Guillermo "Guille" Som

Novedades de Visual Basic 2005

Visual Basic 2005 es, sin lugar a dudas, el lenguaje de los incluidos en la nueva versión de Visual Studio que más novedades presenta, tanto en novedades del propio lenguaje como en las características ofrecidas por el entorno de desarrollo. En este artículo comentaremos esas novedades de la forma más clara posible, con idea de que el lector comprenda por qué Visual Basic 2005 se convierte en la elección más productiva de todas las ofrecidas por Visual Studio 2005.

» **Las novedades que** encontraremos en la nueva versión de Visual Basic, (que a partir de esta versión deja de llamarse .NET para pasar a usar la versión de Visual Studio), las podemos dividir en tres partes:

- Las novedades ofrecidas por el propio lenguaje.
- Las novedades ofrecidas por .NET Framework 2.0.
- Las novedades ofrecidas por el entorno de desarrollo (o editor).

Veamos cada una de estas novedades, aunque en algunas no profundizaremos demasiado, entre otras cosas porque necesitaríamos algo más de las páginas de este especial, y porque otras se tratan en este mismo número o han sido comentadas en números anteriores.

Consideraciones previas

Antes de comentar las novedades propias del lenguaje y del entorno de desarrollo de Visual Basic 2005, quisiéramos indicar unas recomendaciones que cualquier desarrollador que quiera tomarse en serio la escritura de código con Visual Basic.

Option Strict On: Ayuda a escribir mejor código

La primera de ellas es que seleccione la opción de comprobación estricta del código, (`Option Strict On`), ya que esta opción, a pesar de lo que muchos piensan, nos ayudará a evitar muchos errores antes de compilar el código, ya que aprovecha una de las características exclusivas del entorno de desarrollo (o editor) de Visual Basic: la compilación en segundo plano mientras escribimos.

Para que esta opción esté activada en todos los proyectos que creemos, debemos seleccionarla mediante el menú "Herramientas>Opciones>Proyectos y Soluciones> Valores predeterminados de VB", tal como se muestra en la figura 1.

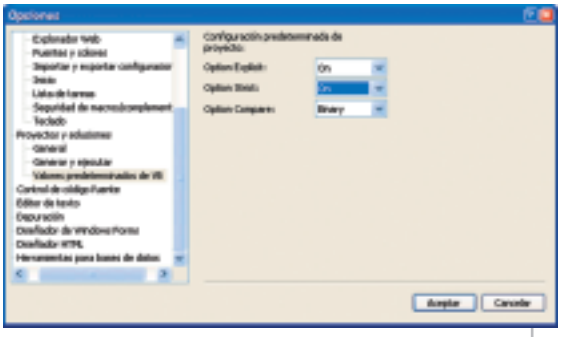

Figura 1.Seleccionar la comprobación estricta del código

¿Dónde están mis ficheros?

La segunda es indicar que, debido a cómo se manejan los proyectos y las configuraciones de los mismos en Visual Studio 2005, hay ciertos ficheros, que aparentemente han desaparecido, entre ellos el fichero `AssemblyInfo`, las declaraciones de los controles añadidos a los formularios y las referencias a los ensamblados externos: referencias y referencias Web.

Realmente estos ficheros no han desaparecido o en el caso de la definición de los controles de los formularios, estas están en un fichero aparte, de forma que aprovecha una de las novedades de VB2005: las *clases parciales*.

Para poder mostrar todos estos elementos que antes teníamos en el explorador de soluciones, tendremos que pulsar en el segundo botón de la barra de herramientas del explorador de soluciones, (ver figura 2).

Guillermo "Guille" Som
es Microsoft MVP de Visual Basic desde 1997. Es redactor de dotNetManía, miembro de *Ineta Speakers Bureau Latin America*, mentor de *Solid Quality Learning* Iberoamérica y autor del libro *Manual Imprescindible de Visual Basic .NET*.
http://www.elguille.info

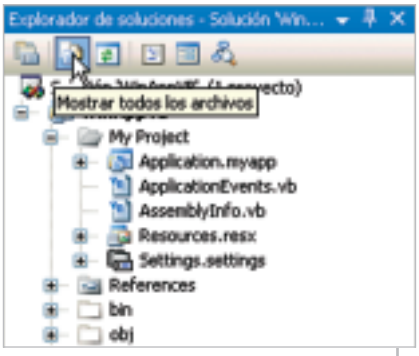

Figura 2. Mostrar todos los archivos de un proyecto de VB2005

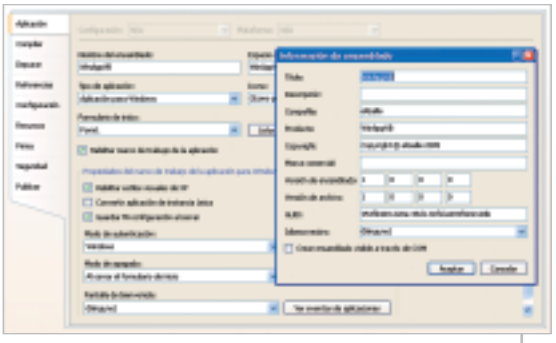

Figura 3. Propiedades del proyecto e información del ensamblado

Como vemos en la figura 2, ese botón tiene también la misma funcionalidad de las versiones anteriores de mostrar todos los ficheros y directorios de la carpeta en la que tenemos nuestro proyecto. Además nos muestra algunos de los ficheros con un símbolo más (+) que nos indica que hay algo más. Aunque ese "algo más" que hay oculto normalmente está generado por el propio entorno, por tanto, no deberíamos modificarlos directamente.

Propiedades del proyecto: Mejor organizado y más asequible

Como hemos comentado, muchas de las opciones que el IDE de Visual Basic 2005 nos oculta es porque podemos asignarlas mediante ventanas de propiedades, como puede ser la de las propiedades del proyecto, que ahora tiene una nueva presentación por fichas que son más fáciles de manejar y que recopila otras configuraciones que anteriormente estaban disponibles en otros menús o en ficheros de configuración.

Por ejemplo, los valores contenidos en el fichero `AssemblyInfo` están asequibles desde un cuadro de diálogo que se accede desde la ficha "Aplicación", tal como podemos ver en la figura 3, en la que además podemos ver el resto de fichas de configuración, entre ellas la de referencias.

Para acceder a esta ventana de configuración, podemos hacerlo de la misma forma que en versiones anteriores, seleccionando el proyecto y bien usando el botón secundario del ratón o bien desde el menú "Proyecto", seleccionaremos la opción "Propiedades", con Visual Basic 2005 también podemos hacer doble clic en el elemento "My Project" del explorador de soluciones (ver figura 2).

Novedades en el lenguaje

Para poder ofrecer mayor funcionalidad, el propio lenguaje de Visual Basic se ha visto mejorado con nuevas instrucciones, de forma que no sea necesario acudir a la librería de clases de .NET para poder disfrutar de nuevas características. Para ser justos, debemos decir que muchas de las novedades incluidas en el propio lenguaje de Visual Basic 2005 son características que ya incluía su compañero de viaje C#; esto lo pone a un nivel que antes no tenía, y por lo que ha sido muy criticado e incluso etiquetado como "lenguaje de juguete". A partir de esta versión de Visual Basic, si bien aún no incluye todas las características que muchos hemos solicitado, podemos afirmar que quién no saque el máximo rendimiento a este lenguaje, no será un problema propio del lenguaje, sino una falta de visión de los desarrolladores de versiones anteriores que aún no acaban de adaptarse. Confiamos que este artículo motive a aquellos que no se deciden a dar el salto final.

Compatibilidad con el CLS

Debido a que algunas de las novedades que incluye Visual Basic 2005 son características que no están definidas en el CLS (*Common Language Specifications*), en esta versión el atributo que indica que el código usado en nuestra aplicación es compatible con esas especificaciones (*CLSCompliant*) ya no se incluye por defecto en el fichero `AssemblyInfo`, cosa que antes siempre ocurría, debido a que el código generado por todas las versiones anteriores a la 2005 eran totalmente compatibles con las especificaciones comunes del lenguaje, por la sencilla razón de que no disponía características que lo hicieran incompatible. Por tanto, si queremos que el compilador compruebe dicha compatibilidad con el CLS, debemos añadir la siguiente línea de código al mencionado fichero:

```
<Assembly: CLSCompliant(True)>
```

Esto hará que el compilador nos advierta de esa incompatibilidad en nuestro código. Aunque no debemos asustarnos por ser "incompatibles", ya que esto solamente afectará a los proyectos de tipo *Librería de clases* (DLL) que queramos usar con otros lenguajes de .NET, y si ese lenguaje es C#, podemos tener la certeza de que nuestra DLL será compatible, ya que las características del lenguaje que hacen a VB2005 incompatible con el CLS son las que han tomado "prestadas" de C#.

Tipos numéricos sin signo

En toda su historia, el lenguaje BASIC nunca ha tenido tipos numéricos sin signo, salvo el tipo `Byte`. En esta versión se incluyen todos los que actualmente existen en la plataforma .NET: `Uint16`, `UInt32` y `UInt64`. Lo novedoso es que se han añadido palabras clave al lenguaje para representar a esos tipos de datos: `UShort`, `UInteger` y `ULong` respectivamente. También se ha incluido el tipo `SByte` que es un tipo de datos entero de 8 bits "con signo".

Todos estos tipos de datos no son compatibles con las especificaciones de .NET, por tanto, debemos tener cuidado a la hora de usarlos como valor devuelto por una función o propiedad o como tipo de datos usado como parámetro de cualquier método.

Para mejorar la lectura, sobre todo cuando trabajamos con constantes

numéricas de estos nuevos tipos de datos, podemos usar los nuevos sufijos numéricos (o caracteres de tipos) que se han añadido para dar soporte a estos nuevos tipos. Estos sufijos empiezan con la letra `U` y pueden ser: `US` para el tipo `UShort`, `UI` para el tipo `UInteger` y `UL` para el tipo `ULong`.

La forma de usarlo sería como se muestra a continuación:

```
Dim us1 As UShort = 3US
Dim ui1 As UInteger = 4UI
Dim ul1 As ULong = 5UL
```

Instrucción Continue

Esta instrucción nos permitirá continuar un bucle sin necesidad de tener que alcanzar el código que hace que se repita. Y su utilidad está en que nos evitará usar condiciones para que en la repetición del bucle no "pase" por una parte del código. Por ejemplo, si tenemos un bucle `For` y parte del código contenido en dicho bucle no siempre debe ejecutarse, lo que hasta ahora hacemos es algo como esto:

```
For...
  Código que siempre se ejecuta
  If <condición> Then
    Código a ejecutar solo si se
    cumple la condición
  End If
Next
```

Si usamos la instrucción `Continue`, el seudo-código quedaría de esta forma:

```
For...
  Código que siempre se ejecuta
  If Not <condición> Then Continue For
  Código a ejecutar solo si se
  cumple la condición
Next
```

NOTA

Esto mismo, en los viejos tiempos del `GoTo`, se solucionaba dando un salto desde el `IF` a la línea en la que está el `Next`, pero... ¿quién usa ya el `GoTo`? Nadie, pero aún así, en Visual Basic 2005 se sigue soportando esta instrucción e incluso utilizando *IntelliSense* para mostrar la línea a la que se realizará el salto. ¡Así nos va!

Visual Basic se ha visto mejorado con nuevas instrucciones, de forma que no sea necesario acudir a la librería de clases de .NET para poder disfrutar de nuevas características

La instrucción `Continue` se puede usar con cualquiera de los bucles soportados por Visual Basic: `Do`/`Loop`, `While` y `For`/`Next`, usándose de la misma forma que la instrucción `Exit`, es decir, para continuar un bucle `Do`/`Loop` usaremos `Continue Do`, y para continuar uno del tipo `While` usaremos `Continue While`.

El operador IsNot

Con este operador las condiciones usadas para comprobar si un tipo por referencia no es de un tipo determinado resultarán más fáciles de escribir y consecuentemente el código resultante será más legible. Hasta ahora si, por ejemplo, queríamos comprobar si una variable no contenía un valor nulo, lo hacíamos así:

```
If Not variable Is Nothing Then ...
```

Usando este operador, el código anterior quedaría de esta forma:

```
If variable IsNot Nothing Then...
```

Como podemos comprobar, resulta más legible esta segunda forma.

Este operador sólo podemos usarlo con variables y tipos por referencia. Ya que para los tipos por valor podemos usar el operador distinto de (`<>`), que sería el equivalente.

La instrucción Using

Esta es una instrucción de bloque, es decir, el contenido estará definido por la instrucción `Using` y acabará con `End Using`. Después de `Using` indicaremos uno o varios objetos que utilizarán algún recurso del sistema. Cuando acabe el código contenido en el bloque, ese recurso se liberará llamando al método `Dispose` del objeto u objetos indicados después de `Using`. Esto nos garantiza que siempre se llamará al método `Dispose`, con la ventaja añadida de que esa llamada al método que liberará los recursos se hará incluso si se produce algún error que no tengamos controlado. `Using` sería equivalente a usar un `Try`/`Finally`, de forma que en el bloque `Finally` llamemos al método `Dispose` del objeto. Por tanto, los objetos que podemos usar con esta instrucción deben implementar la interfaz `IDisposable`, que es la que garantiza que dicho objeto tenga ese método.

La forma de usar esta instrucción es:

```
Using variable As New Recurso
   Código a ejecutar
End Using
```

El ámbito de la variable declarada con la instrucción `Using` será el propio bloque de código.

En caso de que indiquemos más de un objeto con la misma instrucción, los tendremos que separa con comas:

```
Using variable As Recurso, variable2
                As Recurso2, variable3
  Código a ejecutar
End Using
```

En este caso, definimos dos variables junto a la instrucción `Using`, además de usar otra variable definida anteriormente.

Declaración explícita del índice inferior de los arrays

Los arrays de .NET, y por extensión los arrays de Visual Basic 2005, a diferencia de lo que ocurría en Visual Basic 6,0 y anteriores, siempre tienen el valor cero como índice inferior. Esto en principio no supone ningún problema, al menos si hemos "cambiado el chip" que teníamos al usar los arrays de VB6, ya que en esa

versión podíamos indicar un rango de índices de la forma `<valor inferior> To <valor superior>`. Pero al igual que ocurría en VB6, también podemos declarar arrays sin indicar ese rango de valores, el problema es que al usar esa forma de declarar, en la que sólo se indica un valor, es que dicho valor representa el índice superior del array, de forma que si tenemos esta declaración:

```
Dim nombres(5) As String
```

Aparentemente estamos declarando un array con cinco elementos, pero en realidad son seis los elementos: desde 0 hasta 5, ambos inclusive.

Para solventar este tipo de declaraciones, que puede ser confusa para los desarrolladores de C++ o C#, en esta nueva versión de Visual Basic se ha añadido la instrucción (por llamarla de alguna forma), `0 To`, la cual se podrá usar en la declaración de los arrays para que quede más claro que realmente estamos definiendo un array con elementos desde cero hasta el valor indicado después de `To`, usando el ejemplo anterior, ahora quedaría de esta forma:

```
Dim nombres(0 To 5) As String
```

Con lo que podemos comprobar que se consigue mayor claridad.

Sobrecarga de operadores aritméticos y conversiones personalizadas

Esta es otra de las características que Visual Basic ha heredado de los lenguajes de la familia C. Con la primera podemos crear nuestras propias versiones de los operadores aritméticos y lógicos, de forma que podamos definir el comportamiento que tendrá, por ejemplo, el operador suma (+) en una clase o estructura que nosotros definamos; la segunda nos permite indicar cómo se realizarán las conversiones de nuestros tipos con respecto a otros tipos de datos, ya sean definidos en la propia librería de clases de .NET Framework o con otros tipos. Del tema de la sobrecarga de operadores y conversiones, desde el punto de vista de un desarrollador de C#, ya lo vimos en el número 9 de **dotNetManía**, en ese artículo, además de explicar todo lo relacionado con la sobrecarga y conversiones, también se indican algunos consejos de "buen uso". Y cómo no tenemos todo el espacio que sería necesario, veremos sólo unos ejemplos de cómo hacer eso mismo desde Visual Basic 2005.

Definir una sobrecarga

Lo primero que debemos saber es que los operadores que nosotros podemos definir en realidad son funciones estáticas o compartidas, es decir, son métodos que siempre están disponibles en el tipo en el que lo definimos y siempre devuelven un valor. Pero, en lugar de usar la instrucción `Function` para definirlas, usaremos la nueva instrucción `Operator` seguida del operador que queremos sobrecargar. En el fuente 1 tenemos la declaración de una estructura en la que hemos sobrecargado el operador de la suma.

```
Public Structure Punto
  Public X As Integer
  Public Y As Integer

  Public Sub New(ByVal x As Integer, ByVal y As Integer)
    Me.X = x
    Me.Y = y
  End Sub

  ' Sobrecarga del operador +
  Public Shared Operator +(ByVal p1 As Punto, ByVal p2 As Punto) As Punto
    Return New Punto(p1.X + p2.X, p1.Y + p2.Y)
  End Operator
End Structure
```

Fuente 1. Definición de la estructura Punto con sobrecarga del operador +

Como podemos comprobar, la sintaxis usada es:

```
Public Shared Operator +
      (expresión izquierda,
       expresión derecha) As Punto
```

`Public Shared` es porque los operadores siempre estarán disponibles y deben pertenecer al propio tipo de datos.

`Operator +` indica que estamos sobrecargando un operador, en este caso el de la suma.

Los dos parámetros usados representarán a la expresión izquierda y derecha respectivamente; dichas expresiones pueden ser de cualquier tipo, pero siempre deben representar a un valor de tipo `Punto`.

El valor devuelto es del tipo `Punto`, de forma que se genera un nuevo `Punto` con el resultado de sumar los dos indicados como argumentos del operador.

La decisión de qué es lo que ocurre cuando se suman dos valores de este tipo, la tomamos nosotros y es el código que utilizaremos dentro de la declaración del operador. En este caso, creamos un nuevo objeto a partir del resultado de sumar los valores de las propiedades `X` e `Y`, y es ese nuevo objeto el que devolvemos.

Con el resto de operadores haríamos lo mismo.

Y para usar este operador sobrecargado, lo haríamos igual que con cualquier otro tipo de datos definido previamente en las clases de punto NET, tal como vemos en el fuente 2.

```
Dim p1 As New Punto(10, 15)
Dim p2 As New Punto(22, 33)
Dim p3 As Punto
p3 = p1 + p2
```

Fuente 2. Uso del operador sobrecargado del tipo Punto

Una cosa importante que debemos saber es que solamente podremos usar las sobrecargas que hemos definido, en este caso solo tenemos creada una sobrecarga de la suma de dos valores de tipo `Punto`, por tanto si pretendemos hacer esto:

```
p3 = p1 + 15
```

Se producirá un error, por la sencilla razón de que no hemos definido qué es lo que tenemos que hacer para sumar un `Punto` con un entero.

La primera solución será definir esa sobrecarga, es decir, el argumento de la izquierda de tipo `Punto` y el de la derecha de tipo `Integer`:

```
Public Shared Operator +
    (ByVal p1 As Punto,
     ByVal i As Integer) As Punto
  Return New Punto(p1.X + i, p1.Y)
End Operator
```

Mediante la sobrecarga de operadores podemos definir el comportamiento de nuestros tipos al usar los operadores aritméticos y lógicos

El problema es que con esta sobrecarga sólo podemos hacer que el código anterior funcione, pero este otro volverá a fallar:

```
p3 = 25 + p1
```

Porque no hemos definido la sobrecarga de la suma de un entero y un tipo **Punto** (en ese orden). Por tanto, la solución será crear una nueva sobrecarga del operador en la que se indiquen los parámetros en ese mismo orden.

¿Y si quisiéramos definir sobrecargas para más tipos de datos? Pues lo mismo... tendríamos que definir dos sobrecargas para cada uno de los tipos de datos que queramos soportar, cada una de ellas definiendo el tipo en el "lado" correcto de la operación.

Tedioso ¿verdad? Efectivamente hacer esto sería muy "cansado", pero afortunadamente podemos aprovechar una característica del compilador de Visual Basic conocida como *promoción de tipos*. La promoción de tipos no es ni más ni menos que la conversión "automática" de un tipo en otro diferente. Por ejemplo, si tenemos una variable de tipo **Short** (entero de 16 bits) y queremos guardar su valor en otra de tipo **Integer** (entero de 32 bits), no tenemos que hacer ningún tipo de conversión, ya que el compilador "sabe" cómo convertir un valor **Short** en un **Integer**. Y esta sabiduría la podemos aprovechar nosotros usando otra de las nuevas características de Visual Basic 2005: la conversión personalizada de tipos.

Definir conversiones de datos

De la misma forma que podemos sobrecargar un operador, podemos sobrecargar la conversión de datos, particularmente la instrucción **CType**, de forma que podamos indicar cómo actuará el compilador cuando se encuentre con una conversión de cualquier tipo al nuestro o del nuestro a otro tipo.

Como sabemos, existen dos tipos de conversiones: las *implícitas*, también llamadas de *expansión* o *promoción*, es decir, las que usamos sin necesidad de hacer ninguna conversión de datos, por ejemplo, cuando convertimos un valor **Short** en uno de tipo **Integer**, estamos "promocionando" el valor entero de 16 bits en otro de 32 bits; el otro tipo de conversión es la llamada de *reducción* y es el tipo de conversión que hay que hacer explícitamente, es decir, debemos indicar que queremos convertir un tipo en otro diferente. Este segundo tipo de conversiones se usan cuando se puede producir una pérdida de información, por ejemplo, si queremos guardar en una variable de 16 bits el contenido de una de 32 bits, es muy posible que se produzca pérdida de información, ya que el valor de 32 bits posiblemente no pueda convertirse en otro de menos capacidad.

Sabiendo esto, podemos afirmar que las conversiones implícitas se harán siempre que esa conversión no produzca pérdida de información y para realizarla no tenemos que indicar ninguna conversión. Por otra parte, las conversiones explícitas, que son las que tenemos que indicar expresamente que nuestra intención es hacer una conversión entre dos tipos de datos, las usaremos cuando se pueda producir alguna pérdida de información y/o se pueda producir alguna excepción.

En Visual Basic las conversiones implícitas se definen usando la instrucción **Widening** en la declaración del operador **CType**:

```
Public Shared Widening Operator CType(
      ByVal x As Integer) As Punto
      Return New Punto(x, 0)
End Operator
```

En este caso estamos declarando una conversión implícita de **Integer** a **Punto**, con lo cual podemos hacer algo como esto:

```
Dim p4 As Punto = 15
```

Pero si lo que pretendemos es convertir un tipo **Punto** a entero, podíamos devolver el contenido de la propiedad **X**; en este caso es conveniente definirla como una conversión explícita, ya que perderemos parte del contenido del punto. La declaración quedaría de esta forma –en el que usamos la instrucción **Narrowing**, para indicar que estamos "reduciendo" el valor de un tipo **Punto** a uno entero–:

```
Public Shared Narrowing Operator CType(
            ByVal p1 As Punto) As Integer
    Return p1.X '+ p1.Y
End Operator
```

Y podemos usarla haciendo una conversión explícita:

```
Dim i As Integer = CType(p3, Integer)
Console.WriteLine(
    "valor de CType(p3, Integer) = {0}", i)
```

Como lo que estamos haciendo con ese **CType** es convertir la variable **p3** en un entero, podemos usar la función de conversión **CInt** en su lugar, obteniendo el mismo resultado:

```
Dim i As Integer = CInt(p3)
```

NOTA

Cuando hacemos sobrecargas de operadores o conversiones en nuestros propios tipos, al menos uno de los parámetros o el tipo devuelto debe ser del mismo que el tipo en el que se declaran, esto es para que no podamos definir operadores o conversiones de tipos de los que no tengamos la declaración.

Conversiones automáticas

Como hemos comentado anteriormente, el compilador de Visual Basic ya sabe cómo hacer ciertas conversiones implícitas, por tanto, si le "enseñamos" cómo hacer una conversión del tipo **Punto** a un entero de 32 bits, Visual Basic sabrá cómo convertir "automáticamente" un **Punto** a un entero de menor capacidad:

```
Dim s As Short = 12
Dim p5 As Punto = s
Console.WriteLine("valor de p5 = {0}", p5)
```

Pero si queremos asignar un valor de tipo **Long**, tendremos que indicarlo explícitamente, y el compilador lo que hará será convertir ese valor de **Long** a **Integer** y posteriormente utilizar la sobrecarga que asigna un **Integer** a un **Punto**. El código que tendríamos que usar (tal y como tenemos las sobrecargas) sería el siguiente:

```
Dim l As Long = 123456789012345678901L
Dim p6 As Punto = CType(l, Punto)
Console.WriteLine("valor de p6 = {0}", p6)
```

El problema surgirá, como en este caso, si el contenido de la variable **l** es mayor de la capacidad de un entero de 32 bits, lo que producirá un error de desbordamiento (*overflow*) en tiempo de ejecución. La solución sería, por un lado cambiar las opciones de compilación para que no se tengan en cuenta las excepciones causadas por desbordamiento de enteros, o bien, definir la sobrecarga de conversión para que utilice un valor **Long** en lugar de uno entero. Por supuesto, el valor **Long** indicado en la declaración del operador lo tendríamos que convertir en uno entero, y deberíamos hacer nuestras comprobaciones por si se produce un error de desbordamiento. Pero con esta nueva definición del operador de conversión nos aseguramos de que podemos indicar cualquier tipo entero con signo para que se asigne automáticamente a un valor de tipo **Punto**. Lo mismo es aplicable a las conversiones entre tipos **Double** y **Single**, si definimos la conversión de **Double**, el compilador de VB se encargará de hacer la otra conversión.

Sobrecarga de operadores lógicos

Además de los operadores aritméticos, podemos sobrecargar los operadores lógicos, los que utilizamos para hacer comparaciones y, como veremos, también podemos sobrecargar el comportamiento de nuestro tipo en los casos que tenga que devolver un valor *verdadero* o *falso*. Empecemos con los operadores lógicos.

Como sabemos, siempre que hacemos una comparación, ésta se puede hacer de dos formas distintas, por ejemplo, podemos comprobar si dos valores son iguales, y también podemos hacer la comprobación de si son distintos.

> Los tipos generics nos permiten crear colecciones fuertemente tipadas en las que el tipo "real" es el indicado en el constructor

Debido a esta dualidad de los operadores lógicos, cuando sobrecargamos uno de ellos, debemos sobrecargar también el complementario, y en el caso del operador *igual* (y *distinto*) deberíamos crear una sobrecarga del método **Equals**, de forma que también devuelva un valor que se ajuste a la forma de comprobar la igualdad de dos objetos de nuestro tipo de datos.

La forma de declarar estos operadores es similar a la que ya hemos visto, aunque, en estos casos, el valor devuelto debe ser un valor de tipo **Boolean**, ya que eso es lo que se espera cuando hacemos algún tipo de comparación.

Por ejemplo, podríamos definir los operadores *igual* y *distinto* de nuestro tipo **Punto** tal como se muestra en el listado del fuente 3.

NOTA

Es recomendable que siempre que sobrecarguemos los operadores *igual* y *distinto*, sobrecarguemos el método `Equal`, aunque, a diferencia de C#, Visual Basic no nos avisará de que no hemos realizado esta sobrecarga. En estos casos, también se recomienda sobrecargar el método `GetHashCode`.

Operadores IsFalse y IsTrue

Estas dos nuevas instrucciones de Visual Basic 2005 sólo se pueden usar para crear sobrecargas, en este caso para sobrecargar el comportamiento de nuestro tipo cuando debe devolver un valor *verdadero* o *falso*. Estas dos instrucciones equivaldrían a sobrecargar los valores **True** y **False** para que se adecuen a nuestro tipo. Al igual que ocurre con los operadores de comparaciones, se deben sobrecargar los dos operadores.

```
' Sobrecarga de los operadores igual, distinto y sobrecarga de Equals
Public Shared Operator =(ByVal p1 As Punto, ByVal p2 As Punto) As Boolean
    Return (p1.X = p2.X AndAlso p1.Y = p2.Y)
End Operator

Public Shared Operator <>(ByVal p1 As Punto, ByVal p2 As Punto) As Boolean
    ' Podemos llamar a la sobrecarga de la igualdad
    Return Not (p1 = p2)
End Operator

' Equals nunca debe producir una excepción,
' por tanto, si el argumento no es de tipo Punto, devolvemos False
Public Overrides Function Equals(ByVal obj As Object) As Boolean
    If TypeOf obj Is Punto Then
        Return Me = CType(obj, Punto)
    Else
        Return False
    End If
End Function
```

Fuente 3. Definición de los operadores igual y distinto

Como podemos observar en el fuente 3, sólo hemos codificado el operador igual, para el operador distinto, y la sobrecarga del método **Equals** utilizamos la definición de la sobrecarga de igualdad.

```
' Sobrecarga de IsFalse y IsTrue
Public Shared Operator IsFalse(ByVal p1 As Punto) As Boolean
    Return p1.X = 0 AndAlso p1.Y = 0
End Operator

Public Shared Operator IsTrue(ByVal p1 As Punto) As Boolean
    Return p1.X <> 0 OrElse p1.Y <> 0
End Operator
```

Fuente 4. Sobrecarga de los operadores **IsTrue** y **IsFalse**, para usar nuestros tipos en comparaciones

En el fuente 4 podemos ver el código de estas dos sobrecargas en el tipo **Punto**.

¿Qué operadores podemos sobrecargar?

En la tabla 1 podemos ver los operadores que podemos sobrecargar en Visual Basic. Los indicados como *unarios* significa que el operador sólo reciben un argumento, los *binarios*, por otra parte, reciben dos argumentos, que representan a las expresiones situadas a la izquierda y a la derecha del operador.

NOTA

El operador de asignación (=) no se puede sobrecargar, ni tampoco los operadores de incremento, pero si los usamos, el compilador sabrá lo que tiene que hacer con ellos, y si hemos definido una sobrecarga, la usará. Por ejemplo, si hacemos esto: p1 += 1, el compilador usará la sobrecarga de la suma ya que ese código sería equivalente a este otro: p1 = p1 + 1.

Tipo	*Operadores*
Unario	`+, -, IsFalse, IsTrue, Not`
Binario	`+, -, *, /, \, &, ^, >>, <<, =, <>, >, >=, <, <=, And, Like, Mod, Or, Xor`
Conversión (unario)	`CType`

Tabla 1. Los operadores que podemos sobrecargar en Visual Basic

Novedades de .NET Framework 2.0

Con total seguridad, las novedades que más llaman la atención, no sólo de Visual Basic, sino también de C#, son las incorporadas en el propio .NET Framework.

¿Quién a estas alturas no ha oído mencionar los tipos *generics*? Seguro que todos los lectores. Y si, por esas casualidades de la vida, el que lee aún no sabe qué son los tipos *generics*, está de suerte, ya que en este mismo número y de la mano de **Mario del Valle** y de **Miguel Katrib** (ambos del grupo WEBOO) han escrito un estupendo artículo que trata en profundidad esta nueva incorporación de .NET Framework 2.0, aunque desde el punto de vista del desarrollador de C#, pero si quiere saber cómo usar estos tipos de datos en Visual Basic 2005, puede descargar gratuitamente el número 8 de **dotNetManía** en el que tratamos los *generics* desde la perspectiva del desarrollador de Visual Basic.

De los tipos *generics* hablaremos muy poco, ya que, tal como hemos comentado, está ampliamente tratado en este mismo número y en el número 8 de esta revista, pero desde que se publicó el artículo para Visual Basic 2005 hasta ahora han pasado unos meses y el artículo anterior se basaba en la versión beta 1, y como se ha añadido nueva funcionalidad para los desarrolladores de Visual Basic, trataremos esas novedades, aunque sea brevemente.

Pero primero veamos cómo podemos usar los tipos *generics*, para que nos sirva como recordatorio y también de paso para comprender cómo utilizarlos en Visual Basic, por tanto, veamos el código fuente 5 en el que declaramos una colección que sólo acepte tipos de datos **Cliente**.

```
Dim col As New List(Of Cliente)
Dim c As New Cliente("Juan", 33)
col.Add(c)

col.Add(New Cliente)
col(1).Nombre = "Pepe"
```

Fuente 5. Ejemplo de colección *generic* con valores de tipo **Cliente**

Como ya han explicado nuestros compañeros, la ventaja de este tipo de colecciones es que el tipo de datos utilizado por la colección col realmente es de tipo **Cliente**, con lo cual ganamos en rendimiento, ya que .NET no se ve obligado a convertir los valores almacenados en **Object** y tampoco tenemos que hacer conversiones a la hora de utilizarlos, tal como podemos ver en la figura 4, el compilador "sabe" cuál es el tipo de datos que almacena la colección y, por tanto, nos permite el uso de *IntelliSense* a la hora de acceder a los valores almacenados.

Figura 4. *IntelliSense* nos muestra los tipos almacenados en las colecciones *generic*

Los tipos de restricciones que podemos hacer

En Visual Basic 2005, las restricciones que podemos hacer en nuestros tipos *generic* pueden ser como en C#, y entre las diferencias que han surgido desde que se publicó el artículo del número 8 de **dotNetManía**, tenemos que ya se puede indicar si el tipo puede ser un tipo por valor o un tipo por referencia; para estos casos, usaremos en la lista de restricciones las instrucciones **Structure** y **Class** respectivamente.

En el código mostrado en el fuente 6, hemos definido una clase que utiliza generics con restricciones para que el tipo usado sea por referencia e implemente la interfaz **IComparable**.

```
' Tipo Generic con restricciones
' el tipo indicado debe ser un tipo por
' referencia e implementar IComparable
Public Class UnGeneric(Of T As _
                {Class, IComparable})
```

Fuente 6. Definición de una clase *generic* con restricciones

NOTA

Las restricciones las podemos usar tanto en las declaraciones de clases generic como en los métodos que reciban parámetros genéricos

Tipos por valor con valores indefinidos (Nullable types)

En este mismo número, en el artículo de nuestro compañero **Octavio Hernández** sobre las novedades de C#, habla sobre estos tipos de datos, y para no confundir mucho las cosas, utilizaré la misma traducción que él le ha dado al original inglés *Nullable types*: *tipos por valor anulables*, ya que el uso de tipos por valor con valores indefinidos resulta un poco largo. A diferencia de C#, Visual Basic 2005 no incluye instrucciones o palabras propias del lenguaje para tratar estos nuevos tipos de datos, para ello tendremos que esperar a la nueva versión, en la que se incorpora una sintaxis parecida a la de C#.

Pero como no estamos hablando de lo que se incluirá en VB, sino lo que se incluye en la versión que este mismo mes se lanza, veremos cómo usar este tipo de datos, ya que nos resultarán muy útiles particularmente en las aplicaciones con las que accedemos a bases de datos.

La definición que encontramos en la documentación de Visual Studio nos dice: "*Los tipos por valor anulables representan un tipo cuyo valor subyacente es un tipo por valor al que también podemos asignar un valor nulo como a los tipos por referencia.*"

Los *tipos por valor anulables* que utilizaremos en Visual Basic será el tipo *generic* `Nullable(Of T)`, es decir, siempre indicaremos el tipo por valor que queremos que contenga, de forma que, si el valor contenido es un valor nulo, indicará que dicho tipo no contiene un valor válido. Y esa es la verdadera validez de este tipo de datos.

Como hemos comentado anteriormente esta circunstancia es muy útil cuando trabajamos con valores contenidos en bases de datos, ya que uno de los estados de esos datos puede ser el "indefinido", es decir, un valor nulo. Dicho valor nulo no representa un valor real. Por ejemplo, si tenemos esta declaración:

```
Dim nBool As Nullable(Of Boolean)
```

La variable `nBool` puede contener tres valores: `Nothing` (o nulo, que indica que no está asignado), `True` o `False` (que son los dos valores válidos para los tipos `Boolean`). Como vemos, esto difiere de una variable declarada con el tipo Boolean, la cual, por defecto, tendrá un valor `False`, mientras que al definir la variable con `Nullable(Of Boolean)` tenemos la opción de aceptar un valor indefinido (nulo). Pero aquí no acaba todo, ya que podemos usar las propiedades y métodos de esta estructura para averiguar si tiene un valor válido e incluso para asignar un valor por defecto en el caso de que aún no tenga un valor válido.

Para saber si una variable de tipo `Nullable` contiene un valor, podemos usar la propiedad `HasValue`, esta devolverá verdadero si contiene un valor válido para el tipo de datos con el que la hemos definido o un falso si no lo tiene.

También podemos usar el método `GetValueOrDefault` para obtener el valor contenido en la variable (si no es nulo) o el que indiquemos. Y debido a que todos los tipos por valor siempre tienen un valor predeterminado, al usar este método tenemos dos sobrecargas, una en la que no se indica ningún argumento, en cuyo caso devolverá el valor predeterminado y la otra en la que indicamos el valor que se utilizará en el caso de que no esté definido el valor.

> Los tipos por valor anulables que utilizaremos en Visual Basic será el tipo generic Nullable(Of T), es decir, siempre indicaremos el tipo por valor que queremos que contenga

Por ejemplo, si declaramos una variable de tipo anulable que sea de tipo entero, y en el caso de que el valor que contiene sea nulo podríamos indicar en el método `GetValueIrDefault` que devuelva el valor 15, y si tiene un valor válido, que devuelva dicho valor. Esto lo podemos hacer con el siguiente código:

```
Dim nInt As Nullable(Of Integer)
...
Dim i As Integer= nInt.GetValueOrDefault(15)
```

Si en lugar de asignar un valor (en este caso el 15), queremos que devuelva el valor predeterminado para el tipo entero (que es el 0), la asignación anterior la tendremos que hacer así:

```
Dim i As Integer = nInt.GetValueOrDefault()
```

Debido a que la variable `nInt` en el fondo es un tipo entero, podremos usarla en las mismas situaciones que el resto de variables Integer, incluso podemos asignarle valores resultantes de una expresión que de cómo resultado un valor entero. Pero debido a que en realidad no es una variable `Integer` sino del tipo `Nullable(Of Integer)`, también podemos asignarle un valor `Nothing` con lo que conseguiríamos ponerla en un estado "anulable".

Conclusiones

Realmente son muchas las novedades que incorpora Visual Basic 2005, pero lamentablemente es imposible explicarlas todas en el espacio dedicado a un artículo. Afortunadamente en otros artículos de este monográfico sobre la nueva versión de Visual Studio se han tratado otros temas que de alguna forma también son válidos para Visual Basic, como son los distintos niveles de accesibilidad para los bloques `Get` y `Set` de las propiedades o las clases parciales que se ha tratado en el artículo de las novedades de C#, o las novedades en el acceso a datos, que podemos ver en el artículo de **Jorge Serrano**, en el que nos cuenta las novedades de las aplicaciones *Maestro-Detalle*, donde podemos "saborear" algunas de esas novedades y de los nuevos asistentes del entorno integrado. Otro de los temas que tampoco hemos tratado es sobre un "objeto" especial y exclusivo para los desarrolladores de Visual Basic: **My,** el cual fue tratado en el número 8 de esta revista.

En el código de este artículo se incluyen ejemplos de algunas de las novedades que no hemos tenido oportunidad de tratar.