ) قد تم تضمين الواجهة IFile بها، فان باستطاعتنا تعريف متغير من النوع IFile وإسناد اي قيمة له من النوع TextFile، RTFFile، HTMLFile... الخ:

```

Dim textFileObject As New TextFile()
Dim file As IFile

' يمكن عمل ذلك
file = textFileObject
file.FileName = "MyFile.TXT"
...
...

```

كما يمكن تعريف إجراء يستقبل وسيطة من النوع IFile:

```

Sub DoSave(ByVal fileObject As IFile)
    ...
    fileObject.Save ()
    ...
End Sub

```

لنتمكن من إرسال اي قيمة له من النوع TextFile، RTFFile، HTMLFile.... الخ:

```
Dim htmlFileObject As New HTMLFile()
...
...
DoSave (htmlFileObject)
...
...
```

وراثه الواجهات

عند الحديث عن وراثه الواجهات، يمكننا ان ننظر إليها من منظورين مختلفين، المنظور الأول عند اشتقاق فئة قاعدية تحتوي على واجهة:

```
Interface IMyInterface
    Sub MyMethod()
End Interface

Class BaseClass
    Implements IMyInterface

    Sub MyMethod() Implements IMyInterface.MyMethod
        ...
    End Sub
End Class
```

عندما تشتق الفئة BaseClass السابقة، عليك معرفة ان الواجهة IMyInterface المضمنة بها سيتم اشتقاقها أيضا:

```
Class DerivedClass
    Inherits BaseClass

    ...
    ...
End Class
```

لنتعامل مع الكائن من الفئة DerivedClass كما نتعامل مع الكائن BaseClass:

```
Dim DerivedObject As New DerivedClass()
Dim X As IMyInterface

X = DerivedObject

' واجهة من الفئة القاعدية
X.MyMethod()
```

اما المنظور الثاني فهو يتعلق بوراثنة الواجهات، وهو توارث الواجهات فيما بينها باستخدام الكلمة المحجوزة Inherits أيضا:

```
Interface IBaseInterface
    Sub MyMethodInBase()
End Interface

Interface IDerivedInterface
    Inherits IBaseInterface

    Sub MyMethodInDerived()
End Interface
```

عند تضمين الواجهة IDerivedInterface في اي فئة، فيتوجب عليك تعريف جميع الأعضاء التابعة للواجهة القاعدية أيضا:

```
Class TestClass
    Implements IDerivedInterface

    ' الواجهة القاعدية
    Sub MyMethodInBase() Implements IDerivedInterface.MyMethodInBase
        ...
    End Sub

    ' الواجهة المشتقة
    Sub MyMethodInDerived() Implements
        IDerivedInterface.MyMethodInDerived
        ...
    End Sub
End Class
```

واجهات من إطار عمل .NET Framework

في الفصل الثالث الفئات والكائنات تعرفنا على الواجهة IDisposable، وذكرنا حينها انك تضمناها في الفئة لتعرف الإجراء Dispose() لمحاكاة المهذبات Destructors في الفئة:

```
Class TestClass
    Implements IDisposable

    Public Sub Dispose() Implements System.IDisposable.Dispose
        ...
    End Sub
    ...
End Class
```

توجد مئات الواجهات الأخرى والمعرفة في مكتبة فئات إطار عمل .NET Framework. إلا أنني سأحاول في هذا القسم عرض بضعة واجهات هامة تستخدمها في معظم برامجك ومشاريعك بـ Visual Basic .NET.

الواجهة IComparable

عملية إجراء المقارنة بين عددين تعتمد على قيمة العدد، ويمكنك استخدام معاملات المقارنة (>، <، =>... الخ)، ويمكن أيضا استخدام علامات المقارنة بين القيم الحرفية Strings، حيث سيعتمد المترجم في هذه الحالة على المقابل الرقمي للحرف في جداول المحارف ASCII أو UNICODE حسب الحالة:

```
Dim X As Integer = 10
Dim Y As String = "تركي"

If X > 5 Then ...
If Y < "احمد" Then ...
```

ولكن عند الحديث عن أنواع البيانات الأخرى والتي تصممها بفئاتك الخاصة، فلا يمكن تحديد أيهما أكبر وإيهما أصغر، فلو عرفت كائنين من فئة Person فأنت تحدد بنفسك ما هي القيم التي تريد الاعتماد عليها لحظة المقارنة، لذلك سنقوم بتضمين الواجهة IComparable في الفئة لتعرف طريقتها الوحيدة CompareTo() والتي تعود بالقيمة -1 إذا كان الكائن الحالي أصغر من الكائن المرسل، 0 في حالة تساوي الكائنين، أو 1 ان كان الكائن الحالي أكبر من الكائن المرسل:

```
Class Person
    Implements IComparable

    Public Name As String

    Function CompareTo(ByVal obj As Object) As Integer Implements _
        System.IComparable.CompareTo

        Dim tempObj As Person = CType(obj, Person)

        ' الكائن الحالي اصغر من الكائن المرسل
        If Me.Name < tempObj.Name Then
            Return -1
        ' الكائن الحالي اكبر من الكائن المرسل
        ElseIf Me.Name > tempObj.Name Then
            Return 1
        End If
    End Function
End Class
```

```

        الكائنات متساويان '
    Else
        Return 0
    End If
End Function
End Class

```

في المثال السابق، اعتمدت على خاصية الاسم Name لتحديد ناتج المقارنة بين الكائنين، قد تغير أنت رأيك وتعتمد على خاصية العمر Age مثلا في فئاتك الخاصة:



```

Class Person
    Implements IComparable
    Public Age As Integer

    Function CompareTo(ByVal obj As Object) As Integer Implements _
        System.IComparable.CompareTo

        Dim tempObj As Person = CType(obj, Person)

        ' الاعتماد على خاصية العمر Age
        If Me.Age < tempObj.Age Then
            Return -1
        ElseIf Me.Age > tempObj.Age Then
            Return 1
        Else
            Return 0
        End If
    End Function
End Class

```

والآن ففئتنا Person تحتوي على الواجهة IComparable لذلك نستطيع إرسال الكائنات المنشئة من هذه الفئة إلى جميع إجراءات فئات .NET Framework الأخرى (والتي تتطلب الواجهة IComparable طبعا). فمثلا الطريقة Sort() (والتابعة للفئة Array) يمكن ان تستقبل هذا النوع:



```

Dim PersonObject(10) As Person
Dim Counter As Integer

For Counter = 0 To UBound(PersonObject)
    PersonObject(Counter) = New Person()
Next

PersonObject(0).Age = 33
PersonObject(1).Age = 50
PersonObject(2).Age = 14

```

```
...
...
' الطريقة Array.Sort تقبل اي وسيطة تحتوي
' على الواجهة IComparable
Array.Sort(PersonObject)

For Counter = 0 To UBound(PersonObject)
    ArabicConsole.WriteLine(PersonObject(Counter).Age)
Next
```

الواجهة ICloneable

عند التعامل مع المتغيرات المرجعية Reference Type Variables، فقد أخبرتك أكثر من مرة ان عملية إسناد القيم بين متغيرين لا تقوم بإنشاء نسخة من المتغير وإسنادها إلى الآخر، وإنما تقوم بجعل كلا المتغيرين يشيران إلى نفس الكائن:

```
Dim Turki As New Person ()
Dim Ali As Person

...
...

' لا تقوم بإنشاء نسخة جديدة وإنما
' جعل كلا المتغيرين يشيران إلى نفس الكائن
Ali = Turki
```

معظم الفئات في عالم NET، تقوم بتضمين الواجهة ICloneable لتعريف طريقتها الوحيدة Clone() والتي تقوم بنسخ الكائن وإنشاء نسخة جديدة منه وإسنادها إلى متغير آخر، هذا مثال لتعريف الطريقة في فئة:

```
Class Person
    Implements ICloneable

    Public Name As String
    Public Age As Integer

    Function Clone() As Object Implements ICloneable.Clone
        ' إنشاء كائن مؤقت
        Dim tmpObject As New Person()

        ' نسخ قيم الكائن الحالي إلى الكائن
        ' المؤقت يدويا
        tmpObject.Name = Me.Name
        tmpObject.Age = Me.Age
```

```

        العودة بالكائن المنشئ للتو '
        Return tmpObject
    End Function
End Class

```

وبدلاً من قيامك بإنشاء الكائن ونسخ خصائصه يدوياً (حيث أنه يمكن أن توجد متغيرات سكونية Static داخل الطرق والتي لن تتمكن من نسخها بنفسك)، فيمكنك استخدام الطريقة MemberwiseClone() وهي مشتقة من الفئة System.Object (أعود للتذكير بأن جميع البيانات والأنواع وكل شيء في عالم NET، مشتقة وراثياً من الفئة System.Object):

```

Class Person
    Implements ICloneable

    Public Name As String
    Public Age As Integer

    Function Clone() As Object Implements ICloneable.Clone
        العودة بنسخة جديدة من الكائن الحالي '
        Return Me.MemberwiseClone()
    End Function
End Class

```

والآن يمكنك استخدامها بهذا الشكل:

```

Dim Turki As New Person()
Dim Ali As Person

Turki.Name = "تركي"

' نسخ الكائن
Ali = CType(Turki.Clone, Person)

Ali.Name = "علي"

' تحقق ان الكائن الاول لم يتم تغيير قيمته
ArabicConsole.WriteLine(Turki.Name)

```

كما ترى في الشيفرة السابقة، اضطررت إلى استخدام المعامل CType بافتراض ان العبارة Option Strict On مفعلة في الملف الحالي أو على مستوى المشروع، والسبب ان الطريقة Clone() تعود بقيمة من النوع Object. قد يكون استخدام المعامل CType غير محبذ

لمستخدمي فئاتك، لذلك اقترح عليك تغيير اسم الطريقة Clone() في الفئة واجعلها مخفية، ومن ثم عرف طريقة أخرى لتحل محلها:

```
Class Person
    Implements ICloneable
    ...
    ...

    Private Function PrivateClone() As Object Implements
    ICloneable.Clone
        Return Me.Clone()
    End Function

    Public Function Clone() As Person
        Return CType(Me.MemberwiseClone(), Person)
    End Function
End Class
```

هنا سيتمكن مستخدم الفئة من استدعاء الطريقة Clone() دون الحاجة لاستخدام المعامل CType:

```
Ali = Turki.Clone
```

وقبل ان اختتم فقرة الواجهة **ICloneable**، دعني أذكرك وأنبهك بان عمليات النسخ التي قمنا بها سابقاً (باستخدام الطريقة MemberwiseClone()) توجد بها مشكلة خطيرة جداً، ولم اعرضها هنا لان حلها يتم عن طريق ما يعرف بالتسلسل. لذلك، لا تتبع الأسلوب السابق في نسخ الكائنات بمشاريعك الحالية حتى تصل إلى الفصل التاسع تسلسل الكائنات **Object Serialization**. اللهم إني قد بلغت، اللهم فاشهد.

الواجهتان IEnumerable و IEnumerator

ذكرت في الفصل الثاني لغة البرمجة أنك تستطيع تطبيق الحلقة For Each ... Next على المصفوفات Arrays أو المجموعات Collections، وبما أننا نتحدث الآن بلغة الواجهات Interfaces فدعني أكون أكثر دقةً معك، وأخبرك ان الحلقة For Each يمكن تطبيقها أيضاً على جميع أنواع البيانات الأخرى التي تدعم الواجهتين IEnumerable و IEnumerator.

تعمل الواجهتان IEnumerable و IEnumerator جنباً إلى جنب لتمكين المبرمجين من استخدام الحلقة For Each مع فئاتك، وهذا مثال عملي يشرح كيفية استخدام هاتين الواجهتين، وسنبدأ أولاً بتصميم الفئة SplitString والتي تقوم بتقسيم الجملة إلى كلمات، تحتوي هذه الفئة على الخاصية Sentence والتي تمثل الجملة المراد فصل كلماتها:



```

Class SplitString
    Private currentPosition As Integer = 0

    ' خاصية تمثل الجملة المراد فصل كلماتها
    Private m_Sentence As String

    Property Sentence() As String
        Get
            Return m_Sentence
        End Get
        Set(ByVal Value As String)
            m_Sentence = Value
            Me.Reset()
        End Set
    End Property
End Class

```

الخطوة التالية هي تضمين الواجهتين IEnumerable و IEnumerator في الفئة السابقة:



```

Class SplitString
    Implements IEnumerable
    Implements IEnumerator
    ...
    ...
End Class

```

بالنسبة للواجهة IEnumerable، فهي لا تحتوي إلا على طريقة واحدة GetEnumerator() سيتم استدعاؤها بمجرد تنفيذ حلقة For Each، وهي تعود بقيمة لأي كائن يحتوي على الواجهة الأخرى IEnumerator. يكفي كتابة الأمر Return Me لتنفيذها:



```

Class SplitString
    ...
    ...
    Private Function GetEnumerator() As IEnumerator Implements _
        IEnumerable.GetEnumerator
        Return Me
    End Function
End Class

```

اما الواجهة IEnumerator فهي تحتوي على طريقتين وخاصية واحدة، الطريقة الأولى تسمى MoveNext() يتم استدعائها في كل دورة من دورات حلقة التكرار For Each، ويفترض ان تعود بالقيمة True إن بقي عنصر تالي و False إن لم يتبقى اي شيء:



```
Class SplitString
...
...
Private Function MoveNext() As Boolean Implements
IEnumerator.MoveNext
    If Me.currentPosition > Me.Sentence.Length - 1 Then
        Me.Reset()
        Return False
    Else
        Return True
    End If
End Function
End Class
```

والطريقة الثانية هي Reset() والغرض منها إعادة المؤشر الداخلي في الفئة إلى أول عنصر فيها حتى تتمكن من البدء باستخدام الحلقة For Each مرة أخرى:



```
Class SplitString
...
...
Private Sub Reset() Implements IEnumerator.Reset
    Me.currentPosition = 0
End Sub
End Class
```

اما الخاصية فهي تحمل الاسم Current والغرض منها العودة بالقيمة الحالية في الحلقة، لا تنسى ان هذه الخاصية للقراءة فقط ReadOnly:



```
Class SplitString
...
...
Private ReadOnly Property Current() As Object Implements _
IEnumerator.Current
    Get
        Dim counter As Integer
        Dim tmpLength As Integer = 0
```

```

        For counter = Me.currentPosition To Me.Sentence.Length - 1
            If Me.Sentence.Chars(counter) = " " Then
                Exit For
            Else
                tmpLength += 1
            End If
        Next
        Current = Me.Sentence.Substring(Me.currentPosition,
tmpLength)
        Me.currentPosition += tmpLength + 1
    End Get
End Property
End Class

```

والآن يمكنك تطبيق الحلقة For Each على الفئة السابقة SplitString بمرونة كبيرة، لتتمكن من كتابة شيئاً مثل:



```

Dim testObject As New SplitString()
Dim x As String

testObject.Sentence = "سيتم فصل كلمات هذه الجملة في سطور مستقلة"

For Each x In testObject
    ArabicConsole.WriteLine(x)
Next

```

مخرجات الشيفرة السابقة ستكون كالتالي:

```

سيتم
فصل
كلمات
هذه
الجملة
في
سطور
مستقلة

```

ملاحظة

الهدف من إنشاء الفئة SplitString هو تعليمي بحث، أردت أن أوضح من خلاله كيفية تطبيق الواجهتين IEnumerable و IEnumerator، فلا تحاول استخدامها في برامجك الجديدة، حيث يوفر لك إطار عمل .NET Framework فئات أفضل وأسرع بكثير لتقسيم النصوص. نقطة أخرى حول تضمين الواجهتين IEnumerable و IEnumerator، من المفضل ألا تقوم بتضمين هاتين الواجهتين في فئة واحدة، فيفضل تضمين الواجهة IEnumerable في فئة حاضنة، والواجهة IEnumerator في فئة محضونة بها، وذلك لتصميم كائني توجه OOP أفضل.

التفويض

يقصد بكلمة **التفويض Delegates**: بتفويض عملية استدعاء الإجراء إلى متغير يشير إلى ذلك الإجراء، مما يعطيك مرونة كبيرة في اختيار أوقات استدعاء الإجراءات، كما يختصر عليك الكثير من الشيفرات المصدرية والتي تستدعي هذه الإجراءات، فلو كان لدينا الإجراءات التالية:

```
Sub MySub (ByVal X As Integer)
...
End Sub

Sub YourSub (ByVal X As Integer)
...
End Sub
```

فإنه بإمكاننا استدعائها عن طريق متغير يمثل مؤشر إليها:

```
' DlgT سيتم استدعاء الإجراء الذي يشير له المتغير DlgT
' كما ستُرسل إليه قيم الوسيطات
DlgT.Invoke (100)
```

المزيد أيضاً، تستطيع إرسال هذه المؤشرات كوسيلة إلى إجراء آخر وتمكن الإجراء من استدعائها:

```
Sub ShowError(ByVal delegatePointer As OneParameter)
    delegatePointer.Invoke("حدث خطأ في العملية")
End Sub
```

تفيدك الطريقة السابقة كثيراً في اختصار تكرار الشيفرات المتشابهة، حيث ان الإجراء السابق قد يظهر رسالة الخطأ اما على شكل نافذة Message Box، في شاشة Console، عرضها على متصفح Browser (ان كنت تصمم موقعاً)، أو الكتابة في ملف... الخ، دون الحاجة لاعادة استدعاء الإجراء السابق من جديد. ميزة أخرى أكثر إثارة-سترى تطبيقاً عملياً عليها قريباً- هي إمكانية استدعاء إجراءات الفئات (أعضاء الفئات) من داخل الإجراءات المشتركة Shared دون مشاكل.

ان كنت من مبرمجي لغة C\C++ فقد يخطر ببالك مؤشرات الدوال **Function Pointers** والتي تمكنك من استدعاء الدوال عن طريق مؤشراتنا. بشكل مبسّط، مؤشرات الدوال في لغة C\C++ شبيهة بالإجراءات المفوضة في لغة .NET Visual Basic ولكنها تختلف في ميكانيكية عملها وصيغ كتابتها، حيث ان الإجراءات المفوضة في .NET Visual Basic أكثر أماناً من مؤشرات الدوال في لغة C\C++ وذلك لانها تحصر نوعية الإجراءات بحيث تتوافق مع وظيفتها. نقطة أخرى، الإجراءات المفوضة في .NET Visual Basic يمكن ان تتبع لإجراءات ستاتيكية Static Procedures أو تابعة لفئات Instance Procedures، بينما مؤشرات الإجراءات في لغة C\C++ لا يمكن تطبيقها الا مع الإجراءات الستاتيكية فقط.

الإجراءات الستاتيكية

اقصد بعبارة الإجراءات الستاتيكية Static Procedures الإجراءات (سواء كانت Subs أو Functions) التي تعرفها في الفئات Classes بحيث تكون مشتركة (باستخدام الكلمة المحجوزة Shared)، أو التي تعرفها في الوحدات البرمجية Modules. عند تفويض الإجراء إلى مؤشر، عليك اولاً القيام بتعريف نوع مؤشر الإجراء باستخدام الكلمة المحجوزة Delegates:



```
Module Module1
    ' تعريف نوع مؤشر الإجراء لابد ان يكون
    ' على مستوى الوحدة البرمجية
    Delegate Function OneParameter(ByVal X As Integer) As Integer

    Sub Main()
        ...
    End Sub
    ...
End Module
```

ولنفترض الآن أن لدينا الإجرائين التاليين:

```

Function Abs(ByVal x As Integer) As Integer
    If x < 0 Then
        Return -x
    Else
        Return x
    End If
End Function

Function Square(ByVal x As Integer) As Integer
    Return x * x
End Function
    
```

ونريد تفويضها إلى أي مؤشر من النوع OneParameter (الذي عرفناه سابقاً باستخدام الكلمة المحجوزة Delegate) في أي وقت بالطريقة التالية:

```

Sub Main ()
    ' تفويض الإجراء Abs ( )
    Dim Dtgt As New OneParameter(AddressOf Abs)
    ...
    ...
End Sub
    
```

الآن يمكنك استدعاء الإجراء Abs() عن طريق هذا المؤشر باستدعاء الطريقة Invoke() وإرسال الوسائط المطلوبة:

```

Sub Main ()
    Dim Dtgt As New OneParameter(AddressOf Abs)

    ' استدعاء الإجراء
    ArabicConsole.WriteLine( Dtgt.Invoke(-5) ) ' 5
End Sub
    
```

تستطيع الآن تفويض الإجراء الآخر Square() في أي وقت بإسناد عنوانه (باستخدام الكلمة المحجوزة AddressOf) إلى المؤشر Dtgt السابق، وعملية الاستدعاء ستتم بنفس الطريقة :Invoke()

```

Sub Main ()
    ...
    ...
    ' تفويض الإجراء Square ( )
    Dtgt = AddressOf Square
    ArabicConsole.WriteLine( Dtgt.Invoke(-5) ) ' 25
End Sub
    
```

المزيد أيضا، تستطيع تعريف إجراء يستقبل وسيطة من النوع OneParameter لتستدعي الإجراءات التي يشير إليها الكائن المرسل:

```


Sub Main()
    Dim Dtgt As New OneParameter(AddressOf Abs)
    ...
    MySub(Dtgt)
End Sub

' إجراء يستقبل وسيطة من النوع OneParameter
Sub MySub(ByVal delegatePointer As OneParameter)
    ArabicConsole.WriteLine(delegatePointer.Invoke(-5))
End Sub

```

دعنا نرى الآن ماذا يحدث خلف الكواليس، لأوضح لك حقيقة استخدام الكلمة المحجوزة Delegates كما في السطر الاول من المثال السابق عندما عرفت النوع OneParameter:

```
Delegate Function OneParameter(ByVal X As Integer) As Integer
```

تقنيا، قام مترجم اللغة بتحويل العبارة السابقة إلى فئة باسم OneParameter مشتقة وراثيا من الفئة System.MulticastDelegate (والتي هي أيضا مشتقة من الفئة System.Delegate)، أي يمكنك افتراض أن الشيفرة السابقة قد تم كتابتها لتعريف فئة OneParameter:

```

هذا مجرد افتراض، فلا تطبقه في الواقع '
Class OneParameter
    Inherits System.MulticastDelegate
    ...
End Class

```

لذلك، عندما أردنا إنشاء نسخة جديدة من الكائن اضطررنا لاستخدام الكلمة المحجوزة New ومن ثم إرسال القيم لوسيطات المشيد:

```
Dim Dtgt As New OneParameter(AddressOf Abs)
```

ما أود الوصول إليه هو ان DlgT مؤشر لكائن يحتوي على خصائص وطرق إضافية (ذكرت إحداها وهي الطريقة Invoke()) والتي تستدعي الإجراء المشار إليه، كما يمكنك التعامل مع هذا الكائن كما تتعامل من كائنات الفئات التي تعرفها بنفسك لتتمكن من كتابة شيئاً مثل:

```
Dim DlgT As New OneParameter(AddressOf Abs)
Dim X As OneParameter

'إسناد قيم بين كائنين
X = DlgT

ArabicConsole.WriteLine(X.Invoke(-5)) ' 5
```

إجراءات الفئات

في الفقرة السابقة طبقنا التفويض على الإجراءات الستاتيكية، اما إن رغبت في تطبيق التفويض على إجراءات الفئات فالعملية ستنتم بنفس الأسلوب السابق إلا أنك بحاجة إلى استخدام اسم الكائن الذي يتبع له الإجراء عند إرسال عنوان الإجراء (باستخدام AddressOf) إلى المؤشر المفوض:

```
Class TestClass
    Sub TestMethod()
        ArabicConsole.WriteLine("طريقة من الكائن")
    End Sub
End Class

Module Module1
    Delegate Sub NoParameter()

    Sub Main()
        Dim testObject As New TestClass()

        ' لاحظ هنا استخدام اسم الكائن testObject
        Dim dlgT As New NoParameter(AddressOf testObject.TestMethod)

        dlgT.Invoke()
    End Sub
End Module
```

باستخدام التفويض، يمكنك تخطي حاجز المنع من استدعاء إجراءات الفئة من داخل الإجراءات المشتركة في الفئة، حيث وكما ذكرت في الفصل الثالث **الفئات والكائنات** أنه لا يمكنك استدعاء طريقة في الفئة من داخل طريقة مشتركة في نفس الفئة:

```

Class TestClass
    Sub InstanceMethod()
        ...
        ...
    End Sub

    Shared Sub SharedMethod()
        ' لا يمكنك عمل ذلك
        InstanceMethod()
    End Sub
End Class

```

تستطيع تطبيق العملية السابقة باستخدام التفويض بعشرات الأساليب إما باستخدام متغيرات عامة Global Variables أو إرسالها كوسيطات للإجراءات المشتركة:

```

Class TestClass
    Sub InstanceMethod()
        ArabicConsole.WriteLine("تم استدعاء الطريقة")
    End Sub

    Shared Sub SharedMethod(ByVal methodPointer As NoParameter)
        methodPointer.Invoke()
    End Sub
End Class

Module Module1
    Delegate Sub NoParameter()

    Sub Main()
        Dim testObject As New TestClass()
        Dim dlgt As New NoParameter(AddressOf
testObject.InstanceMethod)

        ' سيتم استدعاء الطريقة InstanceMethod()
        ' من خلال الطريقة المشتركة SharedMethod()
        testObject.SharedMethod(dlgt)
    End Sub
End Module

```

محاكاة الأحداث

إن استخدمت المنطق البرمجي قليلا، فسرعان ما ستكتشف ان الأحداث Events التي تعرفها في الفئات تطبق أسلوب التفويض Delegates ولكن بصيغة مختلفة، فالحدث Die التالي:

```

Class Person
    Event Die()

    Sub Kill()
        RaiseEvent Die()
    End Sub
End Class

```

```
Module Module1
    Sub Main()
        Dim Turki As New Person()

        AddHandler Turki.Die, AddressOf PeronHasDied

        Turki.Kill()
    End Sub

    Sub PeronHasDied()
        ArabicConsole.WriteLine("لقد توفي الشخص")
    End Sub
End Module
```

ما هو إلا تفويض متمثل في الاسم Die، فأنت تقوم بتحديد عنوان الإجراء الذي سيتم تنفيذه باستخدام AddHandler. يمكنك أيضا الاستغناء عن صيغة إطلاق الأحداث السابق وتعريف تفويض (باسم NoParameter مثلا) ومن ثم التصريح عن متغير في الفئة يحمل اسم الحدث Die، وان أردت إطلاق هذا الحدث ستقوم باستدعاء الطريقة Invoke() عوضا عن الأمر RaiseEvent. الشيفرة التالية هي نفس الشيفرة السابقة، ولكني هنا استخدمت التفويض لإطلاق الحدث بدلاً من الصيغة المباشرة:

```
Delegate Sub NoParameter()

Class Person
    Public Die As NoParameter

    Sub Kill()
        Die.invoke()
    End Sub
End Class

Module Module1
    Sub Main()
        Dim Turki As New Person()

        Turki.Die = New NoParameter(AddressOf PeronHasDied)

        Turki.Kill()
    End Sub

    Sub PeronHasDied()
        ArabicConsole.WriteLine("لقد توفي الشخص")
    End Sub
End Module
```

دمج التفويضات

من الأشياء الرائعة جدا في استخدام تقنية التفويض، هو إمكانية دمج أكثر من تفويض في مؤشر واحد بحيث يمكنك تنفيذ أكثر من إجراء باستدعاء واحد لتضرب أكثر من عصفور بحجر واحد. فمثلا، لنفترض ان لدينا الاجرائين التاليين:

```
Sub MySub1()
    ArabicConsole.WriteLine("الإجراء الاول")
End Sub

Sub MySub2()
    ArabicConsole.WriteLine("الإجراء الثاني")
End Sub
```

ونريد استدعاء الاجرائين باستخدام التفويض، فإنه يتوجب علينا استدعاء كل إجراء على حدة - باستخدام الطريقة `Invoke()`:

```
Delegate Sub NoParameter()

sub Main ()
    Dim dlgt As New NoParameter(AddressOf MySub1)
    Dim dlgt2 As New NoParameter(AddressOf MySub2)

    dlgt.Invoke()
    dlgt2.Invoke()
End Sub
```

اما عند الدمج فلست بحاجة لكل هذا، حيث يكفي استخدام الطريقة `Combine()` لندمج اكثر من تفويض في مؤشر واحد، لنتمكن من تنفيذ الاجرائين `MySub1()` و `MySub2()` باستدعاء واحد فقط للطريقة `Invoke()`. نستطيع فعل ذلك بإنشاء متغير ثالث للقيام بهذه العملية:

```
Delegate Sub NoParameter()

Sub Main()
    Dim dlgt As New NoParameter(AddressOf MySub1)
    Dim dlgt2 As New NoParameter(AddressOf MySub2)
    Dim dlgt3 As NoParameter

    dlgt3 = dlgt.Combine(dlgt, dlgt2)

    ' سيتم تنفيذ كلا الاجرائين MySub1 و MySub2
    dlgt3.Invoke()
End Sub
```

أو استخدام متغير واحد فقط لتنفيذ هذه العملية، بحيث يتم إنشاء الكائنات لحظة إرسال الوسيطات:

```
Delegate Sub NoParameter()

Sub Main()
    Dim dlgt As NoParameter

    dlgt = dlgt.Combine(New NoParameter(AddressOf MySub1), _
        New NoParameter(AddressOf MySub2))

    ' MySub2 و MySub1 سيتم تنفيذ كلا الاجرائين
    dlgt.Invoke()
End Sub
```

الطريقة Combine() طريقة مشتركة Shared Method لذلك تمكنا من استدعائها قبل إنشاء نسخة جديدة من الكائن في المؤشر dlgt السابق. ودعني هنا أنبهك إلى أن هذه الطريقة ستعود بقيمة من النوع System.Delegate لذلك لابد من استخدام المعامل CType ان كانت العبارة Option Strict On مفعلة:

```
 dlgt = CType(dlgt.Combine( New NoParameter(AddressOf MySub1), _
    New NoParameter(AddressOf MySub2) ), NoParameter)
```

المزيد أيضا، يمكن دمج إجراء ثالث في المؤشر dlgt السابق، وعند استدعاء الطريقة Invoke() سيتم تنفيذ جميع الإجراءات الثلاثة بالتتالي (الأقدم فالأحدث):

```
' دمج إجراء ثالث
dlgt = CType(dlgt.Combine(dlgt, New NoParameter(AddressOf MySub3)), _
    NoParameter)

' سيتم تنفيذ الإجراءات
' MySub1()
' MySub2()
' MySub3()
' بالتسلسل
dlgt.Invoke()
```

نقطة أخرى حول دمج التفويضات، ففي حالة ما اذا كان نوع الإجراء المفوض Function فعليك معرفة أن الإجراء الأخير هو الذي سيعود بالقيمة:



Module Module1

Delegate Function NoParameterFun() As Boolean

Sub Main()

Dim dlgt As New NoParameterFun(AddressOf MyFunction1)

Dim dlgt2 As New NoParameterFun(AddressOf MyFunction2)

dlgt = CType(dlgt.Combine(dlgt, dlgt2), NoParameterFun)

ArabicConsole.WriteLine(dlgt.Invoke()) ' True

End Sub

Function MyFunction1() As Boolean

Return False

End Function

Function MyFunction2() As Boolean

Return True

End Function

End Module

أخيراً، إن رغبت في حذف الإجراء من مؤشر التفويض، فلن تجد أسهل ولا أجمل من الطريقة `:Remove()`



dlgt = CType(dlgt.Remove(dlgt, dlgt2), NoParameterFun)

ArabicConsole.WriteLine(dlgt.Invoke()) ' False

ملاحظة

الطريقة `Remove()` السابقة تحذف آخر إجراء تم إضافته للمتغير المفوض. مع ذلك، جميع الإجراءات المدمجة في المتغير تحفظ في الخاصية `GetInvocationList` وهي مصفوفة. لذلك، الشيفرة التالية تمكنك من حذف الإجراء الأول وليس الأخير:

```
dlgt.Remove(dlgt, dlgt.GetInvocationList(0))
ArabicConsole.WriteLine(dlgt.Invoke()) ' True
```

المواصفات

مصطلح **المواصفات Attributes** التي يوفرها لك إطار عمل .NET Framework جديدة - نسبيا - في لغات البرمجة، الهدف منها هو ان بعض محتويات مشاريعك قد لا تعتمد على كتابة الشيفرات المصدرية Source Codes بشكل كامل، وإنما قد تشمل تعاريف ونصوص حرفية Textual وعبارات توجيه للمترجم Compiler.

فمثلا، المعلومات العامة حول الفئات وأعضائها تعتمد على النصوص لتشرح الغرض منها ومتطلباتها أو حتى الوصول إلى الصفحة المناسبة لملف التعليمات الخاص بها. هناك أيضا بعض اوامر التوجيه الخاصة بالمترجم Compiler والتي تمكنك من التحكم في الشيفرات التي ترغب في ترجمتها ان تحققت شروط معينة. المزيد أيضا، قد تفرض وتحدد مواقع معينة لمتغيرات التركيبات من نوع Structure وطريقة تحديد مواقعها بالذاكرة.

بعض هذه الأمور متوفرة في لغات البرمجة ولكن لكل لغة طريقة خاصة ومعينة في تطبيقها، فنجد في لغة ++C/C# العبارات #pragma للسيطرة على المترجم، واما في لغة Visual Basic 6 فقد كان يعتمد بشكل كبير على صناديق الحوار التي تنشئ ملفات خاصة بها (ك-ctl. و idl). لحفظ هذه المواصفات. ولكن مع إطار عمل .NET Framework، فقد توحدت فكرة تطبيق المواصفات Attributes بين لغات .NET. الأخرى. ليس هذا فقط، بل أصبحت تمكنك من بناء مواصفات خاصة بك.

صيغة كتابة المواصفات في Visual Basic .NET

في البداية عليك معرفة ان المواصفات ما هي الا فئات Classes ولكن لها صيغة قياسية لا بد من إتباعها لتتمكن من استخدامها في مشاريعك، استخدم علامة الأقواس المعقوفة < و > لكتابة المواصفة:

```
<System.ComponentModel.DefaultProperty ("XXX")>
```

وبدلا من كتابة الاسم الكامل للمواصفة كما في المثال السابق، اقترح عليك باستيراد مجال الأسماء Namespace الذي توجد فيه المواصفة باستخدام Imports، لتختصر على نفسك كتابة الاسم الكامل في كل مرة تود فيها استخدام المواصفة:

```
Imports System.ComponentModel
<DefaultProperty ("XXX")>
```

السؤال هنا أين ستكتب الموصفة؟ والجواب يعتمد على نوع الموصفة التي ستستخدمها، فبعض الموصفات يمكن كتابتها في أي جزء أو كتلة من شيفراتك المصدرية، والبعض الآخر محصور في أماكن معينة كما ستري لاحقاً. فمثلاً، الموصفة DebuggerStepThrough لن تستطيع استخدامها إلا عند تعريف الإجراءات فقط:

```
<System.Diagnostics.DebuggerStepThrough()> Sub MySub()  
...  
...  
...  
End Sub
```

قياسياً سنستخدم المعامل _ لفصل الموصفة عن الشيفرة المصدرية بحيث نتوزع على سطرين مختلفين لتسهيل قراءة وتتبع الشيفرة:

```
<System.Diagnostics.DebuggerStepThrough()> _  
Sub MySub()  
...  
...  
...  
End Sub
```

مواصفات من إطار عمل .NET Framework

يوفر لك إطار عمل .NET Framework العشرات من المواصفات والتي يمكن استخدامها في أجزاء متفرقة من البرنامج لأداء وظائف معينة، في الفقرات التالية سنلقي الضوء على بعض هذه المواصفات والموجهة للغة البرمجة .NET Visual Basic، كما سأخصص فقرة كاملة لأعرض لك كيفية بناء موصفة خاصة بك.

الموصفة Conditional Attribute

عملية توجيه المترجم تقتضي ترجمة سطور معينة من الشيفرة المصدرية لحظة تحقق شرط معين، في معظم لغات البرمجة السابقة يستخدم الموجه #If لهذا الغرض:

```
#If MyCondition Then  
Sub MySub ()  
...  
...  
End Sub  
#End If
```

```
Sub Main
    MySub ()
    ...
    ...
    MySub ()
End Sub
```

المشكلة في هذا الأسلوب تبرز إن أردت إلغاء الإجراء MySub() السابق (عن طريق تغيير قيمة الشرط MyCondition ليكون False) فستظهر مئات الأخطاء في باقي سطور البرمجة - لحظة الترجمة - عند كل استدعاء للإجراء MySub()، احد الحلول هو استخدام الموجه #If عند كل استدعاء:

```
Sub Main
    #If MyCondition Then
        MySub ()
    #End If
    ...
    ...
    #If MyCondition Then
        MySub ()
    #End If
    ...
    ...
End Sub
```

صحيح ان الاسلوب السابق غير عملي، لذلك قد تدخل عبارة الشرط للموجه داخل الإجراء MySub() وتسهل عليك المهمة:

```
Sub MySub ()
    #If MyCondition Then
        ...
        ...
    #End If
End Sub
```

مع ذلك، إن كانت نتيجة الشرط MyCondition تساوي False فإن مئات السطور من الشيفرات المصدرية (والتي تستدعي الإجراء MySub() بالتحديد) أصبحت غير ذات قيمة، والتي ستتسبب في بطء عملية الترجمة والتنفيذ أيضا - بالإضافة إلى كبر حجم البرنامج.

يوفر لك إطار عمل .NET Framework المواصفة Conditional Attribute والتي يمكنك من توجيه المترجم عند تحقق شرط معين، وطريقة استخدامها يكون عند تعريف الإجراءات من نوع Sub:

```
<Conditional("MyCondition")> _
Sub MySub()
    ...
End Sub

Sub Main
    MySub ()
    ...
    MySub ()
    ...
End Sub
```

الإجراء MySub() السابق وجميع السطور التي ستستدعيه في كافة ملفات المشروع الأخرى، سيتم تجاهلها لحظة الترجمة ولن يتم ترجمتها وتضمينها في الملف النهائي ان كانت نتيجة الشرط MyCondition تساوي صفر أو False.

ملاحظة

لا يمكنك تطبيق المواصفة Conditional على الإجراءات من نوع Function.

بالنسبة للشرط MyCondition السابق فما هو الا ثابت يمكنك تعريفه وتحديد قيمته من صندوق حوار خصائص المشروع Project Property Pages، والانتقال إلى خانة التبويب Configuration Properties ثم خانة التبويب الفرعية Build ثم كتابة اسم الثابت وقيمه تحت خانة النص Custom Constants (شكل 5-2 بأعلى الصفحة المقابلة).



شكل 5-2: تحديد ثوابت الترجمة في صندوق حوار Project Property Pages.

المواصفة DebuggerStepThrough Attributes

في الفصل السابع اكتشاف الأخطاء سأنتقل إلى بعض طرق التنقيح Debugging التي توفرها بيئة Visual Studio .NET ومنها التنقيح سطرًا سطرًا (لعمل ذلك اختر الأمر Step Into من قائمة Debug أو اضغط على المفتاح [F11]).

في بعض الإجراءات التي تكون متأكدًا من صحتها ولست بحاجة إلى التدقيق فيها سطرًا تلو الآخر، يمكنك استخدام المواصفة System.Diagnostics.DebuggerStepThrough عند بداية الإجراء ليتم تنفيذه دفعة واحد:

```
<System.Diagnostics.DebuggerStepThrough(> _
Sub MySub()
    ...
    ...
    ...
End Sub
```

المواصفة Obsolete Attribute

الغرض من هذه المواصفة هو إظهار رسائل التنبيه Warning أو الخطأ لحظة ترجمة البرنامج، يمكنك الاستفادة منها مثلًا عندما تقوم بإرسال برنامجك إلى آخرين وتود لفت انتباههم إلى جزء معين من الشيفرة أو التحقق منها:

```
<Obsolete("يا عباس، لا تنسى تغيير شيفرة هذا الإجراء قبل التوزيع"  
Sub MySub()  
...  
...  
End Sub
```

عند الترجمة، ستظهر الرسالة السابقة كإنذار Warning في نافذة المخرجات Output (والتي تصل إليها باختيار الأمر View->Other Windows->Output) (شكل 5-3).



شكل 5-3: رسالة التنبيه Warning ظهرت أثناء الترجمة.

المزيد أيضاً، الموصفة Obsolete تستقبل وسيطة أخرى من النوع Boolean، إن أرسلت لها القيمة True ستحول الرسالة من رسالة تنبيه إلى رسالة خطأ لتوقف عملية الترجمة ولن يتم تنفيذ البرنامج:

```
<Obsolete("الم أقل لك عدل الإجراء يا عنيد", True)> _  
Sub MySub()  
...  
...  
End Sub
```

ملاحظة

سواء كانت رسالة تنبيه أو خطأ، فلن يتم عرضها إلا عند السطر الذي يستدعي الإجراء وليس بمجرد استخدام الموصفة Obsolete.

المواصفة StructLayout والمواصفة FieldOffset

استخدام المواصفة StructLayout موجه بشكل حصري إلى التركيبات من النوع Structure، حيث يمكنك من تحديد طريقة ترتيب متغيرات التركيب في الذاكرة، أرسل القيمة LayoutKind.Auto للحصول على أفضل وانسب أداء للمعالج:

استيراد مجال الاسماء لاختصار الشيفرات
Imports System.Runtime.InteropServices

```
<StructLayout(LayoutKind.Auto)> _
Structure RGBValue
    Dim Red As Byte
    Dim Green As Byte
    Dim Blue As Byte
End Structure
```

أما إن أرسلت القيمة LayoutKind.Explicit فعليك تحديد موقع كل متغير من متغيرات التركيب في الذاكرة بنفسك عن طريقة استخدام المواصفة FieldOffset:

```
<StructLayout(LayoutKind.Explicit)> _
Structure RGBValue
    <FieldOffset(0)> Dim Red As Byte
    <FieldOffset(1)> Dim Green As Byte
    <FieldOffset(2)> Dim Blue As Byte
End Structure
```

استخدامك للمواصفة FieldOffset يعطيك مرونة كبيرة للتحكم في مواقع المتغيرات في الذاكرة، فمثلا التركيب التالي لا يحتجز الا بايت واحد تصل اليه من ثلاث متغيرات:

```
<StructLayout(LayoutKind.Explicit)> _
Structure RGBValue
    <FieldOffset(0)> Dim Red As Byte
    <FieldOffset(0)> Dim Green As Byte
    <FieldOffset(0)> Dim Blue As Byte
End Structure
```

ولنتأكد من كلامي:

```
Dim X As RGBValue
X.Blue = 10
ArabicConsole.WriteLine(X.Red) ' 10
```

المزيد أيضاً، تستطيع الوصول إلى قيمة المتغيرات الثلاثة كلها عن طريق متغير واحد، راقب هذا التركيب:

```
<StructLayout(LayoutKind.Explicit)> _
Structure RGBValue
    <FieldOffset(0)> Dim Red As Byte
    <FieldOffset(1)> Dim Green As Byte
    <FieldOffset(2)> Dim Blue As Byte
    <FieldOffset(3)> Dim Extra As Byte
    <FieldOffset(0)> Dim Value As Integer
End Structure
```

لديك الآن الحرية المطلقة في الوصول إلى قيمة التركيب إما عن طريق المتغير Value (وهو يشمل المتغيرات الأربعة) أو عن طريق كل متغير على حدة:

```
Dim X As RGBValue

X.Value = 0
ArabicConsole.WriteLine(X.Red)           ' 0
ArabicConsole.WriteLine(X.Green)         ' 0
ArabicConsole.WriteLine(X.Blue)          ' 0

X.Red = 255
X.Green = 255
X.Blue = 255
ArabicConsole.WriteLine(X.Value)         ' 16777215
```

(الشكل 5-4) يوضح لك مواقع متغيرات التركيب RGBValue السابق في الذاكرة.



شكل 5-4: مواقع متغيرات التركيب RGBValue في الذاكرة.

بناء مواصفات خاصة

المواصفات الخاصة **Custom Attributes** ما هي إلا فئات **Classes** تقليدية تحتوي على جميع الشيفرات التي تود من المواصفة القيام بها، مع ذلك لابد للفئات -التي ترغب في ان تكون مواصفة **Attributes** - ان تستوفي عدة شروط، سأذكرها تباعاً، وسنطبقها معاً على مراحل ليسهل عليك استيعاب الفكرة.

سأقوم معك ببناء مواصفة **CodeInfo** تمكّنك من الاحتفاظ بمجموعة بيانات تتعلق بالشيفرة المصدرية (كاسم كاتبها، اسم مراجعها، تاريخ كتابتها،الخ).

الشرط الأول: يجب أن ينتهي اسم الفئة بالكلمة **Attribute**:

```
Class CodeInfoAttribute
```

```
End Class
```

الشرط الثاني: ان تكون الفئة مشتقة من الفئة **System.Attribute**:

```
Class CodeInfoAttribute
    Inherits System.Attribute
```

```
End Class
```

الشرط الثالث: ضرورة استخدام المواصفة **AttributeUsage** وتحديد المكان الذي يمكن استخدام المواصفة الحالية في شيفرات البرنامج:

```
<AttributeUsage(AttributeTargets.All) _
```

```
Class CodeInfoAttribute
    Inherits System.Attribute
```

```
End Class
```

تلاحظ اني ارسلت القيمة **AttributeTargets.All** وذلك لأجعل المواصفة الحالية قابلة للاستخدام في جميع أماكن الشيفرات المصدرية، يمكنك إرسال قيم أخرى لحصر مجال استخدام هذه المواصفة في موقع معين، الجدول الموجود في الصفحة التالية يوضح لك قيمة المواصفة، ومجال عملها.

القيمة	المكان الذي ستستخدم فيه الموصفة الحالية
AttributeTargets.All	في كل الاماكن.
AttributeTargets.Assembly	في المجمع الحالي.
AttributeTargets.Class	عند تعريف الفئات.
AttributeTargets.Constructor	مشيدات الفئات.
AttributeTargets.Delegate	الاجرائات المفوضة - باستخدام الكلمة المحجوزة Delegate.
AttributeTargets.Enum	التركيبات من النوع Enum.
AttributeTargets.Event	الأحداث.
AttributeTargets.Field	الحقول.
AttributeTargets.Interface	الواجهات.
AttributeTargets.Method	الطرق.
AttributeTargets.Module	الوحدات البرمجية.
AttributeTargets.Parameter	وسيطات الإجراءات.
AttributeTargets.Property	الخصائص.
AttributeTargets.ReturnValue	الاجرائات التي تعود بقيمة Functions.
AttributeTargets.Struct	التركيبات من النوع Structure.

ملاحظة

يمكنك تحديد أكثر من موقع باستخدام المعامل Or لحظة ارسال القيمة للموصفة AttributeUsage، فالقيمة التالية تحصر مجال استخدام الموصفة الحالية على الطرق والخصائص فقط:

```
<AttributeUsage(AttributeTargets.Method Or
AttributeTargets.Property)
```

الشرط الرابع: أنواع البيانات المسموح بها في هذا النوع من الفئات هي: Boolean، Byte، Short، Integer، Long، Char، String، Single، Double، Object، التركيب من نوع Enum، المصفوفات أحادية البعد One-Dimensional Array فقط.

سنقوم الآن بتعريف أعضاء الفئة مع الالتزام بالأنواع المحددة في الشرط الرابع:



```
<AttributeUsage(AttributeTargets.All) _
Class CodeInfoAttribute
    Inherits System.Attribute

    Public ProgrammerName As String ' اسم المبرمج
    Public TesterName As String ' اسم المراجع
    Public Tested As Boolean ' تمت المراجعة؟

    Sub New(ByVal programmerName As String)
        Me.ProgrammerName = programmerName
    End Sub
End Class
```

مبروك! المواصفة جاهزة للاستخدام الآن، أرسل وسيطة المشيد لحظة استخدامها في أي مكان من البرنامج (لاحظ تجاهل كتابة الكلمة Attribute عند استدعاء المواصفة):

```
<CodeInfo("تركبي العسيري") > _
Sub DoSomething()
    ...
    ...
End Sub
```

بالنسبة للخصائص الأخرى TesterName و Tested، فيمكنك إسناد القيم لها بهذه الصيغة:



```
<CodeInfo("تركبي العسيري") > _
Class Person

    <CodeInfo("تركبي العسيري", TesterName:="عباس السريع", Tested:=False) > _
    Sub Move ()
        ...
        ...
    End Sub

    <CodeInfo("تركبي العسيري", TesterName:="عبود اللوح", Tested:=True) > _
    Sub Kill ()
        ...
        ...
    End Sub
    ...
    ...
End Class
```

كنوع من الاقتراح، قد تضيف شيفرات إضافية لحفظ بيانات الموصوفة CodeInfo في ملفات نصية خارجية.

بهذا أكون قد انتهيت من تشييد بنية أساسية لتكون مبرمج .NET Visual Basic حقيقي بعدما تطرقت إلى جميع المبادئ والأساسيات التي لابد على كل مبرمج Visual Basic .NET من معرفتها وإقناعها للإبحار في برمجة واستخدام مكتبة فئات إطار عمل .NET Framework Class Library والآن فإن لديك مطلق الحرية في الانتقال إلى أي جزء ترغب به، رغم أنني أنصحك بشدة بالالتزام بالتسلسل الموضوع في هذا الكتاب والبدء من الفصل التالي **الفئات الأساسية** حتى تتعرف على اكبر قدر من الفئات الأولية قبل تطوير التطبيقات المتقدمة