Visual Basic 2010

Novedades del lenguaje

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 este artículo hacemos una revisión de las más importantes novedades que se presentan en esta nueva versión.

Luis Miguel Blanco

Arquitecto de software en Alhambra-Eidos

Confluencia y evolución conjunta de VB y C#

Antes de comenzar con las novedades del lenguaje, queremos mencionar un hecho especialmente destacable: la evolución paralela de funcionalidades que a partir de Visual Studio 2010 experimentarán y ofrecerán los dos principales lenguajes de la plataforma: Visual Basic 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 Visual Basic un lenguaje más atractivo al desarrollador de aplicaciones de gestión, mientras que C# se pretendía dirigir a los programadores más orientados hacia 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 resultó 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, como:

- La existencia de un entorno de desarrollo integrado y bloques de construcción de aplicaciones comunes a ambos lenguajes.
- La naturaleza orientada a objetos y el sistema de tipos común a los dos lenguajes.
- El hecho de que las principales áreas de innovación presente y futura en el desarrollo de los lenguajes se reflejan en partes "exteriores" de los mismos, como ocurre en el caso de LINQ.

A los elementos anteriores hay que añadir las demandas de las comunidades de desarrolladores, 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 mencionar, que tiene el claro objetivo de que, independientemente del lenguaje que utilicemos, podamos aprovechar toda la potencia que .NET Framework pone a nuestra disposición.

Propiedades auto-implementadas

Antes de la llegada de **Visual Basic 2010** (o Visual Basic 10, como también se denomina), cada vez que en una clase se definía una propiedad, estábamos obligados a codificar por completo sus bloques de acceso/asignación (`Get`/`Set`), aún cuando la propiedad no necesitara una lógica especial para dichas operaciones. A partir de esta nueva versión, es posible crear propiedades auto-implementadas, que se declaran en una simple línea de código sin necesidad de especificar los bloques `Get`/`Set`; con la ventaja adicional de poder asignar al mismo tiempo un valor predeterminado.

Al crear una propiedad de este modo, el compilador genera internamente un campo de respaldo con ámbito 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 1.

```
Public Class Libro
  Public Property Título As String
  Public Property Autor As String
  Public Property Precio As Decimal
  Public Property Editorial
       As String = "Netalia"

  ' La siguiente línea produce un error de compilación:
  ' Public Property Distribuidores Varios(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(_Título)
      Console.WriteLine(_Autor)
      Console.WriteLine(_Precio)
      Console.WriteLine(_Editorial)
      Console.ReadLine()
  End Sub
End Class
```

Listado 1. Propiedades auto-implementadas

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 para esta tarea, 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; internamente, el compilador generará una llamada al método **Add** de la colección por cada uno de los elementos existentes en la lista. El listado 2 presenta un ejemplo basado en la clase **Libro** del listado 1.

Pero, al inicializar colecciones como la del listado 2, ¿no sería estupendo poder pasar solamente los valores para las propiedades de cada objeto de la colección, y que ésta se encargara de instanciar los objetos? Esto es

```
Dim lstLibros1 As List(Of Libro) = New List(Of Libro) From {
  New Libro() With {
    .Título = "El camino", .Autor = "Miguel Delibes",
    .Precio = 18.00D, .Editorial = "Ediciones Destino"
  },
  New Libro() With {
    .Título = "El caballero de Olmedo", .Autor = "Lope de Vega",
    .Precio = 11.50D, .Editorial = "Vicens-Vives"
  }
}
```

Listado 2. Inicializadores de colecciones

```
Dim lstLibros2 As New List(Of Libro) From {
    {"El camino", "Miguel Delibes", 18.00D, "Ediciones Destino"},
    {"El caballero de Olmedo", "Lope de Vega", 11.50D, "Vicens-Vives"} }
'_____________________-
<Extension()>
Public Sub Add(ByVal lstLibros As List(Of Libro),
               ByVal sTítulo As String,
               ByVal sAutor As String,
               ByVal dPrecio As Decimal,
               ByVal sEditorial As String)
    lstLibros.Add(New Libro() With { .Título = sTítulo, .Autor = sAutor,
                                     .Precio = dPrecio, .Editorial = sEditorial })
End Sub
```

Listado 3

perfectamente posible creando un método de extensión con el nombre **Add** para la colección **List(Of Libro)**, lo que hará posible utilizar una sintaxis de inicialización mucho más simple, como muestra el listado 3.

En el caso de que estemos desarrollando una colección propia en la que deseemos que esté disponible esta sintaxis de inicialización, es preciso implementar la interfaz **IEnumerable**, o al menos cumplir con el patrón **IEnumerable**, es decir, implementar métodos **GetEnumerator** y **Add**. Con 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 4 vemos un ejemplo.

Continuación de línea implícita

Suponemos que para la inmensa mayoría de aquellos que programan con Visual Basic siempre ha resultado un fastidio tener que utilizar el carácter de guión bajo (subrayado) para separar en varias líneas físicas una misma línea lógica de código. Gracias a la nueva característica de continuación de línea implícita en Visual Basic 2010, ahora será posible obviar el guión bajo al continuar una instrucción en la línea siguiente en una gran cantidad de lugares de nuestro código. La especificación del lenguaje [3] detalla todas las situaciones en las que el carácter de continuación puede omitirse. Los listados 1-4 incluyen múltiples ejemplos de continuaciones de línea implícitas.

```
Dim colBiblioteca1 As New Biblioteca() From {
    New Libro() With {
        .Título = "El camino", .Autor = "Miguel Delibes",
        .Precio = 18.00D, .Editorial = "Ediciones Destino" },
    New Libro() With {
        .Título = "El caballero de Olmedo", .Autor = "Lope de Vega",
        .Precio = 11.50D, .Editorial = "Vicens-Vives" }
}

Dim colBiblioteca2 As New Biblioteca() From {
    {"El camino", "Miguel Delibes", 18.00D, "Ediciones Destino"},
    {"El caballero de Olmedo", "Lope de Vega", 11.50D, "Vicens-Vives"} }
'_____________-
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 sTítulo As String,
                   ByVal sAutor As String,
                   ByVal dPrecio As Integer
                   ByVal sEditorial As String)
        lstLibros.Add(New Libro() With {
            .Título = sTítulo, .Autor = sAutor,
            .Precio = dPrecio, .Editorial = sEditorial })
    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 4

Expresiones lambda

Las expresiones lambda fueron introducidas en Visual Basic 2008, pero entonces solo podían constar de una única línea de código, siendo también obligatorio que la expresión 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)**, como se muestra en el listado 5.

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

```
' Expresión lambda multi-línea
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))

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 5

```
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 = "AAA", .Autor = "Bea",
       .Destinatario = "Tom"},
  New Carta() With {.Texto = "BBB", .Autor = "María",
       .Destinatario = "Ana"}
}

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

Listado 6

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 6.

Si el intento de asignación de un objeto del tipo **IList(Of Carta)** a una variable del tipo **IList(Of Documento)** no produjera un error en tiempo de ejecución, podríamos reasignar a uno de los elementos de **IList(Of Documento)** un tipo **Acta** e intentar seguidamente extraerlo como un tipo **Carta**, como vemos en el listado 7, 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 7

La versión 4 de .NET Framework levanta en ciertos casos estas restricciones, permitiendo la conversión implícita o varianza entre ciertos tipos de interfaces en dos modalidades diferentes: **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 valor de tipo **IEnumerable(Of T)** cuyo **T** sea un descendiente, sin que se produzca error. El compilador acepta esto debido a que dicha 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 8.

Por otra parte, 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 9 muestra un ejemplo de este caso.

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

Listado 8

```
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 = "XXX", .Autor = "Sole", .Destinatario = "Bob"},
    New Carta With {.Texto = "YYY", .Autor = "Ana", .Destinatario = "Alex"},
    New Carta With {.Texto = "ZZZ", .Autor = "Marta", .Destinatario = "David"}
}
Dim icompDocumentos As IComparer(Of Documento) = New ComparadorDocumentos()
lstCartas.Sort(icompDocumentos)
```

Listado 9

Para una descripción en mayor profundidad de esta nueva característica de Visual Basic, también presente en C#, recomendamos la consulta del artículo sobre este tema que publicó recientemente **dotNetManía** [2].

Otras novedades adicionales

Otras novedades adicionales que incorpora Visual Basic 2010 son las siguientes:

- La opción **/langversion** del compilador nos permite especificar la versión del lenguaje con la que se compilará nuestro código.
- Haciendo uso del enlace tardío del que siempre ha gozado Visual Basic, ahora es posible acceder a objetos creados en lenguajes dinámicos como Iron Python e Iron Ruby.
- Al desarrollar aplicaciones que acceden a objetos COM, tales como los componentes de Office, ahora es posible incrustar directamente en nuestro ensamblado la información de tipos asociada a los objetos COM, en lugar de tener que importarla desde el ensamblado PIA suministrado por el fabricante.

En el centro de recursos del lenguaje [3], encontrará información ampliada y todo tipo de recursos para sacarle el mayor partido a todas las nuevas características.

Conclusiones

Con toda seguridad, la oferta de funcionalidades del nuevo Visual Basic 2010 será bien recibida por todos los desarrolladores que utilizan 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. En **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, septiembre de 2009.

[3] Centro de desarrollo de Visual Basic en MSDN: **http://msdn.microsoft.com/es-ES/vbasic**.