Contenido

- **Lección 1: El sistema de tipos**
    - Tipos primitivos
    - Variables y constantes
    - Enumeraciones
    - Arrays (matrices)

- **Lección 2: Clases y estructuras**
    - Clases
    - Definir una clase
    - Instanciar una clase
    - Estructuras
    - Accesibilidad
    - Propiedades
    - Interfaces

- **Lección 3: Manejo de excepciones**
    - Manejo de excepciones

- **Lección 4: Eventos y delegados**
    - Eventos
    - Definir y producir eventos en una clase
    - Delegados
    - Definir un evento bien informado

- **Lección 5: Atributos**
    - Atributos

Ver vídeo 1 de esta introducción (Configurar el IDE - video en Visual Studio 2005 válido para Visual Studio 2008)
Ver vídeo 2 de esta introducción (Crear un proyecto - video en Visual Studio 2005 válido para Visual Studio 2008)

Lección 1: El sistema de tipos

- Tipos primitivos
- Variables y constantes
- Enumeraciones
- Arrays (matrices)

Introducción

En esta primera lección veremos los tipos de datos que .NET Framework pone a nuestra disposición y cómo tendremos que usarlos desde Visual Basic 2008.

A continuación daremos un repaso a conceptos básicos o elementales sobre los tipos de datos, que si bien nos serán familiares, es importante que lo veamos para poder comprender mejor cómo están definidos y organizados los tipos de datos en .NET.

Tipos de datos de .NET

Visual Basic 2008 está totalmente integrado con .NET Framework, por lo tanto, los tipos de datos que podremos usar con este lenguaje serán los definidos en este "marco de trabajo", por este motivo vamos a empezar usando algunas de las definiciones que nos encontraremos al recorrer la documentación que acompaña a este lenguaje de programación.

Los tipos de datos que podemos usar en Visual Basic 2008 son los mismo tipos de datos definidos en .NET Framework y por tanto están soportados por todos los lenguajes que usan esta tecnología. Estos tipos comunes se conocen como el *Common Type System*, (CTS), que traducido viene a significar el sistema de tipos comunes de .NET. El hecho de que los tipos de datos usados en todos los lenguajes .NET estén definidos por el propio Framework nos asegura que independientemente del lenguaje que estemos usando, siempre utilizaremos el mismo tipo interno de .NET, si bien cada lenguaje puede usar un nombre (o alias) para referirse a ellos, aunque lo importante es que siempre serán los mismos datos, independientemente de cómo se llame en cada lenguaje. Esto es una gran ventaja, ya que nos permite usarlos sin ningún tipo de problemas para acceder a ensamblados creados con otros lenguajes, siempre que esos lenguajes sean compatibles con los tipos de datos de .NET.

En los siguientes enlaces tenemos los temas a tratar en esta primera lección del módulo sobre las características del lenguaje Visual Basic 2008.

- **Tipos primitivos**
    - Sufijos o caracteres y símbolos identificadores para los tipos
    - Tipos por valor y tipos por referencia
    - Inferencia de tipos
- **Variables y constantes**
    - Consejo para usar las constantes
    - Declarar variables
    - Declarar variables y asignar el valor inicial
    - El tipo de datos Char
    - Obligar a declarar las variables con el tipo de datos
    - Aplicar Option Strict On a un fichero en particular
    - Aplicar Option Stict On a todo el proyecto
    - Más opciones aplicables a los proyectos
    - Tipos anulables
    - Tipos anónimos
- **Enumeraciones: Constantes agrupadas**
    - El nombre de los miembros de las enumeraciones
    - Los valores de una enumeración no son simples números
- **Arrays (matrices)**
    - Declarar arrays
    - Declarar e inicializar un array
    - Cambiar el tamaño de un array
    - Eliminar el contenido de un array
    - Los arrays son tipos por referencia

Lección 1: El sistema de tipos

Tipos primitivos
• Variables y constantes
• Enumeraciones
• Arrays (matrices)

Tipos primitivos

Veamos en la siguiente tabla los tipos de datos definidos en .NET Framework y los alias utilizados en Visual Basic 2008.

.NET Framework	VB 2008
System.Boolean	Boolean
System.Byte	Byte
System.Int16	Short
System.Int32	Integer
System.Int64	Long
System.Single	Single
System.Double	Double
System.Decimal	Decimal
System.Char	Char
System.String	String
System.Object	Object
System.DateTime	Date
System.SByte	SByte
System.UInt16	UShort
System.UInt32	UInteger
System.UInt64	ULong

Tabla 2.1. Tipos de datos y equivalencia entre lenguajes

Debemos tener en cuenta, al menos si el rendimiento es una de nuestra prioridades, que las cadenas en .NET son inmutables, es decir, una vez que se han creado no se pueden modificar y en caso de que queramos cambiar el contenido, .NET se encarga de desechar la anterior y crear una nueva cadena, por tanto si usamos las cadenas para realizar concatenaciones (unión de cadenas para crear una nueva), el rendimiento será muy bajo, si bien existe una clase en .NET que es

ideal para estos casos y cuyo rendimiento es superior al tipo *String*: la clase *StringBuilder*.

Las últimas filas mostradas en la tabla son tipos especiales que si bien son parte del sistema de tipos comunes (CTS) no forman parte de la *Common Language Specification* (CLS), es decir la especificación común para los lenguajes "compatibles" con .NET, por tanto, si queremos crear aplicaciones que puedan interoperar con todos los lenguajes de .NET, esos tipos no debemos usarlos como valores de devolución de funciones ni como tipo de datos usado en los argumentos de las funciones, propiedades o procedimientos.

Los tipos mostrados en la tabla 2.1 son los tipos primitivos de .NET y por extensión de Visual Basic 2008, es decir son tipos "elementales" para los cuales cada lenguaje define su propia palabra clave equivalente con el tipo definido en el CTS de .NET Framework. Todos estos tipos primitivos podemos usarlos tanto por medio de los tipos propios de Visual Basic, los tipos definidos en .NET o bien como literales. Por ejemplo, podemos definir un número entero literal indicándolo con el sufijo **I**: ***12345I*** o bien asignándolo a un valor de tipo *Integer* o a un tipo *Sytem.Int32* de .NET. La única excepción de los tipos mostrados en la tabla 1 es el tipo de datos *Object*, este es un caso especial del que nos ocuparemos en la próxima lección.

Sufijos o caracteres y símbolos identificadores para los tipos

Cuando usamos valores literales numéricos en Visual Basic 2008, el tipo de datos que le asigna el compilador es el tipo *Double*, por tanto si nuestra intención es indicar un tipo de datos diferente podemos indicarlos añadiendo una letra como sufijo al tipo, esto es algo que los más veteranos de VB6 ya estarán acostumbrados, e incluso los más noveles también, en Visual Basic 2008 algunos de ellos se siguen usando, pero el tipo asociado es el equivalente al de este nuevo lenguaje (tal como se muestra en la tabla 1), por ejemplo para indicar un valor entero podemos usar la letra **I** o el signo **%**, de igual forma, un valor de tipo entero largo (*Long*) lo podemos indicar usando **L** o **&**, en la siguiente tabla podemos ver los caracteres o letra que podemos usar como sufijo en un literal numérico para que el compilador lo identifique sin ningún lugar a dudas.

Tipo de datos	Símbolo	Carácter
Short	N.A.	S
Integer	%	I
Long	&	L
Single	!	F
Double	#	R
Decimal	@	D
UShort	N.A.	US
UInteger	N.A.	UI
ULong	N.A.	UL

Tabla 2.2. Sufijos para identificar los tipos de datos

El uso de estos caracteres nos puede resultar de utilidad particularmente para los tipos de datos que no se pueden convertir en un valor doble.

> **Nota:**
> Los sufijos pueden indicarse en minúsculas, mayúsculas o cualquier combinación de mayúscula y minúscula.
> Por ejemplo, el sufijo de un tipo ULong puede ser: UL, Ul, ul, uL, LU, Lu, lU o lu.
> Para evitar confusiones, se recomienda siempre indicarlos en mayúsculas, independientemente de que Visual Basic no haga ese tipo de distinción.

Por ejemplo, si queremos asignar este valor literal a un tipo *Decimal*: ***12345678901234567890***, tal como vemos en la figura 1, el IDE de Visual Basic 2008 nos indicará que existe un error de desbordamiento (*Overflow*) ya que esa cifra es muy grande para usarlo como valor *Double*, pero si le agregamos el sufijo **D** o **@** ya no habrá dudas de que estamos tratando con un valor *Decimal*.

```
Option Strict On

Module Module1

    Sub Main()
        Dim m As Decimal = 12345678901234567890
                           Overflow.
    End Sub

End Module
```

Figura 2.1. Error de desbordamiento al intentar asignar un valor Double a una variable Decimal

Tipos por valor y tipos por referencia

Los tipos de datos de .NET los podemos definir en dos grupos:

- Tipos por valor
- Tipos por referencia

Los tipos por valor son tipos de datos cuyo valor se almacena en la pila o en la memoria "cercana", como los numéricos que hemos visto. Podemos decir que el acceso al valor contenido en uno de estos tipos es directo, es decir se almacena directamente en la memoria reservada para ese tipo y cualquier cambio que hagamos lo haremos directamente sobre dicho valor, de igual forma cuando copiamos valores de un tipo por valor a otro, estaremos haciendo copias independientes.

Por otro lado, los tipos por referencia se almacenan en el "monto" (*heap*) o memoria "lejana", a diferencia de los tipos por valor, los tipos por referencia lo único que almacenan es una referencia (o puntero) al valor asignado. Si hacemos copias de tipos por referencia, realmente lo que copiamos es la referencia propiamente dicha, pero no el contenido.

Estos dos casos los veremos en breve con más detalle.

Inferencia de tipos

Una de las características nuevas en Visual Basic 2008 es la inferencia de tipos.

Se conoce como inferencia de tipos a la característica de Visual Basic para inferir el tipo de un dato al ser inicializado.

Para que la inferencia de tipos sea efectiva, deberemos activar la opción **Option Infer** a **True**, aunque por defecto, ese es el valor que tiene el compilador de Visual Basic. Sin embargo, si se hace una migración de una aplicación de Visual Basic a Visual Basic 2008, el valor de esta opción será **False**.

Supongamos por lo tanto, la siguiente declaración:

```
Dim datoDeclarado = 2008
```

En este ejemplo, la variable *datoDeclarado*, será una variable de tipo **Integer (Int32)**.

Si deseamos cambiar el tipo de dato a un tipo **Int64** por ejemplo, el compilador nos devolverá un error. Así, el siguiente ejemplo **no** será válido en Visual Basic 2008 con la opción de inferencia activada:

```
Dim datoDeclarado = 2008

datoDeclarado = Int64.MaxValue
```

Ahora bien, si cambiamos el valor de **Option Infer** a **False**, el mismo ejemplo será correcto.

¿Dónde está la diferencia?.
En este último caso, el caso de tener desactivada la opción de inferencia, la declaración de la variable *datoDeclarado* nos indica que es un tipo de dato **Object** en su origen, y que al darle un valor **Integer**, ésta funciona como una variable entera. Al cambiar su valor a **Long**, esta variable que es de tipo **Object**, cambia sin problemas a valor **Long**.
En todo este proceso, hay un problema claro de rendimiento con acciones de boxing y unboxing que no serían necesarias si tipáramos la variable con un tipo concreto.
Eso es justamente lo que hace la opción **Option Infer** por nosotros. Nos permite declarar una variable con el tipo inferido, y ese tipo de datos se mantiene dentro de la aplicación, por lo que nos da la seguridad de que ese es su tipo de dato, y que ese tipo de dato no va a variar.

Ver vídeo 1 de esta lección (Tipos de datos primitivos - video en Visual Studio 2005 válido para Visual Studio 2008)
Ver vídeo 2 de esta lección (Tipos por valor y por referencia - video en Visual Studio 2005 válido para Visual Studio 2008)

Lección 1: El sistema de tipos

- Tipos primitivos

Variables y constantes

- Enumeraciones
- Arrays (matrices)

Variables y constantes

Disponer de todos estos tipos de datos no tendría ningún sentido si no pudiéramos usarlos de alguna otra forma que de forma literal. Y aquí es donde entran en juego las variables y constantes, no vamos a contarte qué son y para que sirven, salvo en el caso de las constantes, ya que no todos los desarrolladores las utilizamos de la forma adecuada.

Consejo para usar las constantes

Siempre que tengamos que indicar un valor constante, ya sea para indicar el máximo o mínimo permitido en un rango de valores o para comprobar el término de un bucle, deberíamos usar una constante en lugar de un valor literal, de esta forma si ese valor lo usamos en varias partes de nuestro código, si en un futuro decidimos que dicho valor debe ser diferente, nos resultará más fácil realizar un solo cambio que cambiarlo en todos los sitios en los que lo hemos usado, además de que de esta forma nos aseguramos de que el cambio se realiza adecuadamente y no tendremos que preocuparnos de las consecuencias derivadas de no haber hecho el cambio en todos los sitios que deberíamos.

Las constantes se definen utilizando la instrucción *Const* seguida del nombre, opcionalmente podemos indicar el tipo de datos y por último una asignación con el valor que tendrá. Como veremos en la siguiente sección, podemos obligar a Visual Basic 2008 a que en todas las constantes (y variables) que declaremos, tengamos que indicar el tipo de datos.
Para declarar una constante lo haremos de la siguiente forma:

```
Const maximo As Integer = 12345678
```

Declarar variables

La declaración de las variables en Visual Basic 2008 se hace por medio de la instrucción *Dim* seguida del nombre de la constante y del tipo de datos que esta

contendrá. Con una misma instrucción *Dim* podemos declarar más de una variable, incluso de tipos diferentes, tal y como veremos a continuación.

La siguiente línea de código declara una variable de tipo entero:

```
Dim i As Integer
```

Tal y como hemos comentado, también podemos declarar en una misma línea más de una variable:

```
Dim a, b, c As Integer
```

En este caso, las tres variables las estamos definiendo del mismo tipo, que es el indicado al final de la declaración.

> **Nota:**
> Como hemos comentado, en Visual Basic 2008 se pueden declarar las constantes y variables sin necesidad de indicar el tipo de datos que contendrán, pero debido a que eso no es una buena práctica, a lo largo de este curso siempre declararemos las variables y constantes con el tipo de datos adecuado a su uso.

Declarar variables y asignar el valor inicial

En Visual Basic 2008 también podemos inicializar una variable con un valor distinto al predeterminado, que en los tipos numéricos es un cero, en las fechas es el 1 de enero del año 1 a las doce de la madrugada (**#01/01/0001 12:00:00AM#**) y en la cadenas es un valor nulo (*Nothing*), para hacerlo, simplemente tenemos que indicar ese valor, tal como veremos es muy parecido a como se declaran las constantes. Por ejemplo:

```
Dim a As Integer = 10
```

En esa misma línea podemos declarar y asignar más variables, pero todas deben estar indicadas con el tipo de datos:

```
Dim a As Integer = 10, b As Integer = 25
```

Por supuesto, el tipo de datos puede ser cualquiera de los tipos primitivos:

```
Dim a As Integer = 10, b As Integer = 25, s As String = "Hola"
```

Aunque para que el código sea más legible, y fácil de depurar, no deberíamos mezclar en una misma instrucción *Dim* más de un tipo de datos.

> **Nota:**
> Es importante saber que en las cadenas de Visual Basic 2008 el valor de una variable de tipo String no inicializada NO es una cadena vacía, sino un valor nulo (Nothing).

El tipo de datos Char

En Visual Basic 2008 podemos declarar valores de tipo *Char*, este tipo de datos es un carácter Unicode y podemos declararlo y asignarlo a un mismo tiempo. El problema con el que nos podemos encontrar es a la hora de indicar un carácter literal.
Podemos convertir un valor numérico en un carácter o bien podemos convertir un carácter en su correspondiente valor numérico.

```
Dim c As Char

c = Chr(65)

Dim n As Byte

n = Asc(c)
```

En Visual Basic 2008 los tipos *Char* se pueden asignar a variables de tipo *String* y se hará una conversión automática sin necesidad de utilizar funciones de conversión.

Si nuestra intención es asignar un valor *Char* a una variable, además de la función *Chr*, podemos hacerlo con un literal, ese valor literal estará encerrado entre comillas dobles, (al igual que una cadena), aunque para que realmente sea un carácter debemos agregarle una **c** justo después del cierre de las comillas dobles:

```
Dim c As Char = "A"c
```

Obligar a declarar las variables con el tipo de datos

Visual Basic 2008 nos permite, (lamentablemente de forma predeterminada), utilizar las variables y constantes sin necesidad de indicar el tipo de datos de estas, pero, como comentábamos al principio, podemos obligar a que nos avise cuando no lo estamos haciendo, ya que como decíamos en la nota, es una buena costumbre indicar **siempre** el tipo de datos que tendrán nuestras variables y constantes.
Esa obligatoriedad la podemos aplicar a todo el proyecto o a un módulo en particular, para ello tenemos que usar la instrucción *Option Strict On*, una vez indicado, se aplicará a todo el código, no solo a las declaraciones de variables, constantes o al tipo de datos devuelto por las funciones y propiedades, sino también a las conversiones y asignaciones entre diferentes tipos de datos.

No debemos confundir *Option Strict* con *Option Explicit*, este último, sirve para que siempre tengamos que declarar todas las variables, mientras que el primero lo que hace es obligarnos a que esas declaraciones tengan un tipo de datos.

Tanto una como la otra tienen dos estados: conectado o desconectado dependiendo de que agreguemos *On* u *Off* respectivamente.

Insistimos en la recomendación de que siempre debemos "conectar" estas dos opciones, si bien *Option Explicit On* ya viene como valor por defecto, cosa que no ocurre con *Option Strict*, que por defecto está desconectado.

Aplicar Option Strict On a un fichero en particular

Cuando agregábamos un nuevo fichero a nuestro proyecto de Visual Basic 2008 si ya tenemos predefinida las opciones "estrictas", como es el caso de *Option Explicit On*, estas no se añadirán a dicho fichero, (en un momento veremos cómo hacerlo para que siempre estén predefinidas), pero eso no significa que no se aplique, aunque siempre podemos escribir esas instrucciones (con el valor *On* al final) en cada uno de los ficheros de código que agreguemos a nuestro proyecto. Si nos decidimos a añadirlas a los ficheros, esas líneas de código deben aparecer al principio del fichero y solamente pueden estar precedidas de comentarios.

En la figura 2.1 mostrada en la lección anterior, tenemos una captura del editor de Visual Basic 2008 en la que hemos indicado que queremos tener comprobación estricta.

Aplicar Option Stict On a todo el proyecto

También podemos hacer que *Option Strict* funcione igual que *Option Explicit*, es decir, que esté activado a todo el proyecto, en este caso no tendríamos que indicarlo en cada uno de los ficheros de código que formen parte de nuestro proyecto, si bien solamente será aplicable a los que no tengan esas instrucciones, aclaremos esto último: si *Option Strict* (u *Option Explicit*) está definido de forma global al proyecto, podemos desactivarlo en cualquiera de los ficheros, para ello simplemente habría que usar esas declaraciones pero usando *Off* en lugar de *On*. De igual forma, si ya está definido globalmente y lo indicamos expresamente, no se producirá ningún error. Lo importante aquí es saber que siempre se usará el estado indicado en cada fichero, independientemente de cómo lo tengamos definido a nivel de proyecto.

Para que siempre se usen estas asignaciones en todo el proyecto, vamos a ver cómo indicarlo en el entorno de Visual Basic 2008.

Abrimos Visual Studio 2008 y una vez que se haya cargado, (no hace falta crear ningún nuevo proyecto, de este detalle nos ocuparemos en breve), seleccionamos la opción **Herramientas>Opciones...** se mostrará un cuadro de diálogo y del panel izquierdo seleccionamos la opción **Proyectos y soluciones**, la expandimos y seleccionamos **Valores predeterminados de VB** y veremos ciertas opciones, tal como podemos comprobar en la figura 2.2:

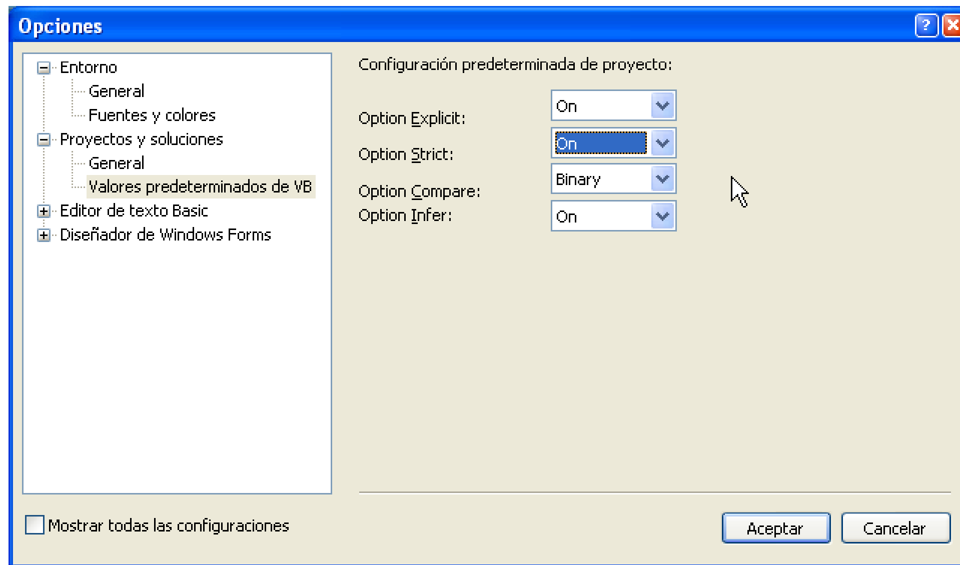


Figura 2.2. Opciones de proyectos (opciones mínimas)

De la lista despegable **Option Strict**, seleccionamos **On.** Por defecto ya estarán seleccionadas las opciones **On** de **Option Explicit** y **Binary** de **Option Compare**, por tanto no es necesario realizar ningún cambio más, para aceptar los cambios y cerrar el cuadro de diálogo, presionamos en el botón **Aceptar**.

Si en la ventana de opciones no aparece toda la configuración pod emos hacer que se muestren todas las disponibles. Para hacerlo, debemos marcar la casilla que está en la parte inferior izquierda en la que podemos leer: **Mostrar todas las configuraciones**, al seleccionar esa opción nos mostrará un número mayor de opciones, tal como podemos ver en la figura 2.3:

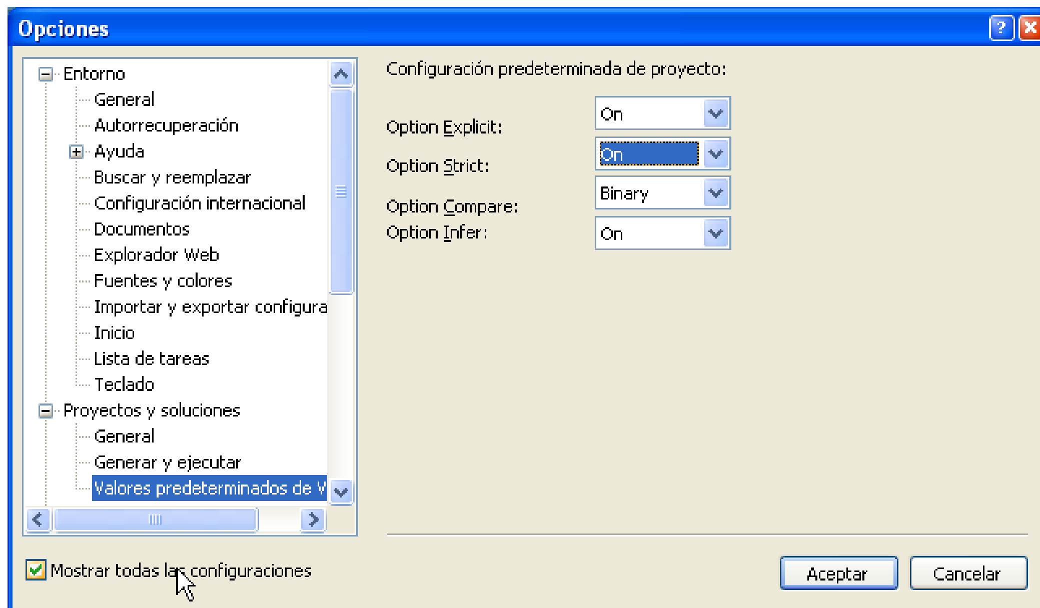


Figura 2.3. Opciones de proyectos (todas las opciones)

Desde este momento el compilador de Visual Basic se volverá estricto en todo lo relacionado a las declaraciones de variables y conversiones, tal como vemos en la figura 2.4 al intentar declarar una variable sin indicar el tipo de datos.

```
Option Strict On

Module Module1

    Sub Main()
        Dim m As Decimal = 12345678901234567890D

        Dim a
            Option Strict On requiere que todas las declaraciones de variables tengan una cláusula 'As'.

    End Sub

End Module
```

Figura 2.4. Aviso de Option Strict al declarar una variable sin tipo

> **Nota:**
> Una de las ventajas del IDE (*Integrated Development Environment*, entorno de desarrollo integrado) de Visual Basic 2008 es que nos avisa al momento de cualquier fallo que cometamos al escribir el código, este "pequeño" detalle, aunque alguna veces puede llegar a parecer fastidioso, nos facilita la escritura de código, ya que no tenemos que esperar a realizar la compilación para que tengamos constancia de esos fallos.

Más opciones aplicables a los proyectos

Aunque en estos módulos no trataremos a fondo el entorno de desarrollo, ya que la finalidad de este curso online es tratar más en el código propiamente dicho, vamos a mostrar otro de los sitios en los que podemos indicar dónde indicar que se haga una comprobación estricta de tipos y, como veremos, también podremos indicar algunas "nuevas peculiaridades" de Visual Basic 2008, todas ellas relacionadas con el tema que estamos tratando.

Cuando tengamos un proyecto cargado en el IDE de Visual Studio 2008, (pronto veremos cómo crear uno), podemos mostrar las propiedades del proyecto, para ello seleccionaremos del menú **Proyecto** la opción **Propiedades de <NombreDelProyecto>** y tendremos un cuadro de diálogo como el mostrado en la figura 2.5.

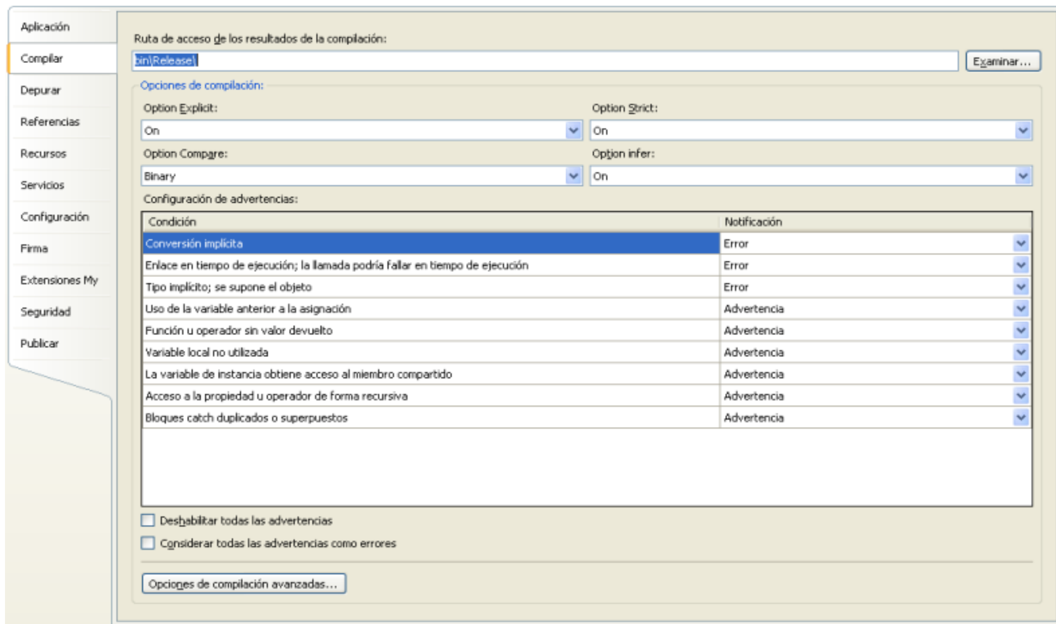


Figura 2.5. Ficha Compilar de las opciones del proyecto actual

Seleccionando la ficha **Compilar**, además de las típicas opciones de **Option Strict**, **Option Explicit**,**Option Compare** y **Option Infer**, (estas asignaciones solo serán efectivas para el proyecto actual), tendremos cómo queremos que reaccione el compilador si se cumple algunas de las condiciones indicadas. Entre esas condiciones, tenemos algo que muchos desarrolladores de Visual Basic siempre hemos querido tener: Que nos avise cuando una variable la hemos declarado pero no la utilizamos (**Variable local no utilizada**). Al tener marcada esta opción (normalmente como una **Advertencia**), si hemos declarado una variable y no la usamos en el código, (siempre que no le hayamos asignado un valor al declararla), nos avisará, tal como podemos ver en la figura 2.6:

```
Option Strict On

Module Module1

    Sub Main()
        Dim m As Decimal = 12345678901234567890D

        Dim a As Integer
        |   Variable local sin utilizar: 'a'.

    End Sub

End Module
```

Figura 2.6. Aviso de variable no usada

Tipos anulables

Otra interesantísima característica de Visual Basic 2008 que conviene conocer, es lo que se denominan tipos anulables.

Los tipos anulables no son nuevos en Visual Basic, de hecho su origen lo encontramos en Visual Studio 2005, aunque eso sí, implementando la clase *Nullable(Of T)*.

Con Visual Basic 2008 no es necesario declarar ninguna clase para implementar tipos anulables dentro de nuestras aplicaciones, y podemos declararlos de forma directa.

Un tipo de dato anulable nos permitirá declarar una variable que podrá tener un tipo de dato nulo.

Si hemos estado atentos hasta ahora, hemos podido ver que las variables numéricas por ejemplo, se inicializan a 0.
Si quisiéramos por la razón que fuera, declarar esa variable como nula para saber si en un determinado momento ha cambiado de valor o cualquier otra acción, deberíamos utilizar los tipos de datos anulables, o bien, utilizar técnicas más rudimentarias como una variable de tipo *Boolean* que nos permitiera saber si ha habido un cambio de valor en una variable, sin embargo, coincidirá conmigo en que el uso de un tipo de datos anulable es más natural y directo.

De esta forma, podríamos trabajar con tipos de datos anulables o que los declararemos como nulos.
A continuación veremos un ejemplo:

```
Dim valor As Integer?
```

Para acceder al valor de un tipo anulable, podríamos hacerlo de la forma habitual, ahora bien, si no sabemos si el valor es nulo o no, podríamos acceder a su valor preguntando por él mediante la propiedad *HasValue*. La propiedad *Value* nos indicará también, el valor de esa variable.
Un ejemplo que aclare esta explicación es el que podemos ver a continuación:

```
Dim valor As Integer?

If Valor.HasValue Then

   MessageBox.Show(valor.Value)

End If
```

Otra característica de los tipos anulables es la posibilidad de utilizar la función *GetValueOrDefault*.
Esta función nos permitirá acceder al valor de la variable si no es nulo, y al valor que le indiquemos si es nulo.

Un breve ejemplo de este uso es el que se indica a continuación:

```
Dim valor As Integer?
```

```
    valor = 2008

    MessageBox.Show(valor.GetValueOrDefault(2012))

End If
```

En este ejemplo, el compilador nos devolvería el valor *2008*, ya que *GetValueOrDefault* sabe que la variable no posee un valor nulo y que por lo tanto, debe obtener el valor no nulo de la variable anulable. En el caso de que no hubiéramos dado ningún valor a la variable, la aplicación obtendría el valor *2012*.

Tipos anónimos

Esta característica de Visual Basic 2008, nos permite declarar los tipos de datos de forma implícita desde el código de nuestras aplicaciones.

Un ejemplo práctico de declaración de tipos anónimos es el siguiente:

```
Dim declaracion = New With {.Nombre = "Carlos", .Edad = 27}

MessageBox.Show(String.Format("{0} tiene {1} años", _

                              declaracion.Nombre,
declaracion.Edad))
```

Como podemos ver en el ejemplo anterior, hemos declarado un objeto al que no hemos indicado ningún tipo de dato concreto, pero a la hora de crear ese objeto, hemos creado implícitamente un miembro *Nombre* y un miembro *Edad*.

Ver vídeo 1 de esta lección (Declarar variables) - video en Visual Studio 2005 válido para Visual Studio 2008
Ver vídeo 2 de esta lección (Definir constantes) - video en Visual Studio 2005 válido para Visual Studio 2008

Lección 1: El sistema de tipos

- Tipos primitivos
- Variables y constantes

Enumeraciones

- Arrays (matrices)

Enumeraciones: Constantes agrupadas

Una enumeración es una serie de constantes que están relacionadas entre sí. La utilidad de las enumeraciones es más manifiesta cuando queremos manejar una serie de valores constantes con nombre, es decir, podemos indicar un valor, pero en lugar de usar un literal numérico, usamos un nombre, ese nombre es, al fin y al cabo, una constante que tiene un valor numérico.

En Visual Basic 2008 las enumeraciones pueden ser de cualquier tipo numérico integral, incluso enteros sin signo, aunque el valor predefinido es el tipo *Integer*. Podemos declarar una enumeración de varias formas:

1- Sin indicar el tipo de datos, por tanto serán de tipo *Integer*:

```
Enum Colores

    Rojo

    Verde

    Azul

End Enum
```

2- Concretando explícitamente el tipo de datos que realmente tendrá:

```
Enum Colores As Long

    Rojo

    Verde

    Azul

End Enum
```

En este segundo caso, el valor máximo que podemos asignar a los miembros de una enumeración será el que pueda contener un tipo de datos *Long*.

3- Indicando el atributo *FlagsAttibute*, (realmente no hace falta indicar el sufijo *Attribute* cuando usamos los atributos) de esta forma podremos usar los valores de la enumeración para indicar valores que se pueden "sumar" o complementar entre sí, pero sin perder el nombre, en breve veremos qué significa esto de "no perder el nombre".

```
<Flags()> _

Enum Colores As Byte

    Rojo = 1

    Verde = 2

    Azul = 4

End Enum
```

Nota:
Los atributos los veremos con más detalle en otra lección de este mismo módulo.

El nombre de los miembros de las enumeraciones

Tanto si indicamos o no el atributo *Flags* a una enumeración, la podemos usar de esta forma:

```
Dim c As Colores = Colores.Azul Or Colores.Rojo
```

Es decir, podemos "sumar" los valores definidos en la enumeración. Antes de explicar con detalle que beneficios nos puede traer el uso de este atributo, veamos una característica de las enumeraciones.

Como hemos comentado, las enumeraciones son constantes con nombres, pero en Visual Basic 2008 esta definición llega más lejos, de hecho, podemos saber "el nombre" de un valor de una enumeración, para ello tendremos que usar el método *ToString*, (el cual se usa para convertir en una cadena cualquier valor numérico). Por ejemplo, si tenemos la siguiente asignación:

```
Dim s As String = Colores.Azul.ToString
```

La variable **s** contendrá la palabra **"Azul"** no el valor 4.

Esto es aplicable a cualquier tipo de enumeración, se haya o no usado el atributo *FlagsAttribute*.

Una vez aclarado este comportamiento de las enumeraciones en Visual Basic 2008, veamos que es lo que ocurre cuando sumamos valores de enumeraciones a las que hemos aplicado el atributo *Flags* y a las que no se lo hemos aplicado. Empecemos por este último caso.

Si tenemos este código:

```
Enum Colores As Byte

    Rojo = 1

    Verde = 2

    Azul = 4

End Enum


Dim c As Colores = Colores.Azul Or Colores.Rojo


Dim s As String = c.ToString
```

El contenido de la variable **s** será **"5"**, es decir, la representación numérica del valor contenido: **4 + 1**, ya que el valor de la constante **Azul** es **4** y el de la constante **Rojo** es **1**.

Pero si ese mismo código lo usamos de esta forma (aplicando el atributo *Flags* a la enumeración):

```
<Flags()> _

Enum Colores As Byte

    Rojo = 1

    Verde = 2

    Azul = 4

End Enum


Dim c As Colores = Colores.Azul Or Colores.Rojo


Dim s As String = c.ToString
```

El contenido de la variable **s** será: **"Rojo, Azul"**, es decir, se asignan los nombres de los miembros de la enumeración que intervienen en ese valor, no el valor "interno".

Los valores de una enumeración no son simples números

Como hemos comentado, los miembros de las enumeraciones realmente son valores de un tipo de datos entero (en cualquiera de sus variedades) tal como podemos comprobar en la figura 2.7:

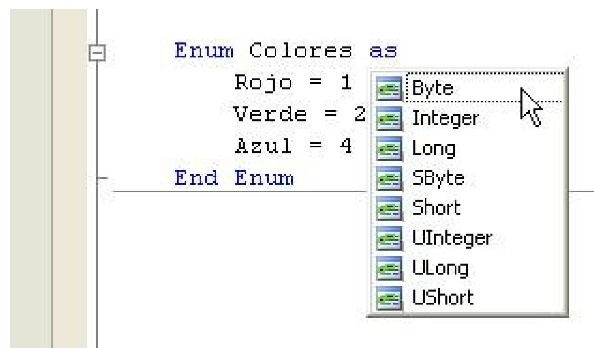


Figura 2.7. Los tipos subyacentes posibles de una enumeración

Por tanto, podemos pensar que podemos usar cualquier valor para asignar a una variable declarada como una enumeración, al menos si ese valor está dentro del rango adecuado.
En Visual Basic 2008 esto no es posible, al menos si lo hacemos de forma "directa" y con **Option Strict** conectado, ya que recibiremos un error indicándonos que no podemos convertir, por ejemplo, un valor entero en un valor del tipo de la enumeración. En la figura 2.8 podemos ver ese error al intentar asignar el valor **3** a una variable del tipo **Colores** (definida con el tipo predeterminado *Integer*).

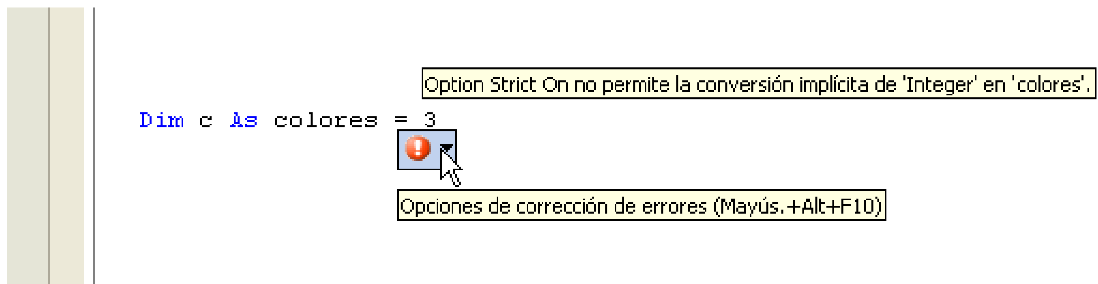


Figura 2.8. Error al asignar un valor "normal" a una variable del tipo Colores

El error nos indica que no podemos realizar esa asignación, pero el entorno integrado de Visual Studio 2008 también nos ofrece alternativas para que ese error no se produzca, esa ayuda se obtiene presionando en el signo de admiración que tenemos justo donde está el cursor del mouse, pero no solo nos dice cómo corregirlo, sino que también nos da la posibilidad de que el propio IDE se encargue de corregirlo, tal como podemos apreciar en la figura 2.9.

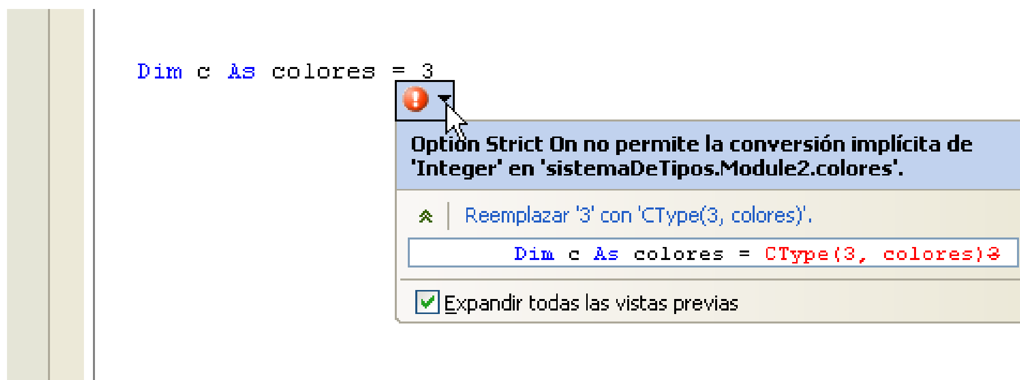


Figura 2.9. Opciones de corrección de errores

Lo único que tendríamos que hacer es presionar en la sugerencia de corrección, que en este caso es la única que hay, pero en otros casos pueden ser varias las opciones y tendríamos que elegir la que creamos adecuada.

El código final (una vez corregido) quedaría de la siguiente forma:

```
Dim c As Colores = CType(3, Colores)
```

CType es una de las formas que nos ofrece Visual Basic 2008 de hacer conversiones entre diferentes tipos de datos, en este caso convertimos un valor entero en uno del tipo **Colores**.

Si compilamos y ejecutamos la aplicación, ésta funcionará correctamente. Aunque sabemos que es posible que usando *CType* no asignemos un valor dentro del rango permitido. En este caso, el valor 3 podríamos darlo por bueno, ya que es la suma de 1 y 2 (**Rojo** y **Verde**), pero ¿que pasaría si el valor asignado es, por ejemplo, 15? En teoría no deberíamos permitirlo.

Estas validaciones podemos hacerlas de dos formas:
1- Con la clásica solución de comprobar el valor indicado con todos los valores posibles.
2- Usando funciones específicas del tipo *Enum*. Aunque en este último caso, solo podremos comprobar los valores definidos en la enumeración.
En el siguiente ejemplo podemos hacer esa comprobación.

```
Sub mostrarColor(ByVal c As Colores)

    ' comprobar si el valor indicado es correcto

    ' si no está¡ definido, usar el valor Azul

    If [Enum].IsDefined(GetType(Colores), c) = False Then

        c = Colores.Azul

    End If

    Console.WriteLine("El color es {0}", c)

End Sub
```

Este código lo que hace es comprobar si el tipo de datos **Colores** tiene definido el valor contenido en la variable **c**, en caso de que no sea así, usamos un valor predeterminado.

> **Nota:**
> La función *IsDefined* sólo comprueba los valores que se han definido en la enumeración, no las posibles combinaciones que podemos conseguir sumando cada uno de sus miembros, incluso aunque hayamos usado el atributo *FlagsAttribute*.

Ver vídeo 1 de esta lección (Enumeraciones 1) - video en Visual Studio 2005 válido para Visual Studio 2008
Ver vídeo 2 de esta lección (Enumeraciones 2) - video en Visual Studio 2005 válido para Visual Studio 2008
Ver vídeo 3 de esta lección (Enumeraciones 3) - video en Visual Studio 2005 válido para Visual Studio 2008
Ver vídeo 4 de esta lección (Enumeraciones 4) - video en Visual Studio 2005 válido para Visual Studio 2008

Lección 1: El sistema de tipos

- Tipos primitivos
- Variables y constantes
- Enumeraciones

Arrays (matrices)

Arrays (matrices)

Los arrays (o matrices) nos permitirán agrupar valores que de alguna forma queremos que estén relacionados entre si.

> **Nota:**
> Esta es la definición usada en la documentación de Visual Studio sobre qué es una matriz:
> *"Una matriz es una estructura de datos que contiene una serie de variables denominadas elementos de la matriz."*
> Aclaramos este punto, porque la traducción en castellano de Array puede variar dependiendo del país, pero aquí utilizaremos la usada a lo largo de la documentación de Visual Studio.

Declarar arrays

En C# los arrays se definen indicando un par de corchetes en el tipo de datos.

En Visual Basic 2008 la declaración de un array la haremos usando un par de paréntesis en el nombre de la variable o del tipo, en el siguiente ejemplo declaramos un array de tipo *String* llamado **nombres**:

```
Dim nombres() As String


Dim nombres As String()
```

Estas dos formas son equivalentes.

También podemos indicar el número de elementos que contendrá el array:

```
Dim nombres(10) As String
```

Pero solo podemos hacerlo en el nombre, si esa cantidad de elementos lo indicamos en el tipo, recibiremos un error indicándonos que *"los límites de la matriz no pueden aparecer en los especificadores del tipo"*.

Al declarar un array indicando el número de elementos, como es el caso anterior, lo que estamos definiendo es un array de 11 elementos: desde cero hasta 10, ya que en Visual Basic 2008, al igual que en el resto de lenguajes de .NET, **todos los arrays deben tener como índice inferior el valor cero**.

Para que quede claro que el límite inferior debe ser cero, en Visual Basic 2008 podemos usar la instrucción ***0 To*** para indicar el valor máximo del índice superior, ya que, tal como podemos comprobar si vemos 0 To 10, quedará claro que nuestra intención es declarar un array con 11 elementos, o al menos nuestro código resultará más legible:

```
Dim nombres(0 To 10) As String
```

Declarar e inicializar un array

En Visual Basic 2008 también podemos inicializar un array al declararlo, para ello debemos poner los valores a asignar dentro de un para de llaves, tal como vemos en el siguiente ejemplo:

```
Dim nombres() As String = {"Pepe", "Juan", "Luisa"}
```

Con el código anterior estamos creando un array de tipo *String* con tres valores cuyos índices van de cero a dos.

En este caso, cuando iniciamos el array al declararlo, no debemos indicar el número de elementos que tendrá ese array, ya que ese valor lo averiguará el compilador cuando haga la asignación. Tampoco es válido indicar el número de elementos que queremos que tenga y solo asignarle unos cuantos menos (o más), ya que se producirá un error en tiempo de compilación.

Si el array es bidimensional (o con más dimensiones), también podemos inicializarlos al declararlo, pero en este caso debemos usar doble juego de llaves:

```
Dim nombres(,) As String = {{"Juan", "Pepe"}, {"Ana", "Eva"}}
```

En este código tendríamos un array bidimensional con los siguientes valores:

nombres(0,0)= Juan
nombres(0,1)= Pepe
nombres(1,0)= Ana
nombres(1,1)= Eva

Como podemos ver en la declaración anterior, si definimos arrays con más de una dimensión, debemos indicarlas usando una coma para separar cada dimensión, o lo que es más fácil de recordar: usando una coma menos del número de dimensiones que tendrá el array. En los valores a asignar, usaremos las llaves encerradas en otras llaves, según el número de dimensiones.

Aunque, la verdad, es que hay algunas veces hay que hacer un gran esfuerzo mental para asociar los elementos con los índices que tendrán en el array, por tanto, algunas veces puede que resulte más legible si indentamos o agrupamos esas asignaciones, tal como vemos en el siguiente código:

```
Dim nomTri(,,) As String = _
        { _
            {{"Juan", "Pepe"}, {"Luisa", "Eva"}}, _
            {{"A", "B"}, {"C", "D"}} _
        }

Console.WriteLine(nomTri(0, 0, 0)) ' Juan
Console.WriteLine(nomTri(0, 0, 1)) ' Pepe
Console.WriteLine(nomTri(0, 1, 0)) ' Luisa
Console.WriteLine(nomTri(0, 1, 1)) ' Eva
Console.WriteLine(nomTri(1, 0, 0)) ' A
Console.WriteLine(nomTri(1, 0, 1)) ' B
Console.WriteLine(nomTri(1, 1, 0)) ' C
Console.WriteLine(nomTri(1, 1, 1)) ' D
```

Tal como podemos comprobar, asi es más legible. Por suerte tenemos el carácter del guión bajo para continuar líneas de código.

Cambiar el tamaño de un array

Para cambiar el tamaño de un array, usaremos la instrucción *ReDim*, esta instrucción solo la podemos usar para cambiar el tamaño de un array previamente declarado, no para declarar un array, ya que siempre hay que declarar previamente los arrays antes de cambiarles el tamaño.

```
Dim nombres() As String
...
ReDim nombres(3)
nombres(0) = "Juan"
nombres(1) = "Pepe"
```

La mayor utilidad de esta instrucción, es que podemos cambiar el tamaño de un array y mantener los valores que tuviera anteriormente, para lograrlo debemos usar *ReDim Preserve*.

```
ReDim Preserve nombres(3)

nombres(2) = "Ana"

nombres(3) = "Eva"
```

En este ejemplo, los valores que ya tuviera el array **nombres**, se seguirían manteniendo, y se asignarían los nuevos.

Si bien tanto *ReDim* como *ReDim Preserve* se pueden usar en arrays de cualquier número de dimensiones, en los arrays de más de una dimensión solamente podemos cambiar el tamaño de la última dimensión.

Eliminar el contenido de un array

Una vez que hemos declarado un array y le hemos asignado valores, es posible que nos interese eliminar esos valores de la memoria, para lograrlo, podemos hacerlo de tres formas:

1. Redimensionando el array indicando que tiene cero elementos, aunque en el mejor de los casos, si no estamos trabajando con arrays de más de una dimensión, tendríamos un array de un elemento, ya que, como hemos comentado anteriormente, los arrays de .NET el índice inferior es cero.
2. Usar la instrucción *Erase*. La instrucción *Erase* elimina totalmente el array de la memoria.
3. Asignar un valor *Nothing* al array. Esto funciona en Visual Basic 2008 porque los arrays realmente son tipos por referencia.

Los arrays son tipos por referencia

Como acabamos de ver, en Visual Basic 2008 los arrays son tipos por referencia, y tal como comentamos anteriormente, los tipos por referencia realmente lo que contienen son una referencia a los datos reales no los datos propiamente dichos.

¿Cual es el problema?

Veámoslo con un ejemplo y así lo tendremos más claro.

```
Dim nombres() As String = {"Juan", "Pepe", "Ana", "Eva"}


Dim otros() As String


otros = nombres


nombres(0) = "Antonio"
```

En este ejemplo definimos el array **nombres** y le asignamos cuatro valores. A continuación definimos otro array llamado **otros** y le asignamos lo que tiene **nombres**. Por último asignamos un nuevo valor al elemento cero del array **nombres**.
Si mostramos el contenido de ambos arrays nos daremos cuenta de que realmente solo existe una copia de los datos en la memoria, y tanto **nombres(0)** como **otros(0)** contienen el nombre **"Antonio"**.

¿Qué ha ocurrido?

Que debido a que los arrays son tipos por referencia, solamente existe una copia de los datos y tanto la variable nombres como la variable otros lo que contienen es una referencia (o puntero) a los datos.

Si realmente queremos tener copias independientes, debemos hacer una copia del array nombres en el array otros, esto es fácil de hacer si usamos el método *CopyTo*. Éste método existe en todos los arrays y nos permite copiar un array en otro empezando por el índice que indiquemos.
El único requisito es que el array de destino debe estar inicializado y tener espacio suficiente para contener los elementos que queremos copiar.

En el siguiente código de ejemplo hacemos una copia del contenido del array **nombres** en el array **otros**, de esta forma, el cambio realizado en el elemento cero de **nombres** no afecta al del array **otros**.

```
Dim nombres() As String = {"Juan", "Pepe", "Ana", "Eva"}


Dim otros() As String


ReDim otros(nombres.Length)


nombres.CopyTo(otros, 0)


nombres(0) = "Antonio"
```

Además del método *CopyTo*, los arrays tienen otros miembros que nos pueden ser de utilidad, como por ejemplo la propiedad *Length* usada en el ejemplo para saber cuantos elementos tiene el array **nombres**.

Para averiguar el número de elementos de un array, (realmente el índice superior), podemos usar la función *UBound*.
También podemos usar la función *LBound*, (que sirve para averiguar el índice inferior de un array), aunque no tiene ningún sentido en Visual Basic 2008, ya que, como hemos comentado, todos los arrays siempre tienen un valor cero como índice inferior.

Para finalizar este tema, solo nos queda por decir, que los arrays de Visual Basic 2008 realmente son tipos de datos derivados de la clase *Array* y por tanto disponen de todos los miembros definidos en esa clase, aunque de esto hablaremos en la próxima lección, en la que también tendremos la oportunidad de profundizar un poco más en los tipos por referencia y en como podemos definir nuestros propios tipos de datos, tanto por referencia como por valor.

Ver vídeo 1 de esta lección (Arrays Parte 1) - video en Visual Studio 2005 válido para Visual Studio 2008
Ver vídeo 2 de esta lección (Arrays Parte 2) - video en Visual Studio 2005 válido para Visual Studio 2008
Ver vídeo 3 de esta lección (Arrays Parte 3) - video en Visual Studio 2005 válido para Visual Studio 2008

Lección 2: Clases y estructuras

- Clases
- Definir una clase
- Instanciar una clase
- Estructuras
- Accesibilidad
- Propiedades
- Interfaces

Introducción

En la lección anterior vimos los tipos de datos predefinidos en .NET Framework, en esta lección veremos cómo podemos crear nuestros propios tipos de datos, tanto por valor como por referencia.
También tendremos ocasión de ver los distintos niveles de accesibilidad que podemos aplicar a los tipos, así como a los distintos miembros de esos tipos de datos. De los distintos miembros que podemos definir en nuestros tipos, nos centraremos en las propiedades para ver en detalle los cambios que han sufrido con respecto a VB6. También veremos temas relacionados con la programación orientada a objetos (POO) en general y de forma particular los que atañen a las interfaces.

Clases y estructuras

- **Clases: Tipos por referencia definidos por el usuario**
  - Las clases: El corazón de .NET Framework
  - La herencia: Característica principal de la Programación Orientada a Objetos
  - Encapsulación y Polimorfismo
  - Object: La clase base de todas las clases de .NET

- **Definir una clase**
  - Una clase especial: Module
  - Los miembros de una clase
  - Características de los métodos y propiedades
    - Accesibilidad, ámbito y miembros compartidos
    - Parámetros y parámetros opcionales
    - Array de parámetros opcionales (ParamArray)
    - Sobrecarga de métodos y propiedades
    - Parámetros por valor y parámetros por referencia

- **Instanciar: Crear un objeto en la memoria**
  - Declarar primero la variable y después instanciarla
  - Declarar y asignar en un solo paso

- El constructor: El punto de inicio de una clase
- Constructores parametrizados
- Cuando Visual Basic 2008 no crea un constructor automáticamente
- El destructor: El punto final de la vida de una clase
- Inicialización directa de objetos

- **Estructuras: Tipos por valor definidos por el usuario**
  - Definir una estructura
  - Constructores de las estructuras
  - Destructores de las estructuras
  - Los miembros de una estructura
  - Cómo usar las estructuras

- **Accesibilidad y ámbito**
  - Ámbito
    - Ámbito de bloque
    - Ámbito de procedimiento
    - Ámbito de módulo
    - Ámbito de espacio de nombres
  - La palabra clave Global
  - Accesibilidad
    - Accesibilidad de las variables en los procedimientos
  - Las accesibilidades predeterminadas
  - Anidación de tipos
    - Los tipos anidables
    - El nombre completo de un tipo
    - Importación de espacios de nombres
    - Alias de espacios de nombres

- **Propiedades**
  - Definir una propiedad
  - Propiedades de solo lectura
  - Propiedades de solo escritura
  - Diferente accesibilidad para los bloques Get y Set
  - Propiedades predeterminadas
    - Sobrecarga de propiedades predeterminadas

- **Interfaces**
  - ¿Qué es una interfaz?
  - ¿Qué contiene una interfaz?
  - Una interfaz es un contrato
  - Las interfaces y el polimorfismo
  - Usar una interfaz en una clase
  - Acceder a los miembros implementados
  - Saber si un objeto implementa una interfaz
  - Implementación de múltiples interfaces
  - Múltiple implementación de un mismo miembro
  - ¿Dónde podemos implementar las interfaces?
  - Un ejemplo práctico usando una interfaz de .NET

Lección 2: Clases y estructuras

Clases

- Definir una clase
- Instanciar una clase
- Estructuras
- Accesibilidad
- Propiedades
- Interfaces

Clases: Tipos por referencia definidos por el usuario

Tal como vimos en la lección anterior, los tipos de datos se dividen en dos grupos: tipos por valor y tipos por referencia. Los tipos por referencia realmente son clases, de la cuales debemos crear una instancia para poder usarlas, esa instancia o copia, se crea siempre en la memoria lejana (*heap*) y las variables lo único que contienen es una referencia a la dirección de memoria en la que el CLR (*Common Language Runtime*, motor en tiempo de ejecución de .NET), ha almacenado el objeto recién creado.

En .NET Framework todo es de una forma u otra una clase, por tanto Visual Basic 2008 también depende de la creación de clases para su funcionamiento, ya que todo el código que escribamos debemos hacerlo dentro de una clase.

Antes de entrar en detalles sintácticos, veamos la importancia que tienen las clases en .NET Framework y como repercuten en las que podamos definir nosotros usando Visual Basic 2008.

Las clases: el corazón de .NET Framework

Prácticamente todo lo que podemos hacer en .NET Framework lo hacemos mediante clases. La librería de clases de .NET Framework es precisamente el corazón del propio .NET, en esa librería de clases está todo lo que podemos hacer dentro de este marco de programación; para prácticamente cualquier tarea que queramos realizar existen clases, y si no existen, las podemos definir nosotros mismos, bien ampliando la funcionalidad de alguna clase existente mediante la herencia, bien implementando algún tipo de funcionalidad previamente definida o simplemente creándolas desde cero.

La herencia: Característica principal de la Programación Orientada a Objetos

El concepto de Programación Orientada a Objetos (POO) es algo intrínsico al propio .NET Framework, por tanto es una característica que todos los lenguajes

basados en este "marco de trabajo" tienen de forma predeterminada, entre ellos el Visual Basic 2008. De las características principales de la POO tenemos que destacar la herencia, que en breve podemos definir como una característica que nos permite ampliar la funcionalidad de una clase existente sin perder la que ya tuviera previamente. Gracias a la herencia, podemos crear una nueva clase que se derive de otra, esta nueva clase puede cambiar el comportamiento de la clase base y/o ampliarlo, de esta forma podemos adaptar la clase, llamémosla, original para adaptarla a nuestras necesidades.

El tipo de herencia que .NET Framework soporta es la herencia simple, es decir, solo podemos usar una clase como base de la nueva, si bien, como veremos más adelante, podemos agregar múltiple funcionalidad a nuestra nueva clase. Esta funcionalidad nos servirá para aprovechar la funcionalidad de muchas de las clases existentes en .NET Framework, funcionalidad que solamente podremos aplicar si previamente hemos firmado un contrato que asegure a la clase de .NET que la nuestra está preparada para soportar esa funcionalidad, esto lo veremos dentro de poco con más detalle.

Encapsulación y Polimorfismo

La encapsulación y el polimorfismo son otras dos características de la programación orientada a objetos.

La encapsulación nos permite abstraer la forma que tiene de actuar una clase sobre los datos que contiene o manipula, para poder lograrlo se exponen como parte de la clase los métodos y propiedades necesarios para que podamos manejar esos datos sin tener que preocuparnos cómo se realiza dicha manipulación.

El polimorfismo es una característica que nos permite realizar ciertas acciones o acceder a la información de los datos contenidos en una clase de forma semi-anónima, al menos en el sentido de que no tenemos porqué saber sobre que tipo objeto realizamos la acción, ya que lo único que nos debe preocupar es que podemos hacerlo, por la sencilla razón de que estamos usando ciertos mecanismos que siguen unas normas que están adoptadas por la clase.
El ejemplo clásico del polimorfismo es que si tengo un objeto que "sabe" cómo morder, da igual que lo aplique a un ratón o a un dinosaurio, siempre y cuando esas dos "clases" expongan un método que pueda realizar esa acción... y como decía la documentación de Visual Basic 5.0, siempre será preferible que nos muerda un ratón antes que un dinosaurio.

Object: La clase base de todas las clases de .NET

Todas las clases de .NET se derivan de la clase *Object*, es decir, lo indiquemos o no, cualquier clase que definamos tendrá el comportamiento heredado de esa clase. El uso de la clase *Object* como base del resto de las clases de .NET es la única excepción a la herencia simple soportada por .NET, ya que de forma implícita, todas las clases de .NET se derivan de la clase *Object* independientemente de que estén derivadas de cualquier otra.
Esta característica nos asegura que siempre podremos usar un objeto del tipo *Object* para acceder a cualquier clase de .NET, aunque no debemos abrumarnos todavía, ya que en el texto que sigue veremos con más detalle que significado tiene esta afirmación.

De los miembros que tiene la clase *Object* debemos resaltar el método *ToString*, el cual ya lo vimos en la lección anterior cuando queríamos convertir un tipo

primitivo en una cadena. Este método está pensado para devolver una representación en formato cadena de un objeto. El valor que obtengamos al usar este método dependerá de cómo esté definido en cada clase y por defecto lo que devuelve es el nombre completo de la clase, si bien en la mayoría de los casos el valor que obtendremos al usar este método debería ser más intuitivo, por ejemplo los tipos de datos primitivos tienen definido este método para devuelva el valor que contienen, de igual forma, nuestras clases también deberían devolver un valor adecuado al contenido almacenado. De cómo hacerlo, nos ocuparemos en breve.

Nota:
Todos los tipos de datos de .NET, ya sean por valor o por referencia siempre están derivados de la clase Object, por tanto podremos llamar a cualquiera de los métodos que están definidos en esa clase. Aunque en el caso de los tipos de datos por valor, cuando queremos acceder a la clase Object que contienen, .NET Framework primero debe convertirla en un objeto por referencia (boxing) y cuando hemos dejado de usarla y queremos volver a asignar el dato a la variable por valor, tiene que volver a hacer la conversión inversa (unboxing).

Ver vídeo de esta lección (Clases 1) - video en Visual Studio 2005 válido para Visual Studio 2008

Lección 2: Clases y estructuras

- Clases

Definir una clase

- Instanciar una clase
- Estructuras
- Accesibilidad
- Propiedades
- Interfaces

Definir una clase

En Visual Basic 2008, todo el código que queramos escribir, lo tendremos que hacer en un fichero con la extensión **.vb**, dentro de ese fichero es donde escribiremos nuestro código, el cual, tal como dijimos anteriormente siempre estará incluido dentro de una clase, aunque un fichero de código de VB puede contener una o más clases, es decir, no está limitado a una clase por fichero.

En Visual Basic 2008 las clases se definen usando la palabra clave *Class* seguida del nombre de la clase, esa definición acaba indicándolo con *End Class*.

En el siguiente ejemplo definimos una clase llamada Cliente que tiene dos campos públicos.

```
Class Cliente

    Public Nombre As String

    Public Apellidos As String

End Class
```

Una vez definida la clase podemos agregar los elementos (o miembros) que creamos conveniente.
En el ejemplo anterior, para simplificar, hemos agregado dos campos públicos, aunque también podríamos haber definido cualquiera de los miembros permitidos en las clases.

Una clase especial: Module

En Visual Basic 2008 también podemos definir una clase especial llamada *Module*, este tipo de clase, como veremos, tiene un tratamiento especial.

La definición se hace usando la instrucción *Module* seguida del nombre a usar y acaba con *End Module*.
Cualquier miembro definido en un *Module* siempre estará accesible en todo el proyecto y para usarlos no tendremos que crear ningún objeto en memoria.
Las clases definidas con la palabra clave *Module* realmente equivalen a las clases en las que todos los miembros están compartidos y por tanto siempre disponibles a toda la aplicación.

De todos estos conceptos nos ocuparemos en las siguientes lecciones, pero es necesario explicar que existe este tipo de clase ya que será el tipo de datos que el IDE de Visual Basic 2008 usará al crear aplicaciones del tipo consola, ya que ese será el tipo de proyecto que crearemos para practicar con el código mostrado en este primer módulo.

Los miembros de una clase

Una clase puede contener cualquiera de estos elementos (miembros):

- Enumeraciones
- Campos
- Métodos (funciones o procedimientos)
- Propiedades
- Eventos

Las enumeraciones, como vimos en la lección anterior, podemos usarlas para definir valores constantes relacionados, por ejemplo para indicar los valores posibles de cualquier "característica" de la clase.

Los campos son variables usadas para mantener los datos que la clase manipulará.

Los métodos son las acciones que la clase puede realizar, normalmente esas acciones serán sobre los datos que contiene. Dependiendo de que el método devuelva o no un valor, podemos usar métodos de tipo *Function* o de tipo *Sub* respectivamente.

Las propiedades son las "características" de las clases y la forma de acceder "públicamente" a los datos que contiene. Por ejemplo, podemos considerar que el nombre y los apellidos de un cliente son dos características del cliente.

Los eventos son mensajes que la clase puede enviar para informar que algo está ocurriendo en la clase.

Características de los métodos y propiedades

Accesibilidad, ámbito y miembros compartidos

Aunque estos temas los veremos en breve con más detalle, para poder comprender mejor las características de los miembros de una clase (o cualquier tipo que definamos), daremos un pequeño adelanto sobre estas características que podemos aplicar a los elementos que definamos.

Accesibilidad y ámbito son dos conceptos que están estrechamente relacionados. Aunque en la práctica tienen el mismo significado, ya que lo que representan es la

"cobertura" o alcance que tienen los miembros de las clases e incluso de las mismas clases que definamos.

Si bien cada uno de ellos tienen su propia "semántica", tal como podemos ver a continuación:

Ámbito

Es el alcance que la definición de un miembro o tipo puede tener. Es decir, cómo podemos acceder a ese elemento y desde dónde podemos accederlo.
El ámbito de un elemento de código está restringido por el "sitio" en el que lo hemos declarado. Estos *sitios* pueden ser:

- **Ámbito de bloque:** Disponible únicamente en el bloque de código en el que se ha declarado.
- **Ámbito de procedimiento:** Disponible únicamente dentro del procedimiento en el que se ha declarado.
- **Ámbito de módulo:** Disponible en todo el código del módulo, la clase o la estructura donde se ha declarado.
- **Ámbito de espacio de nombres:** Disponible en todo el código del espacio de nombres.

Accesibilidad

A los distintos elementos de nuestro código (ya sean clases o miembros de las clases) podemos darle diferentes tipos de accesibilidad. Estos tipos de "acceso" dependerán del ámbito que queramos que tengan, es decir, desde dónde podremos accederlos.
Los modificadores de accesibilidad son:

- **Public:** Acceso no restringido.
- **Protected:** Acceso limitado a la clase contenedora o a los tipos derivados de esta clase.
- **Friend:** Acceso limitado al proyecto actual.
- **Protected Friend:** Acceso limitado al proyecto actual o a los tipos derivados de la clase contenedora.
- **Private:** Acceso limitado al tipo contenedor.

Por ejemplo, podemos declarar miembros privados a una clase, en ese caso, dichos miembros solamente los podremos acceder desde la propia clase, pero no desde fuera de ella.

> **Nota:**
> Al declarar una variable con **Dim**, el ámbito que le estamos aplicando por regla general, es privado, pero como veremos, en Visual Basic 2008 el ámbito puede ser diferente a privado dependiendo del tipo en el que declaremos esa variable.

Miembros compartidos

Por otro lado, los miembros compartidos de una clase o tipo, son elementos que no pertenecen a una instancia o copia en memoria particular, sino que pertenecen al propio tipo y por tanto siempre están accesibles o disponibles, dentro del nivel

del ámbito y accesibilidad que les hayamos aplicado, y su *tiempo de vida* es el mismo que el de la aplicación.

Del ámbito, la accesibilidad y los miembros compartidos nos ocuparemos con más detalle en una lección posterior, donde veremos ciertas peculiaridades, como puede ser la limitación del ámbito de un miembro que *aparentemente* tiene una accesibilidad no restringida.

Parámetros específicos y parámetros opcionales

En Visual Basic 2008, tanto los miembros de una clase, (funciones y métodos Sub), como las propiedades pueden recibir parámetros. Esos parámetros pueden estar explícitamente declarados, de forma que podamos indicar cuantos argumentos tendremos que pasar al método, también podemos declarar parámetros opcionales, que son parámetros que no hace falta indicar al llamar a la función o propiedad, y que siempre tendrán un valor predeterminado, el cual se usará en caso de que no lo indiquemos.
Además de los parámetros específicos y opcionales, podemos usar un array de parámetros opcionales, en los que se puede indicar un número variable de argumentos al llamar a la función que los define, esto lo veremos en la siguiente sección.

> **Nota: Sobre parámetros y argumentos**
> Para clarificar las cosas, queremos decir que los parámetros son los definidos en la función, mientras que los argumentos son los "parámetros" que pasamos a esa función cuando la utilizamos desde nuestro código.

Con los parámetros opcionales lo que conseguimos es permitir que el usuario no tenga que introducir todos los parámetros, sino solo los que realmente necesite, del resto, (los no especificados), se usará el valor predeterminado que hayamos indicado, porque una de las "restricciones" de este tipo de parámetros, es que siempre debemos indicar también el valor por defecto que debemos usar en caso de que no se especifique.

Veamos unos ejemplo para aclarar nuestras ideas:

```
Function Suma(n1 As Integer, Optional n2 As Integer = 15) As Integer

    Suma = n1 + n2

End Function
```

En este primer ejemplo, el primer parámetro es obligatorio (siempre debemos indicarlo) y el segundo es opcional, si no se indica al llamar a esta función, se usará el valor 15 que es el predeterminado.

Para llamar a esta función, lo podemos hacer de estas tres formas:

```
' 1- indicando los dos parámetros (el resultado será 4= 1 + 3)

t = Suma(1, 3)
```

```
' 2- Indicando solamente el primer parámetro (el resultado será 16= 1 + 15)

t = Suma(1)

' 3- Indicando los dos parámetros, pero en el opcional usamos el nombre

t = Suma(1, n2:= 9)
```

El tercer ejemplo solamente tiene utilidad si hay más de un parámetro opcional.

> **Nota:**
> Los parámetros opcionales deben aparecer en la lista de parámetros del método, después de los "obligatorios"
>
> **Nota:**
> En el caso hipotético y no recomendable de que no estemos usando Option Strict On, tanto los parámetros normales como los opcionales los podemos indicar sin el tipo de datos, pero si en alguno de ellos especificamos el tipo, debemos hacerlo en todos.

Array de parámetros opcionales (ParamArray)

En cuanto al uso de *ParamArray*, este tipo de parámetro opcional nos permite indicar un número indeterminado de parámetros, en Visual Basic 2008 **siempre** debe ser una array e internamente se trata como un array normal y corriente, es decir, dentro del método o propiedad se accede a él como si de un array se tratara... entre otras cosas, ¡porque es un array!

Si tenemos activado *Option Strict*, el array usado con *ParamArray* debe ser de un tipo en concreto, si queremos que acepte cualquier tipo de datos, lo podemos declarar con el tipo *Object*.

Si por cualquier razón necesitamos pasar otros valores que no sean de un tipo en concreto, por ejemplo, además de valores "normales" queremos pasar un array, podemos desactivar *Option Strict* para hacerlo fácil.
Por ejemplo, si queremos hacer algo como esto:

```
Function Suma(ParamArray n() As Object) As Integer

    Dim total As Integer

    '

    For i As Integer = 0 To n.Length - 1

        If IsArray(n(i)) Then

            For j As Integer = 0 To n(i).Length - 1

                total += n(i)(j)

            Next

        Else
```

```
            total += n(i)

        End If

    Next

    Return total

End Function



' Para usarlo:

Dim t As Integer

Dim a(2) As Integer = {1, 2, 3}

t = Suma(a, 4, 5, 6)



Console.WriteLine(t)
```

Tal como vemos, el primer argumento que pasamos a la función Suma es un array, después pasamos tres valores enteros normales. El resultado de ejecutar ese código será el valor **21**, como es de esperar.

> **Nota:**
> Como Visual Basic 2008 no sabe que tipo contiene n(i), al escribir este código, no nos informará de que ese "objeto" tiene una propiedad llamada Length.

Pero como **no** debemos "acostumbrarnos" a desactivar *Option Strict*, vamos a ver el código que tendríamos que usar para que también acepte un array como parte de los argumentos indicados al llamar a la función.

```
Function Suma(ByVal ParamArray n() As Object) As Integer

    Dim total As Integer

    '

    For i As Integer = 0 To n.Length - 1

        If IsArray(n(i)) Then

            For j As Integer = 0 To CType(n(i), Integer()).Length
- 1

                total += CType(n(i), Integer())(j)

            Next

        Else

            total += CInt(n(i))

        End If
```

```
    Next

    Return total

End Function



' Para usarlo:

Dim t As Integer

Dim a(2) As Integer = {1, 2, 3}

t = Suma(a, 4, 5, 6)



Console.WriteLine(t)
```

Como podemos comprobar, al tener *Option Strict* activado, debemos hacer una conversión explícita del array a uno de un tipo en concreto, como es natural, en este código estamos suponiendo que el array pasado como argumento a la función es de tipo *Integer*, en caso de que no lo fuera, recibiríamos un error en tiempo de ejecución.

Para comprobar si el contenido de n(i) es un array, hemos usado la función *IsArray* definida en el espacio de nombres *Microsoft.VisualBasic* (que por defecto utilizan todas las aplicaciones de Visual Basic 2008), pero también lo podríamos haber hecho más al "estilo .NET", ya que, como sabemos todos los arrays de .NET realmente se derivan de la clase *Array*:

```
If TypeOf n(i) Is Array Then
```

Nota:
Cuando queramos usar ParamArray para recibir un array de parámetros opcionales, esta instrucción debe ser la última de la lista de parámetros de la función (método).

Tampoco se permite tener parámetros opcionales y ParamArray en la misma función.

Sobrecarga de métodos y propiedades

La sobrecarga de funciones (realmente de métodos y propiedades), es una característica que nos permite tener una misma función con diferentes tipos de parámetros, ya sea en número o en tipo.

Supongamos que queremos tener dos funciones (o más) que nos permitan hacer operaciones con diferentes tipos de datos, y que, según el tipo de datos usado, el valor que devuelva sea de ese mismo tipo.

En este ejemplo, tenemos dos funciones que se llaman igual pero una recibe valores de tipo entero y la otra de tipo decimal:

```
Function Suma(n1 As Integer, n2 As Integer) As Integer

    Return n1 + n2

End Function



Function Suma(n1 As Double, n2 As Double) As Double

    Return n1 + n2

End Function
```

Como podemos comprobar las dos funciones tienen el mismo nombre, pero tanto una como otra reciben parámetros de tipos diferentes.

Con Visual Basic 2008 podemos sobrecargar funciones, pero lo interesante no es que podamos hacerlo, sino cómo podemos usar esas funciones. En el código anterior tenemos dos funciones llamadas **Suma**, la primera acepta dos parámetros de tipo *Integer* y la segunda de tipo *Double*. Lo interesante es que cuando queramos usarlas, no tenemos que preocuparnos de cual vamos a usar, ya que será el compilador el que decida la más adecuada al código que usemos, por ejemplo:

```
' En este caso, se usará la que recibe dos valores enteros

Console.WriteLine(Suma(10, 22) )



' En este caso se usará la que recibe valores de tipo Double

Console.WriteLine(Suma(10.5, 22)
```

El compilador de Visual Basic 2008 es el que decide que función usar, esa decisión la toma a partir de los tipos de parámetros que hayamos indicado. En el segundo ejemplo de uso, el que mejor coincide es el de los dos parámetros de tipo *Double*.

También podemos tener sobrecarga usando una cantidad diferente de parámetros, aunque éstos sean del mismo tipo. Por ejemplo, podemos añadir esta declaración al código anterior sin que exista ningún tipo de error, ya que esta nueva función recibe tres parámetros en lugar de dos:

```
Function Suma(n1 As Integer, n2 As Integer, n3 As Integer) As
Integer

    Return n1 + n2 + n3

End Function
```

Por tanto, cuando el compilador se encuentre con una llamada a la función **Suma** en la que se usen tres parámetros, intentará usar esta última.

> **Nota:**
> Para que exista sobrecarga, la diferencia debe estar en el número o en el tipo de los parámetros, no en el tipo del valor devuelto.

Cuando, desde el IDE de Visual Basic 2008, queremos usar los métodos sobrecargados, nos mostrará una lista de opciones indicándonos las posibilidades de sobrecarga que existen y entre las que podemos elegir, tal como vemos en la figura 2.10:

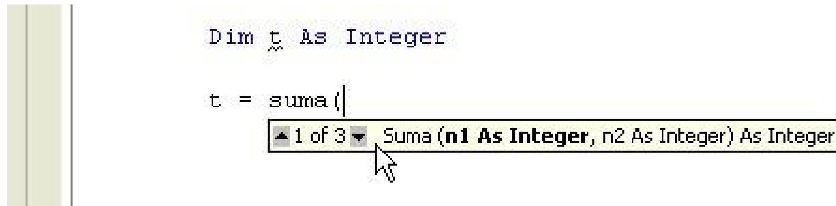


Figura 2.10. Lista de parámetros soportados por un método

Cuando utilicemos parámetros opcionales debemos tener en cuenta que puede que el compilador nos muestre un error, ya que es posible que esa función que declara parámetros opcionales entre en conflicto con una "sobrecargada".

Por ejemplo:

```
Function Suma(n1 As Integer, Optional n2 As Integer = 33) As
Integer

    Return n1 + n2

End Function
```

Si tenemos esta declaración además de las anteriores, el programa no compilará, ya que si hacemos una llamada a la función **Suma** con dos parámetros enteros, el compilador no sabrá si usar esta última o la primera que declaramos, por tanto producirá un error.

Parámetros por valor y parámetros por referencia

Al igual que tenemos dos tipos de datos diferentes, en los parámetros de las funciones también podemos tenerlos, para ello tendremos que usar *ByVal* o *ByRef* para indicar al compilador cómo debe tratar a los parámetros.

Cuando un parámetro es por valor (*ByVal*), el *runtime* antes de llamar a la función hace una copia de ese parámetro y pasa la copia a la función, por tanto cualquier cambio que hagamos a ese parámetro dentro de la función no afectará al valor usado "externamente".

En el caso de que el parámetro sea por referencia (*ByRef*), el compilador pasa una referencia que apunta a la dirección de memoria en la que están los datos, por tanto si realizamos cambios dentro de la función, ese cambio si que se verá reflejado en el parámetro usado al llamar a la función.

Nota:
Hay que tener en cuenta que si pasamos un objeto a una función, da igual que lo declaremos por valor o por referencia, ya que en ambos casos se pasa una referencia a la dirección de memoria en la que están los datos, porque, como sabemos, las variables de los tipos por referencia siempre contienen una referencia a los datos, no los datos en sí.

Cuando en Visual Basic 2008 usamos parámetros en los que no se indica si es por valor (*ByVal*) o por referencia (*ByRef*), serán tratados como parámetros por valor.

Nota:
Si usamos el IDE de Visual Studio 2008 para escribir el código, no debemos preocuparnos de este detalle, ya que si no indicamos si es por valor o por referencia, automáticamente le añadirá la palabra clave ByVal, para que no haya ningún tipo de dudas.

Ver vídeo 1 de esta lección (Clases 2: Los miembros de las clases) - video en Visual Studio 2005 válido para Visual Studio 2008
Ver vídeo 2 de esta lección (Clases 3: Las propiedades) - video en Visual Studio 2005 válido para Visual Studio 2008
Ver vídeo 3 de esta lección (Clases 4: Accesibilidad) - video en Visual Studio 2005 válido para Visual Studio 2008

Lección 2: Clases y estructuras

- Clases
- Definir una clase

Instanciar una clase

- Estructuras
- Accesibilidad
- Propiedades
- Interfaces

Instanciar: Crear un objeto en la memoria

Una vez que tenemos una clase definida, lo único de lo que disponemos es de una especie de plantilla o molde a partir del cual podemos crear objetos en memoria.

Para crear esos objetos en Visual Basic 2008 lo podemos hacer de dos formas, pero como veremos siempre será mediante la instrucción *New* que es la encargada de crear el objeto en la memoria y asignar la dirección del mismo a la variable usada en la parte izquierda de la asignación.

Declarar primero la variable y después instanciarla

Lo primero que tenemos que hacer es declarar una variable del tipo que queremos instanciar, esto lo hacemos de la misma forma que con cualquier otro tipo de datos:

```
Dim c As Cliente
```

Con esta línea de código lo que estamos indicando a Visual Basic es que tenemos intención de usar una variable llamada **c** para acceder a una clase de tipo **Cliente**. Esa variable, cuando llegue el momento de usarla, sabrá todo lo que hay que saber sobre una clase **Cliente**, pero hasta que no tenga una "referencia" a un objeto de ese tipo no podremos usarla.

La asignación de una referencia a un objeto **Cliente** la haremos usando la instrucción *New* seguida del nombre de la clase:

```
c = New Cliente
```

A partir de este momento, la variable **c** tiene acceso a un nuevo objeto del tipo **Cliente**, por tanto podremos usarla para asignarle valores y usar cualquiera de los miembros que ese tipo de datos contenga:

```
c.Nombre = "Antonio"

c.Apellidos = "Ruiz Rodríguez"
```

Declarar y asignar en un solo paso

Con las clases o tipos por referencia también podemos declarar una variable y al mismo tiempo asignarle un nuevo objeto:

```
Dim c As New Cliente
```

O también:

```
Dim c As Cliente = New Cliente
```

Las dos formas producen el mismo resultado, por tanto es recomendable usar la primera.

El constructor: El punto de inicio de una clase

Cada vez que creamos un nuevo objeto en memoria estamos llamando al constructor de la clase. En Visual Basic 2008 el constructor es un método de tipo *Sub* llamado *New*.

En el constructor de una clase podemos incluir el código que creamos conveniente, pero realmente solamente deberíamos incluir el que realice algún tipo de inicialización, en caso de que no necesitemos realizar ningún tipo de inicialización, no es necesario definir el constructor, ya que el propio compilador lo hará por nosotros. Esto es así porque todas las clases deben implementar un constructor, por tanto si nosotros no lo definimos, lo hará el compilador de Visual Basic 2008.

Si nuestra clase **Cliente** tiene un campo para almacenar la fecha de creación del objeto podemos hacer algo como esto:

```
Class Cliente

    Public Nombre As String

    Public Apellidos As String

    Public FechaCreacion As Date

    '

    Public Sub New()

        FechaCreacion = Date.Now

    End Sub

End Class
```

De esta forma podemos crear un nuevo **Cliente** y acto seguido comprobar el valor del campo **FechaCreacion** para saber la fecha de creación del objeto.

En los constructores también podemos hacer las inicializaciones que, por ejemplo permitan a la clase a conectarse con una base de datos, abrir un fichero o cargar una imagen gráfica, etc.

Constructores parametrizados

De la misma forma que podemos tener métodos y propiedades sobrecargadas, también podemos tener constructores sobrecargados, ya que debemos recordar que en Visual Basic 2008, un constructor realmente es un método de tipo *Sub*, y como todos los métodos y propiedades de Visual Basic 2008, también admite la sobrecarga.

La ventaja de tener constructores que admitan parámetros es que podemos crear nuevos objetos indicando algún parámetro, por ejemplo un fichero a abrir o, en el caso de la clase **Cliente**, podemos indicar el nombre y apellidos del cliente o cualquier otro dato que creamos conveniente.

Para comprobarlo, podemos ampliar la clase definida anteriormente para que también acepte la creación de nuevos objetos indicando el nombre y los apellidos del cliente.

```
Class Cliente

    Public Nombre As String

    Public Apellidos As String

    Public FechaCreacion As Date

    '

    Public Sub New()

        FechaCreacion = Date.Now

    End Sub

    '

    Public Sub New(elNombre As String, losApellidos As String)

        Nombre = elNombre

        Apellidos = losApellidos

        FechaCreacion = Date.Now

    End Sub

End Class
```

Teniendo esta declaración de la clase **Cliente**, podemos crear nuevos clientes de dos formas:

```
Dim c1 As New Cliente
```

```
Dim c2 As New Cliente("Jose", "Sánchez")
```

Como podemos comprobar, en ciertos casos es más intuitiva la segunda forma de crear objetos del tipo **Cliente**, además de que así nos ahorramos de tener que asignar individualmente los campos **Nombre** y **Apellidos**.

Esta declaración de la clase **Cliente** la podríamos haber hecho de una forma diferente. Por un lado tenemos un constructor "normal" (no recibe parámetros) en el que asignamos la fecha de creación y por otro el constructor que recibe los datos del nombre y apellidos. En ese segundo constructor también asignamos la fecha de creación, ya que, se instancie como se instancie la clase, nos interesa saber siempre la fecha de creación. En este ejemplo, por su simpleza no es realmente un problema repetir la asignación de la fecha, pero si en lugar de una inicialización necesitáramos hacer varias, la verdad es que nos encontraríamos con mucha "duplicidad" de código. Por tanto, en lugar de asignar los datos en dos lugares diferentes, podemos hacer esto otro:

```
Class Cliente

    Public Nombre As String

    Public Apellidos As String

    Public FechaCreacion As Date

    '

    Public Sub New()

        FechaCreacion = Date.Now

    End Sub

    '

    Public Sub New(elNombre As String, losApellidos As String)

        Nombre = elNombre

        Apellidos = losApellidos

    End Sub

End Class
```

Es decir, desde el constructor con argumentos llamamos al constructor que no los tiene, consiguiendo que también se asigne la fecha.

Esta declaración de la clase **Cliente** realmente no compilará. Y no compila por la razón tan "simple" de que aquí el compilador de Visual Basic 2008 no sabe cual es nuestra intención, ya que *New* es una palabra reservada que sirve para crear nuevos objetos y, siempre, una instrucción tiene preferencia sobre el nombre de un método, por tanto, podemos recurrir al objeto especial *Me* el cual nos sirve, para representar al objeto que actualmente está en la memoria, así que, para que esa declaración de la clase Cliente funcione, debemos usar *Me.New()* para llamar al constructor sin parámetros.

Nota:
El IDE de Visual Basic 2008 colorea las instrucciones y tipos propios del lenguaje, en el caso de Me, lo que no debe confundirnos es que cuando declaramos Sub New, tanto Sub como New se muestran coloreadas, cuando solo se debería colorear Sub; sin embargo cuando usamos Me.New, solo se colorea Me y no New que es correcto, tal como vemos en la figura 2.11, ya que en este caso New es el nombre de un procedimiento y los procedimientos no son parte de las instrucciones y tipos de .NET.

```
    Public Sub New()
        FechaCreacion = Date.Now
    End Sub
    Public Sub New(ByVal elNombre As Strir
        Me.New()
        Nombre = elNombre
        Apellidos = losApellidos
    End Sub
End Class
```

Figura 2.11. Coloreo erróneo de New

Cuando Visual Basic 2008 no crea un constructor automáticamente

Tal como hemos comentado, si nosotros no definimos un constructor (*Sub New*), lo hará el propio compilador de Visual Basic 2008, y cuando lo hace automáticamente, siempre es un constructor sin parámetros.

Pero hay ocasiones en las que nos puede interesar que no exista un constructor sin parámetros, por ejemplo, podemos crear una clase **Cliente** que solo se pueda instanciar si le pasamos, por ejemplo el número de identificación fiscal, (NIF), en caso de que no se indique ese dato, no podremos crear un nuevo objeto **Cliente**, de esta forma, nos aseguramos siempre de que el NIF siempre esté especificado. Seguramente por ese motivo, si nosotros definimos un constructor con parámetros, Visual Basic 2008 no crea uno automáticamente sin parámetros. Por tanto, si definimos un constructor con parámetros en una clase y queremos que también tenga uno sin parámetros, lo tenemos que definir nosotros mismos.

El destructor: El punto final de la vida de una clase

De la misma forma que una clase tiene su punto de entrada o momento de nacimiento en el constructor, también tienen un sitio que se ejecutará cuando el objeto creado en la memoria ya no sea necesario, es decir, cuando acabe la vida del objeto creado.

El destructor de Visual Basic 2008 es un método llamado *Finalize*, el cual se hereda de la clase *Object*, por tanto no es necesario que escribamos nada en él. El propio CLR se encargará de llamar a ese método cuando el objeto ya no sea necesario.

La forma en que se destruyen los objetos pueden ser de dos formas: Porque los destruyamos nosotros asignando un valor *Nothing* a la variable que lo referencia, o bien porque el objeto esté fuera de ámbito, es decir, haya salido de

su espacio de ejecución, por ejemplo, si declaramos un objeto dentro de un método, cuando ese método termina, el objeto se destruye, (hay algunas excepciones a esta última regla, como puede ser que ese mismo objeto también esté referenciado por otra variable externa al método.)

Lo que debemos tener muy presente es que en .NET los objetos no se destruyen inmediatamente. Esto es así debido a que en .NET existe un "sistema" que se encarga de realizar esta gestión de limpieza: El recolector de basura o de objetos no usados (*Garbage Collector*, GC). Este recolector de objetos no usados se encarga de comprobar constantemente cuando un objeto no se está usando y es el que decide cuando hay que llamar al destructor.

Debido a esta característica de .NET, si nuestra clase hace uso de recursos externos que necesiten ser eliminados cuando el objeto ya no se vaya a seguir usando, debemos definir un método que sea el encargado de realizar esa liberación, pero ese método debemos llamarlo de forma manual, ya que, aunque en .NET existen formas de hacer que esa llamada sea automática, nunca tenderemos la seguridad de que se llame en el momento oportuno, y esto es algo que, según que casos, puede ser un inconveniente.

Recomendación:
Si nuestra clase utiliza recursos externos, por ejemplo un fichero o una base de datos, debemos definir un método que se encargue de liberarlos y a ese método debemos encargarnos de llamarlo cuando ya no lo necesitemos.
Por definición a este tipo de métodos se les suele dar el nombre Close o Dispose, aunque este último tiene un significado especial y por convención solo debemos usarlo siguiendo las indicaciones de la documentación.

Inicialización directa de objetos

De todos los modos, en Visual Basic 2008 se ha agregado una característica adicional que tiene que ver con la inicialización de objetos.

Hasta ahora hemos visto como instanciar una clase utilizando su constructor, pero quizás lo que no sabíamos es que en Visual Basic 2008 podemos inicializar la clase a través de su constructor e incluso inicializar directamente las propiedades de la clase.

Supongamos por lo tanto el siguiente código fuente:

```
Class Cliente

    Private m_Nombre As String

    Public Property Nombre() As String

       Get

          Return m_Nombre

       End Get

       Set(ByVal value As String)
```

```
            m_Nombre = value

        End Set

    End Property


    Public Sub New()


    End Sub

End Class
```

A continuación instanciaremos la clase y con ello, inicializaremos adicionalmente la propiedad que hemos declarado.

El código quedará de la siguiente forma:

```
Dim claseCliente As New Cliente() With {.Nombre = "Pedro"}
```

Esta forma de inicializar objetos nos permite ser más productivos y prácticos y ahorrar tiempo y líneas de código en nuestras aplicaciones.

Como podemos apreciar, la forma de inicializar las propiedades o las variables de una clase es utilizando la palabra clave *With* seguida de llaves, y dentro de las llaves, un punto seguido de la propiedad o variable que queremos inicializar con sus valor de inicialización. Si tenemos más de una variable a inicializar, deberemos separarlas por comas.

Intellisense nos ayudará enormemente a inicializar la propiedad o variable que deseemos, ya que vincula de forma directa el código de la clase haciéndonos más fácil el trabajo con los miembros de la clase.

Ver vídeo de esta lección (Clases 5: Constructores) - video en Visual Studio 2005 válido para Visual Studio 2008

Lección 2: Clases y estructuras

- Clases
- Definir una clase
- Instanciar una clase

Estructuras

- Accesibilidad
- Propiedades
- Interfaces

Estructuras: Tipos por valor definidos por el usuario

De la misma forma que podemos definir nuestros propios tipos de datos por referencia, Visual Basic 2008 nos permite crear nuestros propios tipos por valor. Para crear nuestros tipos de datos por referencia, usamos la "instrucción" *Class*, por tanto es de esperar que también exista una instrucción para crear nuestros tipos por valor, y esa instrucción es: *Structure*, por eso en Visual Basic 2008 a los tipos por valor definidos por el usuario se llaman estructuras.

Las estructuras pueden contener los mismos miembros que las clases, aunque algunos de ellos se comporten de forma diferente o al menos tengan algunas restricciones, como que los campos definidos en las estructuras no se pueden inicializar al mismo tiempo que se declaran o no pueden contener constructores "simples", ya que el propio compilador siempre se encarga de crearlo, para así poder inicializar todos los campos definidos.

Otra de las características de las estructuras es que no es necesario crear una instancia para poder usarlas, ya que es un tipo por valor y los tipos por valor no necesitan ser instanciados para que existan.

Definir una estructura

Las estructuras se definen usando la palabra *Structure* seguida del nombre y acaba usando las instrucciones *End Structure*.

El siguiente código define una estructura llamada Punto en la que tenemos dos campos públicos.

```
Structure Punto

    Public X As Integer

    Public Y As Integer

End Structure
```

Para usarla podemos hacer algo como esto:

```
Dim p As Punto

p.X = 100

p.Y = 75
```

También podemos usar *New* al declarar el objeto:

```
Dim p As New Punto
```

Aunque en las estructuras, usar *New*, sería algo redundante y por tanto no necesario.

Las estructuras siempre se almacenan en la pila, por tanto deberíamos tener la precaución de no crear estructuras con muchos campos o con muchos miembros, ya que esto implicaría un mayor consumo del "preciado" espacio de la pila.

Constructores de las estructuras

Tal y como hemos comentado, las estructuras siempre definen un constructor sin parámetros, este constructor no lo podemos definir nosotros, es decir, siempre existe y es el que el propio compilador de Visual Basic 2008 define. Por tanto, si queremos agregar algún constructor a una estructura, este debe tener parámetros y, tal como ocurre con cualquier método o como ocurre en las clases, podemos tener varias sobrecargas de constructores parametrizados en las estructuras. La forma de definir esos constructores es como vimos en las clases: usando distintas sobrecargas de un método llamado *New*, en el caso de las estructuras también podemos usar la palabra clave *Me* para referirnos a la instancia actual.

Esto es particularmente práctico cuando los parámetros del constructor se llaman de la misma forma que los campos declarados en la estructura, tal como ocurre en el constructor mostrado en la siguiente definición de la estructura Punto.

```
Structure Punto

    Public X As Integer

    Public Y As Integer

    '

    Sub New(ByVal x As Integer, ByVal y As Integer)

        Me.X = x

        Me.Y = y

    End Sub

End Structure
```

Nota:
Tanto en las estructuras como en las clases podemos tener

constructores compartidos, (Shared), en el caso de las estructuras, este tipo de constructor es el único que podemos declarar sin parámetros.

Destructores de las estructuras

Debido a que las estructuras son tipos por valor y por tanto una variable declarada con un tipo por valor "contiene el valor en si misma", no podemos destruir este tipo de datos, lo más que conseguiríamos al asignarle un valor nulo (*Nothing*) sería eliminar el contenido de la variable, pero nunca podemos destruir ese valor. Por tanto, en las estructuras no podemos definir destructores.

Los miembros de una estructura

Como hemos comentado, los miembros o elementos que podemos definir en una estructura son los mismos que ya vimos en las clases. Por tanto aquí veremos las diferencias que existen al usarlos en las estructuras.

Campos

Como vimos, las variables declaradas a nivel del tipo, son los campos, la principal diferencia con respecto a las clases, es que los campos de una estructura no pueden inicialiarse en la declaración y el valor que tendrán inicialmente es un valor "nulo", que en el caso de los campos de tipo numéricos es un cero. Por tanto, si necesitamos que los campos tengan algún valor inicial antes de usarlos, deberíamos indicarlo a los usuarios de nuestra estructura y proveer un constructor que realice las inicializaciones correspondientes, pero debemos recordar que ese constructor debe tener algún parámetro, ya que el predeterminado sin parámetros no podemos "reescribirlo".

Los únicos campos que podemos inicializar al declararlos son los campos compartidos, pero como tendremos oportunidad de ver, estos campos serán accesibles por cualquier variable declarada y cualquier cambio que realicemos en ellos se verá reflejado en el resto de "instancias" de nuestro tipo.

Métodos y otros elementos

El resto de los miembros de una estructura se declaran y usan de la misma forma que en las clases, si bien debemos tener en cuenta que el modificador de accesibilidad predeterminado para los miembros de una estructura es *Public*, (incluso si son campos declarados con *Dim*).

Otro detalle a tener en cuenta es que en una estructura siempre debe existir al menos un evento o un campo no compartido, no se permiten estructuras en las que solo tienen constantes, métodos y/o propiedades, estén o no compartidos.

Cómo usar las estructuras

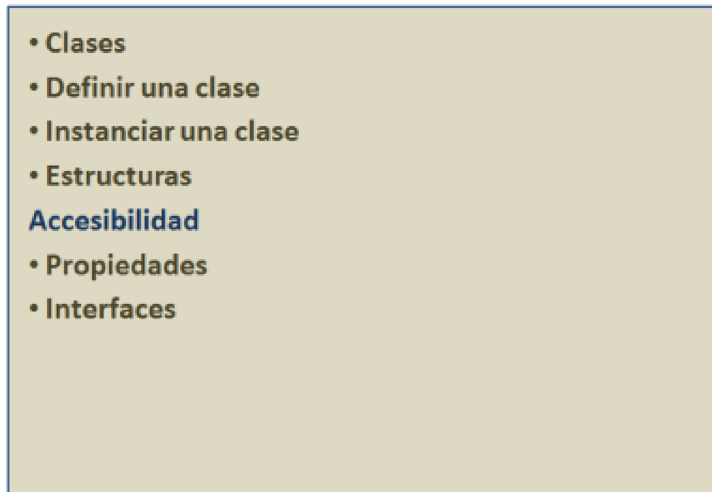
Tal como hemos comentado las estructuras son tipos por valor, para usar los tipos por valor no es necesario instanciarlos explícitamente, ya que el mero hecho de declararlos indica que estamos creando un nuevo objeto en memoria. Por tanto, a diferencia de las clases o tipos por referencia, cada variable definida como un tipo de estructura será independiente de otras variables declaradas, aunque no las hayamos instanciado.

Esta característica de las estructuras nos permite hacer copias "reales" no copia de la referencia (o puntero) al objeto en memoria como ocurre con los tipos por referencia.
Veámoslos con un ejemplo.

```
Dim p As New Punto(100, 75)

Dim p1 As Punto

p1 = p

p1.X = 200

' p.X vale 100 y p1.X vale 200
```

En este trozo de código definimos e *instanciamos* una variable del tipo **Punto**, a continuación declaramos otra variable del mismo tipo y le asignamos la primera, si estos tipos fuesen por referencia, tanto una como la otra estarían haciendo referencia al mismo objeto en memoria, y cualquier cambio realizado a cualquiera de las dos variables afectarían al mismo objeto, pero en el caso de las estructuras (y de los tipos por valor), cada cambio que realicemos se hará sobre un *objeto* diferente, por tanto la asignación del valor **200** al campo **X** de la variable **p1** solo afecta a esa variable, dejando intacto el valor original de la variable **p**.

Ver vídeo de esta lección (Estructuras) - video en Visual Studio 2005 válido para Visual Studio 2008



Accesibilidad y ámbito

Tal y como comentamos anteriormente, dependiendo de dónde y cómo estén declarados los tipos de datos y los miembros definidos en ellos, tendremos o no acceso a esos elementos.

Recordemos que el ámbito es el alcance con el que podemos acceder a un elemento y depende de dónde esté declarado, por otro lado, la accesibilidad depende de cómo declaremos cada uno de esos elementos.

Ámbito

Dependiendo de donde declaremos un miembro o un tipo, éste tendrá mayor alcance o cobertura, o lo que es lo mismo, dependiendo del ámbito en el que usemos un elemento, podremos acceder a él desde otros puntos de nuestro código.
A continuación veremos con detalle los ámbitos en los que podemos declarar los distintos elementos de Visual Basic 2008.

- **Ámbito de bloque:** Disponible únicamente en el bloque de código en el que se ha declarado.
Por ejemplo, si declaramos una variable dentro de un bucle *For* o un *If Then*, esa variable solo estará accesible dentro de ese bloque de código.
- **Ámbito de procedimiento:** Disponible únicamente dentro del procedimiento en el que se ha declarado. Cualquier variable declarada dentro de un procedimiento (método o propiedad) solo estará accesible en ese procedimiento y en cualquiera de los bloques internos a ese procedimiento.
- **Ámbito de módulo:** Disponible en todo el código del módulo, la clase o la estructura donde se ha declarado. Las variables con ámbito a nivel de módulo, también estarán disponibles en los procedimientos declarados en el módulo (clase o estructura) y por extensión a cualquier bloque dentro de cada procedimiento.
- **Ámbito de espacio de nombres:** Disponible en todo el código del espacio de nombres. Este es el nivel mayor de cobertura o alcance, aunque en este

nivel solo podemos declarar tipos como clases, estructuras y enumeraciones, ya que los procedimientos solamente se pueden declarar dentro de un tipo.

Nota:
Por regla general, cuando declaramos una variable en un ámbito, dicha variable "ocultará" a otra que tenga el mismo nombre y esté definida en un bloque con mayor alcance, aunque veremos que en Visual Basic 2008 existen ciertas restricciones dependiendo de dónde declaremos esas variables.

En Visual Basic 2008 podemos definir una variable dentro de un bloque de código, en ese caso dicha variable solo será accesible dentro de ese bloque. Aunque, como veremos a continuación, en un procedimiento solamente podremos definir variables que no se oculten entre sí, estén o no dentro de un bloque de código.

Ámbito de bloque

En los siguientes ejemplos veremos cómo podemos definir variables para usar solamente en el bloque en el que están definidas.

Los bloques de código en los que podemos declarar variables son los bucles, (*For, Do, While*), y los bloques condicionales, (*If, Select*). Por ejemplo, dentro de un procedimiento podemos tener varios de estos bloques y por tanto podemos definir variables "internas" a esos bloques:

```
Dim n As Integer = 3

'

For i As Integer = 1 To 10

    Dim j As Integer

    j += 1

    If j < n Then

        '...

    End If

Next

'

If n < 5 Then

    Dim j As Integer = n * 3

End If

'

Do

    Dim j As Integer

    For i As Integer = 1 To n
```

```
        j += i
    Next
    If j > 10 Then Exit Do
Loop
```

La variable **n** estará disponible en todo el procedimiento, por tanto podemos acceder a ella desde cualquiera de los bloques.

En el primer bucle *For*, definimos la variable **i** como la variable a usar de contador, esta variable solamente estará accesible dentro de este bucle *For*. Lo mismo ocurre con la variable **j**.

En el primer *If* definimos otra variable **j**, pero esa solo será accesible dentro de este bloque *If* y por tanto no tiene ninguna relación con la definida en el bucle *For* anterior.
En el bucle *Do* volvemos a definir nuevamente una variable **j**, a esa variable la podemos acceder solo desde el propio bucle *Do* y cualquier otro bloque de código interno, como es el caso del bucle *For*, en el que nuevamente declaramos una variable llamada **i**, que nada tiene que ver con el resto de variables declaradas con el mismo nombre en los otros bloques.

Lo único que no podemos hacer en cualquiera de esos bloques, es declarar una variable llamada **n**, ya que al estar declarada en el procedimiento, el compilador de Visual Basic 2008 nos indicará que no podemos ocultar una variable previamente definida fuera del bloque, tal como podemos ver en la figura 2.12.

```
Dim n As Integer = 3
'
For i As Integer = 0 To 9
    Dim n As Integer
        La variable 'n' oculta una variable en un bloque de inclusión.

Next
```

Figura 2.12. Error al ocultar una variable definida en un procedimiento

Esta restricción solo es aplicable a las variables declaradas en el procedimiento, ya que si declaramos una variable a nivel de módulo, no habrá ningún problema para usarla dentro de un bloque, esto es así porque en un procedimiento podemos declarar variables que se llamen de la misma forma que las declaradas a nivel de módulo, aunque éstas ocultarán a las del "nivel" superior.

Ámbito de procedimiento

Las variables declaradas en un procedimiento tendrán un ámbito o cobertura que será el procedimiento en el que está declaradas, y como hemos visto, ese ámbito incluye también cualquier bloque de código declarado dentro del procedimiento. Estas variables ocultarán a las que se hayan declarado fuera del procedimiento, si bien, dependiendo del tipo de módulo, podremos acceder a esas variables "externas" indicando el nombre completo del módulo o bien usando la instrucción

Me, tal como vimos en el código del constructor parametrizado de la estructura **Punto**.

Pero mejor veámoslo con un ejemplo. En el siguiente código, definimos una clase en la que tenemos un campo llamado **Nombre**, también definimos un método en el que internamente se utiliza una variable llamada nombre, para acceder a la variable declarada en la clase, tendremos que usar la instrucción o palabra clave *Me*.

```
Public Class Cliente

    Public Nombre As String = "Juan"

    Function Mostrar() As String

        Dim nombre As String = "Pepita"

        Return "Externo= " & Me.Nombre & ", interno= " & nombre

    End Function

End Class
```

En este ejemplo, el hecho de que una variable esté declarada con la letra **ENE** en mayúscula o en minúscula no implica ninguna diferencia, ya que Visual Basic 2008 no hace distinciones de este tipo; aún así, Visual Basic 2008 respetará el nombre según lo hemos escrito y no cambiará automáticamente el "case" de las variables, salvo cuando están en el mismo nivel de ámbito, es decir, si la variable **nombre** que hemos definido en la función **Mostrar** la volvemos a usar dentro de esa función, Visual Basic 2008 la seguirá escribiendo en minúsculas, pero si escribimos "nombre" fuera de esa función, VB se dará cuenta de que hay una variable declarada con la ENE en mayúsculas y automáticamente la cambiará a **Nombre**, aunque nosotros la escribamos de otra forma.

Ámbito de módulo

Cuando hablamos de módulos, nos estamos refiriendo a un "tipo" de datos, ya sea una clase, una estructura o cualquier otro tipo de datos que nos permita definir .NET.
En estos casos, las variables declaradas dentro de un tipo de datos serán visibles desde cualquier parte de ese tipo, siempre teniendo en cuenta las restricciones mencionadas en los casos anteriores.

Ámbito de espacio de nombres

Los espacios de nombres son los contenedores de tipos de datos de mayor nivel, y sirven para contener definiciones de clases, estructuras, enumeraciones y delegados. Cualquier tipo definido a nivel de espacio de nombres estará disponible para cualquier otro elemento definido en el mismo espacio de nombres. Al igual que ocurre en el resto de ámbitos "inferiores", si definimos un tipo en un espacio de nombres, podemos usar ese mismo nombre para nombrar a un procedimiento o a una variable, en cada caso se aplicará el ámbito correspondiente y, tal como vimos anteriormente, tendremos que usar nombres únicos para poder acceder a los nombres definidos en niveles diferentes.

La palabra clave Global

En Visual Basic 2008 podemos definir espacios de nombres cuyos nombres sean los mismos que los definidos en el propio .NET Framework, para evitar conflictos de ámbitos, podemos usar la palabra clave *Global* para acceder a los que se han definido de forma "global" en .NET.
Por ejemplo, si tenemos el siguiente código en el que definimos una clase dentro de un espacio de nombres llamado *System* y queremos acceder a uno de los tipos definidos en el espacio de nombres *System* de .NET, tendremos un problema:

```
Namespace System

    Class Cliente

        Public Nombre As String

        Public Edad As System.Int32

    End Class

End Namespace
```

El problema es que el compilador de Visual Basic 2008 nos indicará que el tipo *Int32* no está definido, ya que intentará buscarlo dentro del ámbito que actualmente tiene, es decir, la declaración que nosotros hemos hecho de *System*, por tanto para poder acceder al tipo *Int32* definido en el espacio de nombres "global" *System* de .NET tendremos que usar la instrucción *Global*, por suerte el IDE de Visual Studio 2008 reconoce este tipo de error y nos ofrece ayuda para poder solventar el conflicto, tal como vemos en la figura 2.13:

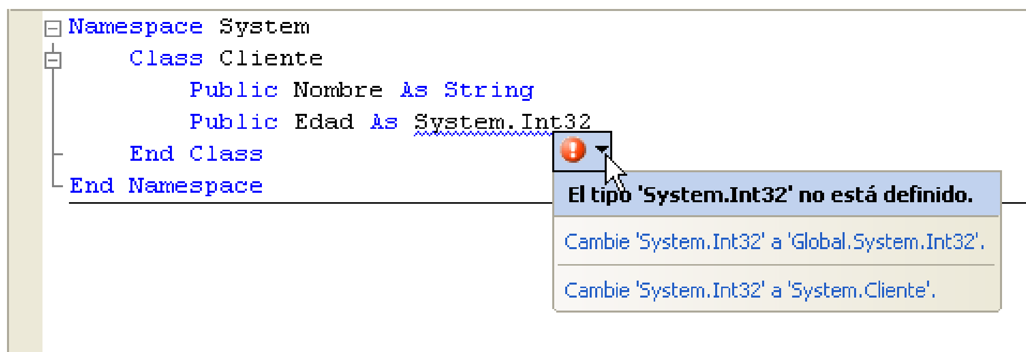


Figura 2.13. Ayuda del IDE en los conflictos de espacios nombres globales

> **Nota:**
> Afortunadamente este conflicto con los espacios de nombres no será muy habitual para los desarrolladores que usemos el idioma de Cervantes, por la sencilla razón de que los espacios de nombres de .NET Framework suelen estar definidos usando palabras en inglés.

Accesibilidad

La accesibilidad es la característica que podemos aplicar a cualquiera de los elementos que definamos en nuestro código. Dependiendo de la accesibilidad declarada tendremos distintos tipos de accesos a esos elementos.

Los modificadores de accesibilidad que podemos aplicar a los tipos y elementos definidos en nuestro código pueden ser cualquiera de los mostrados en la siguiente lista:

- **Public:** Acceso no restringido. Este es modificador de accesibilidad con mayor "cobertura", podemos acceder a cualquier miembro público desde cualquier parte de nuestro código. Aunque, como veremos, este acceso no restringido puede verse reducido dependiendo de dónde lo usemos.
- **Protected:** Acceso limitado a la clase contenedora o a los tipos derivados de esta clase. Este modificador solamente se usa con clases que se deriven de otras.
- **Friend:** Acceso limitado al proyecto actual. Visual Basic 2008 aplica este modificador de forma predeterminada a los procedimientos declarados en las clases.
- **Protected Friend:** Acceso limitado al proyecto actual o a los tipos derivados de la clase contenedora. Una mezcla de los dos modificadores anteriores.
- **Private:** Acceso limitado al tipo contenedor. Es el más restrictivos de todos los modificadores de accesibilidad y en el caso de los campos declarados en las clases (*Class*) equivale a usar *Dim*.

Estos modificadores de accesibilidad los podemos usar tanto en clases, estructuras, interfaces, enumeraciones, delegados, eventos, métodos, propiedades y campos. Aunque no serán aplicables en espacios de nombres (*Namespace*) ni clases de tipo Module, en estos dos casos siempre tendrán cobertura pública, si bien no se permite el uso de ningún modificador.

Accesibilidad de las variables en los procedimientos

Las variables declaradas dentro de un procedimiento solo son accesibles dentro de ese procedimiento, en este caso solo se puede aplicar el ámbito privado, aunque no podremos usar la instrucción *Private*, sino *Dim* o *Static*.

> **Nota:**
> La palabra clave Static, nos permite definir una variable privada (o local) al procedimiento para que mantenga el valor entre diferentes llamadas a ese procedimiento; esto contrasta con el resto de variables declaradas en un procedimiento cuya duración es la misma que la vida del propio procedimiento, por tanto, las variables no estáticas pierden el valor al terminar la ejecución del procedimiento.

Las accesibilidades predeterminadas

La accesibilidad de una variable o procedimiento en la que no hemos indicado el modificador de accesibilidad dependerá del sitio en el que la hemos declarado.

Por ejemplo, en las estructuras si definimos los campos usando *Dim*, estos tendrán un ámbito igual que si le hubiésemos aplicado el modificador *Public*; sin embargo, esa misma variable declarada en una clase (*Class* o *Module*) tendrá una accesibilidad *Private*. Así mismo, si el elemento que declaramos es un procedimiento y no indicamos el modificador de ámbito, éste tendrá un ámbito de tipo *Public* si lo definimos en una estructura y si el lugar en el que lo declaramos es una clase (o *Module*), éste será *Friend*.

En la siguiente tabla tenemos la accesibilidad predeterminada de cada tipo (clase, estructura, etc.), así como de las variables declaradas con *Dim* y de los procedimientos en los que no se indican el modificador de accesibilidad.

Tipo	**del tipo**	**de las variables declaradas con Dim**	**de los procedimientos**
Class	Friend	Private	Friend
Module	Friend	Private	Friend
Structure	Friend	Public	Public
Enum	Public Friend	N.A. (los miembros siempre son públicos)	N.A.
Interface	Friend	N.A. (no se pueden declarar variables)	Public (no se permite indicarlo)

Tabla 2.3. La accesibilidad predeterminada de los tipos

Tal como podemos ver en la tabla 2.3, la accesibilidad predeterminada, (la que tienen cuando no se indica expresamente con un modificador), de todos los tipos es *Friend*, es decir, accesible a todo el proyecto, aunque en el caso de las enumeraciones el modificador depende de dónde se declare dicha enumeración, si está declarada a nivel de espacio de nombres será *Friend*, en el resto de los casos será *Public*.
En la tercera columna tenemos la accesibilidad predeterminada cuando declaramos las variables con *Dim*, aunque en las interfaces y en las enumeraciones no se permiten declarar variables.
La última columna es la correspondiente a los procedimientos, en el caso de las interfaces no se puede aplicar ningún modificador de accesibilidad y de forma predeterminada son públicos.

En esta otra tabla tenemos la accesibilidad permitida en cada tipo así como las que podemos indicar en los miembros de esos tipos.

Tipo	**del tipo**	**de los miembros**
Class	Public Friend Private Protected Protected Friend	Public Friend Private Protected Protected Friend
Module	Public Friend	Public Friend Private
Structure	Public Friend Private	Public Friend Private
Enum	Public Friend Private	N.A.
Interface	Public Friend Private	N.A. Siempre son públicos

	Protected Protected Friend	

Tabla 2.4. Accesibilidades permitidas en los tipos

Algunos de los modificadores que podemos indicar en los tipos dependen de dónde declaremos esos tipos, por ejemplo, tan solo podremos indicar el modificador privado de las enumeraciones cuando estas se declaren dentro de un tipo. En el caso de las clases e interfaces, los modificadores *Protected* y *Protected Friend* solo podremos aplicarlos cuando están declaradas dentro de una clase (*Class*).

Anidación de tipos

Tal como hemos comentado en el párrafo anterior, podemos declarar tipos dentro de otros tipos, por tanto el ámbito y accesibilidad de esos tipos dependen del ámbito y accesibilidad del tipo que los contiene. Por ejemplo, si declaramos una clase con acceso *Friend*, cualquier tipo que esta clase contenga siempre estará supeditado al ámbito de esa clase, por tanto si declaramos otro tipo interno, aunque lo declaremos como *Public*, nunca estará más accesible que la clase contenedora, aunque en estos casos no habrá ningún tipo de confusión, ya que para acceder a los tipos declarados dentro de otros tipos siempre tendremos que indicar la clase que los contiene.
En el siguiente código podemos ver cómo declarar dos clases "anidadas". Tal como podemos comprobar, para acceder a la clase Salario debemos indicar la clase Cliente, ya que la única forma de acceder a una clase anidada es mediante la clase contenedora.

```
Friend Class Cliente

    Public Nombre As String



    Public Class Salario

        Public Importe As Decimal

    End Class

End Class



' Para usar la clase Salario debemos declararla de esta forma:

Dim s As New Cliente.Salario

s.Importe = 2200
```

Los tipos anidables

Cualquiera de los tipos mostrados en la tabla 2.4, excepto las enumeraciones, pueden contener a su vez otros tipos. La excepción es el tipo *Module* que aunque puede contener a otros tipos, no puede usarse como tipo anidado. Una enumeración siempre puede usarse como tipo anidado.

> **Nota:**
> Los espacios de nombres también pueden anidarse y contener a su vez cualquiera de los tipos mostrados en la tabla 2.4, incluso tipos Module.

El nombre completo de un tipo

Tal como hemos visto, al poder declarar tipos dentro de otros tipos y estos a su vez pueden estar definidos en espacios de nombres, podemos decir que el nombre "completo" de un tipo cualquiera estará formado por el/los espacios de nombres y el/los tipos que los contiene, por ejemplo si la clase **Cliente** definida anteriormente está a su vez dentro del espacio de nombres **Ambitos**, el nombre completo será: **Ambitos.Cliente** y el nombre completo de la clase **Salario** será: **Ambitos.Cliente.Salario**.
Aunque para acceder a la clase **Cliente** no es necesario indicar el espacio de nombres, al menos si la queremos usar desde cualquier otro tipo declarado dentro de ese espacio de nombres, pero si nuestra intención es usarla desde otro espacio de nombre externo a **Ambitos**, en ese caso si que tendremos que usar el nombre completo.

Por ejemplo, en el siguiente código tenemos dos espacios de nombres que no están anidados, cada uno de ellos declara una clase y desde una de ellas queremos acceder a la otra clase, para poder hacerlo debemos indicar el nombre completo, ya que en caso contrario, el compilador de Visual Basic 2008 sería incapaz de saber a que clase queremos acceder.

```
Namespace Uno

    Public Class Clase1

        Public Nombre As String

    End Class

End Namespace


Namespace Dos

    Public Class Clase2

        Public Nombre As String

        Sub Main()

            Dim c1 As New Uno.Clase1

            c1.Nombre = "Pepe"

        End Sub

    End Class

End Namespace
```

Esto mismo lo podemos aplicar en el caso de que tengamos dos clases con el mismo nombre en espacios de nombres distintos.

Nota:
En el mismo proyecto podemos tener más de una declaración de un espacio de nombres con el mismo nombre, en estos casos el compilador lo tomará como si todas las clases definidas estuvieran dentro del mismo espacio de nombres, aunque estos estén definidos en ficheros diferentes.

Importación de espacios de nombres

Tal como hemos comentado, los espacios de nombres pueden contener otros espacios de nombres y estos a su vez también pueden contener otros espacios de nombres o clases, y como hemos visto, para poder acceder a una clase que no esté dentro del mismo espacio de nombres debemos indicar el "nombre completo".

Para evitar estar escribiendo todos los espacios de nombres en los que está la clase que nos interesa declarar, podemos usar una especie de acceso directo o para que lo entendamos mejor, podemos crear una especie de "Path", de forma que al declarar una variable, si esta no está definida en el espacio de nombres actual, el compilador busque en todos los espacios de nombres incluidos en esas rutas (paths).

Esto lo conseguimos usando la instrucción *Imports* seguida del espacio de nombres que queremos importar o incluir en el path de los espacios de nombres. Podemos usar tantas importaciones de espacios de nombres como necesitemos y estas siempre deben aparecer al principio del fichero, justo después de las instrucciones *Options*.

Por ejemplo, si tenemos el código anterior y hacemos la importación del espacio de nombres en el que está definida la clase **Clase1**:

```
Imports Uno
```

podremos acceder a esa clase de cualquiera de estas dos formas:

```
Dim c1 As New Uno.Clase1

Dim c1 As New Clase1
```

Alias de espacios de nombres

Si hacemos demasiadas importaciones de nombres, el problema con el que nos podemos encontrar es que el *IntelliSense* de Visual Basic 2008 no sea de gran ayuda, ya que mostrará una gran cantidad de clases, y seguramente nos resultará más difícil encontrar la clase a la que queremos acceder, o también podemos encontrarnos en ocasiones en las que nos interese usar un nombre corto para acceder a las clases contenidas en un espacio de nombres, por ejemplo, si queremos indicar de forma explícita las clases de un espacio de nombres como el de *Microsoft.VisualBasic*, podemos hacerlo de esta forma:

```
Imports vb = Microsoft.VisualBasic
```

De esta forma podemos usar el "alias" **vb** para acceder a las clases y demás tipos definidos en ese espacio de nombres.

En las figuras 2.14 2.15 podemos ver las dos formas de acceder a las clases del espacio de ese espacio de nombres, en el primer caso sin usar un alias y en el segundo usando el alias **vb**.

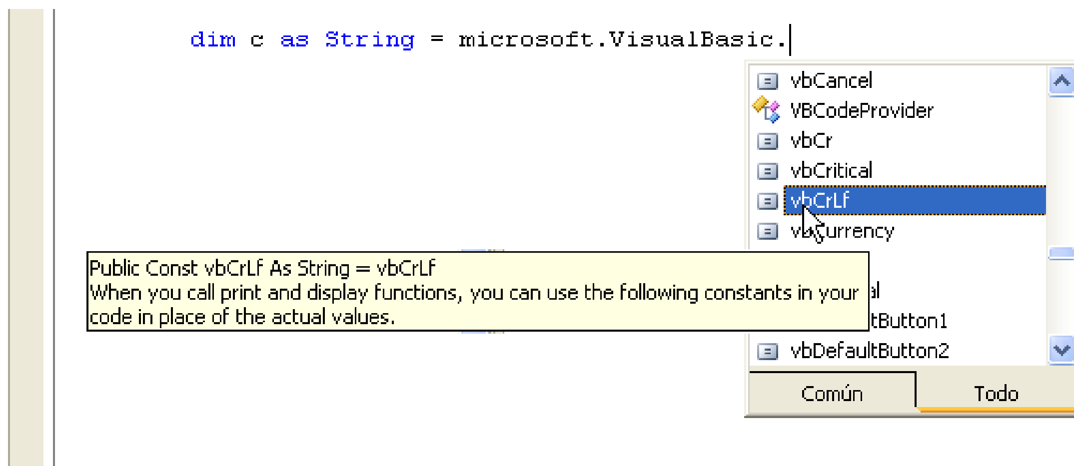


Figura 2.14. Los miembros de un espacio de nombres usando el nombre completo

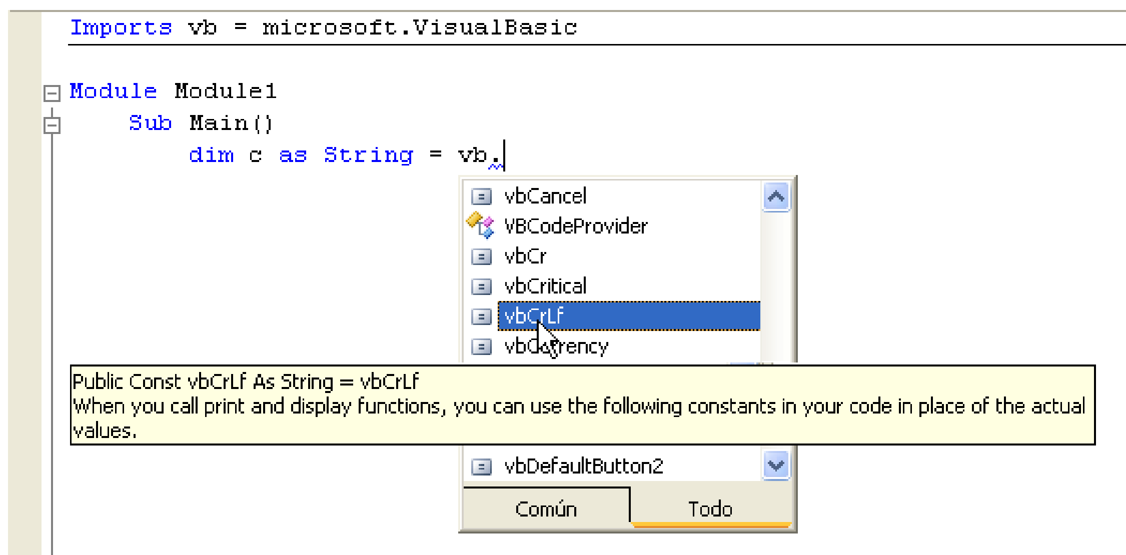


Figura 2.15. Acceder a los miembros de un espacio de nombres usando un alias

Ver vídeo 1 de esta lección (Ámbito - Parte 1) - video en Visual Studio 2005 válido para Visual Studio 2008
Ver vídeo 2 de esta lección (Ámbito - Parte 2) - video en Visual Studio 2005 válido para Visual Studio 2008

Lección 2: Clases y estructuras

- Clases
- Definir una clase
- Instanciar una clase
- Estructuras
- Accesibilidad

Propiedades

- Interfaces

Propiedades

Las propiedades son los miembros de los tipos que nos permiten acceder a los datos que dicho tipo manipula. Normalmente una propiedad está relacionada con un campo, de forma que el campo sea el que realmente contenga el valor y la propiedad simplemente sea una especie de método a través del cual podemos acceder a ese valor.

Debido a que el uso de las propiedades realmente nos permite acceder a los valores de una clase (o tipo), se suelen confundir los campos con las propiedades, de hecho si definimos una variable pública en una clase, ésta se comporta de manera similar, pero realmente un campo (o variable) público no es una propiedad, al menos en el sentido de que el propio .NET Framework no lo interpreta como tal, aunque en la práctica nos puede parecer que es así, ya que se utilizan de la misma forma. Pero no debemos dejarnos llevar por la comodidad y si no queremos perder funcionalidad, debemos diferenciar en nuestro código las propiedades de los campos.

Lo primero que debemos tener presente es que gracias a esta diferenciación que hace .NET Framework, podemos poner en práctica una de las características de la programación orientada a objetos: la encapsulación, de forma, que la manipulación de los datos que una clase contiene siempre se deben hacer de forma "interna" o privada a la clase, dejando a las propiedades la posibilidad de que externamente se manipulen, de forma controlada, esos datos. De esta forma tendremos mayor control sobre cómo se acceden o se asignan los valores a esos datos, ya que al definir una propiedad, tal como hemos comentado, realmente estamos definiendo un procedimiento con el cual podemos controlar cómo se acceden a esos datos.

Definir una propiedad

Debido a que una propiedad realmente nos permite acceder a un dato que la clase (o estructura) manipula, siempre tendremos un campo relacionado con una propiedad. El campo será el que contenga el valor y la propiedad será la que nos permita manipular ese valor.

En Visual Basic 2008, las propiedades las declaramos usando la instrucción *Property* y la definición de la misma termina con *End Property*, dentro de ese bloque de código tenemos que definir otros dos bloques: uno se usará a la hora de leer el valor de la propiedad (bloque *Get*), y el otro cuando queremos asignar un valor a la propiedad (bloque *Set*).

El bloque que nos permite acceder al valor de la propiedad estará indicado por la instrucción *Get* y acaba con *End Get*, por otra parte, el bloque usado para asignar un valor a la propiedad se define mediante la instrucción *Set* y acaba con *End Set*.

Veamos como definir una propiedad en Visual Basic 2008:

```
Public Class Cliente

    Private _nombre As String

    Public Property Nombre() As String

        Get

            Return _nombre

        End Get

        Set(ByVal value As String)

            _nombre = value

        End Set

    End Property

End Class
```

Como podemos comprobar tenemos dos bloques de código, el bloque *Get* que es el que se usa cuando queremos acceder al valor de la propiedad, por tanto devolvemos el valor del campo privado usado para almacenar ese dato. El bloque *Set* es el usado cuando asignamos un valor a la propiedad, este bloque tiene definido un parámetro (*value*) que representa al valor que queremos asignar a la propiedad.

Propiedades de solo lectura

En Visual Basic 2008 podemos hacer que una propiedad sea de solo lectura, de forma que el valor que representa no pueda ser cambiado.

Para definir este tipo de propiedades solo debemos indicar el bloque Get de la propiedad, además de indicar de forma expresa que esa es nuestra intención, para ello debemos usar el modificador *ReadOnly* para que el compilador de Visual Basic 2008 acepte la declaración:

```
Public ReadOnly Property Hoy() As Date

    Get

        Return Date.Now

    End Get

End Property
```

Propiedades de solo escritura

De igual forma, si queremos definir una propiedad que sea de solo escritura, solo definiremos el bloque *Set*, pero al igual que ocurre con las propiedades de solo lectura, debemos indicar expresamente que esa es nuestra intención, para ello usaremos la palabra clave *WriteOnly*:

```
Public WriteOnly Property Password() As String

    Set(ByVal value As String)

        If value = "blablabla" Then

            ' ok

        End If

    End Set

End Property
```

Diferente accesibilidad para los bloques Get y Set

En las propiedades normales (de lectura y escritura), podemos definir diferentes niveles de accesibilidad a cada uno de los dos bloques que forman una propiedad. Por ejemplo, podríamos definir el bloque *Get* como público, (siempre accesible), y el bloque *Set* como *Private*, de forma que solo se puedan realizar asignaciones desde dentro de la propia clase.

Por ejemplo, el salario de un empleado podríamos declararlo para que desde cualquier punto se pueda saber el importe, pero la asignación de dicho importe solo estará accesible para los procedimientos definidos en la propia clase:

```
Public Class Empleado

    Private _salario As Decimal

    Public Property Salario() As Decimal

        Get

            Return _salario

        End Get

        Private Set(ByVal value As Decimal)
```

```
            _salario = value

        End Set

    End Property

End Class
```

Para hacer que el bloque *Set* sea privado, lo indicamos con el modificador de accesibilidad *Private*, al no indicar ningún modificador en el bloque *Get*, éste será el mismo que el de la propiedad.

> **Nota:**
> El nivel de accesibilidad de los bloques Get o Set debe ser igual o inferior que el de la propiedad, por tanto si la propiedad la declaramos como Private, no podemos definir como público los bloques Get o Set.

Propiedades predeterminadas

Las propiedades predeterminadas son aquellas en las que no hay que indicar el nombre de la propiedad para poder acceder a ellas, pero en Visual Basic 2008 no podemos definir como predeterminada cualquier propiedad, ya que debido a como se realizan las asignaciones de objetos en .NET, siempre debemos indicar la propiedad a la que queremos asignar el valor, porque en caso de que no se indique ninguna, el compilador interpretará que lo que queremos asignar es un objeto y no un valor a una propiedad.
Para evitar conflictos o tener que usar alguna instrucción "extra" para que se sepa si lo que queremos asignar es un valor o un objeto, en Visual Basic 2008 las propiedades predeterminadas siempre deben ser parametrizadas, es decir, tener como mínimo un parámetro.
Para indicar que una propiedad es la propiedad por defecto lo debemos hacer usando la instrucción *Default*:

```
Default Public ReadOnly Property Item(ByVal index As Integer) As
Empleado

    Get

        ' ...

    End Get

End Property
```

Como vemos en este ejemplo, una propiedad por defecto puede ser de solo lectura y también de solo escritura o de lectura/escritura.

Para usar esta propiedad, al ser la propiedad por defecto, no es necesario indicar el nombre de la propiedad, aunque si así lo deseamos podemos indicarla, aunque en este caso no tendría mucha utilidad el haberla definido como propiedad por defecto:

```
Dim e As New Empleado

Dim e1 As Empleado = e(2)
```

```
' También podemos usarla indicando el nombre de la propiedad:

Dim e2 As Empleado = e.Item(2)
```

Sobrecarga de propiedades predeterminadas

Debido a que las propiedades predeterminadas de Visual Basic 2008 deben recibir un parámetro, podemos crear sobrecargas de una propiedad predeterminada, aunque debemos recordar que para que esa sobrecarga pueda ser posible, el tipo o número de argumentos deben ser distintos entre las distintas sobrecargas, por ejemplo podríamos tener una sobrecarga que reciba un parámetro de tipo entero y otra que lo reciba de tipo cadena:

```
Default Public ReadOnly Property Item(ByVal index As Integer) As
Empleado

    Get

        ' ...

    End Get

End Property



Default Public Property Item(ByVal index As String) As Empleado

    Get

        ' ...

    End Get

    Set(ByVal value As Empleado)

        '

    End Set

End Property
```

Incluso como vemos en este código una de las sobrecargas puede ser de solo lectura y la otra de lectura/escritura. Lo que realmente importa es que el número o tipo de parámetros de cada sobrecarga sea diferente.

Las propiedades predeterminadas tienen sentido en Visual Basic 2008 cuando queremos que su uso sea parecido al de un array. Por tanto es habitual que las clases de tipo colección sean las más indicadas para definir propiedades por defecto. Aunque no siempre el valor devuelto debe ser un elemento de una colección o array, ya que podemos usar las propiedades predeterminadas para acceder a los miembros de una clase "normal", de forma que se devuelva un valor según el parámetro indicado, esto nos permitiría, por ejemplo, acceder a los miembros de la clase desde un bucle *For*. Si definimos una propiedad predeterminada como en el siguiente código:

```
Public Class Articulo

    Public Descripción As String

    Public PrecioVenta As Decimal

    Public Existencias As Decimal

    Default Public ReadOnly Property Item(ByVal index As Integer)
As String

        Get

            Select Case index

                Case 0

                    Return Descripción

                Case 1

                    Return PrecioVenta.ToString

                Case 2

                    Return Existencias.ToString

                Case Else

                    Return ""

                End Select

        End Get

    End Property

End Class
```

La podemos usar de esta forma:

```
For i As Integer = 0 To 2

    Console.WriteLine( art(i) )

Next
```

Resumiendo:
Las propiedades predeterminadas en Visual Basic 2008 siempre deben tener un parámetro, para que su uso se asemeje a un array, es decir, se use como indizador de la clase. Por convención, cuando se usan como indizador, el nombre de la propiedad predeterminada suele ser **Item**.

Ver vídeo de esta lección (Clases 3: Las propiedades) - video en Visual Studio 2005 válido para Visual Studio 2008

Lección 2: Clases y estructuras

- Clases
- Definir una clase
- Instanciar una clase
- Estructuras
- Accesibilidad
- Propiedades

Interfaces

Interfaces

Las interfaces son un elemento bastante importante en .NET Framework, ya que de hecho se utiliza con bastante frecuencia, en esta lección veremos que son las interfaces y como utilizarlas en nuestros proyectos, también veremos que papel juegan en .NET y cómo aplicar algunas de las definidas en la biblioteca base.

¿Qué es una interfaz?

Las interfaces son una forma especial de una clase, aunque la diferencia principal con las clases es que las interfaces no contienen código ejecutable, solo definen los miembros.

Las interfaces se utilizan para indicar el "comportamiento" que tendrá una clase, o al menos qué miembros debe definir esa clase.

Para definir una interfaz en Visual Basic 2008 tenemos que usar la instrucción *Interface* seguida del nombre de la interfaz y terminar la declaración con *End Interface*:

```
Public Interface IAnimal

    '...

End Interface
```

Nota:
Según las indicaciones de nomenclatura de .NET Framework, se recomienda que todas las interfaces empiecen con una I mayúscula seguida del nombre al que hacer referencia la interfaz.

¿Qué contiene una interfaz?

Al principio de esta lección hemos comentado que las interfaces no contienen código, solo define los miembros que contiene. Esa definición la haremos como cualquier otra, con la diferencia de que no incluimos ningún código, solo la "firma" o el prototipo de cada uno de esos miembros.
En el siguiente código definimos una interfaz que contiene los cuatros tipos de miembros típicos de cualquier clase:

```
Public Interface IPrueba

    Sub Mostrar()

    Function Saludo(ByVal nombre As String) As String

    Property Nombre() As String

    Event DatosCambiados()

End Interface
```

El primer miembro de esta interfaz, es un método de tipo *Sub* que no recibe parámetros.

El siguiente método es una función que devuelve un valor de tipo *String* y recibe un parámetro también de tipo cadena. A continuación definimos una propiedad que devuelve una cadena. Por último, definimos un evento.

Como podemos observar, lo único que tenemos que hacer es indicar el tipo de miembro y si recibe o no algún parámetro o argumento.

Dos cosas importantes sobre las interfaces:

1- No se pueden definir campos.

2- Los miembros de las interfaces siempre son públicos, tal como indicábamos en la tabla 2.3.

Una interfaz es un contrato

Siempre que leemos sobre las interfaces, lo primero con lo que nos solemos encontrar es que *una interfaz es un contrato*. Veamos que nos quieren decir con esa frase.

Tal como acabamos de ver, las interfaces solo definen los miembros, pero no el código a usar en cada uno de ellos, esto es así precisamente porque el papel que juegan las interfaces es el de solo indicar que es lo que una clase o estructura puede, o mejor dicho, debe implementar.

Si en una clase indicamos que queremos "implementar" una interfaz, esa clase debe definir cada uno de los miembros que la interfaz expone. De esta forma nos aseguramos de que si una clase implementa una interfaz, también implementa todos los miembros definidos en dicha interfaz.

Cuando una clase implementa una interfaz está firmando un contrato con el que se compromete a definir todos los miembros que la clase define, de hecho el propio compilador nos obliga a hacerlo.

Las interfaces y el polimorfismo

Como comentamos anteriormente, el polimorfismo es una característica que nos permite acceder a los miembros de un objeto sin necesidad de tener un conocimiento exacto de ese objeto (o de la clase a partir del que se ha instanciado), lo único que tenemos que saber es que ese objeto tiene ciertos métodos (u otros miembros) a los que podemos acceder. También hemos comentado que las interfaces representan un contrato entre las clases que las implementan, por tanto las interfaces pueden ser, (de hecho lo son), un medio para poner en práctica esta característica de la programación orientada a objetos. Si una clase implementa una interfaz, esa clase tiene todos los miembros de la interfaz, por tanto podemos acceder a esa clase, que en principio pude sernos desconocida, desde un objeto del mismo tipo que la interfaz.

Usar una interfaz en una clase

Para poder utilizar una interfaz en una clase, o dicho de otra forma: para "implementar" los miembros expuestos por una interfaz en una clase debemos hacerlo mediante la instrucción *Implements* seguida del nombre de la interfaz:

```
Public Class Prueba

    Implements IPrueba
```

Y como comentábamos, cualquier clase que implemente una interfaz debe definir cada uno de los miembros de esa interfaz, por eso es el propio Visual Basic el encargado de crear automáticamente los métodos y propiedades que la interfaz implementa, aunque solo inserta el "prototipo" de cada uno de esos miembros, dejando para nosotros el trabajo de escribir el código.

Usando la definición de la interfaz **IPrueba** que vimos antes, el código que añadirá VB será el siguiente:

```
Public Class Prueba

    Implements IPrueba



    Public Event DatosCambiados() Implements
IPrueba.DatosCambiados



    Public Sub Mostrar() Implements IPrueba.Mostrar



    End Sub
```

```
    Public Property Nombre() As String Implements IPrueba.Nombre

        Get


        End Get

        Set(ByVal value As String)


        End Set

    End Property


    Public Function Saludo(ByVal nombre As String) As String _

            Implements IPrueba.Saludo


    End Function

End Class
```

Como podemos apreciar, no solo ha añadido las definiciones de cada miembro de la interfaz, sino que también añade código extra a cada uno de esos miembros: la instrucción *Implements* seguida del nombre de la interfaz y el miembro al que se hará referencia.

La utilidad de que en cada uno de los miembros se indique expresamente el método al que se hace referencia, es que podemos usar nombres diferentes al indicado en la interfaz. Por ejemplo, si implementamos esta interfaz en una clase que solo utilizará la impresora, al método **Mostrar** lo podríamos llamar **Imprimir** que sería más adecuado, en ese caso simplemente cambiamos el nombre del método de la clase para que implemente el método **Mostrar** de la interfaz:

```
Public Sub Imprimir() Implements IPrueba.Mostrar


End Sub
```

De esta forma, aunque en la clase se llame de forma diferente, realmente hace referencia al método de la interfaz.

Acceder a los miembros implementados

Una vez que tenemos implementada una interfaz en nuestra clase, podemos acceder a esos miembros de forma directa, es decir, usando un objeto creado a partir de la clase:

```
Dim prueba1 As New Prueba
```

```
prueba1.Mostrar()
```

O bien de forma indirecta, por medio de una variable del mismo tipo que la interfaz:

```
Dim prueba1 As New Prueba

Dim interfaz1 As IPrueba



interfaz1 = prueba1



interfaz1.Mostrar()
```

¿Qué ha ocurre aquí?
Como ya comentamos anteriormente, cuando asignamos variables por referencia, realmente lo que asignamos son referencias a los objetos creados en la memoria, por tanto la variable **interfaz1** está haciendo referencia al mismo objeto que **prueba1**, aunque esa variable solo tendrá acceso a los miembros de la clase Prueba que conoce, es decir, los miembros definidos en **IPrueba**.
Si la clase define otros miembros que no están en la interfaz, la variable **interfaz1** no podrá acceder a ellos.

Saber si un objeto implementa una interfaz

Si las interfaces sirven para acceder de forma anónima a los métodos de un objeto, es normal que en Visual Basic tengamos algún mecanismo para descubrir si un objeto implementa una interfaz.

Para realizar esta comprobación podemos usar en una expresión *If/Then* la instrucción *TypeOf... Is*, de forma que si la variable indicada después de *TypeOf* contiene el tipo especificado después de *Is*, la condición se cumple:

```
If TypeOf prueba1 Is IPrueba Then

    interfaz1 = prueba1



    interfaz1.Mostrar()

End If
```

De esta forma nos aseguramos de que el código se ejecutará solamente si la variable **prueba1** contiene una definición de la interfaz **IPrueba**.

Implementación de múltiples interfaces

En Visual Basic 2008, una misma clase puede implementar más de una interfaz. Para indicar que implementamos más de una interfaz podemos hacerlo de dos formas:

1- Usando nuevamente la instrucción *Implements* seguida del nombre de la interfaz:

```
Public Class Prueba

    Implements IPrueba

    Implements IComparable
```

2- Indicando las otras interfaces en la misma instrucción *Implements*, pero separándolas con comas:

```
Public Class Prueba

    Implements IPrueba, IComparable
```

De cualquiera de las dos formas es válido implementar más de una interfaz, aunque en ambos casos siempre debemos definir los miembros de cada una de esas interfaces.

Múltiple implementación de un mismo miembro

Como acabamos de comprobar, una misma clase puede implementar más de una interfaz, y esto nos puede causar una duda: ¿Qué ocurre si dos interfaces definen un método que es idéntico en ambas? En principio, no habría problemas, ya que el propio Visual Basic crearía dos métodos con nombres diferentes y a cada uno le asignaría la implementación de ese método definido en cada interfaz. Por ejemplo, si tenemos otra interfaz que define el método **Mostrar** y la implementamos en la clase **Prueba**, la declaración podría quedar de esta forma:

```
Public Interface IMostrar

    Sub Mostrar()

End Interface

Public Sub Mostrar1() Implements IMostrar.Mostrar



End Sub
```

Aunque si ambos métodos hacen lo mismo, en este ejemplo mostrar algo, podríamos hacer que el mismo método de la clase sirva para implementar el de las dos interfaces:

```
Public Sub Mostrar() Implements IPrueba.Mostrar, IMostrar.Mostrar
```

```
End Sub
```

Es decir, lo único que tendríamos que hacer es indicar la otra implementación separándola con una coma.

¿Dónde podemos implementar las interfaces?

Para ir acabando este tema nos queda por saber, entre otras cosas, dónde podemos implementar las interfaces, es decir, en que tipos de datos podemos usar *Implements*.

La implementación de interfaces la podemos hacer en las clases (*Class*), estructuras (*Structure*) y en otras interfaces (*Interface*).

Debido a que una interfaz puede implementar otras interfaces, si en una clase implementamos una interfaz que a su vez implementa otras, esa clase tendrá definidas cada una de las interfaces, lo mismo ocurre con una clase que "se derive" de otra clase que implementa alguna interfaz, la nueva clase también incorporará esa interfaz.

> **Nota:**
> Cuando una interfaz implementa otras interfaces, éstas no se pueden indicar mediante Implements, en lugar de usar esa instrucción debemos usar Inherits.
>
> Public Interface IPrueba2
>   Inherits IMostrar

Si en una clase implementamos una interfaz que a su vez implementa otras interfaces, esa clase tendrá definiciones de todos los miembros de todas las interfaces, por ejemplo, si tenemos la siguiente definición de la interfaz **IPrueba2** que "implementa" la interfaz **IMostrar**:

```
Public Interface IPrueba2

    Inherits IMostrar



    Function Saludo(ByVal nombre As String) As String

    Property Nombre() As String

    Event DatosCambiados()

End Interface
```

Y la clase **Prueba2** implementa **IPrueba2**, la definición de los miembros quedaría de la siguiente forma:

```
Public Class Prueba2
```

```
    Implements IPrueba2


    Public Sub Mostrar() Implements IMostrar.Mostrar


    End Sub


    Public Event DatosCambiados() Implements
IPrueba2.DatosCambiados


    Public Property Nombre() As String Implements IPrueba2.Nombre

        Get


        End Get

        Set(ByVal value As String)


        End Set

    End Property


    Public Function Saludo(ByVal nombre As String) As String _

            Implements IPrueba2.Saludo


    End Function

End Class
```

En este código, el método **Mostrar** se indica mediante la interfaz **IMostrar**, pero también se puede hacer por medio de **IPrueba2.Mostrar**, ya que **IPrueba2** también lo implementa (o hereda).

Si dejamos que Visual Basic cree los miembros, no tendremos problemas a la hora de definirlos. Pero si lo hacemos manualmente, aunque dentro del IDE de Visual Basic, éste nos ayuda indicándonos que interfaces implementamos y qué miembros son los que se adecuan a la declaración que estamos usando, tal como podemos comprobar en la figura 2.16:

```
    Public Sub Mostrar() Implements IMostrar.
                                             Mostrar
    End Sub
```

Figura 2.16. IntelliSense solo muestra los métodos que mejor se adecuan a la declaración

Un ejemplo práctico usando una interfaz de .NET

Tal como comentamos al principio, el propio .NET está "plagado" de interfaces, cada una de ellas tiene un fin concreto, por ejemplo, si queremos definir una clase que pueda ser clasificada por el propio .NET, esa clase debe implementar la interfaz *IComparable*, ya que el método *Sort*, (de la clase que contiene los elementos del tipo definido por nosotros), que es el encargado de clasificar los elementos, hará una llamada al método *IComparable.CompareTo* de cada uno de los objetos que queremos clasificar, por tanto, si la clase no ha definido esa interfaz, no podremos clasificar los elementos que contenga.

En el siguiente código tenemos la definición de una clase llamada Empleado que implementa la interfaz *IComparable* y en el método *CompareTo* hace la comprobación de que objeto es mayor o menor, si el de la propia clase o el indicado en el parámetro de esa función:

```
Public Class Empleado

    Implements IComparable


    Public Nombre As String


    Public Sub New(ByVal nombre As String)

        Me.Nombre = nombre

    End Sub


    ' Si el objeto es del tipo Empleado, comparamos los nombres.

    ' Si no es del tipo Empleado, devolvemos un cero

    ' que significa que los dos objetos son iguales.

    Public Function CompareTo(ByVal obj As Object) As Integer _

        Implements System.IComparable.CompareTo

        If TypeOf obj Is Empleado Then

            Dim e1 As Empleado = CType(obj, Empleado)


            Return String.Compare(Me.Nombre, e1.Nombre)
```

```
        Else

            Return 0

        End If

    End Function

End Class
```

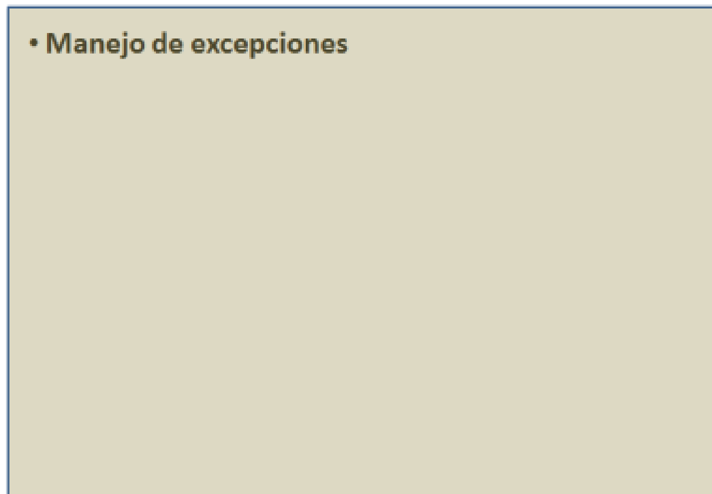
En el método *CompareTo* hacemos una comprobación de que el objeto con el que debemos realizar la comparación es del tipo **Empleado**, en ese caso convertimos el objeto pasado en uno del tipo **Empleado** y comparamos los nombres. Si el objeto que recibe el método no es del tipo **Empleado**, devolvemos un cero, para indicar que no haga ninguna clasificación, ya que ese valor indica que los dos objetos son iguales.

Esta comparación no es estrictamente necesaria, ya que si no indicamos el valor que debe devolver una función, devolverá un valor cero, al menos en este caso, ya que el tipo a devolver es un número entero.

Esta clase la podemos usar de esta forma:

```
' Una colección de datos del tipo Empleado.

Dim empleados As New System.Collections.Generic.List(Of Empleado)



' Añadimos varios empleados a la colección.

empleados.Add(New Empleado("Pepe"))

empleados.Add(New Empleado("Bernardo"))

empleados.Add(New Empleado("Juan"))

empleados.Add(New Empleado("Ana"))



' Clasificamos los empleados de la colección.

empleados.Sort()



' Mostramos los datos una vez clasificados.

For Each e1 As Empleado In empleados

    Console.WriteLine(e1.Nombre)

Next
```

⏩ Ver vídeo 1 de esta lección (Interfaces - Parte 1)- video en Visual Studio 2005 válido para Visual Studio 2008
⏩ Ver vídeo 2 de esta lección (Interfaces - Parte 2)- video en Visual Studio 2005 válido para Visual Studio 2008
⏩ Ver vídeo 3 de esta lección (Interfaces - Parte 3)- video en Visual Studio 2005 válido para Visual Studio 2008

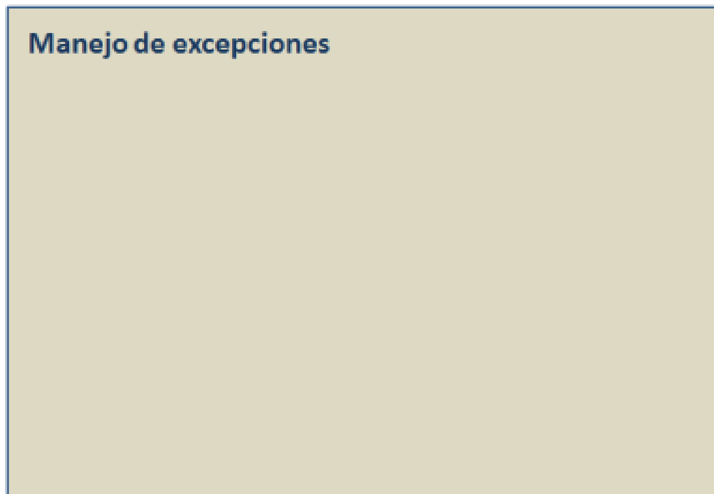


Introducción

Es indiscutible que por mucho que nos lo propongamos, nuestras aplicaciones no estarán libres de errores, y no nos referimos a errores sintácticos, ya que, afortunadamente, el IDE (*Integrated Development Envirnment*, entorno de desarrollo integrado) de Visual Basic 2008 nos avisará de cualquier error sintáctico e incluso de cualquier asignación no válida (al menos si tenemos activado *Option Strict On*), pero de lo que no nos avisará, como es lógico, será de los errores que se produzcan en tiempo de ejecución. Para estos casos, Visual Basic pone a nuestra disposición el manejo de excepcione, veamos pues cómo utilizarlo, sobre todo el sistema de excepciones estructuradas que es el recomendable para cualquier desarrollo con .NET Framework.

Manejo de excepciones

- **Manejo de excepciones**
  - Manejo de excepciones no estructuradas
  - Manejo de excepciones estructuradas
    - Bloque Try
    - Bloque Catch
    - Varias capturas de errores en un mismo bloque Try/Catch
    - Evaluación condicional en un bloque Catch
    - Bloque Finally
  - Captura de errores no controlados



Manejo de excepciones

En Visual Basic 2008 el tratamiento de errores (excepciones) podemos hacerlo de dos formas distintas, dependiendo de que utilicemos el tratamiento de errores heredado de versiones anteriores o el recomendado para la plataforma .NET: el control de errores estructurado, con idea de hacerlo de una forma más "ordenada".

En esta lección solamente veremos cómo tratar los errores de forma estructurada, porque, como hemos comentado es la forma recomendada de hacerlo en .NET Framework.

Manejo de excepciones no estructuradas

Como acabamos de comentar, no trataremos o explicaremos cómo trabajar con el tratamiento de errores no estructurados, pero al menos queremos hacer una aclaración para que no nos llevemos una sorpresa si nos decidimos a usarlo:

No podemos usar los dos sistemas de tratamiento de errores al mismo tiempo, por lo menos en un mismo método o propiedad, o utilizamos *On Error* o utilizamos *Try/Catch*. De todas formas, si escribimos nuestro código con el IDE (entorno integrado) de Visual Studio 2008, éste nos avisará de que no podemos hacer esa mezcla.

Manejo de excepciones estructuradas

Las excepciones en Visual Basic 2008 las podemos controlar usando las instrucciones *Try / Catch / Finally*. Estas instrucciones realmente son bloques de instrucciones, al estilo de *If / Else*.
Cuando queramos controlar una parte del código que puede producir un error lo incluimos dentro del bloque *Try*, si se produce un error, éste lo podemos detectar en el bloque *Catch*, por último, independientemente de que se produzca o no una excepción, podemos ejecutar el código que incluyamos en el bloque *Finally*, para indicar el final del bloque de control de excepciones lo haremos con *End Try*.

Cuando creamos una estructura de control de excepciones no estamos obligados a usar los tres bloques, aunque el primero: *Try* si es necesario, ya que es el que le indica al compilador que tenemos intención de controlar los errores que se produzcan. Por tanto podemos crear un "manejador" de excepciones usando los tres bloques, usando *Try* y *Catch* o usando *Try* y *Finally*.

Veamos ahora con más detalle cada uno de estos bloques y que es lo que podemos hacer en cada uno de ellos.

Bloque Try

En este bloque incluiremos el código en el que queremos comprobar los errores.
El código a usar será un código normal, es decir, no tenemos que hacer nada en especial, ya que en el momento que se produzca el error se usará (si hay) el código del bloque *Catch*.

Bloque Catch

Si se produce una excepción, ésta la capturamos en un bloque *Catch*.

En el bloque *Catch* podemos indicar que tipo de excepción queremos capturar, para ello usaremos una variable de tipo *Exception*, la cual pude ser del tipo de error específico que queremos controlar o de un tipo genérico.
Por ejemplo, si sabemos que nuestro código puede producir un error al trabajar con ficheros, podemos usar un código como este:

```
Try

    ' código para trabajar con ficheros, etc.

Catch ex As System.IO.IOException

    ' el código a ejecutar cuando se produzca ese error

End Try
```

Si nuestra intención es capturar todos los errores que se produzcan, es decir, no queremos hacer un filtro con errores específicos, podemos usar la clase *Exception* como tipo de excepción a capturar. La clase *Exception* es la más genérica de todas las clases para manejo de excepciones, por tanto capturará todas las excepciones que se produzcan.

```
Try
```

```
    ' código que queremos controlar
Catch ex As Exception
    ' el código a ejecutar cuando se produzca cualquier error
End Try
```

Aunque si no vamos usar la variable indicada en el bloque *Catch*, pero queremos que no se detenga la aplicación cuando se produzca un error, podemos hacerlo de esta forma:

```
Try
    ' código que queremos controlar
Catch
    ' el código a ejecutar cuando se produzca cualquier error
End Try
```

Varias capturas de errores en un mismo bloque Try/Catch

En un mismo *Try* podemos capturar diferentes tipos de errores, para ello podemos incluir varios bloques *Catch*, cada uno de ellos con un tipo de excepción diferente.

Es importante tener en cuenta que cuando se produce un error y usamos varios bloques *Catch*, Visual Basic buscará la captura que mejor se adapte al error que se ha producido, pero siempre lo hará examinando los diferentes bloques *Catch* que hayamos indicado empezando por el indicado después del bloque *Try*, por tanto deberíamos poner las más genéricas al final, de forma que siempre nos aseguremos de que las capturas de errores más específicas se intercepten antes que las genéricas.

Evaluación condicional en un bloque Catch

Además de indicar la excepción que queremos controlar, en un bloque *Catch* podemos añadir la cláusula *When* para evaluar una expresión. Si la evaluación de la expresión indicada después de *When* devuelve un valor verdadero, se procesará el bloque *Catch*, en caso de que devuelva un valor falso, se ignorará esa captura de error.

Esto nos permite poder indicar varios bloques *Catch* que detecten el mismo error, pero cada una de ellas pueden tener diferentes expresiones indicadas con *When*.

En el siguiente ejemplo, se evalúa el bloque *Catch* solo cuando el valor de la variable y es cero, en otro caso se utilizará el que no tiene la cláusula *When*:

```
Dim x, y, r As Integer

Try

    x = CInt(Console.ReadLine())

    y = CInt(Console.ReadLine())

    r  = x \ y

    Console.WriteLine("El resultado es: {0}", r)

Catch ex As Exception When y = 0

    Console.WriteLine("No se puede dividir por cero.")

Catch ex As Exception

    Console.WriteLine(ex.Message)

End Try
```

Bloque Finally

En este bloque podemos indicar las instrucciones que queremos que se ejecuten, se produzca o no una excepción. De esta forma nos aseguramos de que siempre se ejecutará un código, por ejemplo para liberar recursos, se haya producido un error o no.

> **Nota:**
> Hay que tener en cuenta de que incluso si usamos Exit Try para salir del bloque de control de errores, se ejecutará el código indicado en el bloque Finally.

Captura de errores no controlados

Como es lógico, si no controlamos las excepciones que se puedan producir en nuestras aplicaciones, estas serán inicialmente controladas por el propio runtime de .NET, en estos casos la aplicación se detiene y se muestra el error al usuario. Pero esto es algo que no deberíamos consentir, por tanto siempre deberíamos detectar todos los errores que se produzcan en nuestras aplicaciones, pero a pesar de que lo intentemos, es muy probable que no siempre podamos conseguirlo. Por suerte, en Visual Basic 2008 tenemos dos formas de interceptar los errores no controlados:
La primera es iniciando nuestra aplicación dentro de un bloque *Try/Catch*, de esta forma, cuando se produzca el error, se capturará en el bloque *Catch*.
La segunda forma de interceptar los errores no controlados es mediante el evento: *UnhandledException*, disponible por medio del objeto *My.Application*.

Nota:
De los eventos nos ocuparemos en la siguiente lección, pero como el evento UnhandledException está directamente relacionado con la captura de errores, lo mostramos en esta, aunque recomendamos al lector que esta sección la vuelva a leer después de ver todo lo relacionado con los eventos.

Este evento se "dispara" cuando se produce un error que no hemos interceptado, por tanto podríamos usarlo para prevenir que nuestra aplicación se detenga o bien para guardar en un fichero **.log** la causa de dicho error para posteriormente actualizar el código y prevenirlo. Ya que cuando se produce el evento *UnhandledException*, podemos averiguar el error que se ha producido e incluso evitar que la aplicación finalice. Esa información la obtenemos mediante propiedades expuestas por el segundo parámetro del evento, en particular la propiedad *Exception* nos indicará el error que se ha producido y por medio de la propiedad *ExitApplication* podemos indicar si terminamos o no la aplicación.

Nota:
Cuando ejecutamos una aplicación desde el IDE, los errores no controlados siempre se producen, independientemente de que tengamos o no definida la captura de errores desde el evento UnhandledException. Ese evento solo se producirá cuando ejecutemos la aplicación fuera del IDE de Visual Basic.

Ver vídeo 1 de esta lección (Excepciones - Parte 1) - video en Visual Studio 2005 válido para Visual Studio 2008
Ver vídeo 2 de esta lección (Excepciones - Parte 2) - video en Visual Studio 2005 válido para Visual Studio 2008
Ver vídeo 3 de esta lección (Excepciones - Parte 3) - video en Visual Studio 2005 válido para Visual Studio 2008

Lección 4: Eventos y delegados

- Eventos
- Definir y producir eventos en una clase
- Delegados
- Definir un evento bien informado

Introducción

La forma que tienen nuestras clases y estructuras de comunicar que algo está ocurriendo, es por medio de eventos. Los eventos son mensajes que se lanzan desde una clase para informar al "cliente" que los utiliza de que está pasando algo.

Seguramente estaremos acostumbrados a usarlos, incluso sin tener una noción consciente de que se tratan de eventos, o bien porque es algo tan habitual que no le prestamos mayor atención, es el caso de las aplicaciones de escritorio, cada vez que presionamos un botón, escribimos algo o movemos el mouse se están produciendo eventos.

El compilador de Visual Basic 2008 nos facilita mucho la creación de los eventos y "esconde" todo el proceso que .NET realmente hace "en la sombra". Ese trabajo al que nos referimos está relacionado con los delegados, una palabra que suele aparecer en cualquier documentación que trate sobre los eventos.

Y es que, aunque Visual Basic 2008 nos oculte, o facilite, el trabajo con los eventos, éstos están estrechamente relacionados con los delegados. En esta lección veremos que son los delegados y que relación tienen con los eventos, también veremos que podemos tener mayor control sobre cómo se interceptan los eventos e incluso cómo y cuando se asocian los eventos en la aplicación cliente, aunque primero empezaremos viendo cómo declarar y utilizar eventos en nuestros tipos de datos.

Eventos y delegados

- **Eventos**
  - Interceptar los eventos de los controles de un formulario
    - Interceptar eventos en Visual Basic 2008
  - Asociar un evento con un control
  - Formas de asociar los eventos con un control
    - 1- Asociar el evento manualmente por medio de Handles
    - 2- Asociar el evento desde la ventana de código
    - 3- Asociar el evento desde el diseñador de formularios
  - Asociar varios eventos a un mismo procedimiento
  - Declarar una variable para asociar eventos con Handles

- **Definir y producir eventos en una clase**
  - Definir eventos en una clase
  - Producir un evento en nuestra clase
  - Otra forma de asociar los eventos de una clase con un método
    - Asociar eventos mediante AddHandler
    - Desasociar eventos mediante RemoveHandler

- **Delegados**
  - ¿Qué ocurre cuando se asigna y se produce un evento?
  - ¿Qué papel juegan los delegados en todo este proceso?
  - Definición "formal" de delegado
  - Utilizar un delegado para acceder a un método

- **Definir un evento bien informado con Custom Event**

Lección 4: Eventos y delegados

Eventos

- Definir y producir eventos en una clase
- Delegados
- Definir un evento bien informado

Eventos

Como hemos comentado en la introducción, en Visual Basic 2008 podemos usar los eventos de una forma bastante sencilla, al menos si la comparamos con otros lenguajes de la familia .NET.

En las secciones que siguen veremos cómo utilizar los eventos que producen las clases de .NET, empezaremos con ver cómo utilizar los eventos de los formularios, que con toda seguridad serán los que estamos más acostumbrados a usar. También veremos las distintas formas que tenemos de asociar un método con el evento producido por un control (que al fin y al cabo es una clase).

Interceptar los eventos de los controles de un formulario

Debido a que aún no hemos visto el módulo dedicado a las aplicaciones de Windows, en las que se utilizan los "clásicos" formularios, no vamos a entrar en detalles sobre cómo crear un formulario ni como añadir controles, etc., todo eso lo veremos en el siguiente módulo. Para simplificar las cosas, veremos cómo se interceptan en un formulario de la nueva versión de Visual Basic, aunque sin entrar en demasiados detalles.

La forma más sencilla de asociar el evento de un control con el código que se usará, es haciendo doble pulsación en el control cuando estamos en modo de diseño; por ejemplo, si en nuestro formulario tenemos un botón, al hacer doble pulsación sobre él tendremos asociado el evento *Click* del botón, ya que ese es el evento predeterminado de los controles *Button*, y desde el diseñador de formularios de Windows, al realizar esa doble pulsación, siempre se muestra el evento predeterminado del control en cuestión.

Interceptar eventos en Visual Basic 2008

En Visual Basic 2008 aunque la forma de asignar los eventos predeterminados de los controles es como hemos comentado anteriormente, es decir, haciendo doble pulsación en el control, la declaración del código usado para interceptar el evento es como el mostrado en el siguiente código:

```
Private Sub Button1_Click(ByVal sender As Object, ByVal e As
EventArgs) _

        Handles Button1.Click



End Sub
```

Lo primero que podemos notar es que en Visual Basic 2008 utiliza dos argumentos, esto siempre es así en todos los eventos producidos por los controles. El primero indica el control que produce el evento, (en nuestro ejemplo sería una referencia al control **Button1**), y el segundo normalmente contiene información sobre el evento que se produce, si el evento en cuestión no proporciona información extra, como es el caso del evento *Click*, ese parámetro será del tipo *EventArgs*. Sin embargo en otros eventos, por ejemplo, los relacionados con el mouse, el segundo argumento tendrá información que nos puede resultar útil, por ejemplo para saber que botón se ha usado o cual es la posición del cursor, en la figura 2.17 podemos ver las propiedades relacionadas con el evento *MouseEventArgs*:

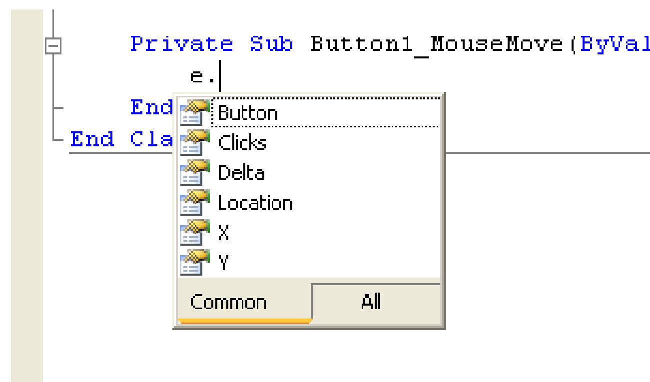


Figura 2.17. Propiedades relacionadas con un evento del mouse

Asociar un evento con un control

Siguiendo con el código que intercepta el evento *Click* de un botón, podemos apreciar que el IDE de Visual Basic 2008 añade al final de la declaración del procedimiento de evento la instrucción *Handles* seguida del control y el evento que queremos interceptar: **Handles Button1.Click**, esta la forma habitual de hacerlo en Visual Basic 2008, aunque también hay otras formas de "ligar" un método con un evento, como tendremos ocasión de comprobar más adelante.

> **Nota:**
> En Visual Basic 2008 el nombre del procedimiento de evento no tiene porqué estar relacionado con el evento, aunque por defecto, el nombre usado es el que habitualmente se ha utilizado por años en otros entornos de desarrollo, y que se forma usando el nombre del control seguido de un guión bajo y el nombre del evento, pero que en cualquier momento lo podemos cambiar, ya que en Visual Basic 2008 no hay ninguna relación directa entre ese nombre y el evento que queremos interceptar.

Tal como resaltamos en la nota anterior, en Visual Basic 2008, el nombre asociado a un evento puede ser el que queramos, lo realmente importante es que indiquemos la instrucción *Handles* seguida del evento que queremos interceptar. Aunque, como veremos a continuación, también hay otras formas de "relacionar" los eventos con el método usado para recibir la notificación.

Formas de asociar los eventos con un control

Cuando estamos trabajando con el diseñador de formularios, tenemos tres formas de asociar un evento con el código correspondiente:

La forma más sencilla es la expuesta anteriormente, es decir, haciendo doble click en el control, esto hará que se muestre el evento predeterminado del control. En el caso del control *Button*, el evento predeterminado es el evento *Click*.

Si queremos escribir código para otros eventos podemos hacerlo de tres formas, aunque la primera que explicaremos no será la más habitual, ya que debemos saber exactamente qué parámetros utiliza el evento.

1- Asociar el evento manualmente por medio de Handles

Con esta forma, simplemente escribimos el nombre del procedimiento (puede ser cualquier nombre), indicamos los dos argumentos que recibe: el primero siempre es de tipo *Object* y el segundo dependerá del tipo de evento, y finalmente por medio de la cláusula *Handles* lo asociamos con el control y el evento en cuestión.

2- Asociar el evento desde la ventana de código

En Visual Basic 2008 también podemos seleccionar los eventos disponibles de una lista desplegable. Esto lo haremos desde la ventana de código. En la parte superior derecha tenemos una la lista con los controles que hemos añadido al formulario, seleccionamos el control que nos interese y en la lista que hay a su izquierda tenemos los eventos que ese control produce. Por tanto podemos seleccionar de esa lista de eventos el que nos interese interceptar, tal como podemos ver en la figura 2.18

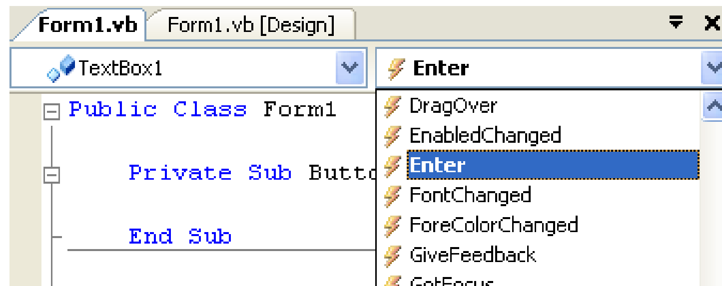


Figura 2.18. Lista de eventos de un control

De esta forma el propio IDE será el que cree el "esqueleto" del procedimiento de evento usando los parámetros adecuados.

3- Asociar el evento desde el diseñador de formularios

La tercera forma de asociar un evento con un control, es hacerlo desde el diseñador de formularios. En la ventana de propiedades del control, (tal como podemos apreciar en la figura 2.19), tenemos una lista con los eventos más importantes de cada control, seleccionando el evento de esa lista y haciendo doble-click en el que nos interese, conseguiremos exactamente el mismo resultado que con el paso anterior.

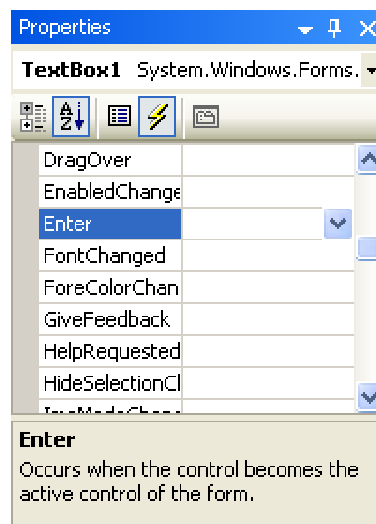


Figura 2.19. Ventana de propiedades con los eventos del control seleccionado

Asociar varios eventos a un mismo procedimiento

Cuando utilizamos *Handles* para asociar eventos, podemos indicar que un mismo procedimiento sirva para interceptar varios eventos. Veamos un caso práctico en el que tenemos varios controles de tipo *TextBox* y queremos que cuando reciban el foco siempre se ejecute el mismo código, por ejemplo, que se seleccione todo el texto que contiene, podemos hacerlo indicando después de la cláusula *Handles* todos los controles que queremos asociar con ese procedimiento. En el siguiente código indicamos tres controles *TextBox*:

```
Private Sub TextBox1_Enter( _

                ByVal sender As System.Object, _

                ByVal e As System.EventArgs) _

                Handles      TextBox1.Enter,      TextBox2.Enter,
TextBox3.Enter



End Sub
```

Esta asociación la podemos hacer manualmente, simplemente indicando en la cláusula *Handles* cada uno de los eventos a continuación del anterior separándolos por comas. O bien desde el diseñador de formularios. En este segundo caso, cuando seleccionamos un control y desde la ventana de propiedades, seleccionamos un evento, nos muestra los procedimientos que tenemos definidos en nuestro código que utilizan los mismos parámetros que el evento en cuestión, si seleccionamos uno de esos procedimientos, el propio IDE añadirá ese control a la lista *Handles*.

Como vemos en la figura 2.20 podemos usar el procedimiento T**extBox1_Enter** para asociarlo al evento *Enter* del control **TextBox2**:

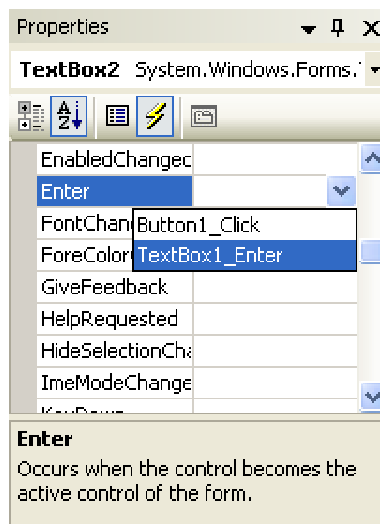


Figura 2.20. Lista de procedimientos "compatibles" con un evento

Declarar una variable para asociar eventos con Handles

Para que podamos usar la instrucción *Handles* para asociar manualmente un procedimiento con un evento, o para que el diseñador de Visual Basic 2008 pueda hacerlo, la variable del control o clase que tiene los eventos que queremos interceptar tenemos que declararla con la instrucción *WithEvents*.

De estos detalles se encarga el propio IDE de Visual Basic 2008, por tanto no debemos preocuparnos de cómo declarar los controles para que se pueda usar *Handles* en el método del procedimiento que recibe la notificación del evento, pero como los controles de .NET realmente son clases, veamos cómo declara el VB los

controles, (en este caso un control llamado **Button1**), para a continuación compararlo con una clase definida por nosotros.

```
Friend WithEvents Button1 As System.Windows.Forms.Button
```

Si en lugar de estar trabajando con formularios y controles, lo hacemos con clases "normales", la forma de declarar una variable que tiene eventos es por medio de la instrucción *WithEvents*. Por ejemplo, en esta declaración indicamos que tenemos intención de usar los eventos que la clase **Empleado** exponga:

```
Private WithEvents unEmpleado As Empleado
```

Y posteriormente podremos definir los métodos de eventos usando la instrucción *Handles*:

```
Private Sub unEmpleado_DatosCambiados() Handles
unEmpleado.DatosCambiados



End Sub
```

Nota:
Usar WithEvents y Handles es la forma más sencilla de declarar y usar una variable que accede a una clase que produce eventos, pero como ya hemos comentado, no es la única forma que tenemos de hacerlo en Visual Basic 2008, tal como tendremos oportunidad de comprobar.

Ver vídeo de esta lección (Eventos - Parte 1) - video en Visual Studio 2005 válido para Visual Studio 2008

Lección 4: Eventos y delegados

- Eventos

Definir y producir eventos en una clase

- Delegados
- Definir un evento bien informado

Definir y producir eventos en una clase

En la lección anterior hemos visto cómo interceptar eventos, en esta veremos cómo definirlos en nuestras clases y cómo producirlos, para que el cliente que los intercepte sepa que algo ha ocurrido en nuestra clase.

Definir eventos en una clase

Para definir un evento en una clase usamos la instrucción *Event* seguida del nombre del evento y opcionalmente indicamos los parámetros que dicho evento recibirá.
En el siguiente trozo de código definimos un evento llamado **DatosModificados** que no utiliza ningún argumento:

```
Public Event DatosModificados()
```

Esto es todo lo que necesitamos hacer en Visual Basic 2008 para definir un evento.

Como podemos comprobar es muy sencillo, ya que solo tenemos que usar la instrucción *Event*. Aunque *detrás del telón* ocurren otras cosas de las que, al menos en principio, no debemos preocuparnos, ya que es el propio compilador de Visual Basic 2008 el que se encarga de esos "pequeños detalles".

Producir un evento en nuestra clase

Para producir un evento en nuestra clase, y de esta forma notificar a quién quiera interceptarlo, simplemente usaremos la instrucción *RaiseEvent* seguida del evento que queremos producir. Cuando escribimos esa instrucción en el IDE de Visual Basic 2008, nos mostrará los distintos eventos que podemos producir, tal como vemos en la figura 2.21:

Figura 2.21. Lista de eventos que podemos producir

Esta es la forma más sencilla de definir y lanzar eventos en Visual Basic 2008.

Otra forma de asociar los eventos de una clase con un método

Tal como hemos estado comentando, la forma más sencilla de declarar una variable para interceptar eventos es declarándola usando *WithEvents* y para interceptar los eventos lo hacemos por medio de la instrucción *Handles*. Esta forma, es a todas luces la más recomendada, no solo por la facilidad de hacerlo, sino porque también tenemos la ventaja de que todas las variables declaradas con *WithEvents* se muestran en la lista desplegable de la ventana de código, tal como podemos apreciar en la figura 2.22:

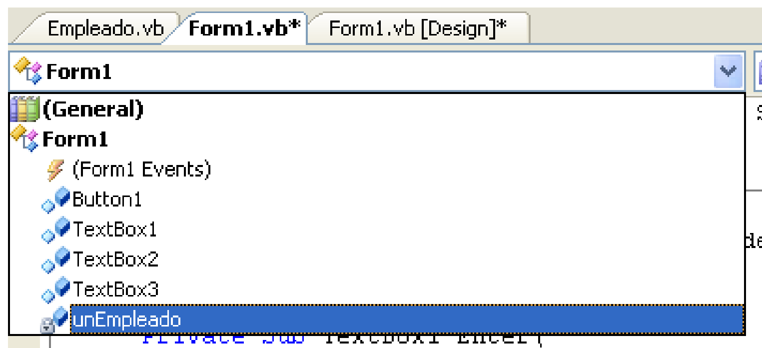


Figura 2.22. Lista de objetos que producen eventos

Y de esta forma podemos seleccionar la variable y posteriormente elegir el evento a interceptar, tal como vimos en la figura 2.18.

Asociar eventos mediante AddHandler

Pero Visual Basic 2008 también proporciona otra forma de asociar un procedimiento con un evento. Aunque en este caso es algo más manual que todo lo que hemos visto y, de alguna forma está más ligado con los delegados, y como los delegados los veremos dentro de poco, ahora solamente mostraremos la forma de hacerlo y después veremos con algo de más detalle cómo funciona.

La forma de de asociar eventos con su correspondiente método es por medio de la instrucción *AddHandler*. A esta instrucción le pasamos dos argumentos, el primero es el evento a asociar y el segundo es el procedimiento que usaremos cuando se produzca dicho evento. Este último parámetro tendremos que indicarlo mediante la instrucción *AddressOf*, que sirve para pasar una referencia a una función o procedimiento, y precisamente eso es lo que queremos hacer: indicarle que procedimiento debe usar cuando se produzca el evento:

```
AddHandler unEmpleado.DatosCambiados, AddressOf
unEmpleado_DatosCambiados
```

En este caso, el uso de *AddressOf* es una forma "fácil" que tiene Visual Basic 2008 de asociar un procedimiento de evento con el evento. Aunque por el fondo, (y sin que nos enteremos), realmente lo que estamos usando es un constructor a un delegado.

La ventaja de usar esta forma de asociar eventos con el procedimiento, es que podemos hacerlo con variables que no están declaradas con *WithEvents*, realmente esta sería la única forma de asociar un procedimiento de evento con una variable que no hemos declarado con *WithEvents*.

Desasociar eventos mediante RemoveHandler

De la misma forma que por medio de *AddHandler* podemos asociar un procedimiento con un evento, usando la instrucción *RemoveHandler* podemos hacer el proceso contrario: desligar un procedimiento del evento al que previamente estaba asociado.
Los parámetros a usar con *RemoveHandler* son los mismos que con *AddHandler*.

Podemos usar *RemoveHandler* tanto con variables y eventos definidos con *AddHandler* como con variables declaradas con *WithEvents* y ligadas por medio de *Handles*.
Esto último es así porque cuando nosotros definimos los procedimientos de eventos usando la instrucción *Handles*, es el propio Visual Basic el que internamente utiliza *AddHandler* para ligar ese procedimiento con el evento en cuestión. Saber esto nos facilitará comprender mejor cómo funciona la declaración de eventos mediante la instrucción *Custom*, aunque de este detalle nos ocuparemos después de ver que son los delegados.

Ver vídeo de esta lección (Eventos - Parte 2) - video en Visual Studio 2005 válido para Visual Studio 2008
Ver vídeo de esta lección (Eventos - Parte 3) - video en Visual Studio 2005 válido para Visual Studio 2008

Lección 4: Eventos y delegados

- Eventos
- Definir y producir eventos en una clase

Delegados

- Definir un evento bien informado

Delegados

Como hemos comentado anteriormente, los eventos son acciones que una clase puede producir cuando ocurre algo. De esta forma podemos notificar a las aplicaciones que hayan decidido interceptar esos mensajes para que tomen las acciones que crean conveniente.

Visual Basic 2008 esconde al desarrollador prácticamente todo lo que ocurre cada vez que definimos, lanzamos o interceptamos un evento, nosotros solo vemos una pequeña parte de todo el trabajo que en realidad se produce, y el que no lo veamos no quiere decir que no esté ocurriendo. También es cierto que no debe preocuparnos demasiado si no sabemos lo que está pasando, pero si somos conscientes de que es lo que ocurre, puede que nos ayude a comprender mejor todo lo relacionado con los eventos.

¿Qué ocurre cuando se asigna y se produce un evento?

Intentemos ver de forma sencilla lo que ocurre "por dentro" cada vez que definimos un método que intercepta un evento y cómo hace el Visual Basic para comunicarse con el receptor de dicho evento.

1. Cuando Visual Basic se encuentra con el código que le indica que un método debe interceptar un evento, ya sea mediante *AddHandler* o mediante el uso de *Handles*, lo que hace es añadir la dirección de memoria de ese método a una especie de array.
   En la figura 2.23 podemos ver un diagrama en el que un mismo evento lo interceptan tres clientes, cuando decimos que un cliente intercepta un evento, realmente nos referimos a que hay un método que lo intercepta y el evento realmente guarda la dirección de memoria de ese método.

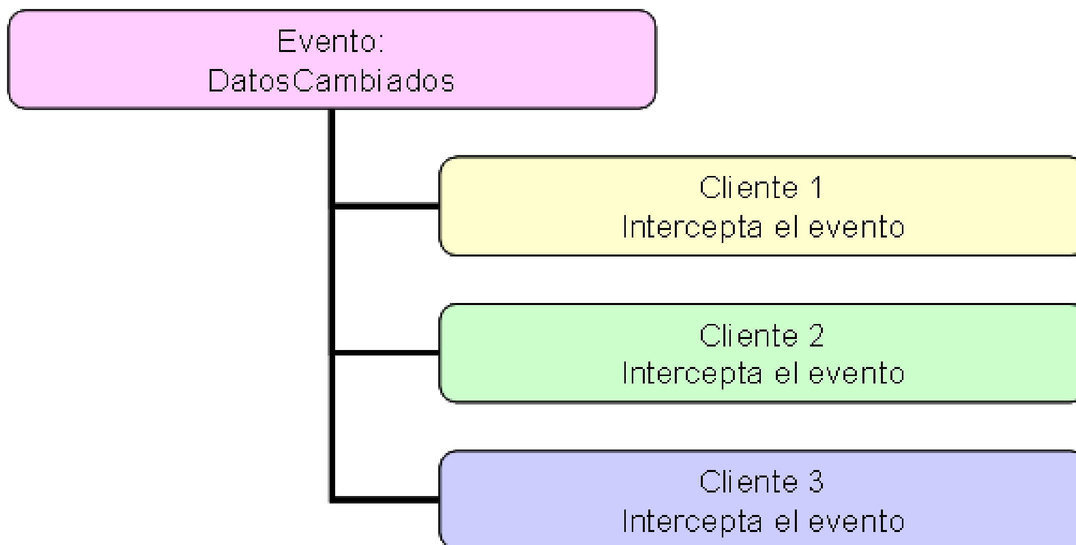


Figura 2.23. El evento guarda la dirección de memoria de cada método que lo intercepta

2. Cuando usamos la instrucción *RaiseEvent* para producir el evento, se examina esa lista de direcciones y se manda el mensaje a cada uno de los métodos que tenemos en el "array". En este caso, lo que realmente ocurre es que se hace una llamada a cada uno de los métodos, de forma que se ejecute el código al que tenemos acceso mediante la dirección de memoria almacenada en la lista.
3. Cuando usamos la instrucción *RemoveHandler*, le estamos indicando al evento que elimine de la lista el método indicado en esa instrucción, de esta forma, la próxima vez que se produzca el evento, solo se llamará a los métodos que actualmente estén en la lista.

Tanto el agregar nuevos métodos a esa lista como quitarlos, lo podemos hacer en tiempo de ejecución, por medio de *AddHandler* y *RemoveHandler* respectivamente. Ya que la instrucción *Handles* solo la podemos usar en tiempo de diseño.

Es más, podemos incluso indicar que un mismo evento procese más de un método en una misma aplicación o que un mismo método sea llamado por más de un evento. Ya que lo que realmente necesita cada evento es que exista un método que tenga una "firma" concreta: la indicada al declarar el evento.

¿Qué papel juegan los delegados en todo este proceso?

Veamos primero que papel tienen los delegados en todo este proceso y después veremos con más detalle lo que "realmente" es un delegado.

1. Cuando definimos un evento, realmente estamos definiendo un delegado, (que en el fondo es una clase con un tratamiento especial), y un método del mismo tipo que el delegado.
2. Cuando indicamos que un método intercepte un evento, realmente estamos llamando al constructor del delegado, al que le pasamos la dirección de memoria del método. El delegado almacena cada una de esas direcciones de memoria para posteriormente usarlas.

3. Cuando se produce el evento, (por medio de *RaiseEvent*), realmente estamos llamando al delegado para que acceda a todas las "direcciones" de memoria que tiene almacenadas y ejecute el código que hayamos definido en cada uno de esos métodos.

Como podemos comprobar, y para decirlo de forma simple, un delegado realmente es la forma que tiene .NET para definir un puntero. La diferencia principal es que los punteros, (no vamos a entrar en demasiados detalles sobre los punteros, ya que no estamos en un curso de C/C++), no tienen forma de comprobar si están accediendo a una dirección de memoria correcta o, para decirlo de otra forma, a una dirección de memoria "adecuada". En .NET, los "punteros" solo se pueden usar mediante delegados, y éstos solamente pueden acceder a direcciones de memoria que tienen la misma "firma" con el que se han definido. Para que lo entendamos un poco mejor, es como si los delegados solo pudieran acceder a sitios en la memoria que contienen un método con la misma "interfaz" que el que ha definido el propio delegado.

Seguramente es difícil de entender, y la principal razón es que hemos empezado la casa por el techo.
Por tanto, veamos a continuación una definición "formal" de qué es un delegado y veamos varios ejemplos para que lo comprendamos mejor.

Definición "formal" de delegado

Veamos que nos dice la documentación de Visual Basic 2008 sobre los delegados:

> *"Un delegado es una clase que puede contener una referencia a un método. A diferencia de otras clases, los delegados tienen un prototipo (firma) y pueden guardar referencias únicamente a los métodos que coinciden con su prototipo."*

Esta definición, al menos en lo que respecta a su relación con los eventos, viene a decir que los delegados definen la forma en que debemos declarar los métodos que queramos usar para interceptar un evento.

Si seguimos buscando más información sobre los delegados en la documentación de Visual Basic 2008, también nos encontramos con esta definición:

> *"Los delegados habilitan escenarios que en otros lenguajes se han resuelto con punteros a función. No obstante, a diferencia de los punteros a función, los delegados están orientados a objetos y proporcionan seguridad de tipos."*

Que es lo que comentamos en la sección anterior: los delegados nos facilitan el acceso a "punteros" (o direcciones de memoria) de funciones, pero hecho de una forma "controlada", en este caso por el propio .NET framework.

Por ejemplo, el evento **DatosCambiados** definido anteriormente, también lo podemos definir de la siguiente forma:

```
Public Delegate Sub DatosCambiadosEventHandler()



Public Event DatosCambiados As DatosCambiadosEventHandler
```

Es decir, el método que intercepte este evento debe ser del tipo *Sub* y no recibir ningún parámetro.

Si nuestro evento utiliza, por ejemplo, un parámetro de tipo *String*, la definición del delegado quedaría de la siguiente forma:

```
Public Delegate Sub NombreCambiadoEventHandler(ByVal nuevoNombre
As String)
```

Y la definición del evento quedaría de esta otra:

```
Public Event NombreCambiado As NombreCambiadoEventHandler
```

Como vemos al definir el evento ya no tenemos que indicar si recibe o no algún parámetro, ya que esa definición la hemos hecho en el delegado.

Si nos decidimos a definir este evento de la forma "normal" de Visual Basic, lo haríamos así:

```
Public Event NombreCambiado(ByVal nuevoNombre As String)
```

Como podemos comprobar, Visual Basic 2008 nos permite definir los eventos de dos formas distintas: definiendo un delegado y un evento que sea del tipo de ese delegado o definiendo el evento con los argumentos que debemos usar.

Declaremos como declaremos los eventos, los podemos seguir usando de la misma forma, tanto para producirlo mediante *RaiseEvent* como para definir el método que reciba ese evento.

Utilizar un delegado para acceder a un método

Ahora veamos brevemente cómo usar los delegados, en este caso sin necesidad de que defina un evento.

Como hemos comentado, un delegado realmente es una clase que puede contener una referencia a un método, además define el prototipo del método que podemos usar como referencia. Sabiendo esto, podemos declarar una variable del tipo del delegado y por medio de esa variable acceder al método que indiquemos, siempre que ese método tenga la misma "firma" que el delegado. Parece complicado ¿verdad? Y no solo lo parece, es que realmente lo es. Comprobemos esta "complicación" por medio de un ejemplo. En este código, que iremos mostrando poco a poco, vamos a definir un delegado, un método con la misma firma para que podamos usarlo desde una variable definida con el mismo tipo del delegado.

Definimos un delegado de tipo *Sub* que recibe un valor de tipo cadena:

```
Delegate Sub Saludo(ByVal nombre As String)
```

Definimos un método con la misma firma del delegado:

```
Private Sub mostrarSaludo(ByVal elNombre As String)
```

```
    Console.WriteLine("Hola, " & elNombre)

End Sub
```

Ahora vamos a declarar una variable para que acceda a ese método. Para ello debemos declararla con el mismo tipo del delegado:

```
Dim saludando As Saludo
```

La variable saludando es del mismo tipo que el delegado **Saludo**. La cuestión es ¿cómo o que asignamos a esta variable?

Primer intento:
Como hemos comentado, los delegados realmente son clases, por tanto podemos usar **New Saludo** y, según parece, deberíamos pasarle un nombre como argumento. Algo así:

```
saludando = New Saludo("Pepe")
```

Pero esto no funciona, entre otras cosas, porque hemos comentado que un delegado contiene (o puede contener) una referencia a un método, y **"Pepe"** no es un método ni una referencia a un método.

Segundo intento:
Por lógica y, sobre todo, por sentido común, máxime cuando hemos declarado un método con la misma "firma" que el delegado, deberíamos pensar que lo que debemos pasar a esa variable es el método, ya que un delegado puede contener una referencia a un método.

```
saludando = New Saludo(mostrarSaludo)
```

Esto tampoco funciona, ¿seguramente porque le falta el parámetro?

```
saludando = New Saludo(mostrarSaludo("Pepe"))
```

Pues tampoco.

Para usar un delegado debemos indicarle la dirección de memoria de un método, a eso se refiere la definición que vimos antes, una referencia a un método no es ni más ni menos que la dirección de memoria de ese método. Y en Visual Basic, desde la versión 5.0, tenemos una instrucción para obtener la dirección de memoria de cualquier método: *AddressOf*.
Por tanto, para indicarle al delegado dónde está ese método tendremos que usar cualquiera de estas dos formas:

```
saludando = New Saludo(AddressOf mostrarSaludo)
```

Es decir, le pasamos al constructor la dirección de memoria del método que queremos "asociar" al delegado.

En Visual Basic esa misma asignación la podemos simplificar de esta forma:

```
saludando = AddressOf mostrarSaludo
```

Ya que el compilador "sabe" que saludando es una variable de tipo delegado y lo que esa variable puede contener es una referencia a un método que tenga la misma firma que la definición del delegado, en nuestro caso, que sea de tipo *Sub* y reciba una cadena.

Si queremos, también podemos declarar la variable y asignarle directamente el método al que hará referencia:

```
Dim saludando As New Saludo(AddressOf mostrarSaludo)
```

Y ahora... ¿cómo podemos usar esa variable?

La variable **saludando** realmente está apuntando a un método y ese método recibe un valor de tipo cadena, por tanto si queremos llamar a ese método (para que se ejecute el código que contiene), tendremos que indicarle el valor del argumento, sabiendo esto, la llamada podría ser de esta forma:

```
saludando("Pepe")
```

Y efectivamente, así se mostraría por la consola el saludo (Hola) y el valor indicado como argumento.

Realmente lo que hacemos con esa llamada es acceder al método al que apunta la variable y como ese método recibe un parámetro, debemos pasárselo, en cuanto lo hacemos, el runtime de .NET se encarga de localizar el método y pasarle el argumento, de forma que se ejecute de la misma forma que si lo llamásemos directamente:

```
mostrarSaludo("Pepe")
```

Con la diferencia de que la variable "**saludando**" no tiene porqué saber a qué método está llamando, y lo más importante, no sabe dónde está definido ese método, solo sabe que el método recibe un parámetro de tipo cadena y aparte de esa información, no tiene porqué saber nada más.

Así es como funcionan los eventos, un evento solo tiene la dirección de memoria de un método, ese método recibe los mismos parámetros que los definidos por el evento (realmente por el delegado), cuando producimos el evento con *RaiseEvent* es como si llamáramos a cada uno de los métodos que se han ido agregando al delegado, si es que se ha agregado alguno, ya que en caso de que no haya ningún método asociado a ese evento, éste no se producirá, por la sencilla razón de que no habrá ningún código al que llamar.

Realmente es complicado y, salvo que lo necesitemos para casos especiales, no será muy habitual que usemos los delegados de esta forma, aunque no está de más saberlo, sobre todo si de así comprendemos mejor cómo funcionan los eventos.

Ver vídeo de esta lección (Delegados) - video en Visual Studio 2005 válido para Visual Studio 2008

Lección 4: Eventos y delegados

- Eventos
- Definir y producir eventos en una clase
- Delegados

Definir un evento bien informado

Definir un evento bien informado con Custom Event

Para terminar con esta lección sobre los eventos y los delegados, vamos a ver otra forma de definir un evento. Esta no es exclusiva de Visual Basic 2008, ya que el lenguaje C#, compañero inseparable en los entornos de Visual Studio 2008, también tiene esta característica, pero aunque pueda parecer extraño, es menos potente que la de Visual Basic 2008; por medio de esta declaración, tal como indicamos en el título de la sección, tendremos mayor información sobre cómo se declara el evento, cómo se destruye e incluso cómo se produce, es lo que la documentación de Visual Basic llama evento personalizado (*Custom Event*).

Cuando declaramos un evento usando la instrucción *Custom* estamos definiendo tres bloques de código que nos permite interceptar el momento en que se produce cualquiera de las tres acciones posibles con un evento:

1. Cuando se "liga" el evento con un método, ya sea por medio de *AddHandler* o mediante *Handles*
2. Cuando se desliga el evento de un método, por medio de *RemoveHandler*
3. Cuando se produce el evento, al llamar a *RaiseEvent*

Para declarar este tipo de evento, siempre debemos hacerlo por medio de un delegado.

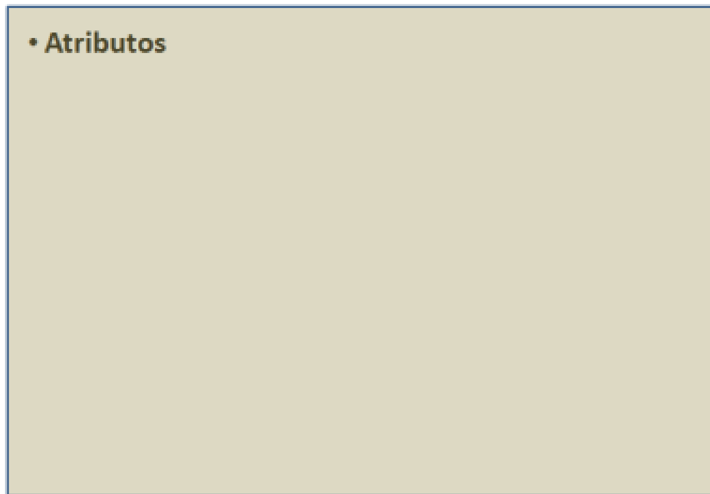
Veamos un ejemplo de una declaración de un evento usando *Custom Event*:

```
Public Delegate Sub ApellidosCambiadosEventHandler(ByVal nuevo As String)



Public Custom Event ApellidosCambiados As ApellidosCambiadosEventHandler
```

```
    AddHandler(ByVal value As ApellidosCambiadosEventHandler)

        ' este bloque se ejecutará cada vez que asignemos un
método al evento

    End AddHandler



    RemoveHandler(ByVal value As ApellidosCambiadosEventHandler)

        ' este bloque se ejecutará cuando se desligue el evento
de un método

    End RemoveHandler



    RaiseEvent(ByVal nuevo As String)

        ' este bloque se ejecutará cada vez que se produzca el
evento

    End RaiseEvent

End Event
```

Como podemos apreciar, debemos definir un delegado con la "firma" del método a usar con el evento.
Después definimos el evento por medio de las instrucciones *Custom Event*, utilizando el mismo formato que al definir un evento con un delegado.
Dentro de la definición tenemos tres bloques, cada uno de los cuales realizará la acción que ya hemos indicado en la lista numerada.

> **Nota:**
> Los eventos Custom Event solamente podemos definirlos e interceptarlos en el mismo ensamblado.



Introducción

Esta es la definición que nos da la ayuda de Visual Basic sobre lo que es un atributo.

Los atributos son etiquetas descriptivas que proporcionan información adicional sobre elementos de programación como tipos, campos, métodos y propiedades. Otras aplicaciones, como el compilador de Visual Basic, pueden hacer referencia a la información adicional en atributos para determinar cómo pueden utilizarse estos elementos.

En esta lección veremos algunos ejemplos de cómo usarlos en nuestras propias aplicaciones y, aunque sea de forma general, cómo usar y aplicar algunos de los atributos definidos en el propio .NET Framework, al menos los que más directamente nos pueden interesar a los desarrolladores de Visual Basic.

Atributos

- **Atributos**
    - Atributos para representar información de nuestra aplicación
        - Mostrar los ficheros ocultos del proyecto
    - Tipos de atributos que podemos usar en una aplicación
        - Atributos globales a la aplicación
        - Atributos particulares a las clases o miembros de las clases
    - Atributos personalizados
        - Acceder a los atributos personalizados en tiempo de ejecución
    - Atributos específicos de Visual Basic
    - Marcar ciertos miembros de una clase como obsoletos

Lección 5: Atributos

Atributos

Como hemos comentado en la introducción, los atributos son etiquetas que podemos aplicar a nuestro código para que el compilador y, por extensión, el propio .NET Framework los pueda usar para realizar ciertas tareas o para obtener información extra sobre nuestro código.

De hecho en cualquier aplicación que creemos con Visual Basic 2008 estaremos tratando con atributos, aunque nosotros ni nos enteremos, ya que el propio compilador los utiliza para generar los metadatos del ensamblado, es decir, la información sobre todo lo que contiene el ejecutable o librería que hemos creado con Visual Basic 2008.

Por otra parte, el uso de los atributos nos sirve para ofrecer cierta funcionalidad extra a nuestro código, por ejemplo, cuando creamos nuestros propios controles, mediante atributos podemos indicarle al diseñador de formularios si debe mostrar ciertos miembros del control en la ventana de propiedades, etc.

Atributos para representar información de nuestra aplicación

De forma más genérica podemos usar los atributos para indicar ciertas características de nuestra aplicación, por ejemplo, el título, la versión, etc. Todos estos atributos los indicaremos como "características" de nuestra aplicación, y lo haremos sin ser demasiados conscientes de que realmente estamos usando atributos, ya que el propio Visual Basic los controla mediante propiedades de la aplicación.

Por ejemplo, si mostramos la ventana de propiedades de nuestro proyecto, ver figura 2.24:

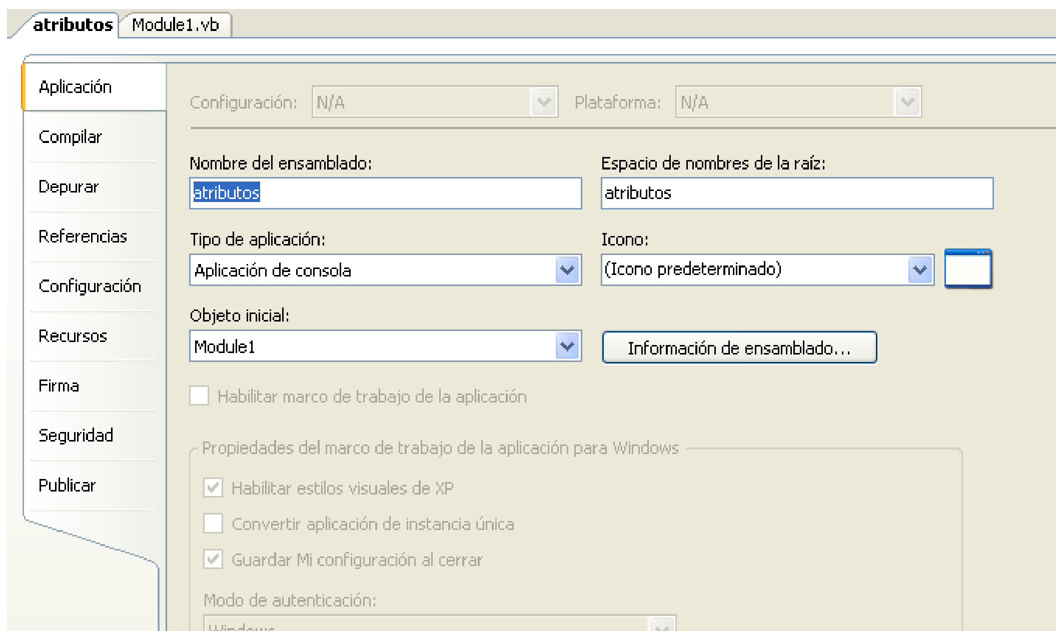


Figura 2.24. Propiedades de la aplicación

Tendremos acceso a las propiedades de la aplicación, como el nombre del ensamblado, el espacio de nombres, etc. Si queremos agregar información extra, como la versión, el copyright, etc. podemos presionar el botón "Assembly Information", al hacerlo, se mostrará una nueva ventana en la que podemos escribir esa información, tal como mostramos en la figura 2.25:

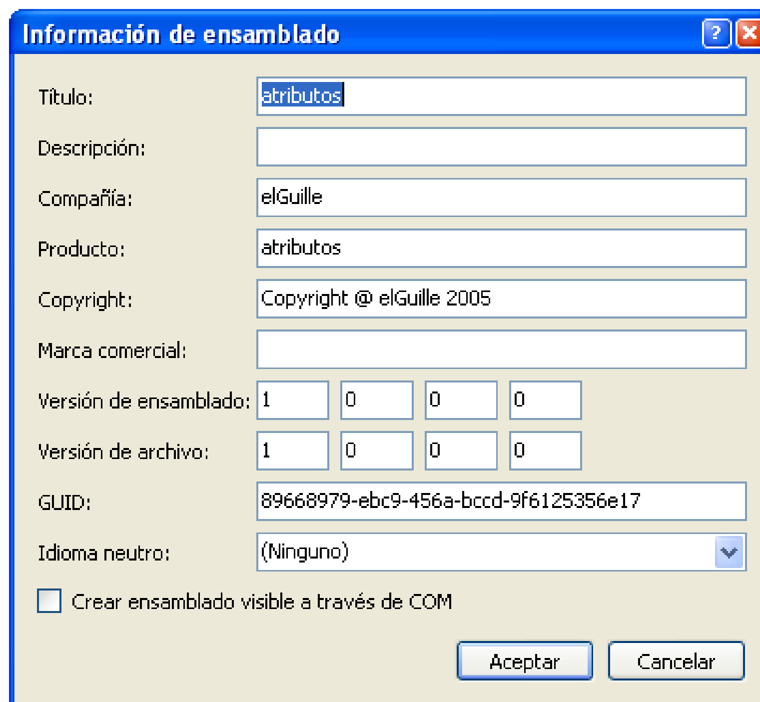


Figura 2.25. Información del ensamblado

Esa información realmente está definida en un fichero del proyecto llamado AssembluInfo.vb, el cual de forma predeterminada está oculto, si lo mostramos, veremos que esa información la contiene en formato de atributos. Parte del código de ese fichero lo podemos ver en la figura 2.26:

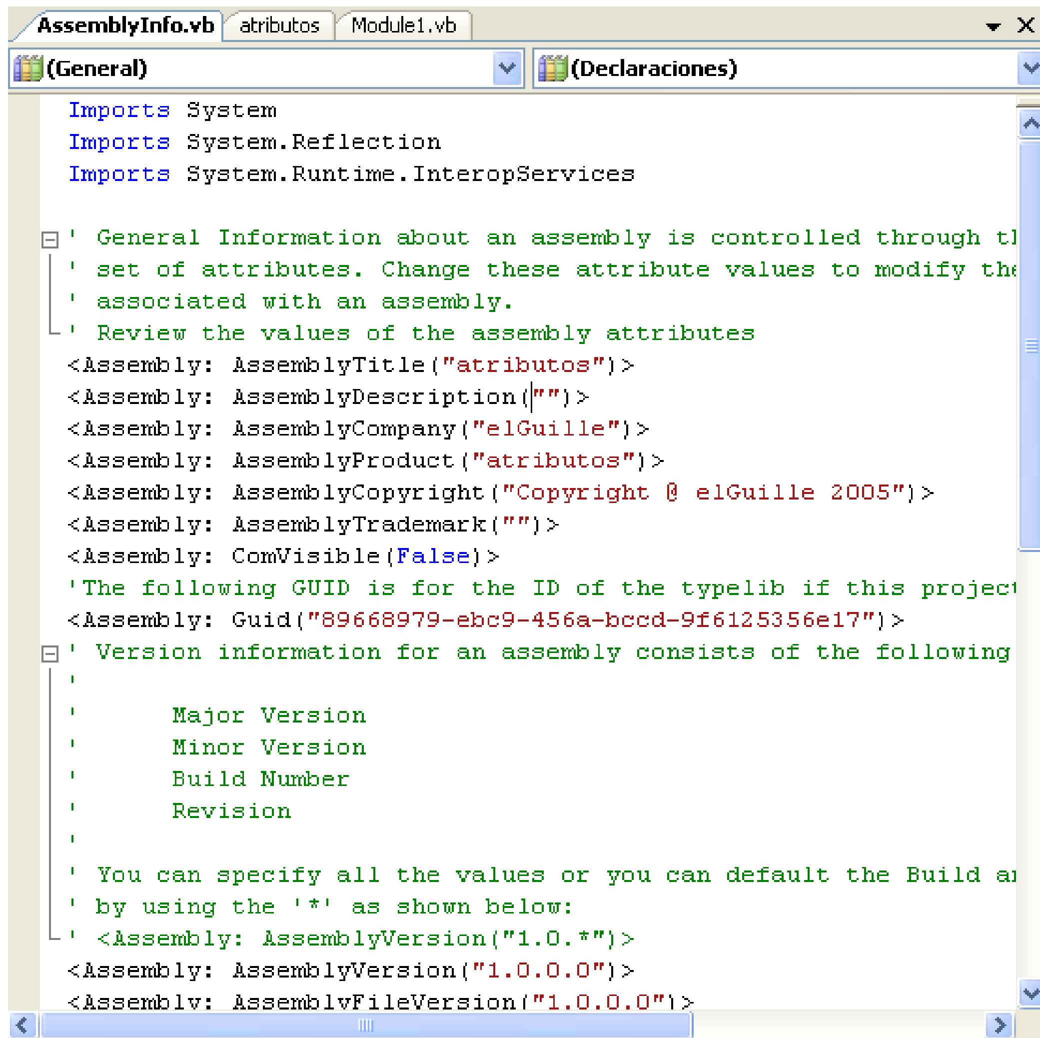


Figura 2.26. Contenido del fichero AssemblyInfo

En este código podemos resaltar tres cosas:

La primera es que tenemos una importación al espacio de nombres **System.Reflection**, este espacio de nombres contiene la definición de las clases/atributos utilizados para indicar los atributos de la aplicación, como el título, etc.

La segunda es la forma de usar los atributos, estos deben ir encerrados entre signos de menor y mayor: **<Assembly: ComVisible(False)>**. La tercera es que, en este caso, los atributos están definidos a nivel de ensamblado, para ellos se añade la instrucción **Assembly:** al atributo. Como veremos a continuación, los atributos también pueden definirse a nivel local,

es decir, solo aplicable al elemento en el que se utiliza, por ejemplo, una clase o un método, etc.

Mostrar los ficheros ocultos del proyecto

Como acabamos de comentar, el fichero AssemblyInfo.vb que es el que contiene la información sobre la aplicación (o ensamblado), está oculto. Para mostrar los ficheros ocultos, debemos hacer lo siguiente:

En la ventana del explorador de soluciones, presionamos el segundo botón, (si pasamos el cursor por encima, mostrará un mensaje que indica "Mostrar todos los ficheros"), de esta forma tendremos a la vista todos los ficheros de la aplicación, incluso el de los directorios en el que se crea el ejecutable, tal como podemos apreciar en la figura 2.27:

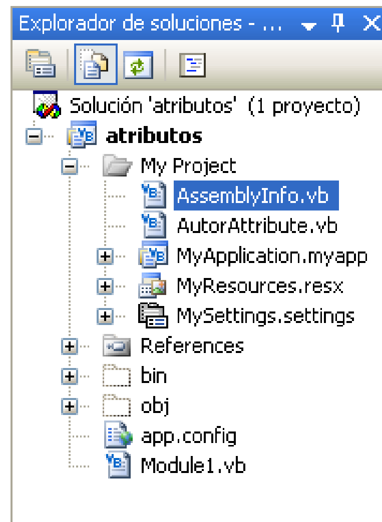


Figura 2.27. Mostrar todos los ficheros de la solución

Tipos de atributos que podemos usar en una aplicación

Como hemos comentado, existen atributos que son globales a toda la aplicación y otros que podremos aplicar a elementos particulares, como una clase o un método.

Atributos globales a la aplicación

Estos se indican usando *Assembly:* en el atributo y los podremos usar en cualquier parte de nuestro código, aunque lo habitual es usarlos en el fichero AssemblyInfo.vb.

> **Nota:**
> La palabra o instrucción Assembly: lo que indica es que el atributo tiene un ámbito de ensamblado.

Atributos particulares a las clases o miembros de las clases

Estos atributos solo se aplican a la clase o al miembro de la clase que creamos conveniente, el formato es parecido a los atributos globales, ya que se utilizan los signos de menor y mayor para encerrarlo, con la diferencia de que en este tipo de atributos no debemos usar *Assembly:*, ya que esta instrucción indica que el atributo es a nivel del ensamblado.

Cuando aplicamos un atributo "particular", este debe estar en la misma línea del elemento al que se aplica, aunque si queremos darle mayor legibilidad al código podemos usar un guión bajo para que el código continúe en otra línea:

```
<Microsoft.VisualBasic.HideModuleName()> _

Module MyResources
```

Atributos personalizados

Además de los atributos que ya están predefinidos en el propio .NET o Visual Basic, podemos crear nuestros propios atributos, de forma que en tiempo de ejecución podamos acceder a ellos mediante las clases del espacio de nombres *Reflection*, aunque debido a que este tema se sale un poco de la intención de este curso, simplemente indicar que los atributos personalizados son clases que se derivan de la clase *System.Attribute* y que podemos definir las propiedades que creamos conveniente utilizar en ese atributo para indicar cierta información a la que podemos acceder en tiempo de ejecución.

En el siguiente código tenemos la declaración de una clase que se utilizará como atributo personalizado, notamos que, por definición las clases para usarlas como atributos deben terminar con la palabra *Attribute* después del nombre "real" de la clase, que como veremos en el código que utiliza ese atributo, esa "extensión" al nombre de la clase no se utiliza.

Veamos primero el código del atributo personalizado:

```
<AttributeUsage(AttributeTargets.All)> _

Public Class AutorAttribute

    Inherits System.Attribute

    '

    Private _ModificadoPor As String

    Private _Version As String

    Private _Fecha As String

    '

    Public Property ModificadoPor() As String

        Get

            Return _ModificadoPor
```

```
        End Get

        Set(ByVal value As String)

            _ModificadoPor = value

        End Set

    End Property

    '

    Public Property Version() As String

        Get

            Return _Version

        End Get

        Set(ByVal value As String)

            _Version = value

        End Set

    End Property

    '

    Public Property Fecha() As String

        Get

            Return _Fecha

        End Get

        Set(ByVal value As String)

            _Fecha = value

        End Set

    End Property

End Class
```

Para usar este atributo lo podemos hacer de la siguiente forma:

```
<Autor(ModificadoPor:="Guillermo 'guille'", _

    Version:="1.0.0.0", Fecha:="13/Abr/2005")> _

Public Class PruebaAtributos
```

Nota:
Cuando utilizamos el atributo, en el constructor que de forma predeterminada crea Visual Basic, los parámetros se indican por el orden alfabético de las propiedades, pero que nosotros podemos alterar usando directamente los nombres de las propiedades, tal como podemos ver en el código de ejemplo anterior.

Acceder a los atributos personalizados en tiempo de ejecución

Para acceder a los atributos personalizados podemos hacer algo como esto, (suponiendo que tenemos la clase **AutorAttribute** y una clase llamada **PruebaAtributos** a la que hemos aplicado ese atributo personalizado):

```
Sub Main()

    Dim tipo As Type

    tipo = GetType(PruebaAtributos)


    Dim atributos() As Attribute

    atributos = Attribute.GetCustomAttributes(tipo)


    For Each atr As Attribute In atributos

        If TypeOf atr Is AutorAttribute Then

            Dim aut As AutorAttribute

            aut = CType(atr, AutorAttribute)

            Console.WriteLine("Modificado      por:      "      &
aut.ModificadoPor)

            Console.WriteLine("Fecha: " & aut.Fecha)

            Console.WriteLine("Versión: " & aut.Version)

        End If
```

```
    Next


End Sub
```

Atributos específicos de Visual Basic

Visual Basic utiliza una serie de atributos para indicar ciertas características de nuestro código, en particular son tres:

- **COMClassAttribute**
  - Este atributo se utiliza para simplificar la creación de componentes COM desde Visual Basic.
- **VBFixedStringAttribute**
  - Este atributo se utiliza para crear cadenas de longitud fija en Visual Basic 2008. Habitualmente se aplica a campos o miembros de una estructura, principalmente cuando queremos acceder al API de Windows o cuando queremos usar esa estructura para guardar información en un fichero, pero utilizando una cantidad fija de caracteres.
- **VBFixedArrayAttribute**
  - Este atributo, al igual que el anterior, lo podremos usar para declarar arrays de tamaño fijo, al menos si las declaramos en una estructura, ya que por defecto, los arrays de Visual Basic son de tamaño variable.

Marcar ciertos miembros de una clase como obsoletos

En ocasiones nos encontraremos que escribimos cierto código que posteriormente no queremos que se utilice, por ejemplo porque hemos creado una versión optimizada.
Si ese código lo hemos declarado en una interfaz, no deberíamos eliminarlo de ella, ya que así romperíamos el contrato contraído por las clases que implementan esa interfaz. En estos casos nos puede venir muy bien el uso del atributo *<Obsolete>*, ya que así podemos informar al usuario de que ese atributo no debería usarlo. En el constructor de este atributo podemos indicar la cadena que se mostrará al usuario. En el siguiente código se declara un método con el atributo *Obsolete*:

```
Public Interface IPruebaObsoleto

    <Obsolete("Este método ya está obsoleto, utiliza el método
Mostrar")> _
```

```
    Sub MostrarNombre()

    Sub Mostrar()

End Interface
```

Si trabajamos con el IDE de Visual Basic, ese mensaje se mostrará al compilar o utilizar los atributos marcados como obsoletos, tal como podemos apreciar en la figura 2.28:

```
    Public Sub MostrarNombre() _
        Implements IPruebaObsoleto.MostrarNombre
                  'Public Sub MostrarNombre()' is obsolete: 'Este método ya está obsoleto, utiliza el método Mostrar'
    End Sub
End Class
```

Figura 2.28. Mensaje de que un método está obsoleto