Dotnet France
Technologies Sharepoint, SQL Server & .NET

Association Dotnet France

Les nouveautés du langage Visual Basic 9.0

Version 1.3

James RAVAILLE
http://blogs.dotnet-france.com/jamesr

Sommaire

1 Introduction

1.1 Présentation

En Février 2008, Microsoft sort officiellement et pour le grand public, Visual Studio 2008, la version 3.5 du Framework .NET, ainsi qu'une nouvelle version du langage Visual Basic .NET. Nous vous proposons dans ce cours, de vous présenter chacune des nouveautés de ce langage, avec pour chacune d'entre elles :

- Une partie théorique afin de vous expliquer en quoi elle consiste, et quel est son but, dans quels cas il est nécessaire/conseillé de l'utiliser…
- Une partie pratique avec des exemples de mise en œuvre.

Ces nouveautés sont les suivantes :

- Les délégués non stricts.
- Le support approfondi du langage XML.
- Les méthodes partielles.
- Les initialiseurs d'objets.
- L'inférence de type.
- Les types anonymes.
- Les méthodes d'extension.
- Les expressions lambda.

2 Les délégués non stricts

2.1 Présentation

Que ce soit dans une application Windows ou Web, nous sommes amenés à implémenter des évènements sur des objets, pouvant être des objets métiers ou des contrôles graphiques d'un formulaire. Dans le langage Visual Basic 8.0, tout évènement est basé sur un délégué. Un délégué peut être défini comme un pointeur de méthode. De manière plus précise, un délégué est un type de données, qui permet de créer des objets qui pointent vers une méthode.

Les développeurs Visual Basic 8.0 n'ont peut être pas l'habitude d'utiliser les délégués pour définir des évènements, puisqu'il est possible de définir un évènement dans une classe, sans avoir à utiliser explicitement un délégué. Alors que dans le langage Visual Basic, on ne peut définir un évènement sans utiliser un délégué. La signature de l'évènement est définie sur l'évènement lui-même (contrairement au langage C#, où la signature d'un évènement est définie au travers du délégué, utilisé pour définir l'évènement).

Lors de la création d'un gestionnaire d'évènement, autrement dit une méthode qui sera abonnée à un évènement, la signature de cette méthode doit respecter la signature du délégué ayant été utilisé pour définir l'évènement.

2.2 Mise en œuvre

Le langage Visual Basic 9.0 introduit la notion de « délégués non stricts », qui permet :

- De ne pas spécifier les arguments des gestionnaires d'évènements, s'ils ne sont pas utilisés.
    - La signature d'un gestionnaire d'évènements classiques, telle qu'on l'écrivait jusqu'en Visual Basic 8.0 :

```
' VB .NET

Private Sub CmdAction1_Click(ByVal sender As Object, ByVal e As
EventArgs) Handles CmdAction1.Click
    ' ...
End Sub
```

L'évènement *Click* définit dans la classe *System.Windows.Forms.Control*, classe de base des contrôles graphiques pour les applications Windows Forms, est défini avec un délégué nommé *EventHandler* acceptant comme paramètres, un objet à l'origine de la levée de l'évènement (de type *Object*), et un argument d'évènement (de type *EventArgs*).

- Dans le langage Visual Basic 9.0, on peut alors omettre ces paramètres s'ils ne sont pas utilisés dans le gestionnaire d'évènement :

```
' VB .NET

Private Sub CmdAction2_Click() Handles CmdAction2.Click
    ' ...
End Sub
```

- De pouvoir assigner à une même méthode, des évènements définis avec des délégués ayant une signature différente. C'est le cas de la méthode *Me_MonEvt*, qui est abonnée aux évènements *MontEvt1* et *MonEvt2*, qui n'ont pas la même signature :

```
' VB .NET

Private Event MonEvt1(ByVal aCode As Integer)
Private Event MonEvt2(ByVal aCode As Integer, ByVal aMessage As String)

Private Sub Me_MonEvt() Handles Me.MonEvt1, Me.MonEvt2
    ' ...
End Sub
```

3 Le support du langage XML approfondi

3.1 Présentation

Le langage Visual Basic 9.0 propose un support étendu de l'utilisation du langage XML, au travers de deux nouveautés :

- Les littéraux XML, qui permettent d'incorporer directement un flux XML dans un bloc de code Visual Basic. Il propose une aide à la saisie, et vérifie qu'il est bien formé lors de la compilation (autrement dit, qu'il respecte les spécifications du langage XML). Le code MSIL Microsoft Intermediate Language) obtenu, génère un code utilisant les classes de l'espace de noms *System.Xml.Linq* du Framework .NET (à savoir les classes *XElement*, *XAttribute*) pour générer ce flux XML.
- Les propriétés d'axe qui permettent d'accéder aux données contenues dans un flux XML. Cette nouveauté est très appréciée dans les requêtes LINQ For Xml, visant à requêter un flux XML :
    - Propriété d'axe d'attribut XML : element.@attribute
    - Propriété d'axe enfant XML : element.<enfant>
    - Propriété d'axe descendant XML : element...<descendant>
    - Propriété d'indexeur d'extension : element(index)
    - Propriété de valeur XML : element.Value

3.2 Mise en œuvre

Voici un exemple de code, permettant d'intégrer un flux XML, dans du code Visual Basic. La variable *oListeLivres* n'est pas de string, mais *System.Xml.Linq.XElement* :

```
' VB .NET

Dim oListeLivres = <livres>
     <livre titre="Microsoft Office SharePoint Server 2007" prix="25,78">
          <auteur>Anthony BIDET</auteur>
     </livre>
     <livre titre="C# 2 et ASP.NET 2.0" prix="25,78">
          <auteur>Anthony BIDET</auteur>
     </livre>
     <livre titre="Windows Server 2008" prix="30,50">
          <auteur>Jean-François APREA</auteur>
     </livre>
</livres>
```

Par la suite, il est alors possible de requêter les données contenues au travers une requête LINQ For Xml. C'est le cas de l'exemple ci-dessous, qui construit un nouveau flux de données au format XML, à partir du flux XML présenté ci-dessus :

```
' VB .NET

Dim sLibrairie As String
sLibrairie = "Rue de l'Ile"

Dim oFluxLibrairie = <librairie>
     <nom><%= sLibrairie %></nom>
     <livres>
          <%= From oLivre In oListeLivres.<livre> _
              Select <livre titre=<%= oLivre.@titre %> auteur=<%=
oLivre.<auteur>.Value %>></livre> %>
     </livres>
</librairie>
```

La variable *oFluxLibrairie* est de type *System.Xml.Linq.XElement*. Il contient le flux de données XML suivant :

```
<librairie>
  <nom>Rue de l'Ile</nom>
  <livres>
    <livre titre="Microsoft Office SharePoint Server 2007"
auteur="Anthony BIDET"></livre>
    <livre titre="C# 2 et ASP.NET 2.0" auteur="Anthony BIDET"></livre>
    <livre titre="Windows Server 2008" auteur="Jean-François
APREA"></livre>
  </livres>
</librairie>
```

4 Les méthodes partielles

4.1 Présentation

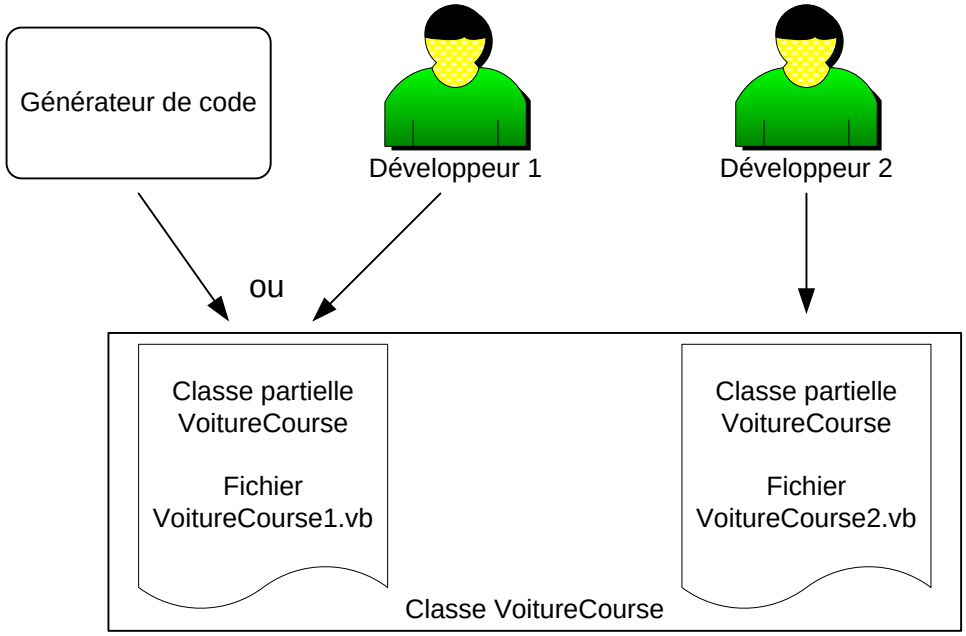
Les méthodes partielles sont obligatoirement contenues dans des classes partielles (nouveautés du langage Visual Basic 8.0), dans le même projet. Pour rappel, deux classes partielles constituent la même classe, séparées physiquement dans deux fichiers ou plus. Une méthode partielle est une méthode dont :

- La signature de la méthode est définie dans une classe partielle.
- L'implémentation de la méthode est définie dans une autre partie de la classe partielle. Cette implémentation est optionnelle.

Voici les caractéristiques des méthodes partielles :

- Elles sont définies avec le mot clé *partial*.
- Elles sont obligatoirement privées. Elles ne peuvent être appelées que dans la classe dans laquelle elles sont définies.

L'intérêt majeur des classes partielles réside dans les générateurs de code, ou bien encore dans des projets qui partagent des assemblies communes :



Par exemple, via un générateur de code (prenons par exemple LINQ for SQL), on génère une classe. Cette classe contient des membres pour lesquels il n'est pas possible de définir une implémentation lors de la génération. Le générateur de code ne génère alors que la signature de la méthode, et les appels de cette méthode dans la même classe. Cette implémentation ne peut être fournie que par le développeur 2. Ce dernier, au lieu de la fournir dans le fichier généré (ces modifications pourraient être perdues lors d'une prochaine génération de la classe), va la fournir en définissant la même méthode partielle dans une autre partie de la classe partielle.

4.2 Exemple de mise en œuvre

Voici un exemple de méthodes partielles :

```
' VB .NET

Partial Public Class Voiture
    Private _NumeroImmatriculation As String

    Public Property NumeroImmatriculation() As String
        Get
            Return Me._NumeroImmatriculation
        End Get
        Set(ByVal value As String)
            Me._NumeroImmatriculation = value
        End Set
    End Property

    Public Sub New(ByVal aNumeroImmatriculation As String)
        Me.NumeroImmatriculation = aNumeroImmatriculation
        Me.Demarrer()
    End Sub

    Partial Private Sub Demarrer()

    End Sub
End Class
```

La méthode *Demarrer* est une méthode partielle. Cette méthode est appelée dans le constructeur de la même classe. Nous avons vu précédemment, que l'implémentation de la méthode partielle dans l'autre classe partielle est optionnelle. Si on observe le code Visual Basic .NET rétro-compilé de l'assembly contenant cette classe avec l'outil Reflector (http://www.red-gate.com/products/reflector), alors on peut observer que l'appel de la méthode *Démarrer*, est présent uniquement si l'implémentation est réalisée dans une autre partie de la classe partielle.

Voici un exemple d'implémentation de la méthode *Démarrer*. Cette méthode aussi ne doit pas être définit avec un niveau de visibilité, et doit avoir la même signature que la méthode précédemment définie :

```
' VB .NET

Private Sub Demarrer()
    Console.WriteLine("La voiture " + Me.NumeroImmatriculation + " a
démarré")
End Sub
```

5 Les initialiseurs d'objets

5.1 Présentation

Le rôle des initialiseurs d'objets, est de « simplifier » l'écriture de la création d'objets à partir d'une classe ou d'un type anonyme, en combinant dans la même instruction :

- L'instruction de la création de l'objet.
- L'initialisation de l'état de l'objet (soit l'ensemble des attributs).

La mise en œuvre des initialiseurs d'objets est réalisée au travers de l'instruction *With {.attribut = valeur, … }*.

5.2 Exemple de mise en œuvre

5.2.1 Les initialiseurs d'objets

Soit la classe *Voiture* suivante :

```
' VB .NET

Partial Public Class Voiture
    Private _NumeroImmatriculation As String
    Private _Marque As String

    Public Property NumeroImmatriculation() As String
        Get
            Return Me._NumeroImmatriculation
        End Get
        Set(ByVal value As String)
            Me._NumeroImmatriculation = value
        End Set
    End Property

    Public Property Marque() As String
        Get
            Return Me._Marque
        End Get
        Set(ByVal value As String)
            Me._Marque = value
        End Set
    End Property

    Public Sub New(ByVal aNumeroImmatriculation As String)
        Me.NumeroImmatriculation = aNumeroImmatriculation
        Me.Demarrer()
    End Sub

    Public Sub New()
        Me.New(String.Empty)
    End Sub

    Partial Private Sub Demarrer()

    End Sub
End Class
```

Voici un bloc d'instruction Visual Basic, permettant de créer une instance de cette classe, en utilisant un constructeur défini dans la classe, et l'initialiseur d'objets :

```
' VB .NET

Dim oVoiture As Voiture

oVoiture = New Voiture() With {.NumeroImmatriculation = "212 YT 44",
.Marque = "Renault"}
```

Via l'initialisation d'objet, il est ainsi possible dans la même instruction, de créer une instance de la classe *VoitureCourse* en utilisant l'un des deux constructeurs présents dans la classe, et d'initialiser tous les attributs souhaités. A noter que le constructeur est toujours appelé avant l'initialisation des attributs.

6 L'inférence de type et les types anonymes

6.1 Présentation de l'inférence de type

L'inférence de type permet de déclarer une variable locale sans préciser son type. Cependant, cette variable doit obligatoirement être initialisée, afin que le compilateur puisse déterminer son type, à partir de la valeur ou l'expression d'initialisation. Une fois la variable déclarée, on peut lui appliquer tous les membres, exposés par le type automatiquement déterminé.

Je pense que l'inférence de type, doit uniquement être utilisée quand il n'est pas possible ou très difficile de déterminé le type de la variable à partir de sa valeur d'initialisation. Cette situation peut se produire dans deux cas :

- Lorsqu'un type anonyme est utilisé dans la valeur d'initialisation. C'est le cas lors de la réalisation de projection de données dans les requêtes LINQ.
- Lorsque le type de la valeur d'initialisation est complexe, et donc difficilement définissable par tout développeur.

L'inférence de type est mise en œuvre au travers du mot clé *Dim*.

6.2 Présentation des types anonymes

Le langage Visual Basic 8.0 introduisait les méthodes anonymes. Le langage Visual Basic 9.0 introduit les types anonymes. Il n'y a pas de relation directe entre ces deux notions.

Les types anonymes permettent de créer des objets et des collections d'objets, sans avoir à définir explicitement dans l'application, la classe utilisée pour créer les objets. C'est le compilateur qui se charge de générer dynamiquement la classe, lors de la compilation du code. L'utilisation de cette nouveauté entraine l'utilisation implicite de l'initialisation d'objets et de collections, ainsi que de l'inférence de types. On retrouve l'utilisation de toutes ces nouveautés dans l'écriture de requêtes LINQ.

Les types anonymes correspondent uniquement à des structures de données en mémoire. Ils ne peuvent contenir des méthodes, applicables aux objets qu'ils permettent de créer. Comme toutes classes, les types anonymes dérivent de la classe *System.Object*. Les seules méthodes alors applicables à ces objets sont ceux hérités de cette même classe.

La mise en œuvre des types anonymes est réalisée au travers de la syntaxe suivante : *New With { propriété = valeur , … }*.

6.3 Exemple de mise en œuvre

Voici un exemple de code, permettant de créer un objet nommé *oVoiture* à partir d'un type anonyme. La « structure » de cet objet est déterminée par les attributs qui sont définis et valorisés entre accolades. Cet exemple met en évidence un exemple d'utilisation des initialiseurs d'objets :

```
' VB .NET

Dim oVoiture = New With {.Marque = "Renault", .NumeroImmatriculation =
"212 YT 44", .Couleur = Color.Black}
```

Il est alors possible de créer des objets, contenant uniquement des données, sans avoir à implémenter dans le code la classe permettant de les créer. On remarque l'utilisation de l'inférence de type avec le mot clé *Dim*.

Voici un autre exemple, qui montre l'utilisation de l'inférence de type, d'un type anonyme et l'initialisation d'objets. La requête LINQ permet d'obtenir une collection d'objets, où chacun contient une marque de voiture et un numéro de département, à partir d'une collection de voitures. Seules les voitures immatriculées dans les départements 44 et 35, doivent être sélectionnées :

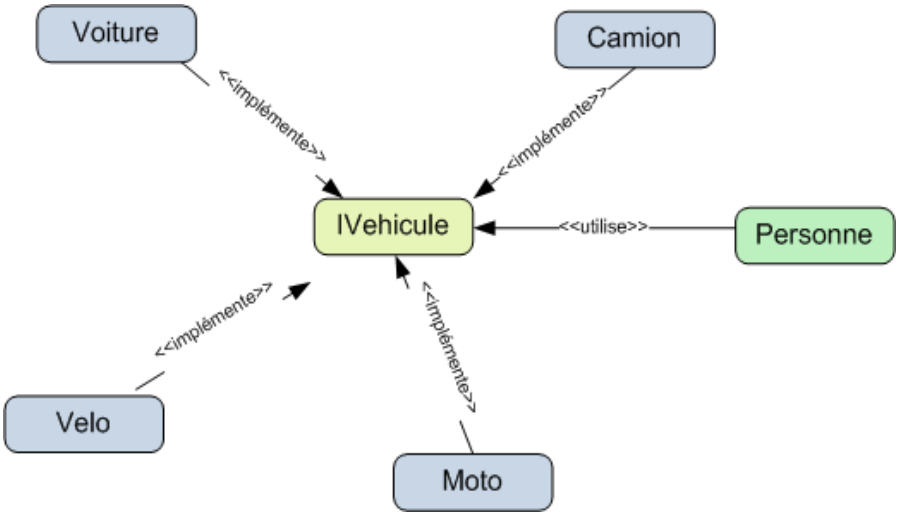
```
' VB .NET

Dim oListeNomsPrenomsEmployesActifs = (From oEmploye In oListeEmployes _
            Where (oEmploye.Actif) _
            Select New With {.NomPrenom = oEmploye.Nom + " " + _
                             oEmploye.Prenom}).ToList()
```

7 Les méthodes d'extension

7.1 Présentation

Les méthodes d'extension permettent d'étendre une classe, en ajoutant de nouvelles méthodes, sans créer de classe dérivée de cette même classe. Elles sont utiles dans les cas suivants :

- Soit une interface *IVehicule*, qui définit les membres de toutes classes permettant de créer des objets, qui se considèrent comme étant des véhicules. Soit les classes *Voiture*, *Camion*, *Velo*, *Moto*, qui implémentent cette interface.



Alors si on étend l'interface *IVehicule* par une méthode d'extension, alors cette méthode d'extension est applicable à toutes les instances, créées à partir des classes implémentant cette interface.

- Vous développez une application, qui utilise des classes contenues dans un assembly qui vous a été fourni. Vous ne possédez pas les sources de cet assembly. Vous savez qu'il contient une classe *Animal*, qui possède des classes dérivées. Vous souhaitez alors ajouter des membres supplémentaires à cette classe, afin d'enrichir les classes dérivées. Comme vous ne possédez pas les sources de l'assembly, vous ne pouvez le faire autrement qu'en créant des méthodes d'extension, qui étendent la classe *Animal*.

En Visual Basic 9.0, une méthode d'extension est une méthode, contenue dans un module de code. Cependant, cette méthode sera utilisée comme toute autre méthode d'instance de la classe étendue.

La signature de cette méthode est particulière :

- Elle doit être définie avec l'attribut de méthode *System.Runtime.CompilerServices.Extension()*.

- Le type du premier paramètre représente le type étendu. Une méthode d'extension peut posséder des paramètres d'entrée, qui sont alors à ajoutés à la suite de ce premier paramètre.

7.2 Exemple de mise en œuvre

Voici une méthode d'extension permettant d'étendre le type *System.Int32* du Framework .NET :

```
' VB .NET

Public Module IntExtension

    <System.Runtime.CompilerServices.Extension()> _
    Public Function MultiplierParDeux(ByVal aNombre As Integer) As String
        Return aNombre.ToString() + " * 2 = " + (aNombre * 2).ToString()
    End Function

End Module
```

Elle permet de multiplier par deux le contenu de toute variable de type *System.Int32*. Voici un exemple :

```
' VB .NET

Dim i As Integer = 10
Console.WriteLine(i.MultiplierParDeux())        => Affiche 10 * 2 = 20
```

Un autre exemple : toutes les classes représentant un dictionnaire de données, dit « générique » implémente l'interface *IDictionary(Of TKey, TValue)* du Framework .NET. Alors voici une méthode permettant d'étendre toute collection de données, implémentant cette interface, de manière à pouvoir accéder à la valeur d'un élément à partir de sa clé, ou obtenir une valeur par défaut, si aucun élément de la collection n'est identifié avec la clé :

```
' VB .NET

Public Module IDictionnaryExtension
    <System.Runtime.CompilerServices.Extension()> _
    Public Function GetValue(Of TKey, TValue)( _
                ByVal aListe As IDictionary(Of TKey, TValue), _
                ByVal key As TKey, ByVal defaultValue As TValue) As
TValue

        Dim aValeurRetour As TValue

        If (aListe.ContainsKey(key)) Then
            aValeurRetour = aListe(key)
        Else
            aValeurRetour = defaultValue
        End If

        Return aValeurRetour
    End Function
End Module
```

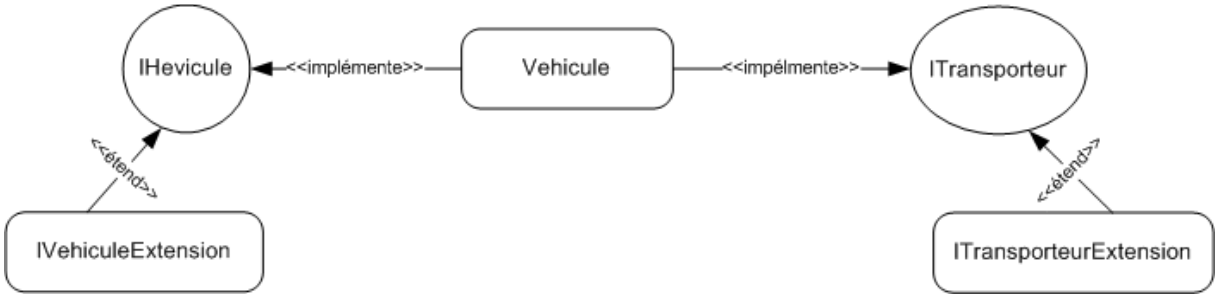
Et voici un exemple d'utilisation de la méthode d'extension, avec la classe générique *IDictionary(Of TKey, TValue)*, qui implémente l'interface générique *IDictionary(Of TKey, TValue)* :

```
' VB .NET

Dim oListeFormations As New Dictionary(Of Integer, String)
oListeFormations.Add(1, "Formation VB .NET")
oListeFormations.Add(2, "Formation ASP .NET")
oListeFormations.Add(3, "Atelier Accès aux données")
oListeFormations.Add(4, "La sécurité dans les applications .NET")
Dim sFormation As String
sFormation = oListeFormations.GetValue(4, "formation inexistante")
Console.WriteLine(sFormation)
```

Le corps des méthodes d'extension peut être « paramétrables », « personnalisables », au travers de l'utilisation d'expressions lambda.

7.3 Règles particulières

Soit le diagramme de classes suivant :



Voici quelques règles à observer lors de l'utilisation de méthodes d'extension :

- Si les classes *IVehiculeExtension* et *ITransporteurExtension* possèdent une méthode d'extension de même nom et avec une signature « analogue », alors une erreur survient lors de la compilation, lorsque cette méthode est appliquée à une instance de la classe *Vehicule*.
- Si la classe *Vehicule* contient une méthode, et que la classe *IVehiculeExtension* et/ou *ITransporteurExtension* propose une méthode d'extension de même nom et avec une signature « analogue », alors l'application compile et la méthode de la classe *Vehicule* « masque » les méthodes d'extension.

8 Les expressions lambda

8.1 Présentation

Une expression lambda est une fonction ne possédant pas de nom, et exécutant un traitement retournant une valeur. Une expression lambda est composée de deux parties :

- Une liste de paramètres, déclarée avec le mot clé *Function*.
- Une expression

Dans quels cas peut-on utiliser les expressions lambda :

- Pour réaliser des fonctions de calcul.
- Largement utilisées dans les méthodes d'extension de l'interface *IEnumerable(Of T)*.

8.2 Exemple de mise en œuvre

Par exemple, vous manipulez une liste d'entiers, créées avec le bloc d'instructions ci-dessous :

```
' VB .NET

Dim oListeNombres As New List(Of Integer)
Dim oListeNombrePaires As New List(Of Integer)

oListeNombres.Add(1)
oListeNombres.Add(34)
oListeNombres.Add(3)
oListeNombres.Add(9)
oListeNombres.Add(12)
oListeNombres.Add(18)
```

Et dans cette liste, vous souhaitez uniquement sélectionner les nombres pairs. Avec le langage Visual Basic 8.0, on écrira le bloc de code suivant :

```
' VB .NET

oListeNombrePaires = New List(Of Integer)()
For Each i As Integer In oListeNombres
    If (i Mod 2 = 0) Then
        oListeNombrePaires.Add(i)
    End If
Next
```

Avec le langage Visual Basic 9.0, en utilisant la méthode d'extension `Where` de l'interface générique *IEnumerable(Of T)*, on écrirait le bloc de code suivant :

```
' VB .NET

oListeNombrePaires = oListeNombres.Where(Function(i) i Mod 2 =
0).ToList()
```

Le bloc de code Visual Basic ci-dessus utilise la méthode d'extension *Where*, étendant l'interface générique *IEnumarable(Of T)* du Framework .NET. Cette méthode permet de filtrer la collection d'entiers, en fonction d'une expression booléenne. Cette expression booléenne est définie via l'expression lambda `Function(i) i Mod 2 = 0`, qui signifie « pour tous les nombres i de la collection pour lesquels le résultat du modulo par deux vaut 0 ».

8.3 Utilisation des expressions lambda comme paramètre de méthode

Dans une méthode « classique » ou une méthode d'extension, il est possible de « paramétrer », « personnaliser » le comportement de la méthode, en utilisant une expression lambda.

Voici une méthode d'extension, permettant de parcourir une liste d'entiers à laquelle elle est appliquée, afin de les traiter. Le traitement de ces entiers n'est pas déterminé dans la méthode d'extension elle-même, mais par le code utilisant cette méthode d'extension :

```
' VB .NET

<System.Runtime.CompilerServices.Extension()> _
Public Sub TraiterElements(ByVal aListe As List(Of Integer), ByVal
aExpression As Func(Of Integer, Integer))
    For i As Integer = 0 To aListe.Count Step 1
        aListe(i) = aExpression(i)
    Next
End Sub
```

Pour définir une expression lambda, il suffit d'utiliser le mot clé *Func* permettant de définir des types de données (une évolution des délégués). Dans l'exemple précédent, *Func(Of Integer , integer)* décrit une expression de traitement acceptant une donnée de type *Integer* en paramètre (le premier type précisé), effectuant un traitement avec ou sur ce paramètre, et retournant une donnée de type *Integer* (second type de donnée défini).

Enfin, voici deux blocs de code utilisant la méthode *TraiterElements* sur une collection d'entiers, afin de multiplier par 2 chacun des nombres de la collection :

```
' VB .NET

' Premier exemple.
Dim oExpressionTraitement As Func(Of Integer, Integer) = Function(i) i*2
oListeNombres.TraiterElements(oExpressionTraitement)

' Second exemple.
oListeNombres.TraiterElements(Function(i) i*2)
```

9 Conclusion

Ce chapitre vous a présenté les nouveautés du langage Visual Basic 9.0. Ces nouveautés ont été nécessaires, afin de permettre d'écrire des requêtes LINQ (sur une grappe d'objets, sur une base de données SQL Server, ou sur un flux XML), et l'utilisation de composants d'accès aux données tels que Classes LINQ For SQL.

Voici un exemple, qui récapitule et combine de nombreuses nouveautés du langage Visual Basic 9.0, au travers d'une requête LINQ : écrire une requête LINQ, permettant de sélectionner la liste des fichiers d'extension *txt* d'un répertoire, dont la taille est strictement supérieure à 10 Ko. En sortie, on doit obtenir uniquement le nom et la taille des fichiers en octets, dans l'ordre alphabétique inversé sur le nom :

```
' VB .NET

Dim oDirInfo As New DirectoryInfo(Application.StartupPath)

Dim oListeFichiers = oDirInfo.GetFiles("*.txt") _
     .Where(Function(oFichier) oFichier.Length > 10 * 1024) _
     .OrderByDescending(Function(oFichier) oFichier.Name) _
     .Select(Function(oFichier) New With {oFichier.Name, .Taille =
oFichier.Length})

For Each oFichier In oListeFichiers
     Console.WriteLine(oFichier.Name + ":" + oFichier.Taille.ToString())
Next
```

La requête LINQ présente dans le bloc de code ci-dessus, utilise pleinement les nouveautés du langage Visual Basic 9.0 que nous avons étudiées dans ce support :

- **Inférence de type** : liée à l'utilisation du type anonyme, et marquée par l'utilisation du mot clé *var*.
- **Méthodes d'extension** : marquées par l'utilisation des méthodes d'extension *Where*, *OrderByDescending*, et *Select* de l'interface générique *IEnumerable(Of T)* du Framework .NET.
- **Type anonyme** : marquée par l'utilisation du mot clé *new*, et la projection de données effectuée en ne sélectionnant que le nom et la taille du fichier en octets.
- **Expressions lambda** : utilisées dans les trois méthodes d'extension. Autrement dit, pour filtrer, trier les fichiers, et réaliser une projection de données sur les informations des fichiers.
- **Initialisation d'objet** : utilisée au sein du type anonyme.