Dotnet France
Technologies Sharepoint, SQL Server & .NET

Association Dotnet France

Les nouveautés du langage Visual Basic 10

Version 1.0

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

Sommaire

1 Introduction

1.1 Présentation

Au début de l'année 2010, Microsoft proposera Visual Studio 10, la version 4.0 du Framework .NET, ainsi qu'une nouvelle version du langage Visual Basic (version 10). 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 propriétés simplifiées.
- Les initialiseurs de collections.
- Les instructions dynamiques.
- Les sauts d'instruction implicites.
- Les instructions Lambda multi-lignes.
- La co-variance et la contre-variance.

Aussi, ce cours est basé sur la version Beta 1 du Framework .NET 4.0 et Visual Studio 2010 Beta 1.

1.2 Pré-requis

Avant de lire ce cours, vous devez avoir lu les précédents cours sur le langage Visual Basic :

- Les nouveautés du langage Visual Basic 9.0.

2 Les propriétés simplifiées

2.1 Présentation

Les propriétés simplifiés, offrent la possibilité de définir, au sein d'une classe, des propriétés sans implémenter les attributs qu'ils « gèrent », ni le corps de l'accesseur en lecture (getter) et l'accesseur en écriture (setter). En effet, en rétro-compilant le code obtenu, on peut observer que l'attribut géré est généré automatiquement, tout comme le code des accesseurs.

Les propriétés simplifiées correspondent aux accesseurs simplifiés, proposés dès la version 3.0 du langage C# dans Visual Studio 2008. En anglais, Elles sont appelées les « Auto Properties », ou bien encore les « Auto-Implemented Properties ».

Quelques « restrictions » toutefois, par rapport aux propriétés implémentées avec le langage Visual Basic 9 :

- Les accesseurs en lecture et en écriture (get/set), doivent obligatoirement être tous les deux présents. Il n'est pas possible de définir un accesseur en lecture ou écriture seule.
- Les accesseurs en lecture et en écriture (get/set) ont le même niveau de visibilité : impossible de réduire le niveau de visibilité de l'accesseur, en lecture ou en écriture.
- Le type de l'attribut généré est obligatoirement celui de la propriété.

L'intérêt des propriétés simplifiées est d'écrire moins de code dans les classes.

2.2 Exemple de mise en œuvre

Soit la classe Voiture suivante :

```
// VB

Public Class Voiture

#Region "Attributs et accesseurs"

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

    Public Property Marque As String

#End Region

End Class
```

La classe ci-dessus présente :

- Un attribut avec sa propriété (aussi appelé propriété « par défaut »), contenant le numéro d'immatriculation d'une voiture.
- Une autre propriété avec une forme particulière, contenant la marque d'une voiture : les accesseurs en lecture et en écriture ne sont pas implémentés. Il s'agit d'une propriété simplifiée. Au final, lors de la génération du code MSIL, notre classe possède deux attributs, et pour chacun d'entre eux, une propriété. Pour l'observer, il suffit de « rétro-compiler » le code MSIL de cette classe, avec un outil tel que Reflector (http://www.red-gate.com/products/reflector). Voici ce qu'on peut observer :

```
Public Class Voiture
    ' Properties
    Public Property Marque As String
        Get
            Return Me._Marque
        End Get
        Set(ByVal AutoPropertyValue As String)
            Me._Marque = AutoPropertyValue
        End Set
    End Property

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


    ' Fields
    <CompilerGenerated, DebuggerBrowsable(DebuggerBrowsableState.Never)> _
    Private _Marque As String
    Private _NumeroImmatriculation As String
End Class
```

3 Les initialiseurs de collection

3.1 Présentation

Le langage Visual Basic 9.0 (cf : voir le cours sur ce langage publié sur Dotnet-France) présentait les initialiseurs d'objets, qui permettaient 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).

Le langage Visual Basic 10 rejoint le langage C# en proposant les initialiseurs de collection. Voici un exemple, montrant commun créer une collection de nombre entiers, via l'utilisation du mot clé *From* :

```
// VB

Dim oListeEntiers As New List(Of Integer) From {1, 3, 4, 6, 90, 34}
```

Voici un autre exemple, montrant comment créer une collection d'objets de type *Voiture* :

```
// VB

Dim oListeVoitures As New List(Of Voiture)
    From {New Voiture("34 YT 54"), New Voiture("17 RE 33"),
          New Voiture("106 IU 75") With {.Marque = "Peugeot"}}
```

L'exemple ci-dessus, vous montre qu'il est aussi possible d'utiliser l'initialisation d'objets (via l'utilisation du mot clé *With*), dans une instruction utilisant l'initialisation de collections.

4 Le typage dynamique

4.1 Présentation

Le typage dynamique des objets permet de créer des instructions dynamiques. Ces instructions permettent déclarer des variables locales, des paramètres de méthodes ou des attributs de classe, qui seront typés lors de l'exécution de l'application (on parle dans ce cas de liaisons tardives). Elles peuvent aussi être utilisées comme type de retour de méthodes.

Le typage dynamique exploite une nouvelle fonctionnalité de la CLR, appelée la DLR (**D**ynamic **L**anguage **R**untime). Cette fonctionnalité permet d'exécuter dans la CLR des langages dynamiques tels que le langage IronPython (« Python » pour .NET).

Une fois déclarée dynamiquement, il est possible d'utiliser ces membres, sans qu'aucune vérification ne soit effectuée par le compilateur.

Les intérêts des instructions dynamiques sont les suivants :

- Utiliser plus facilement des objets venant de langage de programmation dynamique tels que les langages *IronPhyton* et *IronRuby*.
- Faciliter l'accès et l'utilisation d'objets COM.
- Proposer une alternative à la réflexion (mécanismes d'introspection de code MSIL, …).

Toutefois, attention à l'utilisation des instructions dynamiques. Utilisées à mauvais escient, elles rendront les applications moins robustes lors de leur exécution, en provoquant des erreurs uniquement lors de l'exécution (le compilateur ne connaît pas le type des objets manipulés). Dans des cas très précis, elles permettent de simplifier le développement.

Pour mettre en œuvre le typage dynamique, nous utiliserons le mot clé *dynamic*.

4.2 Exemples

Soit le bloc d'instructions ci-dessous, que nous pouvons exécuter dans une application Windows Forms :

```
// VB

Dim var1 As Object = "abc"
Dim var2 As String = var1.ToUpper()
MessageBox.Show(var2)
```

La première instruction permet de déclarer une variable nommée *var1*, et de la typer (lors de l'exécution) en chaîne de caractères (type *string*), ce type étant déduit de la valeur d'affectation "abc".

La seconde instruction permet d'appliquer (dynamiquement) la méthode *ToUpper* à cette variable, de manière à transformer sa valeur en chaîne de caractères majuscules.

La troisième instruction permet d'afficher dans une boite de message le contenu de la variable *VAR1*.

Voici un second exemple. Soit les deux classes suivantes :

```
// VB

Public Class Personne
    Public Property Nom As String
    Public Property Prenom As String

    Public Sub New(ByVal aNom As String, ByVal aPrenom As String)
        Me.Nom = aNom
        Me.Prenom = aPrenom
    End Sub

    Public Function Deplacer() As String
        Return Me.Nom + " " + Me.Prenom + " est arrivé à destination"
    End Function
End Class


Public Class Animal
    Public Property Nom As String
    Public Property Race As String

    Public Sub New(ByVal aNom As String, ByVal aRace As String)
        Me.Nom = aNom
        Me.Race = aRace
    End Sub

    Public Function Deplacer() As Point
        Return New Point(20, 50)
    End Function
End Class
```

Ces deux classes possèdent un accesseur *Nom* (vous remarquerez l'utilisation des accesseurs simplifiés, *cf : cours sur les nouveautés du langage Visual Basic 9.0*) en commun, ainsi qu'une méthode nommée *Deplacer*.

Voici une autre méthode ayant un paramètre défini dynamiquement, appliquant une méthode *Deplacer*() à cet objet, et affiche les informations retournées par cette méthode :

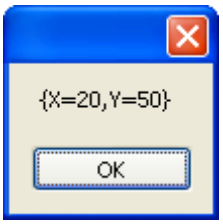
```
// VB

Public Sub ChangerPosition(ByVal aObjet As Object)
    MessageBox.Show(aObjet.Deplacer().ToString())
End Sub
```

Le bloc d'instructions crée une instance de la classe *Personne*, une autre de la classe *Animal*, et exécute deux fois la méthode *ChangerPosition* en passant successivement ces objets en paramètre :

```
// VB

Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button2.Click
    Dim oPersonne As New Personne("RAVAILLE", "James")
    Dim oAnimal As New Animal("Médor", "Chien")

    Me.ChangerPosition(oPersonne)
    Me.ChangerPosition(oAnimal)
End Sub
```

L'instruction *Me.ChangerPosition(oPersonne)* affiche le message suivant :

L'instruction *Me.ChangerPosition(oAnimal)* affiche le message suivant :



Ces deux exemples mettent en évidence que la méthode *Deplacer* est bien appliquée dynamiquement à un objet, dont le type de données n'est pas explicitement défini lors de la compilation, mais lors de l'exécution. Elle est appliquée sur un objet, quelque soit son type, tant que cette méthode est définie au sein de la classe ou héritée d'une classe de base. Cependant :

- Il n'est pas possible d'appliquer de cette manière les méthodes d'extension (*cf : cours sur les nouveautés du langage Visual Basic 9.0*).
- Il n'est pas possible d'utiliser les expressions lambda (*cf : cours sur les nouveautés du langage Visual Basic 9.0*) comme paramètre des méthodes.

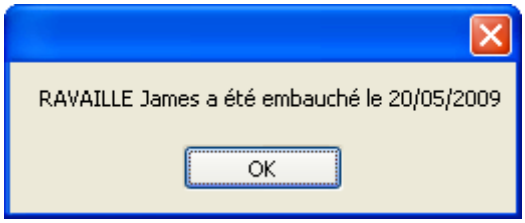
Voici une méthode nommée *Embaucher*, acceptant un paramètre défini dynamiquement, et affichant le *Nom* et le *Prenom* de l'objet passé en paramètre :

```
// VB

Public Sub Embaucher(ByVal aPersonne As Object)
    MessageBox.Show(aPersonne.Nom + " " + aPersonne.Prenom + " a été
embauché le " + DateTime.Now.ToString())
End Sub
```

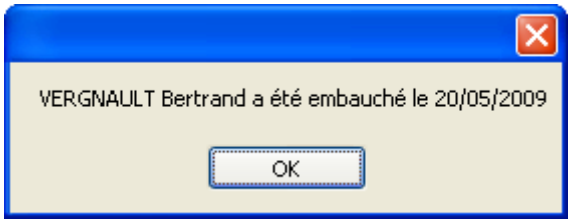
Comme nous l'avons vu, il est alors possible d'appeler cette méthode en passant un objet créé à partir de n'importe quelle classe exposant des propriétés *Nom* et *Prenom*. Mais cet objet peut aussi avoir été créé à partir d'un type anonyme (*cf : cours sur les nouveautés du langage Visual Basic 9.0*) :

```
// VB

Private Sub Button3_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button3.Click
    Dim oPersonne As New Personne("RAVAILLE", "James")
    Dim oPersonne1 = New With {.Nom = "VERGNAULT", .Prenom = "Bertrand"}

    Me.Embaucher(oPersonne)
    Me.Embaucher(oPersonne1)
End Sub
```

L'instruction *Me.Embaucher(oPersonne)* affiche le message suivant :



L'instruction *Me.Embaucher(oPersonne1)* affiche le message suivant :



5 Saut d'instruction d'implicite

Avec le langage Visual Basic 9, pour déclarer et initialiser une chaîne de caractères en une seule instruction sur plusieurs lignes dans l'éditeur, nous devons utiliser le caractère de saut d'instruction « _ ». Voici un exemple :

```
// VB

Dim s As String = "Date 1 : " + DateTime.Now.ToString() + _
    " Date 2 : " + DateTime.Now.AddDays(1).ToString() + _
    " Date 3 : " + DateTime.Now.AddMonths(1).ToString()
```

Le langage Visual Basic 10 permet d'écrire ce bloc d'instruction, sans avoir à utiliser le caractère de saut d'instruction :

```
// VB

Dim s As String = "Date 1 : " + DateTime.Now.ToString() +
    " Date 2 : " + DateTime.Now.AddDays(1).ToString() +
    " Date 3 : " + DateTime.Now.AddMonths(1).ToString()
```

6 Les instructions lambda multi-lignes

Voici un bloc d'instructions permettant d'exécuter de manière asynchrone un traitement sur des chaînes de caractères (valable avec les versions 9 et 10 du langage Visual Basic) :

```
// VB

Private Sub Button6_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button6.Click
    Dim oListeNoms = {"James", "Bertrand", "Julien", "Laurent"}

    Dim oThread As New System.Threading.Thread(AddressOf Traiter)
    oThread.Start(oListeNoms)
    oThread.Join()
End Sub

Private Sub Traiter(ByVal aListeStrings As String())
    For Each s As String In aListeStrings
        ' Bloc d'instructions ...
    Next s
End Sub
```

Dans le bloc de code ci-dessous, nous sommes dans l'obligation de créer une nouvelle méthode dans notre classe, contenant le code exécuté de manière asynchrone, à savoir *Traiter*.

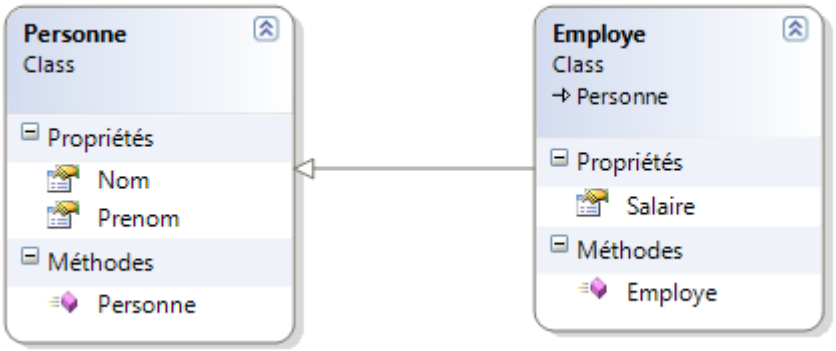
Le langage Visual Basic 10 permet au travers des instructions lambda multi-lignes, de pouvoir écrire un bloc de code iso-fonctionnel, sans avoir à créer de méthode supplémentaire dans la classe :

```
// VB

Private Sub Button6_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button6.Click
    Dim oListeNoms = {"James", "Bertrand", "Julien", "Laurent"}

    Dim oThread As New System.Threading.Thread(
        Sub()
            For Each s As String In oListeNoms
                ' Bloc d'instructions ...
            Next s
        End Sub)
    oThread.Start()
    oThread.Join()
End Sub
```

7 La co-variance et la contre-variance

La co-variance et la contre-variance n'est pas un nouveau concept. En Visual Basic 8.0, il était possible de les appliquer sur des délégués. Le langage Visual Basic 10 étend la co-variance et la contre-variance aux interfaces génériques.

Pour illustrer en quoi consiste la co-variance et la contre-variance, nous allons nous baser sur le diagramme de classes suivant :



La classe *Personne* est caractérisée par un nom et un prénom. La classe *Employe* spécialise la classe *Personne*, et définit un attribut supplémentaire : le salaire.

Voici l'implémentation de ces deux classes :

```
// VB

Public Class Personne
    Public Property Nom As String
    Public Property Prenom As String

    Public Sub New(ByVal aNom As String, ByVal aPrenom As String)
        Me.Nom = aNom
        Me.Prenom = aPrenom
    End Sub
End Class

Public Class Employe
    Inherits Personne

    Public Salaire As Double

    Public Sub New(ByVal aNom As String, ByVal aPrenom As String, ByVal
aSalaire As Double)
        MyBase.New(aNom, aPrenom)
        Me.Salaire = aSalaire
    End Sub
End Class
```

7.1 Co-variance et contre-variance sur les délégués

Nous allons commencer par expliciter les notions de co-variance et de contre-variance sur les délégués, notions déjà existantes dans le langage Visual basic 8.0.

Dans l'exemple présenté ci-dessus, la classe *Employe* dérive de la classe *Personne*. En Visual Basic 8.0, on peut alors écrire le bloc de code suivant :

```
// VB

Public Class VarianceDelegue

    Private Delegate Function PersonneHandler(ByVal aEmploye As Employe)
As Personne

    Private Function Licencier(ByVal aEmploye As Employe) As Employe
        Return Nothing
    End Function

    Private Function Embaucher(ByVal aPersonne As Personne) As Personne
        Return Nothing
    End Function

    Private Sub DemoVariance()
        Dim oDelegue1 As PersonneHandler = AddressOf Licencier
        Dim oDelegue2 As PersonneHandler = AddressOf Embaucher
    End Sub

End Class
```

Détaillons ce bloc de code :

- Dans un premier temps, nous déclarons un délégué nommé *PersonneHandler*. Pour rappel, il s'agit d'un type de données permettant de créer des objets pointant vers une méthode. Ce délégué permet donc de créer des objets pointant vers une méthode, acceptant un employé en paramètre et retournant une personne.
- Puis nous déclarons deux méthodes ayant des signatures différentes. La méthode *Embaucher* accepte une personne en paramètre et retourne une personne. La méthode *Licencier* accepte un employé en paramètre et retourne aussi une personne.
- Dans la méthode *DemoVariance* :
    - Nous déclarons une instance du délégué nommée *oDelegue1* pointant vers la méthode *Licencier*, bien que cette méthode ne respecte pas strictement la signature du délégué : elle retourne un objet de type *Employe* au lieu de Personne. L'utilisation de la **co-variance** permet l'écriture de cette instruction.
    - Puis nous déclarons une autre instance du délégué nommée *oDelegue2* pointant vers la méthode *Embaucher*, bien que cette méthode ne respecte pas strictement la signature du délégué : elle accepte en paramètre de type Personne et non *Employe*. L'utilisation de la **contre-variance** permet l'écriture de cette instruction.

L'utilisation de la co-variance et de la contre-variance permettent aussi d'écrire ces instructions, n'étant qu'une évolution des délégués (toujours valable à partir de Visual Basic 8.0) :

```
// VB

Dim oFunc1 As Func(Of Employe, Personne) = AddressOf Embaucher
Dim oFunc2 As Func(Of Employe, Personne) = AddressOf Licencier
```

L'objet *oFunc1* pointe vers la méthode *Embaucher*. L'objet *oFunc2* pointe vers la méthode *Licencier*.

7.1.1 La co-variance et les interfaces génériques

Dans le Framework .NET 4.0, les interfaces génériques *IEnumerable* et *IEnumerator* sont définies de la manière suivante :

```
// C#

Interface IEnumerable(Of Out T)
    Inherits IEnumerable

    Function GetEnumerator() As IEnumerator(Of T)
End Interface

Interface IEnumerator(Of Out T)
    Inherits IEnumerator, IDisposable

    ReadOnly Property Current() As T
End Interface
```

Vous remarquez l'utilisation du mot clé *out* avant le type générique, qui permet de signifier que le type *T*, pourra uniquement être utilisé comme type de retour des méthodes définies dans ces interfaces. On dit alors que cette interface est « covariante » du type *T*. Ainsi toute énumération d'objets de type *A* (*IEnumerable(Of A)*) pourra être considérée comme une énumération d'objets de type *B* (*IEnumerable(Of B)*), à condition que tout objet A puisse être converti en objet de type *B*.

Appliquons maintenant ce principe sur un cas pratique : voici un bloc de code permettant de créer une liste d'employés :

```
// VB

// Variables locales
Dim oListeEmployes As IList(Of Employe)

// Création et alimentation de la liste des employes.
oListeEmployes = New List(Of Employe)() From {
        New Employe("DURAND", "Alain", 2321.81),
        New Employe("VIRON", "Karl", 1398.22),
        New Employe("HOIN", "Pierre", 1210.09),
        New Employe("HILL", "Tony", 3211.45),
        New Employe("FROT", "Elise", 3232.9),
        New Employe("ZERA", "Laurence", 2129.98)}
```

Maintenant, en Visual Basic 9.0, si nous écrivons ces instructions :

```
// VB

Dim oListePersonnes As IEnumerable(Of Personne)
oListePersonnes = oListeEmployes.ToList()
```

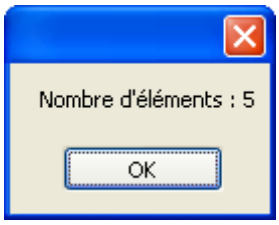
Nous obtenons le message d'erreur suivant « Argument d'instance : conversion impossible de 'System.Collections.Generic.IList(Of Test.Employe)' en 'System.Collections.Generic.IEnumerable(Of Test.Personne)' ».

Dans le langage Visual Basic 10, la co-variance permet de « convertir » une liste d'objets de type *T* en une énumération d'objets de type *T1*, si et seulement si une conversion est possible. En appliquant ce principe à notre exemple, la classe *Employe* dérivant de la classe *Personne*, il est maintenant possible de « convertir » une liste d'employés en une énumération de personnes.

Plus concrètement, cette nouveauté se révèle pratique dans le cas suivant. Voici un bloc d'instructions permettant de créer deux listes d'objets distinctes : l'une de personnes et l'autre d'employés. Puis de fusionner ces deux listes pour n'en former qu'une seule, et d'afficher le nombre d'objets qu'elle contient :

```
// VB

Dim oListePersonnes As New List(Of Personne) from {
    new Personne("RAVAILLE", "James"),
    new Personne("DOLLON", "Julien"),
    new Personne("VERGNAULT", "Laurent")}

Dim oListeEmployes As New List(Of Employe) from {
    new Employe("ASSIS-RANTES", "Laurent", 2239.34),
    new Employe("DORDOLO", "Matthieu", 1989.99)}

Dim oListeFusion = oListePersonnes.Union(oListeEmployes)
MessageBox.Show("Nombre d'éléments : " + oListeFusion.Count().ToString())
```

La co-variance est utilisée dans l'instruction *Dim oListeFusion = oListePersonnes.Union (oListeEmployes)* en permettant de convertir la liste *oListeEmployes* en *IEnumerable(Of Personne)*. Voici le résultat obtenu :



7.1.2 La contre-variance et les interfaces génériques

Dans le Framework .NET 4.0, l'interface générique *IEqualityComparer(Of T)* est définie de la manière suivante :

```
// VB

Public Interface IEqualityComparer (Of In T)
    Function Equals(x As T, y As T) As Boolean
    Function GetHashCode(obj As T) As Integer
End Interface
```

Vous remarquez l'utilisation du mot clé *in* avant le type générique, qui permet de signifier que le type *T*, pourra uniquement être utilisé comme type des paramètres des méthodes de cette interface.

Grâce à cette écriture, un objet créé à partir d'une classe implémentant l'interface *IEqualityComparer(Of Object)*, pourra être considéré comme un objet de type *IEqualityComparer(Of string)*.

Définissons alors une classe nommée *PersonneComparer*, implémentant l'interface *IEqualityComparer(Of Personne)*. Cette classe permet de déterminer si deux instances de la classe *Personne* désigne la même personne :

```
// VB

Public Class PersonneComparer
    Implements IEqualityComparer(Of Personne)

    Public Function Equals1(ByVal aPersonne1 As Personne, ByVal aPersonne2 As
Personne) As Boolean Implements
System.Collections.Generic.IEqualityComparer(Of Personne).Equals
        ' Variables locales.
        Dim bResult As Boolean

        ' Initialisation.
        bResult = True

        ' Comparaison des références des deux personnes.
        If aPersonne1 IsNot aPersonne2 Then
            ' Cas des objets NULL.
            If (aPersonne1 Is Nothing Or aPersonne2 Is Nothing) Then
                bResult = False
            Else
                ' Les deux objets représentent la même personne s'ils ont le
même nom et même prénom.
                bResult = (aPersonne1.Prenom = aPersonne2.Prenom) And
(aPersonne1.Nom = aPersonne2.Nom)
            End If
        End If

        Return bResult
    End Function
```

```
// VB

    Public Function GetHashCode1(ByVal aPersonne As Personne) As Integer
Implements System.Collections.Generic.IEqualityComparer(Of
Personne).GetHashCode
        ' Variables locales.
        Dim iResult As Integer

        ' Initialisation.
        iResult = 0

        ' On vérifie que l'objet est différent de null.
        If aPersonne IsNot Nothing Then

            ' On ontient le HashCode du prénom s'il est différent de null.
            Dim iHashCodePrenom As Integer = 0
            If aPersonne.Prenom IsNot Nothing Then
                iHashCodePrenom = aPersonne.Prenom.GetHashCode()
            End If

            ' Calcul du hashCode du nom
            Dim iHashCodeNom As Integer = 0
            If aPersonne.Nom IsNot Nothing Then
                iHashCodeNom = aPersonne.Nom.GetHashCode()
            End If

            ' Calcul du hascode de la personne à partir du nom et du prénom.
            iResult = iHashCodePrenom Xor iHashCodeNom
        End If

        Return iResult
    End Function
End Class
```

Toute classe implémentant une interface doit proposer une implémentation des membres définis dans cette interface. En implémentant l'interface générique *IEqualityComparer(Of Personne)*, nous devons donc implémenter les membres :

- *Equals* : retourne une valeur booléenne indiquant si deux objets de type *Personne* représente la même personne. Ainsi, elle retourne *True* si les deux personnes passées en paramètre pointent vers la même instance ou s'ils ont le même nom et le même prénom.
- *GetHashCode* : permet d'obtenir le *HashCode* de l'objet de type *Personne* passé en paramètre, à partir de son nom et son prénom.

Voici un bloc d'instructions permettant de créer une liste d'employés, puis de l'apurer afin d'enlever les doublons, et d'afficher le nombre d'employés obtenus :

```
// VB

Private Sub Button7_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button7.Click
    ' Création de la liste des employés.
        Dim oListeEmployes As New List(Of Employe) From {
            New Employe("ASSIS-ARANTES", "Laurent", 2239.34),
            New Employe("ASSIS-ARANTES", "Laurent", 2306.01),
            New Employe("DORDOLO", "Matthieu", 1989.99)}

    ' Suppression des doublons.
        Dim oListeEmpl As IEnumerable(Of Employe) =
    oListeEmployes.Distinct(New PersonneComparer())

    ' Affichage du nombre d'employés distincts.
    MessageBox.Show(oListeEmpl.Count().ToString() + " employés.")

End Sub
```

Pour appliquer le concept de la contre-variance présenté ci-dessus, nous avons utilisé une instance de la classe *PersonneComparer*, permettant de comparer deux personnes, afin de comparer deux employés. Cette contre-variance est applicable car la classe *Employe* dérive de la classe *Personne*.

L'exécution du code ci-dessus affiche la boite de message suivante :

8 Conclusion

Ce cours vous a présenté les principales nouveautés du langage Visual Basic 10 :

- Les propriétés simplifiées.
- Les initialiseurs de collections.
- Les instructions dynamiques.
- Les sauts d'instruction implicites.
- Les instructions Lambda multi-lignes.
- La co-variance et la contre-variance sur les classes génériques.

Ces nouveautés vous offriront plus de possibilités dans le développement de vos applications .NET avec le langage Visual Basic. Attention toutefois au typage dynamique : il est important de ne pas en abuser, au risque de créer des applications dans lesquelles des exceptions sont souvent levées…