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

Equivalencia de instrucciones de C# y VB .NET (I)

Cómo hacer las mismas cosas (o casi) en C# y Visual Basic .NET

» Una cosa que es indiscutible, en lo que a la programación de aplicaciones para .NET Framework se refiere, es que todos los lenguajes se apoyan en las clases que ese marco de trabajo incluye. Ya pasó la época en la que para hacer ciertas tareas comunes, como abrir un fichero, acceder a una base de datos o interactuar con el sistema operativo subyacente cada lenguaje de programación tenía su propia forma de hacerlo. Por suerte para los programadores ahora sólo necesitamos conocer una forma de trabajar: La forma que las clases de .NET Framework tienen para hacerlo.

.NET Framework incluye casi toda la funcionalidad que necesitamos para crear nuestras aplicaciones, y si algo que necesitamos hacer no se incluye entre los numerosos espacios de nombres plagados de clases de .NET Framework, podemos recurrir a componentes creados por terceros e incluso crearlos nosotros mismos.

Ahora, lo importante es que, independientemente del lenguaje que estemos usando, siempre tendremos la oportunidad de hacer las cosas al estilo de .NET Framework: Usar una clase para lo que necesitemos.

Para poder usar una clase, sólo necesitamos saber cómo declararla e instanciarla en la memoria para que esté lista para usar. No necesitamos saber nada más, salvo conocer la sintaxis apropiada del lenguaje que hayamos elegido para desarrollar nuestras aplicaciones para poder hacer esa tarea. Lo mismo ocurrirá si pretendemos crear nuestras propias clases; tendremos que saber cómo quiere .NET Framework que hagamos las cosas, qué requisitos nos exige y una vez que sepamos esas exigencias, podremos crear nuestro propio juego de clases. En ambos casos debemos conocer la sintaxis que utiliza nuestro lenguaje predilecto para hacernos entender con .NET Framework.

No sólo veremos la equivalencia sintáctica entre C# y VB .NET, también veremos algunos conceptos que son exclusivos de cada lenguaje.

Se supone que si estamos programando en Visual Basic .NET, o en C#, sabremos aplicar las reglas sintácticas del lenguaje para poder realizar todas las tareas... o casi todas, que algunas veces no sólo es suficiente conocer la sintaxis del lenguaje, ya que si no conocemos cómo realizar ciertas cosas, por muy bien que

sepamos hacer un bucle FOR, no nos servirá de mucho.

Voy a dar por supuesto que ya sabes programar para .NET, es decir, ya sabes cómo hacer las cosas para trabajar con las clases de .NET Framework, (aunque no conozcas todas las clases incluidas en .NET), ya sabes cómo crear tus propias clases, ya sabes cómo derivar una clase a partir de otra, ya sabes cómo implementar una interfaz, ya sabes cómo crear un bucle o definir una propiedad e incluso un constructor o un destructor (o finalizador); sí, se supone que ya sabes hacer todas esas cosas en el lenguaje que has elegido para programar, (en esta serie de artículos supondré que ha sido el lenguaje C# o Visual Basic .NET), pero lo que seguramente te gustaría es saber cómo hacerlo en el *otro* lenguaje.

…veremos desde cómo declarar una variable hasta cómo llamar desde el constructor de una clase derivada al de la clase base.

Y eso es lo que esta serie de artículos pretende, explicarte cómo hacer las mismas cosas tanto en C# como en Visual Basic .NET. Seguramente serán los programadores de Visual Basic los que tengan más interés en conocer cómo trabajar con sintaxis de C#, por aquello de que en algunos sitios valorarán más a quién sepa programar en C# frente al que sólo sepa programar en el, siempre mal visto, lenguaje de la familia BASIC.

Aquí no voy a enseñarte a programar en ninguno de los dos lenguajes más utilizados por los desarrolladores de .NET Framework, aunque seguramente algunas de las cosas aquí explicadas te ayudarán a clarificar tus ideas. Si quieres aprender a programar, deberías plantearte acudir a algún centro de formación, a algún sitio de Internet o comprarte algún libro.

Sin más historias empecemos viendo las distintas tareas que podemos realizar en .NET Framework comparadas entre los lenguajes C# y Visual Basic .NET, veremos desde cómo declarar una variable hasta cómo llamar desde el constructor de una clase derivada al de la clase base.

Sintaxis básica

La declaración de las variables es una de las cosas que siempre tendremos que hacer en cualquier aplicación.

A lo largo de este y los siguientes artículos te mostraré una serie de tablas en las que usaré ciertos marcadores para indicar instrucciones o tipos de datos que tendremos que especificar. Por ejemplo:

<*tipo*> Será el tipo de datos que vamos a definir. En Visual Basic se puede declarar una variable sin necesidad de indicar el tipo de datos. Debido a que esta no es la forma recomendada de trabajar, voy a suponer que tenemos seleccionada la opción que nos obliga a realizar las declaraciones, asignaciones y conversiones de forma correcta: *Option Strict On*.

<*ámbito*> Será la cobertura que tendrá la variable. Si la declaración se realiza dentro de una propiedad o método (función o procedimiento), en el caso de VB habrá que usar siempre *Dim* o *Static*, en C# no se usa ningún modificador de ámbito. Si la declaración se hace a nivel del tipo (clase o estructura) y no se indica el ámbito en C# o se utiliza *Dim* en VB, será lo mismo que declarar la variable como privada.

En dichas tablas se mostrará tanto el código de C# como el de VB .NET. Puede que dicho código se muestre en varias líneas. En C# no sería un problema, ya que el final de una instrucción se indica con un punto y coma, pero en Visual Basic .NET todo el código debe estar en una línea física y en caso de que queramos que se continúe con la siguiente, habrá que usar un espacio seguido de un guión bajo.

En C# cuando se indique el <*código*> a escribir, podrá ser: una instrucción y acabará con pun-

to y coma; o un bloque de instrucciones (más de una instrucción), en cuyo caso dichas instrucciones deben estar dentro de un par de llaves { y }. Ese par de llaves indicará el bloque de código.

En Visual Basic .NET no existe el concepto de bloque de código independiente, aunque la mayoría de las instrucciones definirán un bloque de código. Dicho bloque de código estará definido por una palabra clave y normalmente acabará con la instrucción *End* seguida de la palabra clave que define dicho bloque. Por ejemplo, la definición de una clase se indica usando la instrucción *Class* y finaliza con las instrucciones *End Class*.

Aunque en VB cada instrucción (o serie de instrucciones) debe ocupar una sola línea, (la que se puede continuar con la siguiente usando un guión bajo), dentro de esa misma línea pueden existir varias instrucciones separadas por dos puntos; aunque esta forma de escribir código no es recomendable porque el código es más difícil de leer y puede causar más de un quebradero de cabeza.

Modificadores de ámbito o visibilidad

Las variables, métodos, propiedades, clases, etc., siempre tendrán un nivel de visibilidad o cobertura, es decir, se tendrá que definir, explícita o implícitamente donde serán visibles. En las siguientes tablas veremos las instrucciones que permitirán definir o modificar el nivel de visibilidad de esos elementos, así como los modificadores de visibilidad que podremos usar.

En el caso de las instrucciones *Dim* o *Static* de Visual Basic, sólo se podrán aplicar a variables.

Modificador de ámbito (visibilidad)	C#	VB .NET
Variables declaradas en métodos, propiedades y bloques de código.	No se indica.	`Dim` o `Static` `Static` hará que la variable mantenga el último valor asignado
Privado a la clase o tipo	`private`	`Private`
Público, siempre accesible	`public`	`Public`
Accesible en el propio ensamblado	`internal`	`Friend`
Accesible en la clase y en las clases derivadas	`protected`	`Protected`
Accesible en la clase, clases derivadas y en el mismo ensamblado	`protected internal`	`Protected Friend`

Tabla 1. Modificadores de accesibilidad (ámbito o visibilidad)

La tabla 1 resume los modificadores de accesibilidad que podemos usar tanto en Visual Basic .NET como en C#.

Dependiendo de dónde se declaren las variables y los tipos (clases), sin indicar expresamente el modificador de visibilidad, es posible que el ámbito de dicho elemento varíe. Incluso ese ámbito puede ser diferente dependiendo del lenguaje usado.

Por ejemplo, si en C# declaramos una clase sin indicar el modificador de visibilidad, será lo mismo que haberla declarado *private*, sin embargo en Visual Basic .NET el ámbito será *Friend*.

La tabla 2 muestra los modificadores que se aplican de forma predeterminada si no se indican expresamente, así mismo muestra los modificadores permitidos según el tipo o miembro que estemos declarando.

sólo necesitamos conocer una forma de trabajar: La forma que las clases de .NET Framework tienen para hacerlo

Tipos y Miembros de los tipos:	Accesibilidad predeterminada		Accesibilidades permitidas	
	C#	VB .NET	C#	VB .NET
enum / Enum	public	Public	public protected internal private protected internal	Public Protected Friend Private Protected Friend
Miembros de enum	public	Public	Sólo public	Sólo Public
class / Class y miembros de las clases.	private	Friend	public protected internal private protected internal	Public Protected Friend Private Protected Friend
Module (sólo VB)	N.A.	Friend	N.A.	Public Friend
Miembros de Module	N.A.	Friend	N.A.	Public Friend Private
interface/Interface miembros de las clases.	private	Friend	public protected internal private protected internal	Public Protected Friend Private Protected Friend
Miembros de *interface*	public	Public	Ninguna	Ninguna
struct / Structure	private	Friend	public protected internal private protected internal	Public Protected Friend Private Protected Friend
Miembros de *struct / Structure*.	private	Siempre hay que indicar el ámbito.	public internal private	Public Friend Private

Tabla 2. Accesibilidades predeterminadas y permitidas

Tipos de datos

Los tipos de datos de los lenguajes de .NET son alias que hacen referencia a los tipos de datos definidos por el sistema de tipos comunes (*Common Type System*, CTS) de .NET.

En la tabla 3 podemos ver los tipos de datos con el nombre usado por .NET y los alias usados en Visual Basic .NET y C#.

Los arrays o matrices

Los arrays (matrices) son un tipo especial de variable por referencia y están basadas en la clase *System.Array*. Aunque sea un tipo por referencia, la creación de matrices no implica la utilización de la instrucción *New*, (que es la que se utiliza para crear una nueva instancia en memoria de una variable de un tipo por referencia), al menos en Visual Basic, ya que en C# sí será obligatorio el uso de la instrucción *new* para crear el array; aunque cuando se declara y asigna al mismo tiempo, no es necesario usar *new* para crear el array.

En Visual Basic, un array se define usando un par de paréntesis en el tipo de datos o en el nombre de la variable. En C# se indica con un par de corchetes, pero siempre se usará en el tipo de datos:

En VB se puede usar de estas dos formas: `Dim var() As Integer` o `Dim var As Integer()`

En C# se usará sólo de esta forma: `int[] var;`

En Visual Basic, si al declarar el array indicamos el número de elementos, el array ya estará listo para poder usarlo: `Dim var(4) As Integer`, aunque no sería correcto hacerlo de esta otra forma: `Dim var As Integer(4)`.

En C#, el equivalente sería: `int[] var = new int[5];`

.NET Framework	C#	VB .NET	Descripción, rango de valores
Tipos por valor:			
System.Boolean	bool	Boolean	Verdadero o falso.
System.DateTime	N.A.	Date	Fecha y hora.
System.SByte	sbyte	N.A.	Entero con signo de 8 bits.
System.Byte	byte	Byte	Entero sin signo de 8 bits.
System.Int16	short	Short	Entero con signo de 16 bits.
System.UInt16	ushort	N.A.	Entero sin signo de 16 bits.
System.Int32	int	Integer	Entero con signo de 32 bits.
System.UInt32	uint	N.A.	Entero sin signo de 32 bits.
System.Int64	long	Long	Entero con signo de 64 bits.
System.UInt64	ulong	N.A.	Entero sin signo de 64 bits.
System.Char	char	Char	Un carácter Unicode de 16 bits.
System.Single	float	Single	Punto flotante de simple precisión (con precisión de 7 dígitos32 bits).
System.Double	double	Double	Punto flotante de doble precisión (64 bits) .
System.Decimal	decimal	Decimal	Tipo moneda con 28-29 dígitos significativos.
Tipos por referencia:			
System.Object	object	Object	Clase base de todos los tipos de .NET.
System.String	string	String	Tipo que representa a cadenas de caracteres

Tabla 3. Tipos de datos predefinidos

Para declarar y asignar valores al mismo tiempo, tanto en VB como en C#, se indicarán los valores de cada elemento entre un par de llaves:

En VB: `Dim var() As Integer = {1, 4, 12, 2}`

En C#: `int[] var = {1, 4, 12, 2};`

En Visual Basic se puede cambiar el tamaño de un array usando *ReDim*:

```
Dim var() As Integer
ReDim var(10)
```

También podemos usar *ReDim Preserve*; en este caso, si el array ya contenía valores, dicho contenido se conserva al cambiar el tamaño:

```
ReDim Preserve var(15)
```

En C# sólo se puede cambiar el tamaño de un array usando *new* y no existe un equivalente para *Preserve*; si queremos conservar el contenido anterior, debemos crear una copia y asignar los valores de forma manual:

```
int[] var;
var = new int[11];
```

Para eliminar un array, podemos asignarle un valor nulo (*null/Nothing*), aunque en VB se puede eliminar el contenido de un array usando la instrucción *Erase*.

Para acceder a un elemento de un array, en VB se usará un índice numérico dentro de un par de paréntesis, mientras que en C# ese índice se indicará dentro de un par de corchetes:

En VB: `i = var(2)`

En C#: `i = var[2];`

Otra diferencia entre Visual Basic .NET y C# es que al definir un número de elementos en un array, en C# se indicará el número de elementos que tendrá el array, mientras que en VB el número indicado será el valor del índice superior del array:

En VB: `Dim var(4) As Integer` define un array de 5 elementos, desde 0 hasta 4.

En C#: `int[] var = new int[4];` define un array de 4 elementos, desde 0 hasta 3.

En todos los lenguajes de .NET, el índice inferior de un array siempre será cero.

En el próximo número seguiremos repasando las diferentes formas de declarar variables, métodos y otros elementos de .NET Framework.

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

Equivalencia de instrucciones de C# y VB .NET (II)

Cómo hacer las mismas cosas (o casi) en C# y Visual Basic .NET

Declaración de variables, métodos, propiedades e indizadores

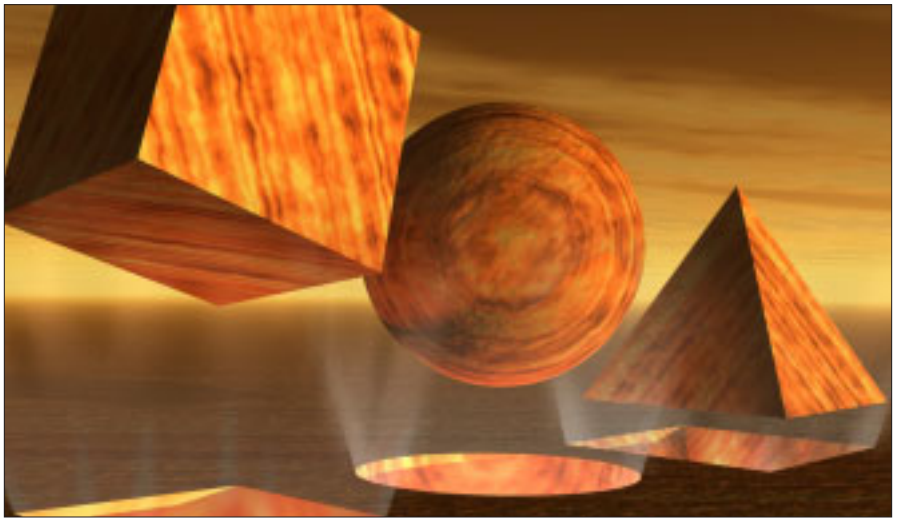
La declaración de las variables, métodos, propiedades, etc. difieren básicamente en la forma de realizar la declaración. En C# se indicará el ámbito, es decir, si es público, privado, etc. seguido del tipo de datos, seguido del nombre de la variable. En VB se indicará el ámbito seguido del nombre y por último la instrucción **`As`** seguida del tipo de datos.

Tanto en VB como en C# se pueden definir constantes, en el caso de C# se usará la instrucción **`const`** antes del tipo de datos, en VB se usará **`Const`** en lugar de **`Dim`** o **`Static`** cuando se declare una constante dentro de un procedimiento. Si esa declaración se hace a nivel de tipo, la instrucción **`Const`** se usará después del modificador de ámbito. Las constantes siempre se definen con un valor.

Las variables también se pueden definir y asignarle un valor predeterminado al mismo tiempo que se declara. Si no se indica de forma explícita, ese valor, en Visual Basic siempre se asignará un valor "cero", pero en C# ese valor predeterminado sólo se asignará si la variable se declara como miembro de un tipo (campo), si la variable se declara dentro de una función, no se asigna ningún valor predeterminado y habrá que asignarle algún valor antes de usarla.

Los métodos serán las funciones y procedimientos declarados en una clase.

En Visual Basic las funciones son procedimientos que devuelven un valor y se declaran usando la instrucción **`Function`**, los métodos que no devuelven un valor, en Visual Basic se declaran como procedimientos **`Sub`**.



En C# todos los métodos son funciones, las que devuelven un valor se declaran con el tipo de datos que devuelve y las que no devuelven un valor (las equivalentes a los procedimientos de tipo **`Sub`** de VB) el tipo se indica con la instrucción **`void`**.

En Visual Basic .NET las propiedades pueden recibir parámetros. En C# las propiedades no pueden recibir parámetros.

En C#, cuando se asigna un valor a una propiedad, (bloque **`set`**), el valor a asignar estará representado por un parámetro implícito llamado **`value`** que no se indica en la declaración. En VB .NET siempre hay que indicar el parámetro en el bloque **`Set`**, aunque ese parámetro puede tener cualquier nombre, se recomienda usar **`value`**.

En VB .NET las propiedades de sólo lectura y sólo escritura, además de sólo indicar el bloque **`Get`** o **`Set`**, se debe indicar que la propiedad es **`ReadOnly`** (solo lectura) o **`WriteOnly`** (solo escritura)

Los indizadores nos permiten usar una clase como si fuese un array: indicando un valor entre corchetes en el caso de C# o entre paréntesis en el caso de VB .NET. En C# los indizadores son propiedades cuyo identificador (nombre) es la propia clase y se usa siempre **`this`**. En Visual Basic .NET los indiza-

Tarea a realizar	C#	VB .NET
Declarar una variable	`int var;`	`Dim var As Integer`
Declarar una constante	`int const var = 10;`	`Const var As Integer = 10`

Declarar un array (matriz)	`int[] var;`	`Dim var() As Integer`
		o
		`Dim var As Integer()`

Declarar e instanciar una variable	`<ámbito> <tipo> var = new <tipo>();`	`<ámbito> var As New <tipo>()`
		o
		`<ámbito> var As <tipo> = New <tipo>()`

Declarar un método que no devuelve un valor	`void método()` `{}`	`Sub método()` `End Sub`
Declarar un método que devuelve un valor	`int método()` `{}`	`Function método() As Integer` `End Function`

Declarar una propiedad de lectura y escritura	`string Nombre{`
	`<bloque get>`
	`<bloque set>`
	`}`
		`roperty Nombre() As String`
		`<bloque Get>`
		`<bloque Set>`
		`End Property`

Declarar una propiedad de solo lectura	`string Nombre{`
	`<bloque get>}`
		`ReadOnly Property Nombre() As String`
		`<bloque Get>`
		`End Property`

Declarar una propiedad de solo escritura	`string Nombre{`
	`<bloque set>}`
		`WriteOnly Property Nombre() As String`
		`<bloque Set>`
		`End Property`

Declarar un indizador (o propiedad por defecto en VB)	`string this[int index]{`
	`<bloque get>`
	`<bloque set>`
	`}`
		`Default Property Item(index As Integer) As String`
		`<bloque Get>`
		`<bloque Set>`
		`End Property`

Declarar un campo de solo lectura con el valor asignado al declararlo	`<ámbito> const <tipo> <identificador> = <valor>;`	`<ámbito> Const <identificador> As <tipo> = <valor>`
Declarar un campo de solo lectura cuyo valor se puede asignar en tiempo de ejecución. Dicha asignación se puede hacer al declarar el campo o en el constructor.	`<ámbito> readonly <tipo> <identificador> [= <valor>];`	`<ámbito> ReadOnly <identificador> As <tipo> [= <valor>]`

Tabla 4. Declaración de variables, métodos, propiedades e indizadores

dores son propiedades predeterminadas y se puede usar cualquier identificador, de forma que se puede acceder usando el identificador o no. Tanto en C# como en VB, se debe indicar como mínimo un parámetro.

Los métodos, propiedades y otras instrucciones se declaran seguidas de un "bloque de código", en C# los bloques de código están incluidos dentro de un par de llaves { y }. En Visual Basic NET el final de dicho bloque de código suele terminar con la instrucción **`End <tipo de bloque>`**, donde **`<tipo de bloque>`** será el tipo de bloque que estamos definiendo, por ejemplo si es un **`Sub`**, el final del bloque se indicará con **`End Sub`**.

Los campos son las variables declaradas en las clases (tipos). Esos campos serán los datos que la clase o tipo utilicen. Se pueden definir campos de sólo lectura de dos formas distintas: definiéndolo como una constante, lo cual hará que dicho campo sea realmente de sólo lectura, no permitirá ninguna asignación de un valor diferente al usado al declararlo por código; y por otro lado, podemos definir campos de sólo lectura a los que podemos asignar valores de dos formas distintas: al definirlo, en este caso actuarán "casi" como una constante, pero también podemos asignarle un valor sólo y exclusivamente en el constructor de la clase. De esta forma, podemos hacer que durante la ejecución sea de sólo lectura, pero ese valor se puede asignar en tiempo de ejecución; aunque sólo al crear la nueva instancia.

La tabla 4 muestra cómo realizar en VB .NET y C# la declaración de variables, métodos, etc.

Parámetros de los métodos

Los métodos pueden recibir parámetros. Esos parámetros pueden ser por valor o por referencia. Los parámetros por valor son copias que se entregan a los métodos de los parámetros usados, cualquier cambio realizado a dichos valores no afectarán a los originalmente usados. Por otro lado, los parámetros por referencia, si se modifican dentro del método, afectarán a los usados al llamar a dicho método.

Por defecto los parámetros son por valor, en el caso de C# no existe ninguna instrucción para indicar esa cualidad; en Visual Basic .NET existe la instrucción **ByVal** para indicarlo expresamente, aunque ese modificador es opcional. Visual Studio .NET siempre lo indicará usando **ByVal**.

Por otro lado, se pueden definir parámetros por referencia, en el caso de VB .NET se indicarán mediante la instrucción **ByRef**. En C# existen dos tipos de modificadores para los parámetros por referencia: **out** y **ref**. La diferencia fundamental es que a los parámetros definidos con **ref** se les debe asignar un valor antes de usarlo como parámetro del método, mientras que los parámetros definidos con **out** no es necesario inicializarlos antes de usarlos, pero se deberán asignar un valor dentro del método antes de usarlo. Otra diferencia de los parámetros por referencia entre Visual Basic y C#, es que en C# siempre habrá que usar el mismo modificador que se ha usado para definir el parámetro, pero en VB no se usará dicho modificador.

También se pueden definir arrays de parámetros. En ese caso, se podrá llamar al método usando un número variable de parámetros, aunque todos esos parámetros serán del mismo tipo y siempre serán por valor. Para declarar este tipo de parámetros, en C# se usará la instrucción **params** y en VB se usará **ParamArray**. Si se utiliza este tipo de parámetros, será el último de la lista de parámetros del método.

En Visual Basic .NET existe el concepto de parámetro opcional. Esos parámetros opcionales pueden ser por valor o por referencia y siempre habrá que definirlos con un valor por defecto, dicho valor se usará en el caso de que no se especifiquen al llamar al método.

La tabla 5 muestra la forma de declarar y usar los distintos tipos de parámetros de los métodos.

Tarea a realizar	C#	VB .NET
Parámetros por valor	`No se indica`	`Es el valor predeterminado, pero se puede usar la instrucción ByVal`
Definir parámetros por referencia	`<método>(ref var)` `{}`	`<método>(ByRef var As <tipo>)` `End <tipo método>`
C#: El valor debe estar inicializado antes de usarlo al llamar al método		
Llamar al método que define un parámetro por referencia	`<método>(ref var);`	`<método>(var)`
Definir parámetros por referencia	`<método>(out var)` `{}`	`<método>(ByRef var As <tipo>)` `End <tipo método>`
C#: No es necesario que el parámetro esté inicializado antes de llamar al método. Dentro del método, hay que asignar un valor al parámetro out antes de usarlo		
Llamar al método que define un parámetro por referencia	`<método>(out var);`	`<método>(var)`
Debido a la forma que se tratan las variables en Visual Basic, que siempre son inicializadas antes de usarlas, no hay esa diferencia en los parámetros por referencia		
Definir un array de parámetros	`<método>(params array[]){}`	`<método>(ParamArray var)` `End <tipo método>`
Llamar a un método que define un array de parámetros	`<método>([param1][, param2][, etc.])`	`<método>([param1][, param2][, etc.])`
Definir parámetros opcionales	`No aplicable, usar sobrecarga de métodos`	`método>([ByVal|ByRef]Optional var As <tipo> = <valor>)`
Sólo VB puede definir parámetros opcionales. Se pueden definir tantos parámetros opcionales como necesitemos, pero los parámetros opcionales siempre habrá que indicarlos después de los parámetros no opcionales y en caso de que se utilice **ParamArray**, éste debe aparecer después de todos los parámetros opcionales.		

Tabla 5. Parámetros de los métodos

Declaración de espacios de nombres, clases, estructuras, interfaces y enumeraciones

En nuestros programas podemos crear nuestros propios tipos de datos, incluso espacios de nombres.

Los tipos de datos y las interfaces las podemos declarar dentro de otros tipos de datos, pero lo habitual es que las declaremos dentro de espacios de nombres o directamente en el fichero de código. En este último caso, si estamos trabajando con Visual Studio .NET, el espacio de nombres que contendrá nuestros tipos, de forma predeterminada será el nombre del proyecto. En el caso de VB .NET ese espacio de nombres no se muestra en el código, estará "oculto" en las propiedades del proyecto; por otro lado, si estamos usando C#, el espacio de nombres siempre se mostrará en el código.

Las clases se pueden derivar de otras clases y también pueden implementar interfaces, así mismo las interfaces y estructuras pueden implementar otras interfaces. Aquí veremos cómo realizar todas esas declaraciones de tipos de datos.

Visual Basic .NET define un tipo de clase especial en la que todos los miembros de la clase son estáticos (están compartidos), es el tipo **`Module`**. En C# no existe un equivalente a un tipo **`Module`** de VB .NET, lo más parecido sería una clase en la que todos los miembros (campos, métodos, propiedades, etc.) están compartidos (son estáticos).

Los miembros estáticos pertenecen al tipo que lo define y están compartidos por todos los objetos (instancias) que se hayan creado en memoria.

La tabla 6 muestra cómo declarar espacios de nombres, clases, estructuras, interfaces y enumeraciones, así como la forma de implementar una interfaz o derivar una clase.

Definir los miembros de una interfaz y cómo se deben implementar en las clases que se deriven de esas interfaces

En las interfaces sólo se declaran los miembros (métodos, propiedades, indizadores y eventos), pero sin código que los haga operativos, ese código tendrá que definirlo la clase que implemente la interfaz.

Cuando una clase implementa una interfaz, la clase está obligada a definir los miembros definidos en dicha interfaz. En C# sólo es necesario declarar un miembro que tenga el mismo nombre que tiene en la interfaz. En Visual Basic .NET el nombre del miembro puede ser diferente al usado en la interfaz, pero debe indicarse explícitamente que ese miembro es el que se utiliza para definir el indicado por la interfaz.

En la tabla 7 vemos la forma de definir los miembros de una interfaz y cómo se implementan en las clases derivadas de esa interfaz.

Tarea a realizar	C#	VB .NET
Declarar un espacio de nombres	`namespace Guille.Pruebas {}`	`Namespace Guille.Pruebas End Namespace`
Declarar una clase (class)	`class Prueba {}`	`Class Prueba End Class`
Declarar una clase que se deriva de otra	`class Prueba2 : Prueba {}`	`Class Prueba2 Inherits Prueba End Class`
Declarar una clase que implementa una interfaz	`class Prueba3 : IPrueba {}`	`Class Prueba3 Implements IPrueba End Class`
Declarar una clase que se deriva de una clase e implementa una interfaz	`class Prueba4 : Prueba, IPrueba {}`	`Class Prueba4 Inherits Prueba Implements IPrueba End Class`
Declarar una clase que se deriva de una clase e implementa más de una interfaz	`class Prueba5 : Prueba, IPrueba, IPruebaB {}`	`Class Prueba5 Inherits Prueba Implements IPrueba, IPruebaB End Class`
Declarar una clase de tipo Module	No aplicable. En C# sería como class, pero todos los miembros usarán el modificador static.	`Module Pruebas End Module`
Declarar una estructura	`struct MiEstructura {}`	`Structure MiEstructura End Structure`
Declarar una estructura que implementa una o más interfaces	`struct MiEstructura2 : IPrueba, IPrueba2 {}`	`Structure MiEstructura2 Implements IPrueba, IPrueba2 End Structure`
Declarar una interfaz	`interface IPrueba {}`	`Interface IPrueba End Interface`
Declarar una interfaz que implemente otras interfaces	`interface IPrueba2 : IPrueba {}`	`Interface IPrueba2 Implements IPrueba End Interface`
Declarar una enumeración	`enum MiEnumeración {Lu, Ma, Mi};`	`Enum MiEnumeración Lu : Ma Mi End Enum`

La herencia de .NET sólo permite la herencia simple, sólo podemos derivar nuestras clases de una sola clase. Pero se permite la implementación de múltiples interfaces.En C#, no se indica explícitamente si se deriva de una clase o se implementa una interfaz, el único requisito es que la clase base se indique antes que las interfaces.

En VB .NET hay que indicar siempre si se está derivando la clase de otra, mediante **`Inherits`** o si se están implementando interfaces, mediante **`Implements`**.

Siempre debe aparecer Inherits antes de **`Implements`** y ambas antes de cualquier otro código que contenga la clase.

En C# las constantes (miembros de la enumeración) se separan con comas.En VB .NET cada una de las constantes que forman parte de una enumeración deben estar definidas en líneas independientes o en instrucciones diferentes (separándolas con dos puntos).

Tabla 6. Declaración de espacios de nombres, clases, estructuras, interfaces y enumeraciones

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

Equivalencia de instrucciones de C# y VB .NET (y III)

Tercera y última entrega de esta serie de artículos en la que hemos pretendido explicarle cómo hacer las mismas cosas (o casi) tanto en C# como en Visual Basic .NET

Instrucciones de decisión y operadores de comparación

En algunas de estas instrucciones se utilizan expresiones que devolverán un valor verdadero (true) o falso (false).

En esas expresiones podemos utilizar cualquiera de los operadores condicionales mostrados en la tabla 8. También podemos formar expresiones múltiples usando los operadores condicionales mostrados en esa misma tabla.

En la tabla 9 se muestran algunos ejemplos de cómo usar las instrucciones de selección o de tomas de decisiones según usemos `if... else` o `switch... case / Select Case`.

En los comentarios se indican algunas de las peculiaridades de C# y de Visual Basic .NET.

Las instrucciones para realizar bucles

Las instrucciones para realizar bucles nos permiten iterar un número determinado (o indeterminado) de veces sobre una parte del código. El código lo incluiremos dentro de dicho bucle.

En C# el código a usar en un bucle puede ser una sola instrucción, terminada con un punto y coma, o un bloque de código, incluido dentro de un par de llaves.

En VB .NET los bucles siempre estarán dentro de un bloque de código bien delimitado, es decir,

Descripción	C#	VB .NET
Operador de igualdad	`==`	`=`
Distinto	`!=`	`<>`
Menor, menor o igual, mayor o mayor o igual son los mismos operadores en ambos lenguajes:		`<, <=, >, >=`
Y	`&`	`And`
Or	`|`	`Or`
Xor	`^`	`Xor`
Negación (not)	`!`	`Not`
Y cortocircuitado	`&&`	`AndAlso`
Or cortocircuitado	`||`	`OrElse`

Tabla 8. Instrucciones (operadores) de comparación

Tarea a realizar	C#	VB .NET
Toma de decisiones con if	`if(a != b) <código>;`	`If a <> b Then <código>`

Toma de decisiones con if con varias instrucciones	`if(a > b){`
	`  <código>`
	`}`
		`If a > b Then`
		`  <código>`
		`End If`

Instrucción if que si se cumple haga una cosa y si no se cumple, haga otra, usando varias líneas	`if(a > b)`
	`  <código>`
	`else`
	`  <código>`
		`If a > b Then`
		`  <código>`
		`Else`
		`  <código>`
		`End If`

Varias instrucciones if... else asociadas a otro if.	`if(a != b)`
	`  <código>`
	`else`
	`  if(b > i && a > b){`
	`  <código>`
	`}`
		`If a <> b Then`
		`  <código>`
		`ElseIf b > i AndAlso a > b Then`
		`  <código>`
		`Else`
		`  <código>`
		`End If`

Varias instrucciones if anidadas.	`if(a != b)`
	`  if(b > i)`
	`  <código>;`
	`else`
	`  <código>;`
		`If a <> b Then`
		`  If b > i Then`
		`  <código>`
		`  End If`
		`Else`
		`  <código>`
		`End If`

En C# no se distingue entre un if de simple línea o multilínea, pero si queremos usar varias instrucciones en lugar de una sola acabada con un punto y coma, las incluiremos dentro de un bloque entre un par de llaves {}. En VB .NET podemos crear un bloque If multilínea acabándola con End If, tanto en el bloque If como en el bloque Else o ElseIf podemos indicar una o más líneas con instrucciones. Si no se indica End If se tomará como una instrucción de una línea, en la que se puede incluir también la parte Else, pero siempre en la misma línea física.		

Seleccionar entre varias opciones usando switch...	`switch(<expresión>){`
	`  case <constante1>:`
	`  <código>`
	`  break;`
	`  case <constante2>:`
	`  case <constante3>:`
	`  <código>`
	`  break;`
	`  default:`
	`  <código>`
	`  break;`
	`}`
		`Select Case <expresión>`
		`  Case <valor1>`
		`  <código>`
		`  Case <valor2>, <valor3>`
		`  <código>`
		`  Case <valorA> To <valorB>`
		`  <código>`
		`  Case Is <expresión>`
		`  <código>`
		`  Case Else`
		`  <código>`
		`End Select`

En C# sólo se pueden usar valores constantes con cada cláusula case. Podemos anidar una tras otra indicando varios case seguidos. Después de cada bloque case hay que usar la instrucción break o bien se debe salir del bloque de código, ya que no se permite pasar de un case a otro, salvo que usemos goto case <constante>. En VB .NET en cada cláusula Case se pueden indicar varios valores separados por comas, incluso un rango de valores usando To o un valor condicional usando Is. Esos valores no tienen porqué ser constantes, pueden ser también expresiones.		

Tabla 9. Instrucciones de decisión

habrá una parte inicial y otra instrucción que marcará el final de dicho bucle.

En la tabla 10 podemos ver cómo crear bucles `for`, `do`, `while` así como algunos aspectos a tener en cuenta, ya que

Constructores, destructores y cómo invocar a los constructores de una clase y de la clase base

Los constructores son el punto de inicio de las clases o tipos de datos, si en una clase no definimos un constructor, se creará uno predeterminado que simplemente asignarán un valor inicial a los campos (variables) que la clase contiene. También podemos crear nuestros propios constructores, de forma que puedan recibir parámetros para asignar valores a ciertos campos de nuestra clase. Incluso podemos crear distintas versiones de los constructores para permitir diferentes formas de instanciar (o crear) nuevos objetos.

En Visual Basic .NET un constructor se define por medio de un procedimiento (`Sub`) llamado `New`. En C# el constructor será un procedimiento "especial" que se llama igual que la propia clase (o tipo) que estamos definiendo.

Por otro lado, un destructor se utiliza cada vez que destruimos un objeto, en .NET Framework se llama finalizador, de hecho, en Visual Basic .NET se utiliza como destructor una sobrecarga del método `Finalize` declarado en la clase `Object`. En C# el destructor se define usando el nombre de la clase precedida con `~`. Un destructor se llamará cuando un objeto deje de estar en "ámbito" o se asigne un valor nulo a la variable. Hay que tener en cuenta que en .NET los objetos no se destruyen inmediatamente, sino que cuando dejan de ser

Tarea a realizar	C#	VB .NET

Bucle for	`for(<inicio>; <final>; <incremento>)`
	`  <código>`
		`For <contador> = <inicio> To <final>`
		`  <código>`
		`Next`

Bucle for infinito **Nota:** Espero que a nadie en su sano juicio se le ocurra hacer esto	`for(;;) ;`	`For i = 0 To 0 Step 0:` `Next`

Bucle for con incremento distinto de uno.	`for(int i = 0; i<10; i += 2)`
	`  <código>`
		`For i = 0 To 9 Step 2`
		`  <código>`
		`Next`

Bucle for para recorrer de mayor a menor.	`for(int i = 10; i>0; i–)`
	`  <código>`
		`For i = 10 To 1 Step -2`
		`  <código>`
		`Next`

Salir de un bucle for	`break;`	`Exit For`
En C# podemos indicar varias instrucciones después de if o else incluyéndolas dentro de un bloque entre un par de llaves {} o bien una sola instrucción acabada con punto y coma. En VB .NET podemos crear un bloque If acabándola con End If, tanto en la parte If como en la parte Else podemos indicar una o más líneas con instrucciones. Si no se indica End If se tomará como una instrucción en una sola línea.		

Bucle sin condición de término	`do{`
	`  <código>`
	`}while(true)`
		`Do`
		`  <código>`
		`Loop`

Bucle con una condición después de cada iteración, se repetirá mientras se cumpla la condición	`do{`
	`  <código>`
	`}while(<expresión>)`
		`Do`
		`  <código>`
		`Loop While <expresión>`

Bucle con una condición al principio	`while(<expresión>)`
	`  <código>;`
		`Do While <expresión>`
		`  <código>`
		`Loop`
		
		`While <expresión>`
		`  <código>`
		`End While`

Bucle que continúe la ejecución hasta que se cumpla la condición	`do{`
	`  <código>`
	`}while(! <expresión>)`
		`Do`
		`  <código>`
		`Loop Until <expresión>`

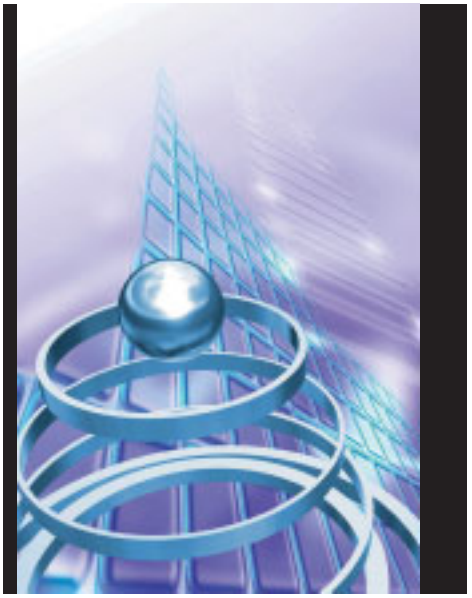
Bucle que continúe la ejecución hasta que se cumpla la condición, realizando la comprobación al principio del bucle	`while(! <expresión>){`
	`  <código>`
	`}`
		`Do Until <expresión>`
		`  <código>`
		`Loop`

Salir de un bucle do o while	`break;`	`Usar Exit seguida del tipo de bucle:`
		`Exit Do para Do... Loop`
		`Exit While para While...End While`

En C# los bucles do se utilizan con una instrucción while al final del bucle, esta instrucción es la que se encarga de comprobar si el bucle debe seguir ejecutándose o no. Si queremos que el bucle se repita indefinidamente podríamos usar una expresión que siempre devuelva un valor verdadero. En Visual Basic .NET podemos usar la instrucción While o la instrucción Until, en C# no existe la instrucción Until, pero se puede simular usando un while en el que se niega la expresión usada. En VB .NET se puede usar While como instrucción asociada a Do... Loop o como instrucción independiente, en ese caso el final del bloque del código se indicará con End While.		

Tabla 10. Instrucciones de bucles

"útiles", el recolector de basura (recolector de objetos no usados) se encarga de ellos y será el propio GC (`Garbage Collector`) se encargará de destruirlo, aunque esa destrucción no



se hará inmediatamente, este punto es importante ya que si nuestro objeto mantiene recursos externos éstos no se liberarán inmediatamente, en esos casos, es recomendable definir un método al que llamemos de forma explícita para liberar esos recursos justo cuando ya no los necesitemos.

Los constructores siempre llamarán a un constructor de la clase derivada, si no lo indicamos expresamente, el compilador intentará llamar a un constructor sin parámetros. En caso de que la clase base no tenga definido un constructor sin parámetros, tendremos que realizar nosotros esa llamada, indicando el constructor adecuado, si no lo hacemos se producirá un error de compilación.

Por otra parte, los destructores siempre llaman al método `Finalize` de la clase base, de forma que se destruyan todos los objetos creados. En este caso no es necesario que lo llamemos de forma explícita.

En la tabla 11 podemos ver cómo definir los constructores y destructores, así como la forma de invocar a otra sobrecarga de un constructor de la misma clase e incluso de la clase de la que se deriva.

Tarea a realizar	C#	VB .NET
Definir un constructor de una clase. En estos ejemplos, supondremos que la clase se llama Cliente	`public Cliente()` `{}`	`Public Sub New()` `End Sub`
Definir un constructor que recibe un parámetro	`public Cliente(int id)` `{}`	`Public Sub New(id As Integer)` `End Sub`
Definir un destructor o finalizador	`~Cliente()` `{}`	`Public Overrides Sub Finalize()` `End Sub`
En C# el constructor siempre se llama como la clase. En VB .NET el constructor siempre es un método Sub llamado New. Los destructores sólo se pueden usar en las clases no en las estructuras.		

Definir un constructor que llama a otro constructor de la propia clase.	`public Cliente(int id, string nombre) : this(id)`
	`{}`
		`Public Sub New(id As Integer, nombre As String)`
		`Me.New(id)`
		`End Sub`

Definir un constructor que llama a otro constructor de la clase base.	`public Cliente(int id, string nombre) : base(id)`
	`{}`
		`Public Sub New(id As Integer, nombre As String)`
		`MyBase.New(id)`
		`End Sub`

Tabla 11. Constructores y destructores.

Tarea a realizar	C#	VB .NET
Definir una clase abstracta	`abstract class Prueba` `{}`	`MustInherit Class Prueba` `End Class`
Definir una clase sellada	`sealed class Prueba2` `{}`	`NotInheritable Class Prueba2` `End Class`
Definir un miembro abstracto	`abstract void Prueba();`	`MustOverride Sub Prueba()`
Definir un miembro virtual	`virtual void Prueba2()` `{}`	`Overridable Sub Prueba2()` `End Sub`
Los miembros abstractos sólo definen el método o propiedad, pero no contienen código que lo defina.		
Redefinir un miembro abstracto o virtual	`override void Prueba()` `{}`	`Overrides Sub Prueba()` `End Sub`
Definir un miembro que oculta a otro de la clase base	`new void Prueba3()` `{}`	`Shadows Sub Prueba3()` `End Sub`
En VB .NET si se quiere ocultar un miembro virtual, además de usar Shadows debemos usar la instrucción Overloads. **Nota sobre seguridad:** Los miembros declarados como virtual internal (Overridable Friend en VB) en teoría sólo se pueden reemplazar en clases definidas en el propio ensamblado, pero esa "restricción" sólo es aplicable a los compiladores de C# y de VB, el CLR no tiene esa restricción, por tanto es teóricamente posible reemplazar esos miembros desde otro ensamblado, al menos por compiladores que no tengan dicha restricción. Para más información: /cpguide/html/cpconkeyconceptsinsecurity.htm /cpguide/html/cpconsecurityconcernsforinternalvirtualoverloadsoverridablefriendkeywords.htm		

Tabla 11. Constructores y destructores.

Definir clases abstractas y selladas, miembros abstractos y virtuales, redefinir y ocultar métodos

Las clases abstractas son clases que sólo se pueden utilizar para derivar nuevas clases, no se podrán usar para crear nuevos objetos. Una clase abstracta puede contener métodos y propiedades normales así como abstractos, los métodos abstractos sólo se definen como en las interfaces: sin código que los hagan operativos.

Por otro lado los métodos virtuales son los que podremos redefinir en la clase derivada, para dar la funcionalidad adecuada que creamos conveniente. Las referencias a las instancias creadas en memoria siempre usarán las versiones redefinidas de los métodos (o miembros) virtuales. Por defecto, los métodos y propiedades de una clase no son virtuales, es decir, no se pueden redefinir en las clases derivadas, sin embargo podemos ocultarlos. Esos miembros ocultados sólo pertenecerán a la instancia de la clase que los define, no a las referencias obtenidas a través de tipos de la clase base.

Los miembros abstractos siempre son virtuales.

También podemos definir clases selladas, lo contrario de las clases abstractas, es decir, clases que no se pueden usar para derivar nuevas clases a partir de ellas.

También podemos ocultar tipos además de los miembros de una clase.

En la tabla 12 podemos ver una lista de las instrucciones usadas para declarar clases abstractas, miembros abstractos, virtuales, así como las instrucciones usadas para ocultar miembros y para redefinir los miembros virtuales de las clases base.

Por supuesto no se han cubierto todas las posibilidades sintácticas entre los dos lenguajes más usados de la plataforma .NET, pero espero que al menos ahora tengas una idea bastante aproximada de cómo hacer las tareas más comunes tanto en C# como en Visual Basic .NET, de forma que en cualquier momento te resulte fácil poder escribir código en cualquiera de estos dos lenguajes.