Por Guillermo 'Guille' Som
Visual Basic MVP desde 1997
www.elguille.info

Generics y Visual Basic .NET

Es mucho lo que se ha escrito (y seguramente se seguirá escribiendo) sobre esta nueva característica de la próxima versión de .NET Framework. Pero en casi todas las ocasiones es referente a que es una nueva característica de C#. Para los que aún no lo sepan, espero que se enteren de que Visual Basic también existe y que forma parte de la familia de lenguajes de .NET y por tanto Visual Basic .NET también podrá usar y crear tipos genéricos, por la sencilla razón de que los tipos genéricos forman parte del .NET Framework 2.0.

» Los tipos genéricos: Introducción

Los tipos genéricos nos dan la posibilidad de tener tipos que permitan almacenar datos de distintos tipos sin perder la funcionalidad y sin la sobrecarga extra de tener que realizar conversiones (*casting*) al recuperar un elemento. A los genéricos también se les conocen como tipos con argumentos de tipos anónimos, tipos parametrizados o tipos con parámetros polimórficos.

Por ejemplo, si tenemos una lista en la que queramos almacenar datos de tipo entero, podemos crear una lista específica que sólo acepte valores de tipo entero. El problema es que si esa lista la construimos con una colección, por ejemplo del tipo *ArrayList*, el entero se guardará como un dato de tipo *Object*, por tanto cada vez que queramos recuperar el valor tendremos que hacer una conversión de *Object* al tipo almacenado. El problema principal es que aunque podemos hacer comprobaciones y demás chequeos de que el tipo de datos almacenado es el adecuado, esas comprobaciones se tendrán que hacer en tiempo de ejecución, no en el de compilación.

La solución que actualmente tenemos es crear esa lista usando un array, de esta forma no será necesario hacer ningún tipo de conversión al recuperar el valor, además de que el compilador se encargará de avisarnos de que estamos haciendo algo mal, ya que toda la comprobación de tipos se realiza en tiempo de compilación. El inconveniente es que en la mayoría de los casos es más práctico usar una colección en lugar de un array, ya que así conseguimos mayor funcionalidad, sobre todo si es una lista que no tiene un número fijo de elementos.

¿Y si en lugar de tener una colección cuyo tipo de datos interno sea *Object* pudiera ser el que nosotros quisiéramos? En ese caso el rendimiento estaría garantizado, ya que obtendríamos todas las ventajas de las colecciones sin pagar un precio en cuanto a rendimiento se refiere, porque toda la comprobación de los tipos almacenados se haría en tiempo de compilación. Y aquí es donde entran los tipos genéricos o los tipos con parámetros genéricos. Usando los tipos genéricos podemos definir una colección del tipo de datos que necesitemos. Si la colección va a almacenar datos de tipo entero podemos crear, por ejemplo una lista de tipo entero, de forma que el tipo "interno" de dicha lista sea precisamente un tipo entero y así no habrá necesidad de hacer ningún tipo de comprobación al almacenar un dato en la lista ni tendremos que hacer una conversión al recuperar dicho dato. Si posteriormente pensamos en crear una lista para almacenar datos de tipo *Cliente*, pues la creamos, y sin necesidad de tener que hacer nada especial, ni tener que crear una nueva lista especialmente para almacenar los datos del tipo *Cliente*. Lo único que tendremos que hacer es definir una colección que admita cualquier tipo de datos y cuando creemos una nueva instancia de dicha colección, le indicamos qué tipo de datos va a almacenar y así será el propio CLR el que se encargue de hacer todos los preparativos para que dicha colección sólo almacene el tipo de datos que hemos indicado. Es como si le dijéramos

al compilador que la colección es del tipo de datos tal o cual y que sólo admita valores de ese tipo en concreto.

Utilizando código de C#, si tenemos una colección declarada de la siguiente forma: *List<T>* lo que tenemos es una lista de *T*, es decir una lista del tipo de datos *T*. *T* no es ningún tipo de datos nuevo, simplemente es una forma de decirle al compilador que cuando se cree un nuevo objeto del tipo *List*, el tipo de datos que almacenará será del que se indique al instanciarlo, por ejemplo: *List<int>* crearía un objeto del tipo *List* cuyo tipo interno será *int*; por otro lado, también podemos crear listas de cualquier otro tipo, por ejemplo: *List<Cliente>*, en este caso, el compilador tendrá en cuenta que nuestra intención es almacenar objetos del tipo *Cliente*.

Los tipos genéricos en Visual Basic .NET

La ventaja de que los tipos de datos genéricos formen parte del propio .NET Framework es que se pueden usar con cualquiera de los lenguajes de .NET, entre ellos Visual Basic.

Como ya estamos acostumbrados, la sintaxis usada en Visual Basic casi siempre difiere de la usada en C#. En el caso de *generics* no es una excepción y para crear una colección genérica tendremos que usar la instrucción *Of* seguida del tipo de datos que se utilizará en dicha colección. Por ejemplo, si queremos que el tipo de datos que almacene la clase *List* sea de tipo entero, la declaración la haremos de esta forma: *List(Of Integer)*. Si es de tipo *Cliente*, la declaramos de esta otra: *List(Of Cliente)* y desde entonces será el propio compilador el que se encargue de todo lo necesario para generar el código IL que utilice solamente el tipo de datos indicado.

Supongo que la justificación de usar *Of* es porque las instrucciones del lenguaje Basic siempre se han caracterizado por ser lo más parecidas al inglés hablado, por tanto si tenemos la declaración *List(Of T)* la podamos leer como: una lista de *T*, que es como se "recomienda" que se lea este tipo de declaraciones para que nuestras neuronas se vayan acostumbrando a pensar de forma diferente, y así poder asimilar mejor el significado de los tipos *generics*.

Lo que sí es cierto, es que ese pensamiento es más fácil de conseguirlo viendo la forma en que se usa en VB, ya que, como suele ser habitual, el C# es algo más críptico: *List<T>*, pero independientemente del lenguaje que usemos, lo importante es que la introducción de los tipos genéricos o tipos anónimos en .NET Framework nos facilitará la creación de ciertos tipos de datos, además de que en muchas ocasiones mejorará el rendimiento de nuestras aplicaciones.

Veamos algunas de las posibilidades que nos da el uso de *generics*, que como tendremos la posibilidad de comprobar no sólo se utiliza para crear tipos, sino que también lo podremos usar para crear métodos con parámetros de tipos anónimos (genéricos).

> Los tipos genéricos nos permiten tipos que almacenan datos de distintos tipos sin perder funcionalidad y sin la sobrecarga de realizar conversiones al recuperar un elemento.

Las colecciones del espacio de nombres *Generic*

La nueva versión de .NET Framework incluye un espacio de nombres con colecciones que utilizarán los tipos genéricos: *System.Collections.Generic*. En este espacio de nombres se incluyen clases como *Collection*, *Dictionary*, *List*, *LinkedList*, *Queue*, *Stack* además de interfaces como *ICollection*, *IDictionary*, *IList*, *IComparable*, *IEnumerator*, etc. Todas estas clases e interfaces nos permitirán crear objetos que acepten tipos de datos anónimos (o genéricos) que nos proporcionarán las ventajas de los tipos genéricos: seguridad de tipos y rendimiento.

Cuando creamos un objeto de cualquiera de estas clases es cuando indicamos el tipo de datos que queremos que contenga, de esta forma el compilador sabrá que tipos de datos puede contener y nos alertará cuando intentemos añadir algún dato que no sea del tipo adecuado. Esto último es así incluso si tenemos desconectado *Option Strict*, aunque sea en modo advertencia (*warning*) lo cual es de agradecer, ya que si no se hiciera esa comprobación, no tendría muchas ventajas usar las clases del espacio de nombres *Generic*.

Ventajas de usar las colecciones *Generic*

Es habitual que al definir una clase de un tipo, por ejemplo *Cliente*, también definamos una clase-colección para almacenar ese tipo. En estos casos, también es habitual que esa clase-colección (*Clientes*) se derive de algunas de las clases base del espacio de nombres *Collections*, como *CollectionBase* o

> Los elementos que podemos declarar en nuestro código como *generics* son clases, estructuras, interfaces, delegados y métodos (*Sub* o *Function*) pero no propiedades.

```
Imports System.Collections

Public Class Clientes
    Inherits CollectionBase
    '
    Public Sub Add(cli As Cliente)
        MyBase.List.Add(cli)
    End Sub
    Default Public Property Item(index As Integer) As Cliente
        Get
            Return CType(MyBase.List(index), Cliente)
        End Get
        Set(value As Cliente)
            MyBase.List(index) = value
        End Set
    End Property
End Class
```

Fuente 1

DictionaryBase, dependiendo del tipo de colección que queramos crear, en nuestra clase *Clientes* tendremos que definir ciertos métodos, por ejemplo, el método *Add*, para que se asigne un valor del tipo que queremos almacenar: *Cliente*. De igual forma, cuando accedemos a uno de los clientes que contiene nuestra colección, tendremos que hacer una conversión de tipos (*casting*) para que devuelva uno de tipo correcto, ya que internamente las colecciones almacenan los datos usando el tipo *Object*.

Veamos un pequeño ejemplo que aclare este punto. En el fuente 1 vemos cómo sería la clase-colección *Clientes*, para que sólo acepte elementos de tipo *Cliente*.

Como podemos comprobar, el método *Add* añade sólo elementos del tipo *Cliente* a la colección, por tanto ni en tiempo de compilación nos permitirá añadir un elemento que no sea del tipo adecuado, pero la propiedad *Item*, como devuelve un elemento del tipo *Cliente*, forzosamente debe hacer una conversión desde *Object* a *Cliente*, ya que internamente la colección almacena los elementos como el tipo *Object*, no como de tipo *Cliente*.

En el fuente 2 podemos ver un ejemplo de cómo usar esta colección.

La ventaja de usar una colección del espacio de nombres *Generic* es que no necesitamos definir ninguna clase para este propósito, ya que como este tipo de colecciones permiten almacenar cualquier tipo de datos, tendremos la seguridad de que sólo se almacenarán elementos del tipo que indiquemos, en nuestro caso del tipo *Cliente*. Pero aún hay más, al recuperar un elemento, no habrá que hacer ninguna conversión, ya que el tipo de datos "interno" de la colección será del tipo *Cliente*.

En el fuente 3 tenemos un ejemplo de cómo usar una colección *Generic* que acepte elementos del tipo *Cliente*.

```
Private Sub pruebaCLientes()
    Dim clis As New Clientes
    Dim c1 As New Cliente("Guillermo", "Som")
    clis.Add(c1)
    clis.Add(New Cliente("Pepe", "López"))
    '
    ' Esto dará error en tiempo de compilación
    'clis.Add("pepe")
    '
    c1 = clis(2)
    Console.WriteLine("Elemento 2: {0}", c1)
End Sub
```

Fuente 2

Como podemos comprobar, el código para usar los dos tipos de colecciones es bastante similar. La diferencia está en que en el fuente 2 estamos usando

```
Private Sub pruebaClientesGeneric()
    Dim clis As New List(Of Cliente)
    Dim c1 As New Cliente("Guillermo", "Som")
    clis.Add(c1)
    clis.Add(New Cliente("Pepe", "López"))
    '
    ' Esto dará error en tiempo de compilación
    'clis.Add("pepe")
    '
    c1 = clis(2)
    Console.WriteLine("Elemento 2: {0}", c1)
End Sub
```

Fuente 3

la clase-colección que nosotros hemos definido para almacenar solamente elementos del tipo *Cliente* y en

el fuente 3 usamos la colección *List* del espacio de nombres *Generic*, la cual al instanciarla, le hemos indicado que el tipo de datos que debe almacenar es del tipo *Cliente*:

```
Dim clis As New _
    System.Collections.Generic.List(Of Cliente)
```

Con esta declaración, el compilador sabe que sólo debe almacenar elementos del tipo indicado después de *Of*, y lo que es más importante, internamente no usará objetos del tipo *Object* sino del tipo *Cliente*.

Los datos genéricos en las colecciones de tipo *IDictionary*

La colección *Dictionary* del espacio de nombres *Generic* también tiene sus ventajas ya que, como sabemos, las colecciones basadas en *IDictionary* manejan los datos con el par clave (*key*) y valor (*value*), y la versión genérica admite dos tipos de datos anónimos, el primero para almacenar la clave y el segundo para almacenar el valor. En las colecciones de tipo *Dictionary* clásicas, tanto la clave como el valor son de tipo *Object*, por tanto es obvio que usar este tipo de colección en la que se pueden definir tanto el tipo de la clave como el valor, es una gran ventaja, no sólo por la seguridad de tipos sino también en lo que a rendimiento se refiere.

> La ventaja de que los tipos de datos genéricos formen parte del propio .NET Framework es que se pueden usar con cualquiera de los lenguajes de .NET, entre ellos el Visual Basic.

Si queremos utilizar una colección *Dictionary* cuyas claves sean de tipo *String* y el valor de tipo *Cliente*, lo haremos de esta forma:

```
Dim clis As New Dictionary(Of String, Cliente)
```

Si preferimos utilizar claves numéricas, la declaramos de la siguiente forma:

```
Dim clis As New Dictionary(Of Integer, Cliente)
```

Los datos internos de las colecciones del tipo *IDictionay* se almacenan en objetos del tipo *DictionaryEntry*, el cual nos permite acceder tanto a la clave (*key*) como al valor (*value*), por tanto es habitual usar un objeto *DictionaryEntry* para recorrer todos los elementos de la colección, pero en el espacio de nombres *Collections.Generic* no existe una definición de este tipo que acepte parámetros anónimos. Aunque sí que existe la estructura *KeyValuePair*, que nos servirá para acceder al par de datos que cada elemento de las colecciones genéricas *Dictionary* almacenan.

Para poder usar esta estructura, debemos indicar los mismos tipos de datos que se utilizaron para definir la colección, tal como podemos ver en el fuente 4:

```
For Each de As KeyValuePair(Of String,
Cliente) In clis
    Console.WriteLine("{1},{0}",de.Key,de.Value)
Next
```

Fuente 4

Tipos genéricos definidos por el usuario

Como hemos comprobado, solamente con las colecciones del espacio de nombres *Generic* ya tendríamos mucha funcionalidad y rendimiento en los nuevos proyectos creados con Visual Studio 2005, pero la implementación en .NET Framework 2.0 de estos tipos anónimos o tipos parametrizados no acaba con las colecciones "genéricas", ya que también nos permite crear nuestros propios tipos genéricos además de poder definir incluso simples métodos que acepten parámetros anónimos.

Métodos genéricos o métodos con parámetros de tipo anónimo

Los métodos de nuestra aplicación (*Sub* o *Function*) pueden declarar tipos anónimos para que se puedan usar parámetros o argumentos de los tipos que definamos. De esta forma podríamos crear métodos con argumentos genéricos, los cuales se usarán directamente y el compilador sustituirá el tipo usado al llamar a dicho método por los parámetros anónimos indicados en la definición del método.

Por ejemplo, si tenemos un procedimiento llamado *pruebaParametroGeneric* que define un parámetro de tipo genérico, podríamos llamar a dicho método usando cualquier tipo de dato sin necesidad de crear sobrecargas que acepten esos tipos de datos diferentes, por ejemplo:

```
pruebaParametroGeneric(22)
pruebaParametroGeneric(43.50)
pruebaParametroGeneric("hola")
```

La nueva versión de .NET Framework incluye un espacio de nombres con colecciones de tipos genéricos: *System.Collections.Generic*. En este espacio de nombres se incluyen clases como *Collection*, *Dictionary*, *List*, *LinkedList*, *Queue*, *Stack* además de interfaces como *ICollection*, *IDictionary*, *IList*, *IComparable*, *IEnumerator*, etc.

La forma de definir este procedimiento sería la siguiente:

```
Private Sub pruebaParametroGeneric(Of T)_
                        (ByVal uno As T)
```

Es decir, usamos la instrucción *Of* justo después del nombre del método e indicamos un nombre "ficticio" para indicar el tipo de dato anónimo. Si ese método recibe parámetros del tipo anónimo los declararemos usando el habitual *As* seguido del nombre usado después de *Of*. Por tanto *Of T* indicará que este método acepta datos genéricos del tipo *T*; ese tipo se conocerá al usar el método y el compilador insertará el código correspondiente para que cada vez que se haga referencia a *T* dentro del método se use el tipo utilizado para llamarlo.

Por ejemplo, para la primera llamada a este método genérico en el que el tipo de datos usado es *Integer*, el código IL generado por el compilador es el siguiente:

```
call void Module1::pruebaParametroGeneric<int32>(!!0)
```

Es decir, el compilador genera el código para usar el tipo adecuado.

Así, a primera vista, podría parecer que esta forma de declarar métodos es más eficiente que declarar métodos sobrecargados, ya que sólo tendríamos que declarar un solo método en lugar de uno diferente para cada uno de los tipos de datos que vayamos a usar. Si tenemos las mismas tres llamadas anteriores, usando la sobrecarga nos veríamos obligados a declarar tres métodos, uno para cada uno de los tipos de datos usados: *Integer*, *Double* y *String*.

El único problema que hay con los métodos de parámetros genéricos (o anónimos) es que dentro del método no se sabe que tipo de dato se usará, por tanto estamos limitados en cuanto a las cosas que podemos hacer con el parámetro. Por ejemplo, no podríamos hacer ningún tipo de operación aritmética, no podríamos hacer comparaciones, y sólo podríamos usar los métodos definidos en la clase *Object*, ya que al fin y al cabo todos los tipos definidos en .NET se derivan de la clase *Object*. Por tanto, lo que podamos hacer con los parámetros anónimos de un método genérico será lo mismo que podamos hacer con un dato de tipo *Object*.

En el fuente 5 podemos ver la declaración del método genérico usado en los ejemplos anteriores:

```
Private Sub pruebaParametroGeneric(Of T) (ByVal uno As T)
  Console.Write("El valor es: {0} ", uno)
  Console.WriteLine("y el tipo de datos es: {0}",_
                    uno.GetType.Name)
  '
  ' esto dará error ya que uno * 2 se convierte en T * 2
  ' y el compilador nos informará de que:
  ' "el operador * no está definido para
  ' los tipos T e Integer"
  'Console.WriteLine("El doble de {0} es {1}", uno, uno * 2)
End Sub
```

Fuente 5

¿Desilusionado? Tampoco iba a ser todo perfecto... algún fallo debía tener...

De todas formas, no está todo perdido, como tendremos oportunidad de ver más adelante, hay una forma de indicarle al .NET Framework que queremos que nuestro tipo anónimo tenga ciertas características.

Antes de ver cómo hacer que el tipo anónimo no lo sea tanto, vamos a ver cómo podemos usar más de un parámetro anónimo.

De igual forma que podemos declarar una lista con varios parámetros, también podemos definir un método que admita más de un tipo anónimo, para ello indicaremos después de *Of* varios tipos que posteriormente serán sustituidos por los tipos reales.

Por ejemplo podríamos definir un método que reciba 2 parámetros (o argumentos) de dos tipos diferentes, la declaración sería tal como se muestra en el fuente 6.

Es decir, después de *Of* indicamos los tipos a usar separados con comas. Por supuesto, aquí seguimos

teniendo las restricciones de que no podemos hacer demasiadas cosas con esos parámetros, incluso si sabemos que ambos serán de tipo *Integer* no podríamos hacer una simple suma, por la sencilla razón de que el compilador no nos permitirá hacer una conversión explícita. Por supuesto que siempre nos quedaría el recurso de declarar los parámetros de tipo *Object*, pero... en ese caso ya no estaríamos usando tipos genéricos.

```
Private Sub pruebaVariosGeneric(_
        Of T1,T2) (ByVal uno As T1, _
                   ByVal otro As T2)
    Console.WriteLine("uno: {0}, _
                       otro: {1}", _
                       uno, otro)
End Sub
```

Fuente 6

Nuestros propios tipos genéricos

De la misma forma que podemos definir métodos (que no propiedades) con argumentos genéricos, también podemos declarar nuestros propios tipos de datos (clases, estructuras, delegados e interfaces). La forma de definir una clase genérica es parecida a lo que hasta ahora hemos estado viendo. En el fuente 7 tenemos una clase que acepta tipos genéricos:

```
Public Class tipoGenericVB(Of tipo)
 Private _unaPropiedad As tipo
 Public Property unaPropiedad() As tipo
     Get
         Return _unaPropiedad
     End Get
     Set(ByVal value As tipo)
         _unaPropiedad = value
     End Set
 End Property
End Class
```

Fuente 7

Esta clase nos permitirá hacer cosas como las mostradas en el fuente 8.

Por supuesto, las clases también pueden definir varios tipos genéricos e incluso declarar constructores, etc. En el código del fuente 9 tenemos una clase con varios tipos anónimos y constructores, con y sin parámetros.

```
Sub pruebaTipoGenericVB()
 Dim t1 As tipoGenericVB(Of String)
 t1 = New tipoGenericVB(Of String)
 t1.unaPropiedad = "Hola"
 '
 Dim t2 As New tipoGenericVB(Of Integer)
 t2.unaPropiedad = 22
 '
End Sub
```

Fuente 8

Pero siempre con las mismas restricciones indicadas anteriormente: que el tipo de datos usado sólo se conocerá cuando se haya declarado una variable de la clase, por tanto en el código interno de la clase no podremos hacer llamadas a métodos que "posiblemente" estén en el tipo finalmente usado.

```
Public Class tipoGenericVB2(Of tipo1, tipo2)
    Sub New()
    End Sub
    Sub New(ByVal uno As tipo1, ByVal otro As tipo2)
        Me.unaPropiedad = uno
        Me.otraPropiedad = otro
    End Sub
    '
    Private _unaPropiedad As tipo1
    Public Property unaPropiedad() As tipo1
        Get
            Return _unaPropiedad
        End Get
        Set(ByVal value As tipo1)
            _unaPropiedad = value
        End Set
    End Property
    '
    Private _otraPropiedad As tipo2
    Public Property otraPropiedad() As tipo2
        Get
            Return _otraPropiedad
        End Get
        Set(ByVal value As tipo2)
            _otraPropiedad = value
        End Set
    End Property
End Class
```

Fuente 9

Restricciones en los tipos genéricos

Como hemos visto anteriormente uno de los problemas que tenemos con los tipos genéricos que hemos declarado es que no podemos utilizar ninguno de los métodos que "posiblemente" tendrán, salvo los expuestos por la clase *Object*; tampoco podemos realizar ningún cálculo aritmético, ni siquiera podemos realizar conversiones explícitas; todo esto es debido a que el compilador no conoce de antemano el tipo que se usará en el parámetro anónimo, ya que ese conocimiento solamente lo tiene al declarar una clase o al llamar a un método.

Como vemos esto es una limitación muy importante, ya que si el código que tenemos en uno de estos tipos o métodos anónimos debe realizar alguna comprobación o algún tipo de acción sobre uno de los parámetros, no podríamos hacerlo.

Para solventar estos problemas podemos declarar tipos genéricos que restrinjan los tipos de datos que podemos usar como parámetros anónimos. De esta forma, podemos "forzar" a que el tipo usado cumpla ciertas condiciones, por ejemplo, que implementen ciertas interfaces o que se deriven de tal o cual clase e incluso que el tipo usado tenga al menos un constructor sin parámetros; en C# además podemos indicar

que el tipo anónimo sea un tipo por referencia o por valor. Todo esto es posible gracias a las restricciones (*constraints*) de los parámetros usados en nuestros tipos genéricos.

Veamos cómo podemos obligar que se usen ciertos tipos.

Las restricciones (*constrains*) se hacen como si declarásemos el tipo anónimo con la restricción que queremos usar, por ejemplo si los tipos usados deben implementar la interfaz *IComparable* se declararía de la siguiente forma: *Of T As IComparable*.

Por ejemplo, en el fuente 10 tenemos un método con tipos anónimos que deben implementar la interfaz *IComparable*:

```
Private Sub prueba(Of T As IComparable)_
                     (ByVal uno As T)
  Console.WriteLine(_
           "{0} implementa IComparable", _
           GetType(T).Name)
End Sub
```

Fuente 10

Aunque esto tampoco sería nada del otro mundo, ya que esa misma funcionalidad la podemos conseguir simplemente declarando el parámetro del tipo *IComparable*, tal como se muestra en el fuente 11 y el compilador también nos avisaría si quisiéramos pasar como parámetro un objeto que no implemente esa interfaz.

```
Private Sub pruebaSin(ByVal uno As IComparable)
    Console.WriteLine(_
              "{0} implementa IComparable", _
              CObj(uno).GetType.Name)
End Sub
```

Fuente 11

Pero lo que no podremos hacer sin el uso de los *generics* es poder comprobar que no sólo implemente una interfaz, sino que implemente varias interfaces o que el parámetro disponga de un constructor.

Restringir a tipos que implementen varias interfaces

Para poder restringir a tipos que implementen varias interfaces debemos incluir todas las interfaces entre un par de llaves después de *As*, tal como podemos ver en la siguiente declaración:

```
Private Sub pruebaVarios(Of T As {ICloneable,_
                IComparable})(ByVal uno As T)
```

En C#, además de las restricciones que hemos visto, también se pueden hacer otras dos más:

1- Que el tipo sea un tipo por referencia.

2- Que el tipo sea un tipo por valor

La forma de hacer estas restricciones será usando *class* o *struct* respectivamente y se usarán tal como se muestra a continuación:

```
// el tipo usado debe ser un tipo por referencia
class GenericCS4<T> where T : class

// el tipo usado debe ser un tipo por valor
class GenericCS4V<T> where T : struct
```

Restringir a tipos que se deriven de una clase

Además de restringir una o más interfaces, también podemos indicar que dicho tipo se derive de una clase en concreto, dicha clase la indicaríamos de igual forma que con las interfaces, la única restricción en el uso de clases es que solamente se puede indicar una clase. Esto, en parte, podría parecer lógico ya que los tipos de .NET no se pueden derivar de más de un tipo, aunque sí podemos hacer que una clase se derive indirectamente de otra, por ejemplo si la clase *Contenido* se deriva de (hereda) la clase CID y la clase *Palabra* se deriva de *Contenido*, *Palabra* también hereda los miembros de CID. Por supuesto en esta jerarquía de clases, las interfaces implementadas por los tipos bases también forman parte de las clases derivadas.

La forma de hacer esa restricción sería igual que con las interfaces y el orden en el que se indique la

> Los tipos genéricos no forman parte de las especificaciones comunes de .NET (CLS), por tanto si queremos que nuestra aplicación sea compatible con esas especificaciones no podremos usar componentes que expongan clases genéricas.

clase no tiene importancia, aunque deberíamos indicarla al principio, (antes de cualquier otra restricción), de esta forma, si algún programador de C# ve nuestro código le resultará más fácil entenderlo, ya que en C# las restricciones a clases deben aparecer antes del resto de tipos restringidos.

El siguiente código muestra un método que restringe los tipos a usar como parámetros a los que implementen dos interfaces (*ICloneable* e *IComparable*) y estén derivadas (directa o indirectamente) de una clase llamada CID.

```
Sub prueba2(Of T As {CID, ICloneable,
IComparable}) _
              (ByVal uno As T)
```

Restringir a tipos que implementen un constructor

La última restricción que podemos indicar es que el tipo anónimo tenga un constructor que no reciba parámetros, con idea de que se pueda crear usando *As New Tipo*.

En este caso, utilizaremos la propia instrucción *New*, la cual se agregará en la lista de restricciones, como si de una interfaz o clase se tratara:

```
Sub prueba3(Of T As {CID, New })(uno As T)
```

Sólo queda aclarar una cosa más, los tipos de datos que se pueden usar para realizar restricciones deben ser tipos por referencia y que no estén "sellados" es decir, que no estén marcados como *NotInheritable*, al menos en Visual Basic, ya que C# puede restringir los tipos genéricos a tipos por valor.

> **Nota a tener en cuenta**
>
> En estos ejemplos estamos mostrando solamente la declaración de métodos, pero las restricciones también las podemos aplicar de la misma forma a clases, estructuras, interfaces o delegados.

Y esto es todo lo que por ahora se puede decir sobre *generics* desde el punto de vista del programador de Visual Basic .NET, no sin recordar que aún estamos tratando sobre la versión beta 1 de .NET Framework; por tanto es posible que la sintaxis usada pueda variar, como ya lo hizo desde la primera "alfa" de Visual Studio 2005 (entonces llamada *Whidbey*).

El código usado en los ejemplos de este artículo, en versión completa, así como los mismos ejemplos para C# se pueden descargar como siempre en el material de apoyo de este artículo en nuestra página web en : *www.dotnetmania.com/articulos/008/apoyo/generics.html*

trucos.trucos.trucos

Manipular el texto de un TextBox

Por *Guillermo "Guille" Som (elguille.info)*

Seleccionar todo el texto de un *TextBox*:

```
TextBox1.SelectAll()
```

Copiar el texto seleccionado de un *TextBox* en el portapapeles:

```
TextBox1.Copy()
```

Pegar el contenido del portapapeles en un *TextBox*:

```
TextBox1.Paste()
```

Cortar el texto seleccionado de un *TextBox* en copiarlo en el portapapeles:

```
TextBox1.Cut()
```

Comprobar si un *TextBox* puede deshacer:

```
VB: If TextBox1.CanUndo Then ...
C#: if(TextBox1.CanUndo) ...;
```

Deshacer el último cambio realizado en un *TextBox*:

```
TextBox1.Undo()
```

Procedimientos de eventos

Por *Guillermo "Guille" Som (elguille.info)*

Asignar dinámicamente un evento a un procedimiento:

```
VB: AddHandler TextBox1.Enter, AddressOf TextBox1_Enter
C#: TextBox1.Enter += new EventHandler(TextBox1_Enter);
```

Quitar un procedimiento de evento:

```
VB: RemoveHandler TextBox1.Enter, AddressOf TextBox1_Enter
C#: TextBox1.Enter -= new EventHandler(TextBox1_Enter);
```

NOTA: Salvo que se indique lo contrario, el código mostrado es válido tanto para VB .NET como para C#, si bien para usarlo con C# habrá que añadir un punto y como (;) al final.