Visual Basic 2010. Novedades del lenguaje

Luis Miguel Blanco Ancos

La llegada de Visual Studio 2010 marca también un nuevo paso en la evolución de Visual Basic. Uno de los grandes clásicos entre los lenguajes de programación desembarca con energías renovadas y un notable conjunto de nuevas características, que a buen seguro serán bienvenidas por toda la comunidad de desarrolladores. En el presente artículo realizamos una revisión de las más importantes que acompañan a esta nueva versión.

Confluencia y evolución conjunta de VB y C#

Antes de comenzar con las novedades del lenguaje, queremos mencionar un hecho especialmente destacable como es la evolución paralela de funcionalidades, que a partir de Visual Studio 2010 experimentarán y ofrecerán los dos lenguajes principales de la plataforma: VB y C#.

Desde la primera versión de .NET Framework, los equipos de desarrollo de estos dos lenguajes han procurado marcar algunas diferencias entre ambos, siendo su intención la de hacer de VB un lenguaje más atractivo al desarrollador de aplicaciones de gestión; mientras que C# se pretendía orientar a un programador cuyo objetivo fuera el desarrollo a más “bajo nivel”: componentes, servicios, etc.

Scott Wiltamuth, uno de los directores de la división de lenguajes de Visual Studio, menciona [1] que llevar estos objetivos a la práctica ha resultado más complicado de lo esperado, debido a la presencia de lo que él denomina “poderosas fuerzas de unificación”, que han propiciado un cambio de orientación hacia el desarrollo en paralelo de funcionalidades para los dos lenguajes.

Las mencionadas fuerzas de unificación se organizan en tres grupos:

- La existencia de un entorno de desarrollo integrado y bloques de construcción de aplicaciones comunes a ambos lenguajes.
- La naturaleza orientada a objeto y el sistema de tipos de los dos lenguajes.
- La existencia de una especificación común para los lenguajes (CLS), y la innovación en el desarrollo de funcionalidades externas a los propios lenguajes, tales como genéricos y LINQ.

A los tres puntos anteriores se unían las demandas de las comunidades de desarrolladores sobre las diferencias entre los lenguajes, ya que los programadores de

VB querían aquellas funcionalidades disponibles en C#, de las que VB carecía, y viceversa.

Todo ello ha propiciado el cambio de estrategia que acabamos de describir, con la clara intención de que independientemente del lenguaje que utilicemos, podamos aprovechar toda la potencia que la plataforma .NET Framework pone a nuestra disposición.

Continuación de línea implícita

Suponemos que para la inmensa mayoría de aquellos que programan con Visual Basic, resulta un fastidio tener que utilizar el carácter de guión bajo para separar en varias líneas físicas una misma línea lógica de código.

Gracias a la nueva característica de continuación de línea implícita, resulta posible obviar el guión bajo en una gran cantidad de lugares de nuestro código: paréntesis, llaves, consultas LINQ, atributos, etc. El listado 1 muestra algunos ejemplos de código en los que podemos suprimir el carácter de continuación de línea.

```
Dim lstNombres As List(Of String) = New List(Of String) From {
"José Luis",
"Elena",
"María Luisa",
"Luis Alberto",
"María"
}
'------------------------------
Dim qryNombres = From sNombre In lstNombres
   Where sNombre.Contains(
      "Luis"
   )
   Select sNombre
'------------------------------
Dim oLibro As Libro = New Libro() With {.Titulo = "Dune",
.Autor = "Frank Herbert", .Precio = 12}

Dim oXElem = <Libro>
             <Titulo>
               <%=
                  oLibro.Titulo
               %>
             </Titulo>
             <Autor>
               <%=
                  oLibro.Autor
               %>
             </Autor>
           </Libro>
'------------------------------
<
```

```
Serializable()
>
Public Class Libro
  '....
End Class
```

Listado 1.

Propiedades Auto Implementadas

Antes de la llegada de Visual Studio 2010, cada vez que en una clase escribíamos el código para una propiedad, estábamos obligados a codificar por completo sus bloques de acceso/asignación (Get/Set), a pesar de que la propiedad no necesitara una lógica especial de validación para dichas operaciones, lo cual, en el caso de propiedades sencillas, suponía una tediosa tarea de codificación.

Con Visual Basic 2010 es posible crear propiedades auto implementadas, es decir, propiedades que se declaran en una simple línea de código, sin necesidad de especificar los bloques Get/Set; con la ventaja adicional de poder asignar un valor predeterminado al mismo tiempo.

Al crear una propiedad de este modo, Visual Basic genera internamente un campo de respaldo con ámbito a nivel de clase, cuyo nombre se compone de un guión bajo y el nombre de la propiedad. Dicho campo es perfectamente accesible desde el código de la clase, aunque no es expuesto a través de IntelliSense.

Las propiedades auto implementadas sufren algunas restricciones: no pueden ser declaradas con los modificadores ReadOnly ni WriteOnly, y en el caso de que la propiedad vaya a contener un array, no podemos especificar la dimensión del mismo en la declaración, aunque sí es posible inicializarlo, como vemos en los ejemplos del listado 2.

```
Public Class Libro
  Public Property Titulo As String
  Public Property Autor As String
  Public Property Precio As Integer
  Public Property Editorial As String = "Netalia"

  ' la siguiente línea produce error de compilación
  ' Public Property DistribuidoresVarios(10) As String

  ' la siguiente línea es correcta para crear una propiedad
  ' que contenga un array
  Public Property Distribuidores As String() = New String() {
    "Distribuidor01", "Distribuidor02"}

  Public Sub VerCamposRespaldo()
    Console.WriteLine(_Titulo)
    Console.WriteLine(_Autor)
```

```
        Console.WriteLine(_Precio)
        Console.WriteLine(_Editorial)
        Console.ReadLine()
    End Sub
End Class
```

Listado 2.

Inicializadores de colecciones

La manera que hasta ahora teníamos de inicializar una colección con un conjunto de valores consistía en llamar sucesivamente a su método Add, pero Visual Basic 2010 aporta una nueva sintaxis más sucinta –que vemos en el listado 3— para este tipo de operaciones, consistente en utilizar la palabra clave From en el momento de crear la colección, seguida de una lista con los valores de inicialización encerrados entre llaves; esto produce internamente una llamada al método Add de la colección por cada uno de los elementos existentes en la lista.

```
Dim lstNombres1 As List(Of String) = New List(Of String) From {"Eva", "Ana", "María"}
Dim lstNombres2 As New List(Of String) From {"Miguel", "Fernando", "Alberto"}

Dim lstLibros As List(Of Libro) = New List(Of Libro) From {
    New Libro() With {.Titulo = "El camino", .Autor = "Miguel Delibes", .Precio = 18},
    New Libro() With {.Titulo = "El caballero de Olmedo", .Autor = "Lope de Vega", .Precio = 12}}
'-----------------------------------
Public Class Libro
    Public Property Titulo As String
    Public Property Autor As String
    Public Property Precio As Integer
End Class
```

Listado 3.

Si trabajamos con tipos propios como la clase Libro del listado 3, al crear una colección de este tipo ¿no sería estupendo poder pasar solamente los valores para las propiedades de cada objeto Libro de la colección, y que ésta se encargara de instanciar el objeto?. Esto es perfectamente posible creando un método de extensión con el nombre Add para la colección List(Of Libro), lo que permitirá la escritura de una sintaxis de inicialización mucho más simple, como vemos en el listado 4.

```
Dim lstLibros As New List(Of Libro) From {
    {"El camino", "Miguel Delibes", 18},
    {"El caballero de Olmedo", "Lope de Vega", 12}}
'-----------------------------------------------------
<Extension()>
Public Sub Add(ByVal lstLibros As List(Of Libro),
    ByVal sTitulo As String,
    ByVal sAutor As String,
    ByVal nPrecio As Integer)
    lstLibros.Add(New Libro() With {.Titulo = sTitulo,
            .Autor = sAutor, .Precio = nPrecio})
```

```
End Sub
```

Listado 4.

En el caso de que estemos desarrollando una colección propia en la que necesitemos que esté disponible esta sintaxis de inicialización, es preciso implementar la interfaz IEnumerable, o al menos que cumpla con el patrón IEnumerable, es decir, que tenga los métodos GetEnumerator y Add. Respecto al método Add, podemos crear una sobrecarga que facilite la sintaxis de inicialización para nuestra colección, o bien un método de extensión como en el caso anterior. En el listado 5 vemos un ejemplo.

```
Dim colBiblioteca1 As New Biblioteca() From {
   New Libro() With {.Titulo = "El camino", .Autor = "Miguel Delibes", .Precio = 18},
   New Libro() With {.Titulo = "El caballero de Olmedo", .Autor = "Lope de Vega", .Precio = 12}
}

Dim colBiblioteca2 As New Biblioteca() From {
   {"El camino", "Miguel Delibes", 18},
   {"El caballero de Olmedo", "Lope de Vega", 12}}
'-----------------------------------
Public Class Biblioteca
   Implements IEnumerable(Of Libro)

   Private lstLibros As List(Of Libro)

   Public Sub New()
      Me.lstLibros = New List(Of Libro)
   End Sub

   Public Sub Add(ByVal oLibro As Libro)
      lstLibros.Add(oLibro)
   End Sub

   Public Sub Add(ByVal sTitulo As String,
                  ByVal sAutor As String,
                  ByVal nPrecio As Integer)
      lstLibros.Add(New Libro() With {.Titulo = sTitulo,
                  .Autor = sAutor, .Precio = nPrecio})
   End Sub

   Public Function GetEnumerator() As System.Collections.Generic.IEnumerator(Of Libro) Implements
System.Collections.Generic.IEnumerable(Of Libro).GetEnumerator
      Return lstLibros.GetEnumerator()
   End Function
   '....
End Class
```

Listado 5.

Expresiones Lambda

Las expresiones lambda, introducidas en Visual Basic 2008, solamente permitían ser creadas utilizando una única línea de código, siendo también obligatorio que la expresión siempre devolviera un valor.

Esta restricción ha sido superada en Visual Basic 2010, donde podemos escribir expresiones lambda compuestas por varias líneas de código. Como novedad adicional, además de crear expresiones que devuelvan un valor (comportamiento habitual), podemos crear otras que no devuelvan resultado alguno (al estilo de un procedimiento Sub) utilizando un delegado de tipo Action(Of T). Ver el listado 6.

```
' expresión lambda multilinea
Dim Lambda01 As Func(Of Integer, String) =
   Function(nNumero As Integer)
      Dim nNuevoNumero As Integer
      Dim sResultado As String
      nNuevoNumero = nNumero * 7
      sResultado = "El resultado es: " & nNuevoNumero.ToString()
      Return sResultado
   End Function
Console.WriteLine(Lambda01(123))

' si no tipificamos la variable
' podemos tipificar el valor de retorno al declarar la expresión lambda
Dim Lambda02 = Function(nDiasAgregar As Double) As DateTime
         Dim dtFechaActual As DateTime = DateTime.Today
         Dim dtFechaNueva As DateTime = dtFechaActual.AddDays(nDiasAgregar)
            Return dtFechaNueva
          End Function
Console.WriteLine(Lambda02(5).ToString("dd-MMMM-yyyy"))

' expresión lambda de tipo Sub
Dim Lambda03 = Sub(sNombre As String, dtFechaNacimiento As DateTime)
            Dim sMensajeCompleto As String = sNombre &
              " nacido en " &
              dtFechaNacimiento.ToString("yyyy")
            Console.WriteLine(sMensajeCompleto)
         End Sub
Lambda03("Ernesto Naranjo", New DateTime(1970, 10, 18))

' expresión lambda de tipo Sub con declaración estricta
' usando Action(Of T)
Dim Lambda04 As Action(Of String) =
   Sub(sMensaje)
      Console.WriteLine(sMensaje)
   End Sub
Lambda04("hola mundo")
```

Listado 6.

Covarianza y contravarianza

Cuando trabajamos con tipos genéricos que mantienen una relación de herencia debemos tener en cuenta ciertas restricciones impuestas por la plataforma, de las que a priori podemos no ser conscientes, ya que asumimos que deberían funcionar por una simple cuestión de principios lógicos en los que se basa la OOP. Tomemos como ejemplo el listado 7.

```
Public Class Documento
   Public Property Texto As String
   Public Property Autor As String
End Class

Public Class Carta
   Inherits Documento

   Public Property Destinatario As String
End Class

Public Class Acta
   Inherits Documento

   Public Property DepartamentoEmisor As String
   Public Property Fecha As DateTime
End Class
'---------------------------------------
Dim ilstCartas As IList(Of Carta) = New List(Of Carta) From {
   New Carta() With {.Texto = "aaaa", .Autor = "Bea", .Destinatario = "Tom"},
   New Carta() With {.Texto = "bbbb", .Autor = "María", .Destinatario = "Ana"}
}

Dim ilstDocumentos As IList(Of Documento) = ilstCartas  ' error de ejecución
```

Listado 7.

Si el intento de asignación del tipo IList(Of Carta) sobre un tipo IList(Of Documento) no produjera un error en tiempo de ejecución, como vemos en el listado 8, podríamos reasignar a uno de los elementos de IList(Of Documento) un tipo Acta e intentar seguidamente extraerlo como un tipo Carta, lo que provocaría una ruptura en el sistema de seguridad de tipos de la plataforma.

```
ilstDocumentos(1) = New Acta() With {.Texto = "cccc", .Autor = "Ignacio",
   .DepartamentoEmisor = "Contabilidad", .Fecha = DateTime.Today}
Dim oCarta As Carta = ilstCartas(1)
```

Listado 8.

A pesar de estas restricciones, la versión 4 de .NET Framework permite la conversión implícita o varianza entre un cierto número de interfaces en sus dos modalidades: covarianza y contravarianza.

La covarianza permite asignar a un tipo como IEnumerable(Of T), en el que T esté situado en un nivel superior de la jerarquía, un tipo IEnumerable(Of T) cuyo T sea un

descendiente, sin que se produzca error. Esto es posible debido a que la interfaz está definida dentro de la plataforma como IEnumerable(Of Out T), lo que indica que el tipo T solamente podrá ser manipulado en operaciones "de salida". De esta manera es posible escribir el código del listado 9.

```
Dim ienumCarta As IEnumerable(Of Carta) = New List(Of Carta) From {
   New Carta() With {.Texto = "aaaa", .Autor = "Bea", .Destinatario = "Tom"},
   New Carta() With {.Texto = "bbbb", .Autor = "María", .Destinatario = "Ana"}
}

Dim ienumDocumento As IEnumerable(Of Documento) = ienumCarta
```

Listado 9.

La contravarianza produce, en cierto sentido, un efecto opuesto al anterior, ya que permite, por ejemplo, que en un tipo derivado T del que hemos creado una colección List(Of T), una operación/método como Sort, sea llevada a cabo por un tipo superior en la jerarquía de clases de T mediante la interfaz IComparer(Of T). Ello es posible porque la interfaz está definida dentro de la plataforma como IComparer(Of In T), lo que indica que T solamente podrá ser manipulado en operaciones "de entrada". El listado 10 muestra un ejemplo de este caso.

```
Public Class ComparadorDocumentos
   Implements IComparer(Of Documento)

   Public Function Compare(ByVal x As Documento, ByVal y As Documento) As Integer Implements
System.Collections.Generic.IComparer(Of Documento).Compare
      '....
   End Function
End Class
'--------------------------------
Dim lstCartas As List(Of Carta) = New List(Of Carta) From {
   New Carta With {.Texto = "aaaa", .Autor = "Sole", .Destinatario = "Bob"},
   New Carta With {.Texto = "zzzz", .Autor = "Ana", .Destinatario = "Alex"},
   New Carta With {.Texto = "hhhh", .Autor = "Marta", .Destinatario = "David"}
}

Dim icompDocumentos As IComparer(Of Documento) = New ComparadorDocumentos()
lstCartas.Sort(icompDocumentos)
```

Listado 10.

Recomendamos la consulta del artículo [2] que sobre este tema publicaron los compañeros de grupo WEBOO, para una descripción en mayor profundidad de este nuevo aspecto de la plataforma.

Otras características adicionales

La opción /langversion del compilador nos permite especificar la versión del lenguaje con la que se compilará nuestro código. También es posible acceder a objetos creados

en lenguajes dinámicos como IronPython y IronRuby, así como el soporte para la equivalencia de tipos, mediante el que podemos instalar una aplicación que contiene incrustada la información de tipos, en vez de importarla desde un ensamblado PIA. En el centro de recursos del lenguaje [3], encontraremos información ampliada y todo tipo de recursos para sacar el mayor partido a todas las nuevas funcionalidades

Concluimos

Con toda seguridad, la oferta de funcionalidades del nuevo Visual Basic 2010 será bien recibida por todos los desarrolladores que trabajan con este lenguaje. Esperamos que este rápido repaso sirva para conseguir una mejor toma de contacto con las novedades aquí presentadas.

REFERENCIAS

[1] Wiltamuth, Scott. VB and C# Coevolution.

http://blogs.msdn.com/scottwil/archive/2010/03/09/vb-and-c-coevolution.aspx

[2] Katrib, Miguel y del Valle, Mario. La danza de las varianzas en C# 4.0. En dotNetManía nº 62.

[3] Recursos de Visual Basic 2010.

http://msdn.microsoft.com/es-es/vbasic/dd819153.aspx