# Curso de iniciación a la programación con Visual Basic .NET

**Guilermo Som "el Guille"**

**(Adaptación a Word, Marc Salvador. Rev A)**

# Curso de iniciación a la programación con Visual Basic .NET

**(http://guille.costasol.net/NET/cursoVB.NET/tutorVBNET01.htm)**

## 0.1. Introducción:

Debido a que la nueva versión de Visual Basic no es sólo una mejora con respecto a las versiones anteriores, sino que cambia mucho, tanto como si de otro lenguaje de programación se tratara, creo que se merece que se explique de forma más o menos fácil de comprender para que cualquiera que se decida a elegirlo como su lenguaje de programación lo tenga, valga la redundancia, fácil.

Tan fácil como permitan las circunstancias, y además, (para que esto de estudiar no resulte algo tedioso), tan ameno como me sea posible, ya que las cosas se pueden explicar de muchas formas y, a pesar de parecer que peco de falta de modestia, estoy seguro que este curso de iniciación a la programación con **Visual Basic .NET** te va a resultar ameno y fácil de comprender... ¡seguro!

Pero no sólo vas a aprender a programar con VB.NET, sino que al estar "basado" en el **.NET Framework**, conocerás lo suficiente de este marco de desarrollo que podrás atreverte con otros lenguajes .NET, tales como C#, ya que al fin y al cabo, el corazón de los lenguajes .NET es el .NET Framework.
Para ir aclarando ideas, veamos algunos conceptos que habrá que tener claros desde el principio:
Visual Basic .NET usa una jerarquía de clases que están incluidas en el .NET Framework, por tanto conocer el .NET Framework nos ayudará a conocer al propio Visual Basic .NET, aunque también necesitarás conocer la forma de usar y de hacer del VB ya que, aunque en el fondo sea lo mismo, el aspecto sintáctico es diferente para cada uno de los lenguajes basados en .NET Framework, si no fuese así, ¡sólo existiría un solo lenguaje!

Me imagino que la primera pregunta a la que habría que responder es:

## 0.2. ¿Qué es el .NET Framework?

Voy a intentar dar una respuesta que sea fácil de "asimilar", a ver si lo consigo...

Primer intento, lo que se dice en el *eBook* ***Microsoft .NET Framework***, cuya versión en Castellano puedes conseguir usando este link: (este link está actualizado, al menos a fecha de hoy 10 de noviembre de 2002)

*".NET Framework es un entorno para construir, instalar y ejecutar servicios Web y otras aplicaciones.*
*Se compone de tres partes principales: el Common Language Runtime, las clases Framework y ASP.NET"*

Aunque dicho libro está basado en la Beta1 es válido para aclarar conceptos sobre lo que es el .NET Framework además de otros conceptos como el *Common Language Runtime (CLR)*, *Common Language Specification (CLS)*, *Common Type System (CTS)*, *Microsoft Intermediate Language (MSIL),* los ensamblados o *assemblies*, así como sobre *ASP.NET*, conceptos que si bien no son imprescindibles para poder usar Visual Basic .NET, es conveniente leer un poco sobre ellos, para no estar totalmente perdidos cuando nos encontremos con esos *conceptos*...

Segundo intento, lo que dice la MSDN Library:

*"El .NET Framework es un entorno multi-lenguaje para la construcción, distribución y ejecución de Servicios Webs y aplicaciones."*
*"El .NET Framework es una nueva plataforma diseñada para simplificar el desarrollo de aplicaciones en el entorno distribuido de Internet."*
*"El .NET Framework consta de dos componentes principales: el Common Language Runtime y la librería de clases .NET Framework."*

Tercer intento, aclarando las cosas, para que se te "queden" grabadas:

El .NET Framework es el corazón de .NET, cualquier cosa que queramos hacer en cualquier lenguaje .NET debe pasar por el filtro cualquiera de las partes integrantes del .NET Framework.

El Common Lenguage Runtime (CLR) es una serie de librerías dinámicas (DLLs), también llamadas assemblies, que hacen las veces de las DLLs del API de Windows así como las librerías runtime de Visual Basic o C++. Como sabrás, y si no lo sabes ahora te lo cuento yo, cualquier ejecutable depende de una forma u otra de una serie de librerías, ya sea en tiempo de ejecución como a la hora de la compilación. Pues el CLR es eso, una serie de librerías usadas en tiempo de ejecución para que nuestros ejecutables o cualquiera basado en .NET puedan funcionar. Se acabó eso de que existan dos tipos de ejecutables: los que son autosuficientes y no dependen de librerías externas o los que necesitan de librerías en tiempo de ejecución para poder funcionar, tal es el caso de las versiones anteriores de Visual Basic.

Por otro lado, la librería de clases de .NET Framework proporciona una jerarquía de clases orientadas a objeto disponibles para cualquiera de los lenguajes basados en .NET, incluido el Visual Basic. Esto quiere decir que a partir de ahora Visual Basic ya no será la "oveja negra" de los lenguajes de programación, sino que tendrá a su disposición todas las clases disponibles para el resto de los lenguajes basados en .NET, (o casi), con lo cual sólo nos diferenciará del resto de programadores en la forma de hacer las cosas: **¡más fáciles!**

VB.NET ahora es totalmente un lenguaje orientado a objetos con herencia y todo. También permite crear Threads o hilos o tramas de ejecución y otras cosas que antes nos estaban vetadas. De todo esto veremos en esta serie de "entregas", espero que, aunque es un poco más complicado que el Visual Basic de "siempre", confío en que te sea fácil de asimilar. ¡A ver si lo consigo!

## 0.3. Sobre la versión de Visual Basic .NET:

A la hora de escribir estas líneas, la versión de Visual Basic .NET que hay disponible es la que se incluye en la Beta2 de Visual Studio .NET. Pero según dicen, la versión final tendrá pocos cambios con respecto a la Beta 2, así que, espero que todo lo que aquí explique sea válido para la versión definitiva de Visual Basic .NET.

## 0.4. Algunas aclaraciones preliminares:

Antes de empezar a ver el código, un par de aclaraciones, que aunque ahora puede ser que te suenen a chino, *(si eres chino o conoces ese idioma, sólo decirte que es una frase hecha: "me suena a chino" es como decir: "no sé de que me estás hablando")*, pronto serán tan usuales que acabarás por asimilarlas como si toda tu vida las hubieras estado usando... o casi...

**Extensión de los ficheros de código.**
En Visual Basic .NET a diferencia de lo que ocurría en las versiones anteriores de Visual Basic, sólo existe un tipo de fichero de código, el cual tiene la extensión **.vb**, en este tipo de fichero pueden coexistir distintos tipos de *elementos*, por ejemplo: un módulo de clase, un formulario, un módulo de código, un control, etc.; mientras que en las versiones anteriores de Visual Basic, cada uno de estos elementos tenían su propio tipo

de fichero con su respectiva extensión. Si no sabes o no quieres saber de lo que ocurría en las versiones anteriores, me parece muy bien... pero esto sólo es para que lo sepas y no te sorprenda, si es que hay algo que aún puede sorprenderte, claro.

**Tipos de ejecutables.**

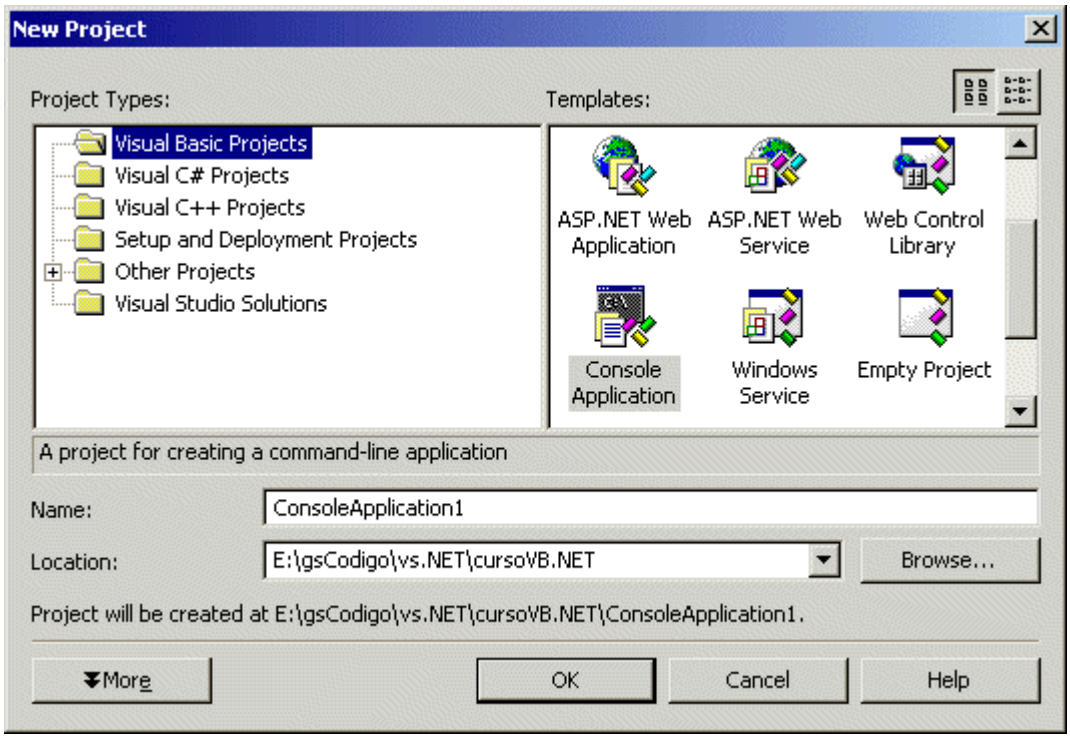
Con Visual Basic .NET puedes crear básicamente estos dos tipos de ejecutables:
de consola, no gráfico, al estilo del viejo MS-DOS, y gráficos, como los que normalmente estamos acostumbrados a ver en Windows.
Existen otros tipos de aplicaciones que se pueden crear con Visual Basic .NET: aplicaciones ASP.NET, (realmente no es una aplicación o ejecutable, sino un compendio de distintos tipos de elementos...), servicios Web, servicios Windows, etc.

# 1. Nuestra primera aplicación con Visual Basic .NET.: Primera entrega

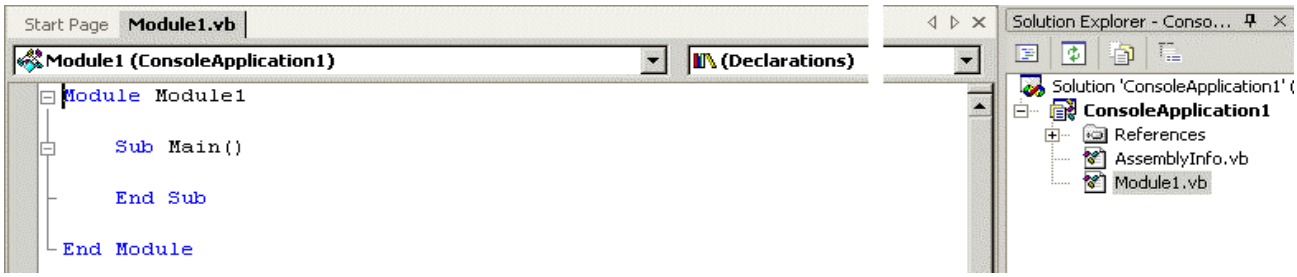
Para ir calentando motores, creo que lo mejor es empezar creando una pequeña aplicación con VB.NET, después iremos aclarando los distintos conceptos usados... así te resultará menos complicado todo lo que tengo preparado para ti.

Inicia el Visual Studio .NET, por defecto te mostrará la "página de inicio" desde la cual pueden crearse nuevos proyectos o bien abrir alguno de los más recientemente abiertos. Pulsa en Nuevo proyecto


Te mostrará los diferentes tipos de proyectos que se pueden crear, en el panel izquierdo selecciona Proyectos de Visual Basic (Visual Basic Projects) y de los que muestra en el panel de la derecha, selecciona Console Application



Tendrás que especificar el directorio en el que se guardará el proyecto, así como el nombre del mismo, (creando un directorio con el nombre del proyecto indicado), deja el nombre que muestra por defecto, en la versión inglesa de Visual Studio .NET se llamará ConsoleApplication1. Pulsa en OK (Aceptar) y se creará el proyecto.
Por defecto te mostrará lo siguiente:




Es decir, creará un fichero llamado Module1.vb, (mostrado a la derecha en el Solution Explorer), con el código necesario para empezar a escribir. Fíjate que además del procedimiento **Sub Main**, el cual se usará como punto de entrada de nuestro ejecutable, también ha creado una "definición" llamada **Module Module1** con su respectivo **End Module**, el cual indica dónde termina la definición del módulo. Esto es así, porque, como te dije hace un rato, en un mismo fichero .vb, pueden existir distintos tipos de elementos. Por ahora, dejémoslo así... ya habrá tiempo de complicarnos la vida...

> **Una aclaración:** lo que estamos creando es una aplicación tipo consola, es decir, no se creará ninguna ventana gráfica, sino que el ejecutable que vamos a crear funciona desde una ventana de MS-DOS (o consola). Esto lo comprobaremos cuando ejecutemos el proyecto.

Lo que queremos, (o mejor dicho, lo que YO QUIERO), mostrar, es un mensaje que diga algo así como: Hola mundo .NET ¡que original! ¿verdad?, por tanto para mostrar un texto en la "consola", usaremos una función, método o instrucción, (como prefieras llamarla), que si bien no es nativa de Visual Basic .NET, la usaremos como si lo fuese... como veremos más tarde, TODO esto es posible gracias a los assemblies o a las clases incluidas en el .NET Framework. Por ahora simplemente confía en mí y escribe lo que te voy a decir.

La función en cuestión, (realmente todo lo que se usa en .NET son funciones), es **Console.Write** y se usa de la siguiente forma:

**Console.Write("Hola mundo .NET")**, es decir incluiremos dentro de paréntesis lo que queremos que se muestre en la consola, en este caso queremos mostrar un texto, el cual hay que incluirlo dentro de comillas dobles.
Escríbelo entre el *Sub Main()* y el End Sub. Comprueba que cuando escribas Console y el punto, se mostrarán las funciones que Console pone a nuestra disposición, así como una pequeña ayuda, en modo de ToolTip, aunque a esto, o a algo parecido, ya estarás acostumbrado si has usado alguna vez el Visual Basic 5/6.

Bien, ya tenemos todo lo que necesitamos. Ahora tendremos que indicarle al "Entorno Integrado" (IDE) que compile el proyecto y lo ejecute, y después de compilarse el proyecto, se deberá mostrar el texto en una ventana de DOS (o consola).
(Guille, ¿por qué me da la impresión de que no se va a mostrar nada? te gustaría preguntarme en este preciso momento)
Para salir de dudas, pulsa F5 (o a la flecha azul o botón con figura de PLAY de un reproductor)

**Pregunta:** ¿Que ha pasado?
**Respuesta:** Realmente se ha mostrado el mensaje en una ventana de consola...

(Salvo que hayas cometido algún error, cosa que sólo habrá ocurrido si en lugar de estar leyendo, te has dedicado a hacer tus propias pruebas, así que... ¡HAZ EL FAVOR DE ATENDER EN CLASE! ¡Ya tendrás tiempo de hacer tus propias pruebas!)
**P:** Entonces, ¿por qué no se ve?
**R:** Porque después de mostrarse se ha cerrado la ventana.
**P:** ¿Cómo podemos ver el mensaje?
**R:** Ejecutando el EXE desde una ventana de DOS (o consola)
Pero lo mejor sería hacer que el programa se pare hasta que pulsemos la tecla Intro. Para ello, añade la siguiente línea a continuación de la anterior:
**Console.Read()**

Pulsa de nuevo F5 y verás como esta vez si que se muestra el mensaje, además de que la ventana no se cierra hasta que pulses Intro.
Realmente puedes escribir lo que te de la gana y se irá mostrando en la ventana de consola, pero hasta que pulses Intro no dejará de mostrarse. (Tampoco iba a ser el primer ejemplo tan perfecto... ¡que te crees!).

Pues ésta es nuestra primera aplicación con el Visual Basic .NET.
Realmente tan inútil como poco práctica, pero... queda muy bien eso de saber que ya somos capaces de crear nuestros propios ejecutables. La verdad es que a estas alturas (o mejor dicho bajuras) del curso o tutorial no pretenderás hacer cosas más "sofisticadas", entre otras razones, porque se supone que no sabes nada de nada... ¿cómo? que si que sabes... que ya has trabajado antes con el Visual Basic... que incluso te has leído mi Curso Básico de VB... entonces... tendrás que esperar algunas entregas o unirte al grupo de estudiantes noveles (o principiantes o novatos o... como quieras llamarlos) y esperar a que los conceptos básicos estén aclarados, ya que este curso es un curso de iniciación y si los que lo siguen ya supieran tanto como tú, no sería un curso de iniciación... pues eso... (¡Que borde (desagradable) eres algunas veces Guille!)

Olvidemos a *los otros Guilles* y sigamos...

Antes de continuar, vamos a conocer un poco sobre el entorno de desarrollo de Visual Studio .NET, (que es el que se usa con Visual Basic .NET), para que podamos configurar algunos aspectos, por ejemplo para indicar cómo se comportará el compilador e intérprete sobre el código que escribamos o para configurar las librerías (assemblies) que se usarán en nuestras aplicaciones. Recuerda que Visual Basic .NET usa una serie de librerías (de clases) con las funciones que necesitemos en cada momento...

¿Te parece complicado? No te preocupes... ahora simplemente lee y pronto entenderás, pero por favor: ¡lee! no intentes pasar todo este "rollo" por alto, ya que si no te enteras de lo que te estoy contando, seguramente acabarás preguntándomelo por e-mail y la única respuesta que recibirás por mi parte es que te vuelvas a leer toda esta parrafada... gracias.

Por ejemplo, para poder mostrar un texto en la consola, necesitamos tener disponible la librería en la cual está declarada la clase **Console**, para que podamos acceder a las funciones que dicha clase pone a nuestra disposición, (por ejemplo Write o Read); en este caso la *librería* en la que está la clase Console es: **System**. System realmente es un **Namespace** o espacio de nombres, no es una librería o assembly.

## 1.1. ¿Que es un Namespace (o espacio de nombres)?

*"Un espacio de nombres es un esquema lógico de nombres para tipos en el que un nombre de tipo simple, como MiTipo, aparece precedido por un nombre jerárquico separado por puntos. [...]"*

Así es como lo definen en el eBook de .NET Framework que mencioné al principio.

Para que nos entendamos, un Namespace, (prefiero usar el nombre en inglés, ya que así es como aparecerá en el código), es una forma de agrupar clases, funciones, tipos de datos, etc. que están relacionadas entre sí. Por ejemplo, entre los Namespaces que

podemos encontrar en el .NET Framework encontramos uno con funciones relacionadas con Visual Basic: **Microsoft.VisualBasic**. Si te fijas, Microsoft y VisualBasic están separados por un punto, esto significa que **Microsoft** a su vez es un Namespace que contiene otros "espacios de nombres", tales como el mencionado **VisualBasic**, **CSharp** y **Win32** con el cual podemos acceder a eventos o manipular el registro del sistema...

Para saber que es lo que contiene un Namespace, simplemente escribe el nombre con un punto y te mostrará una lista desplegable con los miembros que pertenecen a dicho espacio de nombres.

Por regla general se deberían agrupar en un Namespace funciones o clases que estén relacionadas entre sí. De esta forma, será más fácil saber que estamos trabajando con funciones relacionadas entre sí.

Pero el que distintos espacios de nombres pertenezcan a un mismo Namespace, (viene bien esto de usar la traducción castellana e inglesa de una palabra, para no ser redundante), no significa que todos estén dentro de la misma librería o assembly. Un Namespace puede estar repartido en varios assemblies o librerías. Por otro lado, un assembly, (o ensamblado), puede contener varios Namespaces.

Pero de esto no debes preocuparte, ya que el IDE de Visual Studio .NET se encarga de "saber" en que assembly está el Namespace que necesitamos.

## 1.2. ¿Que es un assembly (o ensamblado)?

*"Un ensamblado es el bloque constructivo primario de una aplicación de .NET Framework. Se trata de una recopilación de funcionalidad que se construye, versiona e instala como una única unidad de implementación (como uno o más archivos). [...]"*

Para que nos entendamos, podríamos decir que un assembly es una librería dinámica (DLL) en la cual pueden existir distintos espacios de nombres. Aunque esto es simplificar mucho, por ahora nos vale.

Un ensamblado o assembly puede estar formado por varios ficheros DLLs y EXEs, pero lo más importante es que todos los ensamblados contienen un manifiesto (o manifest), gracias al cual se evitan muchos de los quebraderos de cabeza a los que Windows nos tiene acostumbrados, al menos en lo referente a las distintas versiones de las librerías y ejecutables, seguramente habrás oído hablar de las ***DLL Hell*** (o librerías del demonio) expresión que se usa cuando hay incompatibilidad de versiones entre varias librerías que están relacionadas entre si. Por ejemplo, supongamos que tenemos una librería DLL que en su primera versión contenía X funciones. Al tiempo, se crea la segunda versión de dicha librería en la que se cambian algunas funciones y se añaden otras nuevas, para mejorar el rendimiento de las funciones contenidas en esa librería se usa otra DLL que es usada por algunas de las funciones contenidas en esa segunda versión. Esa otra librería puede ser una librería del sistema, la cual a su vez se actualiza con nueva funcionalidad y puede que dicha funcionalidad dependa a su vez de una tercera librería.

Resulta que instalamos un programa que usa las últimas versiones de todas estas librerías. Todo va bien, el programa funciona a las mil maravillas y nosotros estamos "supersatisfechos" de ese programa que no se cuelga ni una sola vez... (¿Quién habrá hecho ese programa tan maravilloso?, sin comentarios...)

Ahora llega a nuestras manos otra aplicación que necesitamos instalar y la instalamos, pero resulta que esa aplicación usa la primera versión de nuestra famosa librería. Si el programa de instalación está bien hecho, no ocurrirá nada malo, ya que al descubrir que tenemos una versión más reciente de la librería, deja la que ya está instalada. Probamos el programilla de marras y todo funciona bien. Probamos el maravilloso programa anterior y también funciona bien. ¿Cual es el problema? Por ahora ninguno, pero espera... Después instalamos un programa que usa una de las librerías del sistema u otra que también usa nuestra "flamante" librería, pero ese programa se ha instalado de "mala manera", bien porque el programa de instalación sea una caca o bien porque

simplemente se ha instalado mal... como quiera que ha instalado una librería anterior a la que nuestros dos maravillosos ejecutables usan, se puede dar el caso de que ninguno de los dos programas funcionen correctamente... esto ocurrió cuando salió el Internet Explorer 4 y a más de uno nos trajo de cabeza, aunque también ha ocurrido con otros programas que no han tenido en cuenta a la hora de instalar que ya existe una versión más reciente de la librería. Por suerte, esto ya es menos común que hace unos años, sobre todo si los programas de instalación están creados con el Windows Installer o estamos usando el Windows 2000/XP. Pero es que .NET mejora aún esa "imposibilidad" de meter la pata ya que cada assembly contiene un manifiesto en el cual se indica: -el nombre y la versión del assembly, -si este assembly depende de otros ensamblados, con lo cual se indica hasta la versión de dichos ensamblados,

- los tipos expuestos por el assembly (clases, etc.),
- permisos de seguridad para los distintos tipos contenidos en el assembly.

También se incluyen en los assemblies los datos del copyright, etc.

Nuevamente he de decirte que no debes preocuparte demasiado por esto, ya que es el propio .NET el que se encarga de que todo funciones a las mil maravillas, o al menos esa es la intención.

La ventaja de los ensamblados es que "realmente" no necesitan de una instalación y un registro correcto en el registro del sistema de Windows, ya que es el "intérprete" de .NET el que se encarga de hacer las comprobaciones cuando tiene que hacerlas. Por tanto podríamos distribuir una aplicación sin necesidad de crear un programa de instalación. Pero, (¿por qué siempre hay un pero?), si la aplicación usa ensamblados compartidos, puede que sea necesario usar una instalación.

Los ensamblados compartidos se pueden usar por varias aplicaciones diferentes y deben estar "debidamente" instalados en el directorio asignado por el propio .NET Framework.

Ejemplo de ensamblados compartidos son los que definen las clases (tipos) usados por el propio .NET Framework.

Para terminar esta primera entrega introductoria al mundo .NET vamos a ver algunos conceptos que usaremos con bastante frecuencia en el resto de las entregas:

**Nota: Las palabras o conceptos están en la página del glosario.**

# 2. Segunda entrega

En la entrega anterior vimos algunos conceptos generales que acompañan a esta versión de Visual Basic, también vimos algunas **palabras** que usaremos durante todo este curso de iniciación, dichas palabras están en el glosario, al cual iré añadiendo nuevas definiciones, conforme surjan o yo me vaya acordando, así que te recomiendo que de vez en cuando le eches un vistazo.

Ahora vamos a empezar con esta segunda entrega del Curso de Iniciación a la programación con Visual Basic .NET

Lo primero que vamos a aprender es a saber manejarnos un poco con el entorno de desarrollo (IDE) de Visual Studio .NET, sí he dicho entorno de desarrollo de Visual Studio, no de Visual Basic... y no me he confundido, ya que dicho IDE es el mismo para cualquier lenguaje .NET. Por tanto, si además de usar el VB.NET quieres hacer tus "pinitos" con el C# o con cualquier otro, no tendrás que abrir otro programa para cada uno de esos lenguajes. Incluso podrás trabajar con varios proyectos a un mismo tiempo, aunque sean de lenguajes diferentes. Además, si así lo quieres y "sabes" cómo hacerlo, puedes crear tus propios complementos para que se integren con el IDE de Visual Studio .NET.

Aunque eso, casi con total seguridad, no lo veremos en este curso de iniciación.

Lo primero que vamos a hacer es cargar el entorno de desarrollo del Visual Studio .NET, así que... ya puedes abrirlo, si es que no la has hecho ya.

En la pantalla de inicio, selecciona un nuevo proyecto. Pero esta vez vamos a seleccionar una aplicación de Windows:

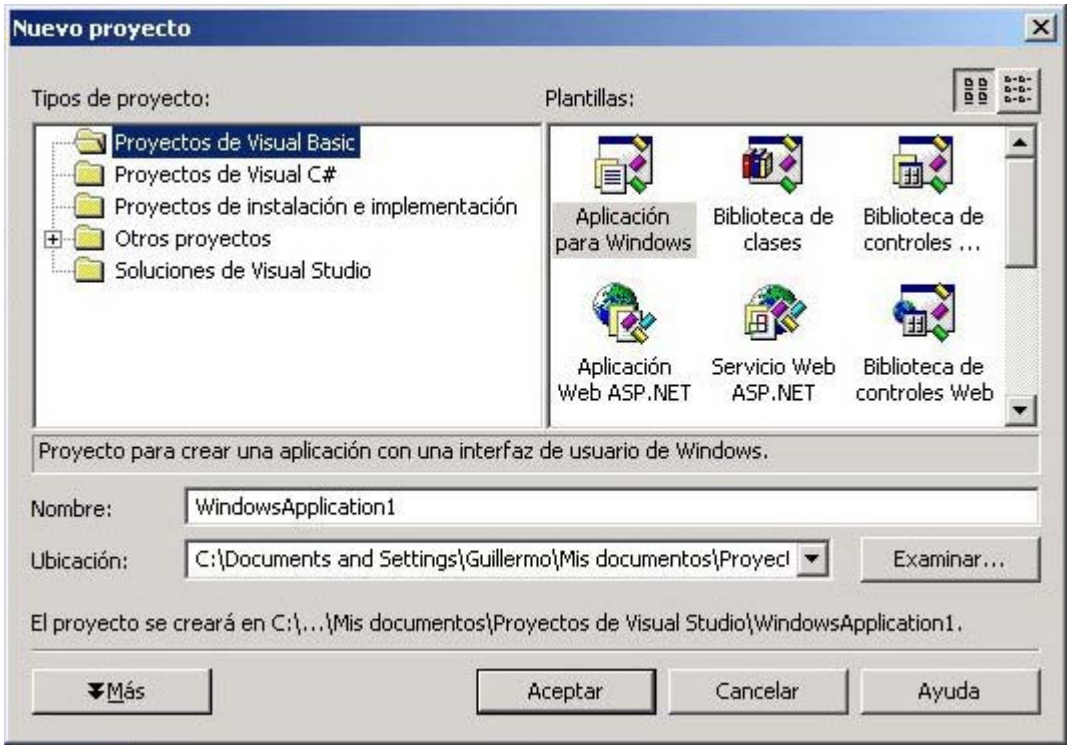



**Figura 1**

No te preocupes, que no te voy a complicar la vida, simplemente vamos a tener algún tipo de proyecto cargado en el entorno de desarrollo, ya que lo que vamos a ver es cómo modificar y configurar algunos de los aspectos del mismo.

Déjale el nombre que te muestra, ya que nos da igual cómo se llame. (el Guille está un poco *pasota*, ¿verdad?)

En el proyecto, se habrá creado un formulario, el cual seguramente te lo mostrará de forma automática. Si no se mostrara nada y siguiera estando la pantalla de inicio... dale un respiro ya que seguramente se lo estará pensando... en caso de que después de pensárselo no lo mostrara tampoco... fíjate en el lado derecho de la pantalla, verás que hay un "panel" o ventana en la que se indica el proyecto actual y se muestran los ficheros que lo componen. Ese panel es el "**Solution Explorer**" o **Explorardor de Soluciones**, (te digo cómo se llama en los dos idiomas, ya que, algunas de las imágenes que te voy a mostrar están capturadas de la versión en castellano del Visual Studio .NET y otras de la versión en inglés... así cuando la veas en un idioma o en otro, sabrás de que estoy hablando).

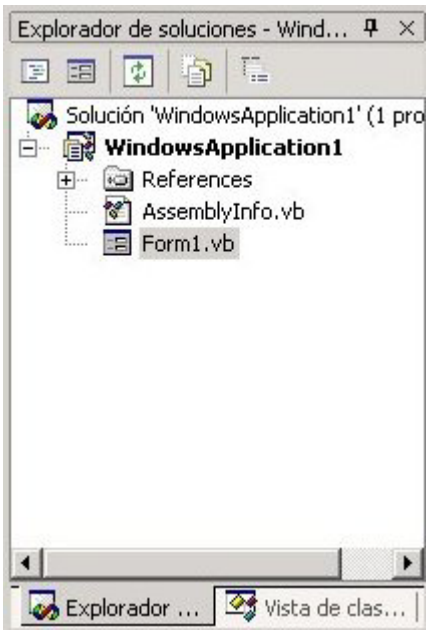



**Figura 2**

Para que se muestre el formulario (Form1), haz doble click en dicho "elemento" del explorador de soluciones.

Lo que vamos a hacer ahora es cambiar la separación de los puntos que se muestran en el formulario, ese "grid" o grilla, servirá para ajustar los controles (ya sabrás de qué hablo) que se añadan a dicho formulario.

Por defecto la separación es de 8x8 puntos o pixels... y vamos a ponerlo en 4x4.

Selecciona el formulario, (el cual se supone que ya si puedes verlo), simplemente con hacer un click en él es suficiente. Verás que tiene este aspecto:

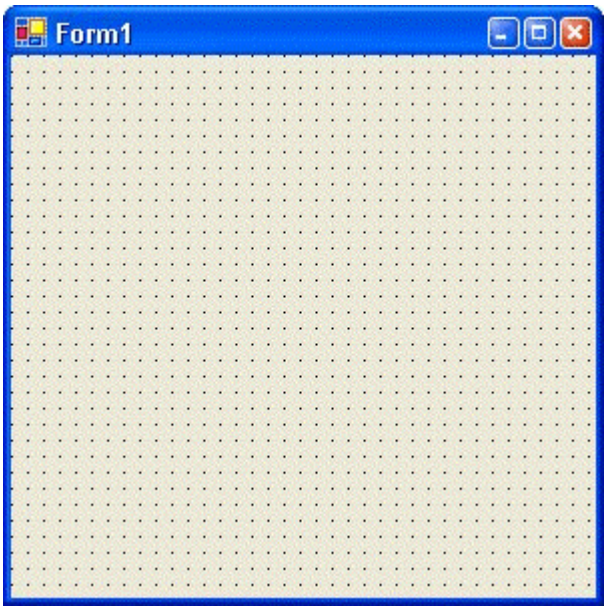



**Figura 3**

Realmente no tendrá ese aspecto, salvo que tengas el *Windows XP*, pero al aspecto que me refiero es al de la separación de los puntos.

En el panel de la derecha, debajo del explorador de soluciones, está la ventana de propiedades del elemento que actualmente esté seleccionado, en nuestro caso son las propiedades del Form1. Vamos a buscar el elemento **GridSize** para poder cambiar el tamaño de separación. Verás que se muestran dos valores separados por punto y coma, pero también hay una cruz a la izquierda de la palabra GridSize, si pulsas en esa cruz, se mostrarán los tamaños a lo que se refiere esos valores separados por punto y coma:

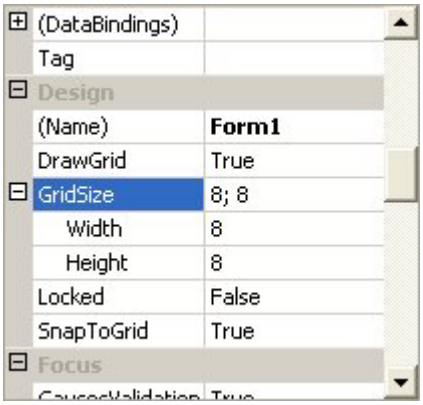



**Figura 4**

Posiciónate en cualquiera de ellos y asígnale el valor 4, de esta forma, tendremos un formulario con una cuadrícula más pequeña, en la que será más fácil posicionar los controles que queramos añadir al formulario en cuestión.

Fíjate que después de haber cambiado esos valores, los mismos se muestran en "negrita", indicándonos de esta forma que son valores que nosotros hemos asignado, los que tiene por defecto. También te habrás fijado que ahora el "grid" tiene los puntos más juntos. Si no te gusta así como está ahora, ponlos a tu gusto. Yo los configuro a 4x4, pero tu eres libre de ponerlos como mejor te plazca...

El aspecto habrá cambiado a este otro, te muestro el aspecto de Windows XP y el del Windows 2000 (clásico), aunque en resumidas cuentas, lo que hay que ver es que *los punticos esos están más arrejuntaos*...

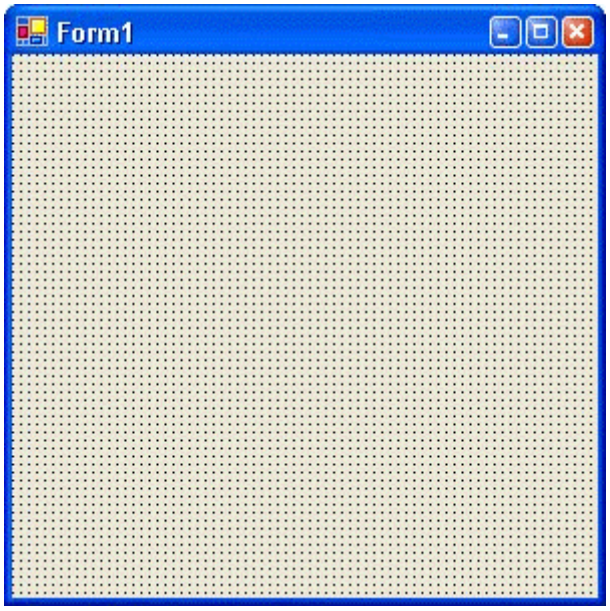



**Figura 5**

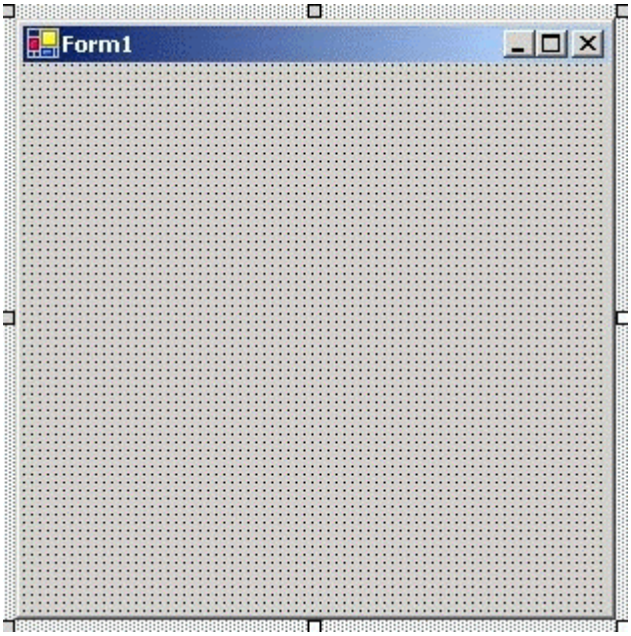



**Figura 6**

Para añadir controles al formulario, hay que usar la barra de herramientas que está situada en la parte izquierda del IDE de Visual Studio .NET, por ejemplo para añadir una etiqueta (Label) y una caja de texto (TextBox), simplemente haz doble-click sobre esos elementos de la barra de herramientas y se añadirán al formulario.

Para poder situarlos en el sitio que más te apetezca, simplemente pulsa en ellos y manteniendo el ratón pulsado, ponlos donde más te guste... todo esto deberías saber hacerlo, ya que son cosas básicas de Windows, así que en próximas ocasiones no esperes tantas explicaciones... ¿vale?

Añade ahora un botón (Button) y sitúalo debajo del textbox. Para cambiarle el texto que muestra el botón, que por defecto será el nombre que el IDE le ha asignado, en esta ocasión será Button1, hay que usar la ventana de propiedades, en esta ocasión el elemento que nos interesa de esa ventana de propiedades es **Text**, escribe Mostrar y cuando pulses Intro, verás que el texto del botón también ha cambiado. Si antes has trabajado con el Visual Basic "clásico", esa propiedad se llamaba Caption. Decirte que ya la propiedad Caption no existe en ningún control, ahora se llama Text. Haz lo mismo con la etiqueta, recuerda que tienes que seleccionarla (un click) para que se muestren las propiedades de la etiqueta, escribe **Nombre:** y pulsa intro.

Ahora vamos a escribir código para que se ejecute cada vez que se haga click en el botón que hemos añadido.

Para ello, selecciona el botón Mostrar y haz doble click en él, se mostrará una nueva ventana, en este caso la ventana de código asociada con el formulario que tenemos en nuestro proyecto.

Te mostrará esto: (realmente te mostrará más cosas, pero por ahora centrate sólo en este código)

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
End Sub
```

Lo que vamos a hacer ahora, como ya te he dicho, es escribir el código que se ejecutará cuando se haga click en ese botón, lo cual producirá el **evento** Click asociado con dicho botón, ese evento se producirá si se hace un click propiamente dicho, es decir con el

ratón, o bien porque se pulse intro o la barra espaciadora cuando el botón tenga el foco. La nomenclatura, (forma de llamar a las cosas), para los eventos de Visual Basic siguen el siguiente "esquema": [nombre del control] [guión bajo] [nombre del evento] Pero esto sólo es una sugerencia que Visual Basic nos hace, en las versiones anteriores no era una sugerencia, era una imposición. Podemos dejar el nombre que Visual Basic nos sugiere o podemos poner el nombre que nosotros queramos; lo importante aquí es la parte final de la línea de declaración del procedimiento: ***Handles Button1.Click***, con esto es con lo que el compilador/intérprete de Visual Basic sabe que este procedimiento es un evento y que dicho evento es el evento Click del objeto Button1.
Un detalle: el nombre Button1 es porque hemos dejado el nombre que por defecto el IDE de Visual Studio asigna a los controles que se añaden a los formularios. Si quieres que se llame de otra forma, simplemente muestra el formulario, selecciona el control al que quieres cambiarle el nombre, busca la propiedad Name en la ventana de propiedades y cambia el nombre que allí se sugiere por el que tu quieras... o casi, ya que para los nombres de los controles, así como para otras cosas que usemos en Visual Basic hay que seguir ciertas normas:

- El nombre debe empezar por una letra o un guión bajo.
- El nombre sólo puede contener letras, números y el guión bajo.

Por tanto, si quieres cambiarle el nombre al evento que se produce cuando se hace click en el botón, escribe ese nombre después de Private Sub, aunque no es necesario cambiar el nombre del evento, ya que, al menos por ahora, nos sirve tal y como está.

Lo que si importa es lo que escribamos cuando ese evento se produzca, en este caso vamos a a hacer que se muestre un cuadro de diálogo mostrándonos el nombre que previamente hemos escrito en el cuadro de texto.

Escribe lo siguiente en el hueco dejado por el Visual Basic, la línea que hay entre Private Sub... y End Sub

```
MsgBox("Hola " & TextBox1.Text)
```

Antes de explicarte que es lo que estamos haciendo, pulsa F5 para que se ejecute el código que hemos escrito o pulsa en el botón "play" que está en la barra de botones.

Escribe algo en la caja de textos, que por defecto tendrá "TextBox1", (que es el valor que por defecto Visual Basic le asigna a la propiedad Text de dicho control), pulsa en el botón Mostrar y verás que se muestra un cuadro de diálogo diciéndote Hola y a continuación lo que hayas escrito en el TextBox.

Algo así:

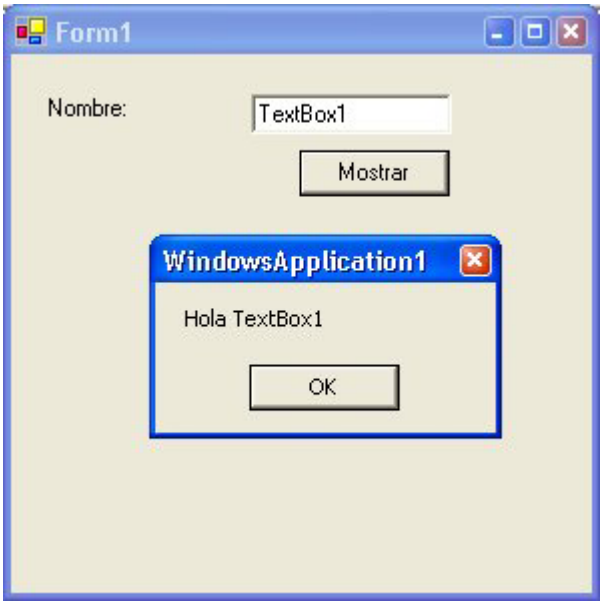


**Figura 7**

Pues sí: ¡esta es tu primera aplicación para Windows creada con el Visual Basic .NET! (pfiuuuuu, pfiiiiuuuu, bang! (sonido de cohetes y esas cosas))

Pulsa en el botón OK (Aceptar) para quitar el cuadro de diálogo y para cerrar el formulario, pulsa en el botón con la "x".

Ahora vamos a añadir otro botón, el cual se llamará cmdCerrar y el texto que debe mostrar es: Cerrar.

Y este es el código que debe ejecutar cuando se haga click en el... ¿te atreves a hacerlo sin ayuda? Si no te atreves, tendrás que hacerlo sin mi ayuda... creo que ya sabes cómo hacerlo... venga, no me seas holgazán... (u holgazana, puntualicemos).

```
Me.Close()
```

Pulsa F5 y cuando pulses en el botón cerrar, se cerrará el formulario.

Ahora veamos con detalle el código que hemos usado en los dos eventos:

```
MsgBox("Hola " & TextBox1.Text)
```

En este código tenemos lo siguiente:

**MsgBox** que es una función o método, (realmente es una clase, como casi todo en .NET), cuya tarea es mostrar en un cuadro de diálogo lo que le indiquemos en el primer parámetro, también tiene parámetros opcionales, pero por ahora usemos sólo el primero que es obligatorio.

En Visual Basic.NET todos los procedimientos que reciban parámetros deben usarse con los paréntesis, esto no era así en las versiones anteriores de VB, por tanto, para indicarle que es lo que queremos que muestre, tendremos que hacerlo dentro de los paréntesis, en este caso, queremos que se muestre la palabra **"Hola "** y lo que haya en la caja de texto.

La palabra Hola (seguida de un espacio) es una constante, es decir siempre será la palabra Hola seguida de un espacio, salvo que nosotros queramos que sea otra cosa... pero una vez que el programa esté compilado, siempre será lo mismo... por eso se llama constante, porque no cambia... en este caso una constante alfanumérica o de cadena, por eso va entrecomillada, ya que todos los *literales* o cadenas que queramos usar en nuestros proyectos deben ir dentro de comillas dobles.

Por otro lado, **TextBox1.Text** representa el texto que haya en la caja de textos y por tanto devolverá lo que en él hayamos escrito.

Por último, para que esas dos cadenas de caracteres, la constante Hola y el contenido de la propiedad Text del control TextBox1, se puedan unir para usarla como una sola cadena, usaremos el signo & (ampersand) el cual sirve para eso, para concatenar cadenas de caracteres y hacer que Visual Basic entienda que es una sola.

Por tanto, si la propiedad Text del control TextBox1 contenía la cadena TextBox1, (tal y como se muestra en la figura 7), al unir las dos cadenas, resultará una nueva con el siguiente contenido: **"Hola TextBox1"** que no es ni más ni menos que la "suma" de las dos cadenas que teníamos... (sí, ya se que soy un poco pesado con estos y que me estoy repitiendo, pero tú lee y deja que yo piense en cómo hacer que todo esto te entre en tu cabecita... ¿vale? gracias por la confianza...)

En el método del evento Click del botón cerrar hemos escrito: Me.Close()

**Me** representa al objeto o clase Form1 (el formulario) y el método **Close** lo que hace es cerrar el formulario, igual que cuando pulsamos en el botón cerrar del formulario.

Vamos a ver ahora el código completo del evento Click del botón cerrar, así como los pasos que se supone que has realizado para poder añadirlo al formulario, tal y como te dije un poco antes:

Para añadir un botón nuevo:

- haz doble click en el elemento Button de la ventana de herramientas que está a la izquierda,
- selecciona el control recién añadido, (haciendo un click simple)
- en la ventana de propiedades selecciona la propiedad Text y escribe en ella la palabra Cerrar,
- en la ventana de propiedades selecciona la propiedad Name y escribe en ella la palabra cmdCerrar,
- haz doble-click en el botón para que se muestre la ventana de código y te mostrará:

```
Private Sub cmdCerrar_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles cmdCerrar.Click

End Sub
```

- escribe el código para cerrar el formulario: Me.Close()

**Nota**: Para que tengas más fácil buscar las propiedades en la ventana de propiedades, puedes hacer que se muestren por orden alfabético, simplemente pulsando en el botón AZ:

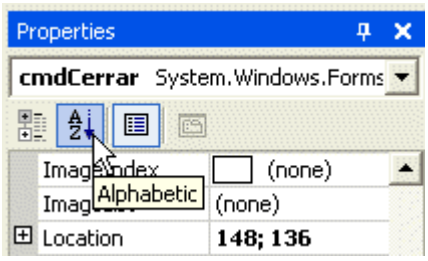



**Figura 8**

Seguramente te estarás preguntando porqué algunas imágenes están en español y otras en inglés, (y si no te lo estás preguntando, te lo va a decir igualmente), es porque algunas capturas las hice en de la versión en castellano de la Beta 2 de Visual Studio .NET y otras en la versión en inglés, ¿la razón o motivo de esta dualidad?, la partición en la que tenía instalado la versión en español, la he formateado y cuando quise volver a instalarla, me dio tantos errores que desistí y ahora estoy usando la versión inglesa que ya tenía instalada en la partición del Windows XP. Si más adelante vuelvo a instalarla, (cosa que volveré a intentar, aunque no con tanta celeridad como me hubiese gustado, ya que al no tener la documentación en español, me da lo mismo que el entorno de trabajo esté o no en inglés... cosa que cambiará cuando esté la versión definitiva... o cuando me envíen la versión con la documentación en español... lo que antes ocurra). Sé que estos detalles a lo mejor ni te interesan, pero tenía ganas de explicarlo... je, je.

Para terminar esta segunda entrega, vamos a crear un proyecto igual al que hasta ahora hemos usado, pero con el lenguaje C# (c sharp), para que veas que en algunas cosas es igual de sencillo usarlo, aunque en algunos aspectos es más estricto que el Visual Basic y

así de camino te demuestro que no era falso eso que te dije de que en el entorno integrado del Visual Studio .NET podíamos tener varios proyectos en varios de los lenguajes soportados por .NET.

Sin cerrar el proyecto que ya tenemos, despliega el menú File/Archivos y selecciona la opción Add project (Añadir proyecto), del submenú que te mostrará selecciona Nuevo proyecto... y del cuadro de diálogo que te muestra, (como el de la Figura 1), selecciona del panel izquierdo **Visual C# Pojects** y del derecho **Windows Application**, y pulsa en el botón Aceptar.(en la imagen siguiente te muestro también el nombre que yo le he dado: WindowsApplication1cs)

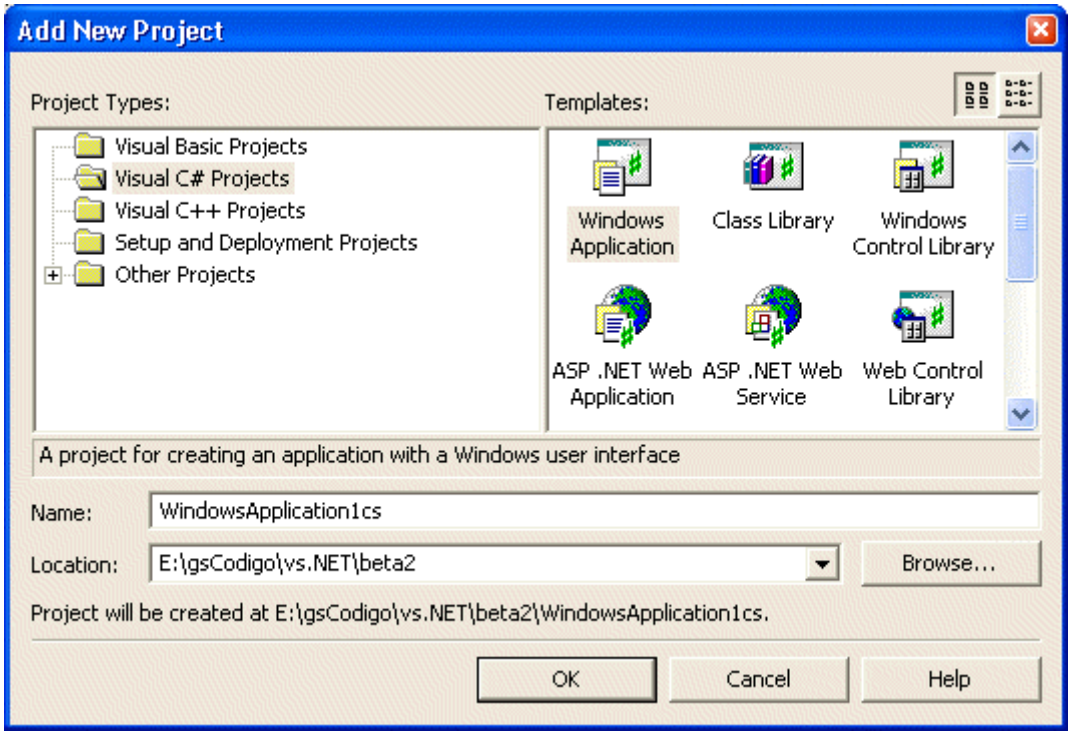



**Figura 9**

Se añadirá un nuevo proyecto al Explorador de soluciones, fíjate que la extensión del formulario es .cs en lugar de .vb que es la extensión que se usa en los ficheros de Visual Basic .NET.

Añade los mismos controles que en el formulario de Visual Basic y también cámbiale el nombre al botón cerrar.

Cámbiale el texto que muestra el formulario para que indique esto: **Form1 para C#**, ya sabes, selecciona el formulario y en la ventana de propiedades eliges la propiedad Text.

Haz doble click en el botón Mostrar y escribe este código en el procedimiento que te muestra:

**MessageBox.Show("Hola " + textBox1.Text);**

Fíjate que en lugar de usar MsgBox, aquí hemos usado MessageBox con la particularidad de que hay que especificar el método que queremos usar, en este caso: Show. El resto se usa igual que con VB, con la salvedad de que la concatenación de cadenas se hace usando el signo de suma (+) en lugar del ampersand (&), (eso mismo también podemos hacerlo con Visual Basic, pero **te recomiendo que uses el signo & en lugar de + cuando quieras *sumar* cadenas de caractes en VB**), y otro detalle: todas las instrucciones en C# deben acabar en punto y coma (;).

Esta función MessageBox también podemos usarla en nuestros proyectos de VB, en lugar de MsgBox, ya que es una clase que pertenece al .NET Framework y todas las clases del .NET Framework podemos usarlas en nuestros proyectos de Visual Basic .NET.

Ahora haz doble click en el botón cerrar y escribe esto otro: **this.Close();**

En C# no se utiliza **Me**, sino **this**, que para el caso viene a representar lo mismo: la clase en la que se usa.

Si pulsas F5, verás que funciona de igual forma que antes... tan igual porque es el mismo proyecto de VB, la razón es porque al tener varios proyectos en el panel del explorador de soluciones, tenemos que indicarle cual de los proyectos es el que se ejecutará al pulsar F5, para cambiarlo al de C#, selecciona el nuevo proyecto añadido y pulsando con el botón derecho, selecciona del menú desplegable: **Set as StartUp Project** (seleccionarlo como proyecto de inicio o algo parecido en la versión española)

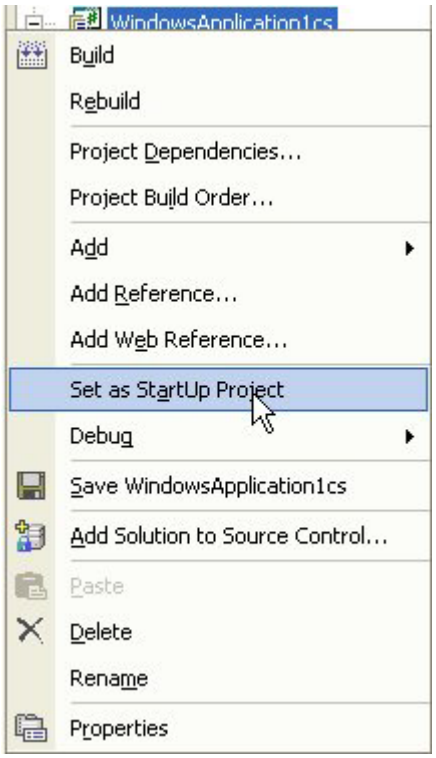



**Figura 10**

Pulsa F5 y se ejecutará ese proyecto, te darás cuenta que es el de c# por el caption del formulario, ya que el aspecto es idéntico al de VB, otra cosa por la que te darás cuenta de que no es el proyecto de VB es cuando pulses en el botón mostrar, el cuadro de diálogo no muestra el nombre de la aplicación, como ocurría con el proyecto de VB, (ver la figura 7), para eso ocurra, tendremos que decirle explícitamente que lo muestre:

MessageBox.Show("Hola " + textBox1.Text, **Application.ProductName**);

Esto mismo también es válido si queremos usar esa función desde Visual Basic.

Creo que lo vamos a dejar aquí.

Te resumo lo que hasta ahora hemos aprendido: (dice hemos, porque ***él*** también va experimentando mientras te explica, que conste, no sea que te creas que ***el Guille*** ha nacido sabiendo...)

- Crear un nuevo proyecto en Visual Basic y en C#

- Manejar varios proyectos a la vez, incluso de lenguajes distintos.
- Añadir controles a un formulario e interactuar con ellos mediante eventos.
- Codificar o decirle qué queremos que haga cuando esos eventos se produzcan.
- Usar la función MsgBox y MessageBox para mostrar cuadros de aviso.
- Concatenar (unir varias en una) cadenas de caracteres.
- Uso de la clase Application para mostrar el nombre de la aplicación.
- Uso de Me (o this en c#) para representar a la clase actual.
- Uso de las propiedades de los controles, en este caso la propiedad Text del control TextBox.
- Saber manejarte, aunque sólo sea un poco, por el entorno integrado del Visual Studio .NET
- Conceptos tales como: método, propiedad, evento, parámetro, parámetro opcional, de los cuales tienes una explicación o definición en la página del glosario, sitio al que te mandan los links que has encontrado en algunas de esas palabras.

# 3. Tercera entrega

En esta tercera entrega del curso de iniciación a la programación con Visual Basic .NET vamos a seguir viendo algunas cosillas más del entorno integrado de Visual Studio .NET, en esta ocasión vamos a hacer que los controles que tengamos en un formulario se adapten de forma automática al nuevo tamaño de la ventana (**formulario**) así como a los distintos tamaños de fuentes que el usuario de nuestras aplicaciones .NET prefiera usar, ya que no todo el mundo usa la misma resolución que nosotros ni los mismos tamaños de letras.

Hacer esto con las versiones anteriores de Visual Basic era un trabajo duro y algunas veces bastante engorroso, que requería bastante código y casi nunca se lograba lo que queríamos... ¿que no sabes de qué estoy hablando? bueno, no te preocupes que, aunque ahora no sepas la utilidad de todo lo que te voy a explicar pueda tener, en algún momento lo necesitarás y aquí tendrás la explicación de cómo hacerlo.

Para entrar en calor, te expondré un caso típico:

El tamaño de una ventana de Windows, (que al fin y al cabo es un formulario), se puede hacer redimensionable, es decir que el usuario puede cambiar de tamaño, en estos casos, lo adecuado sería que los controles que dicho formulario contenga, se adapten al nuevo tamaño de la ventana, con idea de que no queden huecos vacíos al cambiar el tamaño de la ventana.

Por ejemplo, si tenemos esta ventana (o formulario):

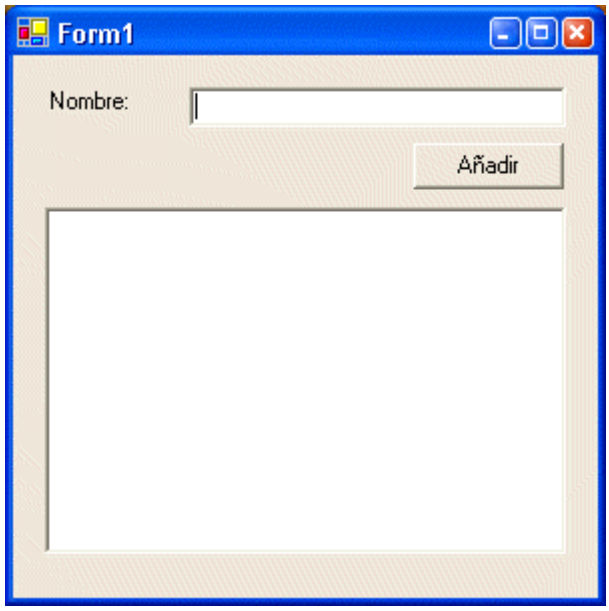



**Figura 1**

y la agrandamos, por ejemplo para que tenga este otro aspecto:

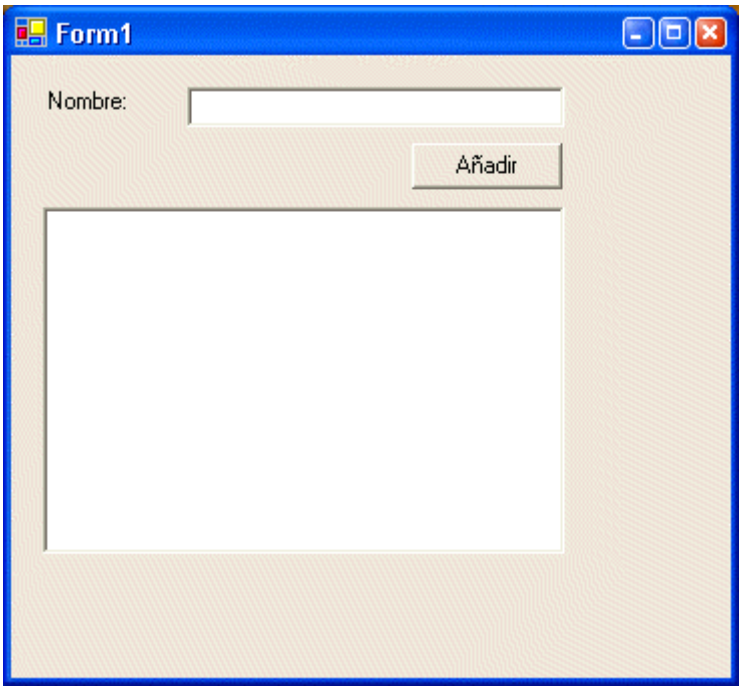



**Figura 2**

Comprobaremos que la ventana se ha agrandado, pero los controles que hay en ella siguen teniendo el mismo tamaño y la misma posición que en la ventana anterior.

Pues bien, la idea de lo que te quiero explicar es que al cambiar el tamaño de la ventana se ajusten los controles al nuevo tamaño, para que tuviesen este otro aspecto:

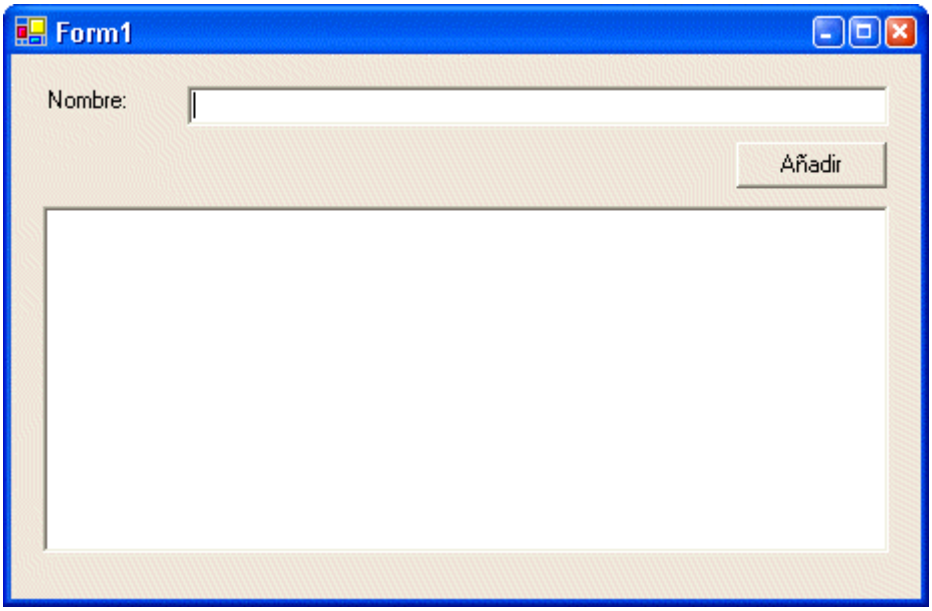



**Figura 3**

Para que esto sea posible de forma automática, hay que hacer unas cuantas asignaciones a los controles, de forma que podamos indicarle qué tienen que hacer cuando el tamaño de la ventana varíe.

En este ejemplo, lo correcto sería que:

- La caja de texto superior se agrandase hacia el lado derecho.

- El botón Añadir se moviese hacia el extremo derecho del formulario.
- La lista se ajustara al ancho y también al alto de la ventana.

Todo esto lo podemos hacer en tiempo de diseño, es decir cuando tenemos el formulario en el entorno integrado o bien lo podemos codificar dentro del propio formulario, dónde hacerlo queda a tu criterio, yo te voy a explicar cómo hacerlo en los dos casos y después tu decides cómo hacerlo.

Antes de empezar a explicarte, vamos a crear un nuevo proyecto.

Crea un proyecto del tipo Windows, (aplicación Windows o Windows Application), nómbralo como WinApp3.

Añade una etiqueta (Label), una caja de textos (TextBox), un botón (Button) y una lista (ListBox)

Deja los nombres que el IDE ha puesto, salvo para el botón, el cual se llamará cmdAdd.

(Realmente puedes ponerles los nombres que quieras, pero los que yo voy a usar son: Label1, TextBox1, ListBox1 y cmdAdd)

Posiciona esos controles tal y como se muestra en la figura 1.

Selecciona la caja de textos (TextBox1) y en la ventana de propiedades, selecciona **Anchor**, verás que por defecto estarán los valores Left y Top, esto quiere decir que la caja de textos estará "anclada" a la izquierda y arriba, pero ahora vamos a seleccionar también la derecha. Cuando pulses en la lista desplegable verás que se muestra una imagen con cuatro líneas, (como la mostrada en la figura 4), dos de las cuales son obscuras (están seleccionadas) y las otras dos están blancas (no seleccionadas), pulsa en la línea de la derecha, para que se ponga gris, de esta forma estaremos indicándole que también se "ancle" a la derecha.

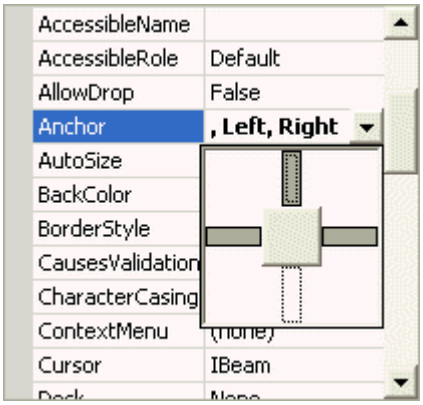



**Figura 4**

Vamos a comprobarlo. Pulsa F5 para ejecutar el proyecto.

Una vez que se muestre el formulario en ejecución, agrándalo hacia la derecha, verás que la caja de textos se adapta al nuevo tamaño. Ahora "achícalo", es decir haz que se haga más pequeño hacia la izquierda, incluso haz que no se vea el botón, comprobarás que la caja de texto sigue estando "proporcionalmente" igual de separada de los extremos superior, derecho e izquierdo y se adapta al tamaño de la ventana.

Incluso si intentas hacer la ventana muy pequeña, el ancho se quedará justo en la parte izquierda de la caja de texto, con el alto puedes hacer que casi desaparezca, (salvo el caption de la ventana, la barra de arriba, la cual se mantiene).

Ahora vamos a "anclar" el botón.

Una vez que hayas terminado de probar... vamos a seguir... ¡valeeee! deja ya de juguetear con la dichosa ventanita.

Ahora, de vuelta al IDE, selecciona el botón y en la ventana de propiedades selecciona la propiedad Anchor.

En este caso, lo que nos interesa es que el botón se desplace a la derecha, pero que no se haga más grande.

Para ello, debes seleccionar las líneas de la derecha y la de arriba.

Es decir: ánclate en la parte de arriba y en la derecha, de forma que si cambiamos el tamaño del formulario, el botón se desplazará a la derecha o a la izquierda, pero no cambiará de tamaño, como le ocurre al textbox.

Pulsa F5 y compruébalo. Agranda el formulario, (hacia la derecha) y achícalo, (hacia la izquierda), verás que el botón llegará casi a tocar el lado izquierdo del formulario y allí se quedará, no permitiendo que se haga más pequeño.

Por último vamos a anclar el listbox... ¿cómo? ¿que quieres intentarlo por tu cuenta? vale... me parece bien...

Sólo decirte que el listbox debe hacerse grande tanto hacia la derecha como hacia la izquierda e incluso cuando se estira el formulario desde la parte inferior, pero en la parte superior debe mantenerse en la misma posición.

Fíjate en la figura 3...

Bueno, espero que lo hayas conseguido. Si no es así, no te preocupes, ahora veremos la solución.

Vamos a complicar un poco más la cosa y vamos a añadirle otro botón. En este caso, dicho botón estará en la parte inferior derecha del formulario, será el botón cerrar y al pulsarlo hay que cerrar el formulario... ¿recuerdas cómo se hace?

¡Exacto! usando Me.Close en el evento Click de dicho botón, el cual yo voy a llamar cmdCerrar.
Como te decía, este botón se debería anclar en la parte inferior derecha, por tanto los valores que hay que asignar en Anchor son precisamente esos: Right y Botton (derecha y abajo).

Como habrás notado, con el Label1 no hay que hacer nada, ya que por defecto el tamaño se ajusta por la derecha y por abajo, por tanto se quedará en la misma posición... aunque realmente está anclada arriba y a la izquierda, que son los valores por defecto de la propiedad Anchor, por eso no es necesario asignarle nada.

Ahora vamos a ver cómo hacerlo mediante código... ¿que qué sentido tiene hacerlo por código? pues... esto... bueno, porque puede ser que quieras hacerlo... (je, je, ¡tan pillao guille!)

Haz que se muestre el formulario y haz doble-click en él, (no hagas doble-click en ninguno de los controles que tiene el formulario, sino en el propio formulario). Cuando estamos en el IDE y hacemos doble-click en el formulario se muestra el evento Form_Load que es el que se ejecuta cuando el formulario "se carga" en la memoria, justo antes de mostrarse, por tanto aquí es un buen sitio para hacer algunas "inicializaciones" o asignaciones que nuestro formulario necesite.

Por ejemplo, podemos limpiar el contenido de la lista, el de la caja de texto, etc. e incluso hacer las asignaciones para que los controles se queden "anclados" en la posición que nosotros le indiquemos.

Vamos a ver primero cómo se "declara" este evento, aunque el VB lo hace automáticamente por nosotros, es conveniente verlo para ir aclarando conceptos... que tal vez ahora no necesites, pero en un futuro casi seguro que te hubiese gustado haberlo sabido.

```
    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        Me.TextBox1.Text = ""
        Me.ListBox1.Items.Clear()
    End Sub
```

Lo primero que hay que notar es que **Handles** es la palabra que le indica al compilador de Visual Basic .NET qué evento es el que "manipula" o maneja este procedimiento. Siempre lo digo... o lo pienso, que los anglosajones (los que hablan inglés) lo tienen muchísimo más fácil que los que no hablamos la lengua de Shakespeare, ya que para ellos eso de Handles es una palabra que tiene sentido y precisamente quiere decir eso "maneja", manipula, se encarga de, etc. con lo cual tienen superclaro que es lo que quiere decir esa palabreja... Esto, (la declaración del Sub), se encarga de manejar el evento Load del objeto MyBase.

Aunque también hay que decirlo... algunos de esos "hablantes" de inglés, aún a pesar de tener el lenguaje (idioma) a su favor... no tienen ni repajolera idea de Basic... en fin... ese consuelo nos queda... así que, no te desanimes y tira "pa lante", que "pa trás" no hay que ir ni para coger carrerilla...

El objeto **MyBase** se refiere al objeto base del que se deriva el formulario, recuerda que en .NET todo está basado en objetos y en programación orientada a objetos y todo objeto se deriva de un objeto básico o que está más bajo en la escala de las clases... es decir, un formulario se basa en la clase **System.Windows.Forms.Form** y a esa clase es a la que hace referencia el objeto MyBase, mientras que **Me** se refiere a la clase actual, la que se ha derivado de dicha clase Form o por extensión a cualquier clase, como veremos en futuras ocasiones.

¿Cómo? ¿Que has acabado por liarte más? ¿Que no has captado lo que acabo de decir?

Pues lo siento por ti... pero no te lo voy a explicar mejor... simplemente déjalo estar y poco a poco acabarás por comprenderlo... (je, je, ¡que malo que soy algunas veces!)

Veamos ahora el código prometido para hacer que los controles se anclen al formulario de forma que se adapten al nuevo tamaño del mismo:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Me.TextBox1.Text = ""
    Me.ListBox1.Items.Clear()
    ' Asignar los valores para "anclar" los controles al formulario
    ' El TextBox1 se anclará a Arriba, Izquierda y Derecha
    TextBox1.Anchor = AnchorStyles.Top Or AnchorStyles.Left Or
AnchorStyles.Right
    ' El botón Añadir lo hará Arriba y a la derecha:
    cmdAdd.Anchor = AnchorStyles.Top Or AnchorStyles.Right
    ' El listbox lo hará en los cuatro vértices:
    ListBox1.Anchor = AnchorStyles.Top Or AnchorStyles.Left Or
AnchorStyles.Right Or AnchorStyles.Bottom
    ' El botón cerrar sólo lo hará a la derecha y abajo
    cmdCerrar.Anchor = AnchorStyles.Right Or AnchorStyles.Bottom
End Sub
```

Para ir terminando la presente entrega, vamos a ver lo otro que comenté al principio, que el formulario y los controles se adapten también a otros tamaños de fuentes, no a los que nosotros tenemos en nuestro equipo... ya que hay gente que por necesidades tienen que poner tamaños de fuentes más grandes e incluso más pequeñas... que

también hay quién tiene un monitor de 19 o 21 pulgadas y lo mismo tienen que usar letras de tamaños más pequeñín...

La propiedad que hace eso posible es **AutoScale**, esta propiedad sólo está disponible en los formularios y por defecto tiene el valor True (verdadero), por tanto los formularios, sin necesidad de que hagamos nada, se auto ajustarán al tamaño de las fuentes.

Esto no lo he comprobado, pero me fío de lo que dice la documentación, (aunque esté en inglés)

Otra cosilla interesante que tienen los formularios es la propiedad **AutoScroll**. Si asignamos el valor True (verdadero) a esta propiedad, hacemos que cuando el formulario se haga muy pequeño o muy estrecho, se muestren unas barras de desplazamiento (scrolls) para que pulsando en ellas podamos ver el contenido del mismo.

Por ejemplo si no hubiésemos "anclado" nuestros controles, al hacer el formulario más estrecho se mostrarían unas barras de desplazamiento para que podamos ver los controles que están contenidos en el formulario.

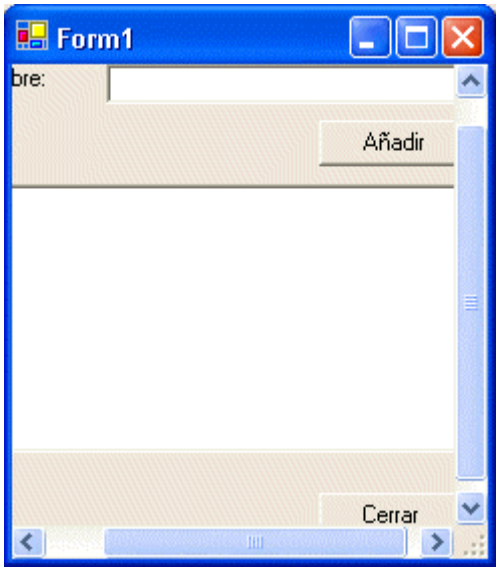



**Figura 5**

Si quieres probarlo, quita el código que hemos añadido, o coméntalo todo, para ello selecciona todas las líneas que quieres comentar, las que asignan los valores a la propiedad Anchor y en el menú Edición, selecciona Avanzado y Comentar Selección, también puedes usar las teclas: Ctrl+K seguidas de Ctrl+C, (yo tengo las opciones de Comentar Selección y Quitar los comentarios puestas en la barra de herramientas, ya que de vez en cuando las utilizo para hacer alguna que otra prueba y no tener que borrar el texto que quiero quitar)

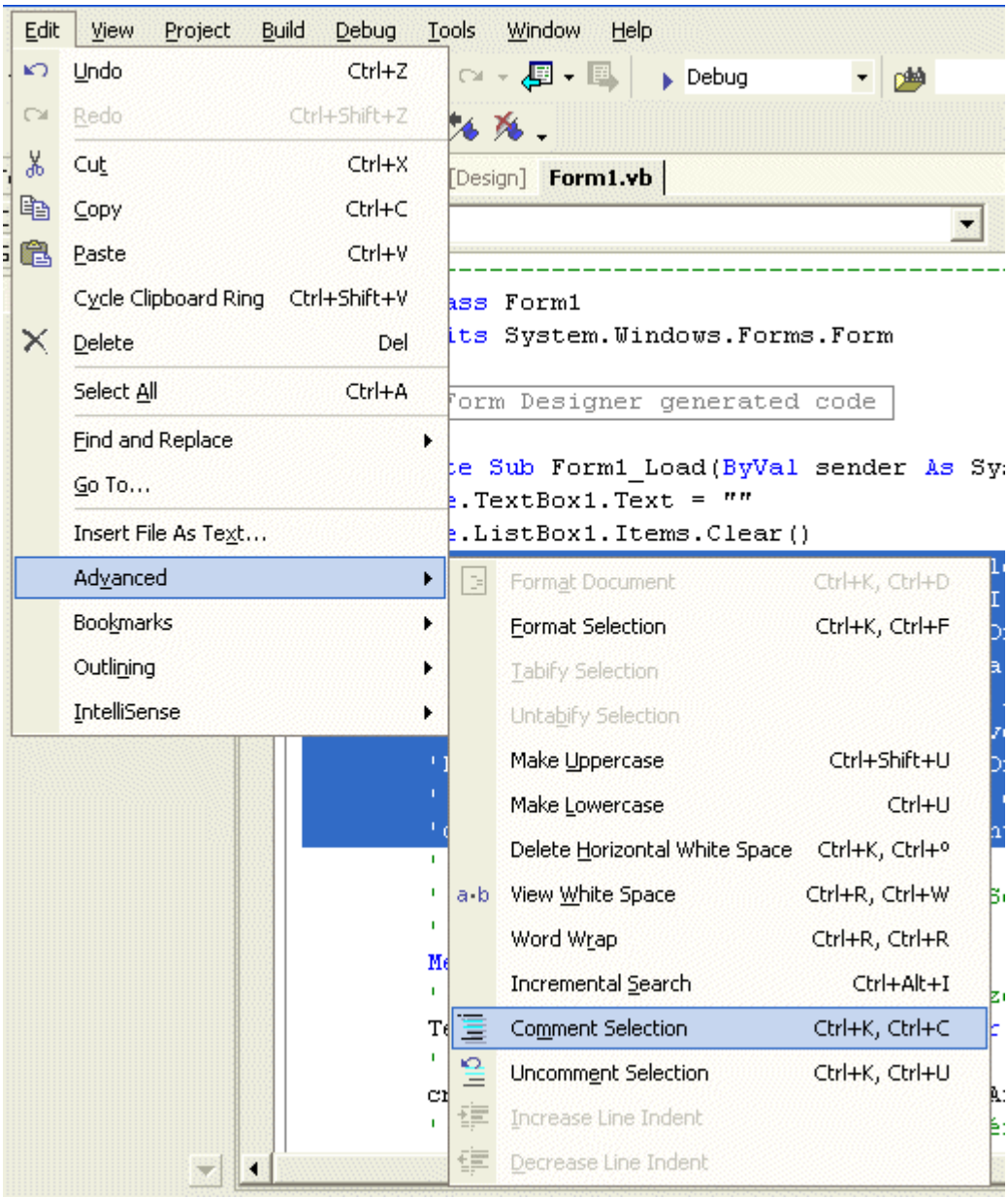



**Figura 6**

Lo dicho, comenta ese código y añade el siguiente, ejecuta la aplicación y pruébalo haciendo la ventana más pequeña, tal y como te muestro en la figura 5.

```
' Asignamos True a la propiedad AutoScroll y dejamos los controles
' como están por defecto:
Me.AutoScroll = True
' El TextBox1 se anclará a Arriba, Izquierda y Derecha
TextBox1.Anchor = AnchorStyles.Top Or AnchorStyles.Left
' El botón Añadir lo hará Arriba y a la derecha:
cmdAdd.Anchor = AnchorStyles.Top Or AnchorStyles.Left
' El listbox lo hará en los cuatro vértices:
ListBox1.Anchor = AnchorStyles.Top Or AnchorStyles.Left
' El botón cerrar sólo lo hará a la derecha y abajo
cmdCerrar.Anchor = AnchorStyles.Top Or AnchorStyles.Left
```

Todas estas cosas ya nos hubiese gustado tenerlas en las versiones anteriores de Visual Basic, ya que para hacerlo o bien te tenías que "comer" el coco o bien te tenías que crear un control que hiciera ese trabajo...

Sólo comentarte que los **Or**s que se están utilizando sirven para "sumar" y el resultado sería el mismo que si usáramos el signo de suma, pero la razón de usar **Or** es porque lo que queremos hacer es una suma de bits... realmente da lo mismo usar la suma que Or en este caso, pero... dejemos el Or que es lo apropiado... y no me preguntes porqué... ya que, aunque no te hayas enterado, te lo acabo de explicar... je, je.

¡UF! vaya entrega más larga y en resumidas cuentas ¿qué es lo que has aprendido?

Sí... claro que no ha sido en vano… ¿Te crees que iba a gastar yo tanto tiempo para explicarte algo que no sirve para nada...?

De todas formas, vamos a ver algo de código para que no se te quede mal sabor de boca.

El código que te voy a mostrar hará lo siguiente:

- Al pulsar en el botón Añadir, se añadirá a la lista lo que hayas escrito en la caja de textos.
- Al pulsar Intro será como si hubieses pulsado en el botón Añadir.
- Al pulsar Esc es como si hubieses pulsado en el botón Cerrar.
- Al pulsar en uno de los elementos de la lista, éste se mostrará en la caja de textos.
- Al seleccionar un elemento de la lista y pulsar la tecla Supr (o Del si tu teclado está en inglés), dicho elemento se borrará de la lista, pero se quedará en la caja de textos, ya que al seleccionarlo para poder pulsar la tecla suprimir se habrá mostrado...

Veamos cómo hacer esto.

Lo de pulsar en Añadir y hacer algo, está claro: simplemente codificamos lo que haya que codificar en el evento Click del botón cmdAdd. Para que nos muestre ese evento, simplemente haz doble-click en el botón y el VB te mostrará el evento en cuestión, añade este código y ahora te lo explico:

```
Private Sub cmdAdd_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles cmdAdd.Click
    ListBox1.Items.Add(TextBox1.Text)
End Sub
```

Lo que te voy a explicar es lo que está dentro del evento Click, ya que lo de Handles te lo he explicado hace un rato.

Si te has fijado en el código que te mostré del evento Form_Load, seguramente habrás visto que teníamos:

**Me.ListBox1.Items.Clear()**

**Me** hace referencia a la clase actual, es decir al formulario.

**Items** son los elementos que tiene el objeto ListBox

**Clear** es un método de Items que se encarga de limpiar los elementos de la lista, es decir: los borra.

Por tanto esa línea lo que hace es borrar los elementos del listbox.

Ahora lo que necesitamos no es borrarlos, sino añadir nuevos elementos a la lista, por tanto, como ya sabemos que **Items** es el sitio en el que se guardan los elementos de la lista, lo único que tenemos que saber es ¿cómo se añaden nuevos elementos a dicha

lista? y para eso estoy yo aquí: para contártelo; pero como me imagino que eres lo suficientemente observador, te habrás "percatado" que **Add** es lo que necesitamos para añadir elementos a la lista de un ListBox. Si no eres tan observador (u observadora) como yo me creía, te lo explico un poco:

***Para añadir elementos a un listbox, se usa el método Add de Items.***

Ya está dicho.

En la colección Items se puede añadir lo que queramos, cualquier objeto, en este caso lo que añadimos es el contenido (el texto) del TextBox, por eso es por lo que hacemos:

**Items.Add(TextBox1.Text)**

Añadimos a Items el contenido de la caja de textos.

## 3.1. ¿Qué es una colección?

Realmente es una lista de objetos o elementos que están agrupados en un objeto, en este caso un objeto colección (o Collection), aunque en este caso Items no es del todo lo que en el VB de antes era una colección, ya que en vb.NET existen varios tipos de colecciones o arrays (¿arreglos?), pero no vamos a complicarnos la vida... Items es una colección de elementos, es decir los elementos están guardados en una "lista" de datos.

En su momento veremos los diferentes tipos de colecciones o listas.

Veamos ahora cómo saber cual es el elemento que se ha seleccionado de la lista y cómo asignarlo al TextBox:

```
Private Sub ListBox1_SelectedIndexChanged(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles ListBox1.SelectedIndexChanged
    ' Cuando pulsamos en un elemento de la lista...
    With ListBox1
        TextBox1.Text = .GetItemText(.SelectedItem)
    End With
End Sub
```

Fíjate que en este caso no es el evento Click, como era de esperar (y como es en las versiones anteriores de VB, incluso como era en la Beta 1 de vb.NET), sino que el evento en cuestión es **SelectedIndexChanged**.

Ya te he comentado que lo que podemos añadir a los elementos del listbox son objetos, pero lo que a nosotros nos interesa mostrar es el "texto" de dicho elemento, ya que no nos interesa otra cosa, más que nada porque lo que hemos añadido son textos y no objetos... aunque, como ya te he comentado en otras ocasiones TODO lo que se maneja en .NET son objetos, incluso las cadenas de textos son objetos... pero... en fin... dejemos las cosas así por ahora.

Lo que en este evento hacemos es asignar a la caja de textos el texto del elemento seleccionado: la propiedad **SelectedItem** representa al elemento seleccionado y **GetItemText** es una función, (o método), que devuelve el texto (o la representación en formato texto del elemento indicado dentro de los paréntesis).

Este evento se consigue haciendo doble click en el listbox, pero...

## 3.2. ¿Cómo hacemos para escribir código en otros eventos?

Para poder usar otros eventos de un objeto, muestra la ventana de código, de la lista desplegable de la izquierda, selecciona dicho objeto y en la lista desplegable de la derecha elige el evento que quieras usar, en nuestro próximo ejemplo será el evento

KeyPress, tal y como se muestra en la figura 7, los eventos se distinguen por el rayo que se muestra a la izquierda del nombre del elemento...

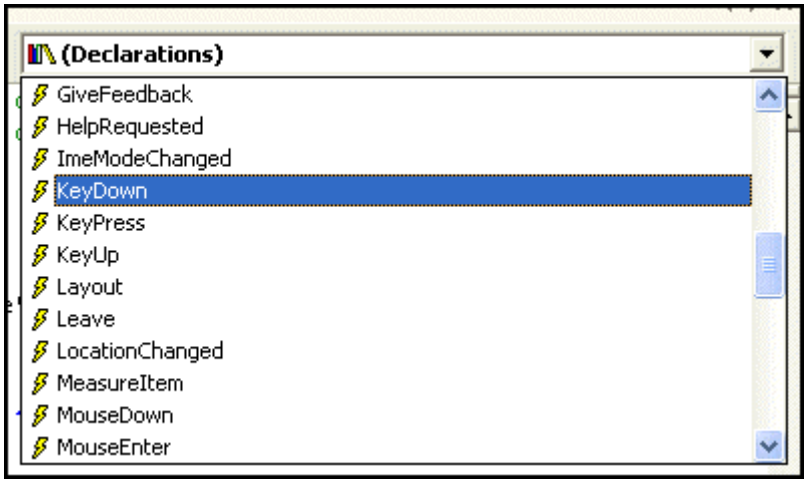



**Figura 7**

Como dato curioso, en los formularios de C# se pueden seleccionar los eventos en la misma ventana de propiedades del formulario u objeto en cuestión, incluso a veces ocurre que en vb.NET se muestra ese símbolo en la mencionada ventana de propiedades, aunque no siempre y no sé exactamente porqué se muestra... pero espero y confío que en la versión definitiva de vb.NET dicha opción estará disponible, para facilitarnos un poco más el trabajo. A día de hoy (26/Oct/2001), aún no he visto la Release Candidate 1, pero según he leído en las páginas de Microsoft no se diferencia mucho de la Beta 2, (que es la que estoy usando en estos momentos), así que... me da mala espina esto de que aún no esté la opción de elegir eventos en la ventana de propiedades, pero esperemos y confiemos que algún día esté.

**Nota del 24/Dic/2002:**
Pues ni está en la versión final ni tampoco estará en la versión 2003

Al seleccionar ese evento, se mostrará el siguiente código:

```
Private Sub ListBox1_KeyDown(ByVal sender As Object, ByVal e As
System.Windows.Forms.KeyEventArgs) Handles ListBox1.KeyDown

End Sub
```

**¿Por qué he elegido KeyDown y no KeyUp o incluso KeyPress?**

Porque la tecla Supr (Del) es una tecla especial y no se detecta en el evento KeyPress, podía haber usado KeyUp también, pero ese evento se produce cuando se suelta la tecla... (que sería lo lógico), pero yo siempre tengo la costumbre de interceptar esas pulsaciones en el evento KeyDown (cuando se empieza a pulsar la tecla) y es por ese motivo que voy a codificar el tema este de borrar un elemento de la lista en ese evento... cosas mías.

Este sería el código a usar para eliminar el elemento que está seleccionado:

```
Private Sub ListBox1_KeyDown(ByVal sender As Object, ByVal e As
System.Windows.Forms.KeyEventArgs) Handles ListBox1.KeyDown
    If e.KeyCode = Keys.Delete Then
        With ListBox1
            .Items.Remove(.SelectedItem)
```

```
        End With
    End If
End Sub
```

Es decir, comprobamos si la tecla en cuestión es la de suprimir, si es así, eliminamos el elemento que está seleccionado.

Recuerda que **SelectedItem** nos indica el elemento que actualmente está seleccionado y usando el método **Remove** de la colección Items, lo quitamos de la lista.

Así de fácil.

Pero como no estoy dispuesto a terminar aún esta tercera entrega... para que te empaches y no te entre el mono cuando veas que tardo en publicar la siguiente entrega... vamos a permitir múltiple selección de elementos y vamos a ver cómo borraríamos los elementos que estén seleccionados.

Cuando permitimos múltiple selección en un ListBox, podemos seleccionar un elemento o varios.
Si son varios, estos pueden estar consecutivos o no.

Por ejemplo, si seleccionas un elemento de la lista y manteniendo pulsada la tecla Shift (mayúsculas), pulsas en otro que está más arriba o más abajo, se seleccionan todos los elementos intermedios, (esto deberías saberlo, ya que es una cosa habitual de Windows); también puedes seleccionar elementos no contiguos, si pulsas la tecla Control y con el ratón vas haciendo click en elementos no consecutivos.

Lo primero que necesitamos saber es:

**¿Cómo hacer que un ListBox permita múltiple selección?** ya que por defecto sólo se puede seleccionar un elemento a un mismo tiempo.

Para que un ListBox permita múltiple selección de elementos, hay que asignar a la propiedad **SelectionMode** el valor **MultiExtended**, por tanto selecciona el ListBox y en la ventana de propiedades, asigna dicho valor a la propiedad SelectionMode.

Ahora tenemos que hacer más cosas cuando se detecta la pulsación de la tecla suprimir en el evento KeyDown, ya que tenemos que saber qué elementos están seleccionados para poder borrarlos.

Lo primero que tenemos que hacer es recorrer todos los elementos del ListBox para saber si está o no seleccionado, pero ese recorrido hay que hacerlo desde atrás hacia adelante... ¿por qué? porque si lo hiciéramos desde el principio de la lista, al eliminar un elemento de dicha lista, el número de elementos variaría y tendríamos problemas cuando llegásemos al final, ya que no será el mismo número de elementos después de haber borrado alguno... mientras que al recorrer los elementos desde el final hacia adelante, no importará que borremos alguno del final, ya que el siguiente que comprobaremos estará más al principio que el recién borrado y no tendremos problemas... sé que no te has enterado, pero no importa, confía en mi, (¡que remedio te queda!), y ya tendrás tiempo de comprobarlo por tu cuenta.

Veamos primero el código que habría que usar y después lo comento.

```
Private Sub ListBox1_KeyDown(ByVal sender As Object, ByVal e As
System.Windows.Forms.KeyEventArgs) Handles ListBox1.KeyDown
    If e.KeyCode = Keys.Delete Then
        'Borrar las palabras seleccionadas del listbox
        Dim i As Integer
        '
        With ListBox1
            For i = .SelectedItems.Count - 1 To 0 Step -1
                .Items.Remove(.SelectedItems.Item(i))
```

```
            Next
        End With
    End If
End Sub
```

La parte del **If e.KeyCode = Keys.Delete Then** ya la vimos antes, aunque no te expliqué qué es lo que significa esto... bueno que es lo que significa si que te lo dije, lo que no te expliqué es porqué eso sirve para saber "qué es lo que significa".

Sólo decirte que IF... THEN... sirve para hacer comparaciones o para comprobar si una o más cosas están relacionadas de alguna forma, en este caso, queremos saber si el valor de e.KeyCode, que es la tecla que has pulsado, es igual a Keys.Delete, éste último valor es un valor "predefinido" que representa a la tecla suprimir.

Por tanto, si la tecla que hemos pulsado es igual a la tecla suprimir, entonces hacer lo que viene a continuación...¿todo lo que viene a continuación?

No, sólo hasta que encontremos **End If**, pero de esto ya hablaremos en otra ocasión.

**Dim i As Integer** esto le indica al VB que vamos a usar un número y que ese número lo guarde en la *variable* i, de esto también tendrás ocasión de enterarte mejor, por ahora dejémoslo así.

**With ListBox1** el With se utiliza para simplificarnos las cosas y lo que viene a significar es que donde se tendría que usar el objeto ListBox1, se ponga un punto... más o menos... ahora veremos cómo se escribiría el código sin usar el With ListBox1.

¿Qué? ¿Te estás liando? ¿Sí? Pues no desesperes, que aunque todo esto sea un follón, dentro de un par de años acabarás por comprenderlo... je, je... no, que no... que no será tanto tiempo, confía en el Guille...

**For i = .SelectedItems.Count - 1 To 0 Step -1** con esto le estamos indicando que use la *variable* i para ir guardando los valores que resulten de contar desde el número de elementos que hay seleccionados hasta cero. El Step -1 se usa para contar hacia atrás, (de mayor a menor), pero eso, al igual que el For, también lo veremos más adelante.

SelectedItems es una colección en la cual están los elementos que hay seleccionados en el ListBox.

¿Recuerdas la colección Items? Pues en este caso, SelectedItems son los elementos seleccionados y para que lo sepas, todas las colecciones suelen tener un método **Count** que nos dice el número de elementos que hay en ellas.

**.Items.Remove(.SelectedItems.Item(i))** Esto es parecido a lo visto antes, es decir, elimina el elemento que está seleccionado y que ocupa la posición **i** dentro de la colección de elementos seleccionados.

Esto también lo podríamos haber escrito de esta otra forma:

**.Items.Remove(.SelectedItems(i))** que para el caso es lo mismo.

En su momento veremos porqué.

Por ahora quédate con la copla de que Item es la propiedad o método predeterminado de la colección SelectedItems.

**Next** indica que continúe el bucle o la cuenta que estamos llevando con la variable i.

De esta forma, al haber usado el Step -1, lo que hacemos es contar hacia atrás y si por ejemplo i valía 3, al llegar aquí, valdrá 2, es decir restamos 1: el valor indicado en Step

**End With** indica hasta dónde llega el tema ese del With ListBox1

**End If** le dice que hasta aquí llega la comprobación que hicimos de si la tecla pulsada era la de suprimir.

¡Menudo embrollo!

¿De verdad piensas que no acabarás nunca aprendiendo?

Date una oportunidad y de paso dámela a mi también... y confía, que dentro de poco todo esto te parecerá una tontería.

Vamos a ver qué significa eso de With.

Si no hubiésemos usado el With, ese código tendríamos que haberlo escrito de esta forma:

```
For i = ListBox1.SelectedItems.Count - 1 To 0 Step -1
    ListBox1.Items.Remove(ListBox1.SelectedItems.Item(i))
Next
```

Es decir, tendríamos que haber escrito el nombre del objeto en cada una de las partes correspondientes.

Por tanto, si usamos **With Objeto**, podemos sustituir a **Objeto** por el punto, siempre y cuando ese punto, (y la propiedad o método correspondiente), esté dentro del par With... End With.

Seguramente preferirás usarlo de esta otra forma, ya que se ve "claramente" que es lo que estamos haciendo, pero con el tiempo te acostumbrarás a usar With/End With, entre otras cosas porque yo los uso bastante y al final será como de la familia...

Y ahora sí que hemos terminado... ¡por fin!

(Pst! Guille... se te ha olvidado explicar lo de que al pulsar Intro sea como si hubiese pulsado en el botón Añadir y al pulsar Esc como si hubiese pulsado en Cerrar)

Pues sí... ¿en que estaría pensando?

Por suerte para tus neuronas, eso es más fácil de asimilar... je, je, je.

Para que un botón intercepte la tecla Intro, hay que decirle al VB que ese botón es el botón de aceptar, lo mismo ocurre con la "captura" de la tecla ESC, pero en lugar de ser el botón por defecto, será el botón de cancelación.

Selecciona el formulario y en la ventana de propiedades busca la propiedad **AcceptButton**, habrá una lista desplegable con los botones disponibles en el formulario, (tal y como te muestro en la figura 8), selecciona cmdAdd y ya está.

Ahora al pulsar Intro es lo mismo que si pulsaras en el botón Añadir.

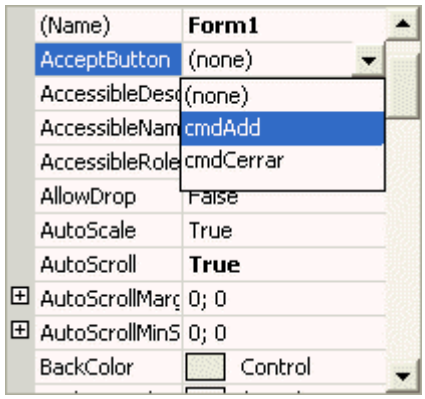



**Figura 8**

Lo mismo para el de cancelar, pero en esta ocasión, la propiedad en cuestión es: **CancelButton**.

Por tanto, selecciona de la lista desplegable el botón cmdCerrar y así al pulsar Esc se cerrará la aplicación.

**Nota:** En las versiones anteriores de VB, esas propiedades pertenecían al propio botón, es decir, eran propiedades del objeto CommandButton. Aunque, al menos para mí, es más lógico que estén en el formulario, entre otras cosas porque sólo puede haber un botón de aceptar y uno de cancelar.

Pruébalo y así lo comprobarás.

Escribe algo en la caja de textos y pulsa Intro, verás que se añade a la lista, después pulsa ESC y verás que se cierra el formulario o ventana con lo que se da por terminada la aplicación.

Ahora sí. Ya hemos terminado por hoy.

La próxima vez veremos con más detalles qué es eso de las variables y puede que incluso veamos algo de lo que ya hemos visto aquí, pero con más detalles y mejor explicado. Pero eso será en la próxima entrega, que esta ya está bien servida.

Sólo me queda hacerte un "pequeño" resumen de lo que hemos visto en esta entrega:

- Usar la propiedad Anchor de los controles para que se ajusten automáticamente al cambiar el tamaño del formulario en el que están contenidos.
- Usar la propiedad AutoScale de los formularios para que los controles se ajusten al tamaño de la fuente predeterminada de Windows.
- Usar la propiedad AutoScroll de los formularios para mostrar las barras de desplazamiento (scroll) cuando el formulario se hace más pequeño y alguno de los controles queda oculto.
- Mención y pequeña descripción/explicación de las palabras MyBase, Me, Or, With, If, For...
- También hemos visto, aunque sea de pasada una variable para usarla en un bucle For.
- Qué es una colección, así como los métodos Clear, Count, Add, Remove e Item de las colecciones.
- Cómo asignar a un ListBox el contenido de un TextBox y viceversa:
- Cómo asignar a un TextBox el contenido del elemento seleccionado de un ListBox.
- Cómo saber cual es el elemento seleccionado de un ListBox.
- Cómo hacer que un ListBox permita múltiples selecciones de elementos.
- Cómo saber cuantos y cuales son los elementos que están seleccionados en el ListBox.
- Cómo eliminar el elemento que está actualmente seleccionado y
- Cómo eliminar todos los elementos que están seleccionados.
- Cómo hacer que un botón sea el botón predeterminado de aceptar de un formulario.
- Cómo hacer que un botón sea el botón predeterminado de cancelar de un formulario.
- Cómo seleccionar cualquier evento para un control determinado.
- Eventos vistos: Form_Load, Click, KeyDown, SelectedIndexChanged

# 4. Cuarta entrega

Como te dije en la entrega anterior de este curso de iniciación a la programación con Visual Basic .NET, en esta ocasión vamos a ver qué es eso de una variable y algunos otros conceptos que necesitarás para poder tomarte esto de programar con vb.NET un poco más en serio... bueno, tampoco hace falta que sea demasiado en serio, simplemente que te enteres de las cosas y sobre todo que sepas aplicarlas en el momento en que sea oportuno o necesario... (que bien ta quedao ezo Guille)

## 4.1. Variables, constantes y otros conceptos relacionados

El concepto o descripción de lo que es una variable es fácil de asimilar, si es que has estudiado en el colegio, al menos hace unos diez años o más, ya se explicaba, pero hace 25 ó 30 años, por poner una cifra, a mi no me lo explicaron... o lo mismo el "profe" lo explicó, pero yo estaría pensando en las *altabacas de Rio Seco*, (otras cosas)... como siempre... que lo que es prestar atención en clase, no era lo mío... (así te va Guille), en fin... no tomes malos ejemplos y aplícate... al menos en esta "clase", en las otras... de ti depende... no te voy a echar el sermón de que *debes estudiar para que en el futuro*... eso lo dejo para otros...

A lo que vamos... el concepto de constante, ya lo vimos hace un par de entregas, una constante es algo que permanece inalterable, por eso se llama constante, porque siempre es constante: inalterable, siempre es lo mismo...

Sin embargo una variable puede alterarse, es decir, se le puede cambiar el valor que tiene... por eso se llama variable, como el estado anímico que algunas veces tenemos, hoy estamos de buenas y mañana lo mismo tenemos los cables cruzados y no hay Dios que nos pueda dirigir la palabra... en fin... con una tila se suele solucionar... pero... no es de eso de lo que se trata cuando hablamos de variables en esto de la programación, ya sea en vb.NET o en cualquier otro lenguaje.

Las variables son "nombres" que pueden contener un valor, ya sea de tipo numérico como de cualquier otro tipo.

Esos nombres son convenciones que nosotros usamos para facilitarnos las cosas, ya que para los ordenadores, (o computadores, según te guste o estés acostumbrado a llamarlos), una variable es una dirección de memoria en la que se guarda un valor o un objeto, te vuelvo a recordar por enésima vez que en .NET todo es un objeto.

Existen distintos tipos de valores que se pueden asignar a una variable, por ejemplo, se puede tener un valor numérico o se puede tener un valor de tipo alfanumérico o de cadena, (string para los que inventaron esto de los lenguajes de programación), pero en cualquier caso, la forma de hacerlo siempre es de la misma forma... al menos en .NET ya no hay las distinciones que antes había en las versiones anteriores de Visual Basic... no voy a entrar en detalles, pero si has trabajado anteriormente en VB con objetos, sabrás de que estoy hablando.

Por ejemplo si queremos guardar el número 10 en una variable, haremos algo como esto:

**i = 10**

En este caso **i** es la variable, mientras que 10 sería una constante, (10 siempre vale 10), la cual se asigna a esa "posición" de memoria a la que llamamos i, para facilitarnos las cosas... ya que, realmente no nos interesa saber dónde se guarda ese valor, lo único que nos interesa es saber que se guarda en algún lado para en cualquier ocasión poder volver a usarlo.

Pensarás, que tal y como están las cosas, i también vale 10, por tanto ¿por qué no es una constante? por la sencilla razón de que podemos alterar su valor, por ejemplo, si en cualquier ocasión posterior hacemos esto: i = 25, el valor de la variable i cambiará, de forma que el valor anterior se esfumará y el que se almacenará será el nuevo.

También podemos aplicar expresiones al asignar una variable, una expresión es algo así como un cálculo que queremos hacer, por ejemplo: i = x * 25, en este caso x * 25 se dice que es una expresión, cuyo resultado, (el resultante de multiplicar lo que vale la variable x por la constante 25), se almacenará en la variable i.

Si x vale 3, (es decir el valor de la variable x es tres), el resultado de multiplicarlo por 25, se guardará en la variable i, es decir i valdrá 75.

Pero no es suficiente saber qué es una variable, lo importante es saber cómo decirle al vb.NET que queremos usar un espacio de memoria para almacenar un valor, ya sea numérico, de cadena o de cualquier otro tipo.

Para que vayas entrando en calor, te diré que las cadenas de caracteres (o valores alfanuméricos) se representan por algo que está contenido dentro de comillas dobles: "hola" sería una constante de cadena, ya que "hola" será siempre "hola", lo mismo que el número 10 siempre vale 10.

Para asignar esa constante de caracteres a una variable, se haría algo como esto:

s = "Hola"

De esta forma, la variable s contiene el valor *constante* "Hola".

Podemos cambiar el valor de s, asignándole un nuevo valor: s = "adiós", pero no podemos cambiar el valor de "Hola", ya que si lo cambiamos dejará de ser "Hola" y se convertirá en otra cosa...

Como ya te he dicho, existen distintos tipos de datos que vb.NET maneja, para que podamos usar una variable para almacenar cualquiera de esos tipos, tenemos que decirle al VB que "reserve" espacio en la memoria para poder guardarlo.

Esto se consigue mediante la "**declaración de variables**", es necesario, aunque no obligatorio, declarar las variables según el tipo de datos que va a almacenar.

Por ejemplo, en el caso anterior, la variable **i** era de tipo numérico y la variable **s** era de tipo cadena. Esas variables habría que declararlas de la siguiente forma: (después veremos otras formas de declarar las variables numéricas)

**Dim i As Integer**

**Dim s As String**

Con esto le estamos diciendo al vb.NET que reserve espacio en su memoria para guardar un valor de tipo Integer, (numérico), en la variable **i** y que en la variable **s** vamos a guardar valores de cadena de caracteres.

Antes de seguir con esta "retahíla" de conceptos, vamos a ver cuales son los tipos de datos que .NET soporta y esas cosillas, así veremos los tipos de variables que podemos tener en nuestros programas.

La siguiente tabla te muestra algunos de ellos y los valores mínimos y máximos que puede contener, así como el tamaño que ocupa en memoria; también te comento algunas otras cosas, que aunque ahora no te parezcan "aclaratorios", en un futuro si que lo serán, como por ejemplo a que tipo de datos se puede convertir sin recibir un mensaje **overflow**, (o los que se aceptan usando **Option Strict**), que signo se puede usar para "aclarar" el tipo de datos que representa, e incluso que signo se puede usar con variables

para que el VB sepa el tipo de datos que es... (Aunque esto último no es recomendable, lo muestro para que lo sepas):

## 4.2. Tipos de datos de Visual Basic.NET y su equivalente en el Common Language Runtime (CLR)

| Tipo de Visual Basic | Tipo en CLR (Framework) | Espacio de memoria que ocupa | Valores que se pueden almacenar y comentarios |
|---|---|---|---|
| **Boolean** | **System.Boolean** | 2 bytes | Un valor verdadero o falso.<br>Valores: **True** o **False.**<br>En VB se pueden representar por -1 o 0, en CLR serán 1 y 0, aunque no es recomendable usar valores numéricos, es preferible usar siempre True o False.<br>Dim b As Boolean = True |
| **Byte** | **System.Byte** | 1 byte | Un valor positivo, sin signo, para contener datos binarios.<br>Valores: de 0 a 255<br>Puede convertirse a: **Short**, **Integer**, **Long**, **Single**, **Double** o **Decimal** sin recibir overflow<br>Dim b As Byte = 129 |
| **Char** | **System.Char** | 2 bytes | Un carácter Unicode.<br>Valores: de 0 a 65535 (sin signo).<br>No se puede convertir directamente a tipo numérico.<br>Para indicar que una constante de cadena, realmente es un Char, usar la letra C después de la cadena:<br>Dim c As Char = "N" |
| **Date** | **System.DateTime** | 8 bytes | Una fecha.<br>Valores: desde las 0:00:00 del 1 de Enero del 0001 hasta las 23:59:59 del 31 de Diciembre del 9999.<br>Las fechas deben representarse entre almohadillas # y por lo habitual usando el formato norteamericano: #m-d-yyyy#<br>Dim d As Date = #10-27-2001# |
| **Decimal** | **System.Decimal** | 16 bytes | Un número decimal.<br>Valores:<br>de 0 a +/- 79,228,162,514,264,337,593,543,950,335 sin decimales;<br>de 0 a +/- 7.9228162514264337593543950335 con 28 lugares a la derecha del decimal;<br>el número más pequeño es: +/-0.0000000000000000000000000001 (+/-1E-28).<br>En los literales se puede usar la letra D o |

| | | | |
|---|---|---|---|
| | | | el signo @ para indicar que el valor es Decimal.<br><br>Dim unDecimal As Decimal = 9223372036854775808D<br><br>Dim unDecimal2 As Decimal = 987654321.125@ |
| **Double** | **System.Double** | 8 bytes | Un número de coma flotante de doble precisión.<br>Valores:<br>de -1.79769313486231570E+308 a -4.94065645841246544E-324 para valores negativos;<br>de 4.94065645841246544E-324 a 1.79769313486231570E+308 para valores positivos.<br><br>Se puede convertir a **Decimal** sin recibir un overflow.<br>Se puede usar como sufijo el signo almohadilla # o la letra R para representar un valor de doble precisión:<br><br>Dim unDoble As Double = 125897.0235R<br><br>Dim unDoble2 As Double = 987456.0125# |
| **Integer** | **System.Int32** | 4 bytes | Un número entero (sin decimales)<br>Valores:<br>de -2,147,483,648 a 2,147,483,647.<br><br>Se puede convertir a **Long**, **Single**, **Double** o **Decimal** sin producir overflow.<br><br>Se puede usar la letra I o el signo % para indicar que es un número entero:<br><br>Dim unEntero As Integer = 250009I<br><br>Dim unEntero2 As Integer = 652000% |
| **Long** (entero largo) | **System.Int64** | 8 bytes | Un entero largo (o grande)<br>Valores:<br>de -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807.<br><br>Se puede convertir a **Single**, **Double** o **Decimal** sin producir overflow.<br><br>Se puede usar la letra L o el signo & para indicar que es un número Long:<br><br>Dim unLong As Long = 12345678L<br><br>Dim unLong2 As Long = 1234567890& |
| **Object** | **System.Object** (class) | 4 bytes | Cualquier tipo se puede almacenar en una variable de tipo **Object**.<br>Todos los datos que se manejan en .NET están basados en el tipo Object. |
| **Short** | **System.Int16** | 2 bytes | Un entero corto (sin decimales) |

| | | | |
|---|---|---|---|
| (entero corto) | | | Valores:<br>de -32,768 a 32,767.<br>Se puede convertir a: **Integer**, **Long**, **Single**, **Double** o **Decimal** sin producir un overflow.<br>Se puede usar la letra S para indicar que es un número entero corto:<br>Dim unShort As Short = 32000S |
| **Single** | **System.Single** | 4 bytes | Número de coma flotante de precisión simple.<br>Valores:<br>de -3.4028235E+38 a -1.401298E-45 para valores negativos;<br>de 1.401298E-45 a 3.4028235E+38 para valores positivos.<br>Se puede convertir a: **Double** o **Decimal** sin producir overflow.<br>Se pueden usar la letra F y el símbolo ! para indicar que es un número Single:<br>Dim unSingle As Single = 987.125F<br>Dim unSingle2 As Single = 65478.6547! |
| **String** (cadenas de longitud variable) | **System.String** (clase) | Depende de la plataforma | Una cadena de caracteres Unicode.<br>Valores:<br>de 0 to aproximadamente 2 billones (2^31) de caracteres Unicode.<br>Se puede usar el símbolo $ para indicar que una variable es un String. |
| **Tipos definidos por el usuario** (estructuras) | (heradada de **System.ValueType**) | Depende de la plataforma | Cada miembro de la estructura tiene su rango, dependiendo del tipo de dato que representa. |

En la tabla anterior tienes los tipos de datos que podemos usar en vb.NET y por tanto, de los que podemos declarar variables.

Por ejemplo, si queremos tener una variable en la que guardaremos números enteros, (sin decimales), los cuales sabemos que no serán mayores de 32767 ni menores de -32768, podemos usar el tipo Short:

Dim unShort As Short

Después podemos asignar el valor correspondiente:

unShort = 15000

## 4.3. Sobre la necesidad u obligatoriedad de declarar las variables:

He de aclarar que el Visual Basic no "obliga" a que se declaren todas las variables que vayamos a usar.

Existe una instrucción, (**Option Explicit**), que gracias a las fuerzas de la Naturaleza, (por no decir gracias a Dios, que hay mucho *ateo* por ahí suelto), ahora viene puesta por defecto; la cual si que obliga a que declaremos las variables, pero si quitamos esa instrucción, entonces podemos hacer ***perrerías*** y declarar las variables si nos da la gana, etc...

**¡Que bien!** estarás diciendo... **¡la libertad total!**

¡Pues no! tanto anarquismo no es bueno... porque después te acostumbras y ¿que pasa? palos vienen, palos van... y no me refiero a palos físicos... sino a los palos que te dará el programa por no haber sido un poco más "conservador"... (espero que tantos símiles políticos no te alteren...)

La cuestión es que **siempre debes declarar las variables,** e incluso te diría más: **siempre debes declarar las variables del tipo que quieres que dicha variable contenga**, lo resalto porque esto te evitará quebraderos de cabeza... ¡créeme!

Ya que, puedes declarar variables sin tipo específico:

Dim unaVariable

que en realidad es como si la hubieses declarado del tipo Object, (As Object), por tanto aceptará cualquier tipo de datos, pero esto, acuérdate de lo que te digo, no es una buena práctica.

Si quieres llevarte bien conmigo, declara siempre las variables.

De todas formas, voy a decirte cómo hacer que no tengas que declarar las variables, por si eres *masoquista* y te gusta sufrir... de camino, también te voy a decir dónde está la otra opción que te pido encarecidamente que siempre uses, me refiero a la opción **Option Strict**, ésta opción, si se activa, se indica con **Option Strict On**, obligará a que los tipos de datos que uses sean del tipo adecuado, de esta forma, aunque es un verdadero "*peñazo*", (por no decir *coñazo*, ya que a algunos les puede parecer improcedente), hará que las cosas las hagas o las programes cómo debes... seguramente acabarás dejando esa opción en el valor que el vb.NET trae por defecto... valor que no creo que cambien en la versión definitiva del Visual Studio .NET, cosa que me agradaría un montón... (en este caso no se si el *masoca* soy yo, ya que usando Option Strict On es más complicado hacer las conversiones entre tipos diferentes de datos, pero...)

Por ejemplo, con el Option Strict On no podemos hacer esto:

**Dim unChar As Char = "N"**, ya que "N" es una constante del tipo String.

El compilador de Visual Basic nos diría algo así:

**Option Strict no permite la conversión entre Char y String**

Ahora veremos algunos ejemplos.

Primero vamos a ver dónde se puede cambiar esas opciones, así como alguna otra más, que en su momento también veremos.

Para acceder a las propiedades del proyecto, debes seleccionar el proyecto en la ventana del explorador de proyectos, una vez seleccionado, puedes usar el botón derecho del ratón y del menú que te muestra, seleccionar Propiedades... o bien, en el menú Proyecto, seleccionar Propiedades... (recuerda que si no está seleccionado el proyecto en el explorador de proyectos, ese menú no mostrará la opción de Propiedades).

Te mostrará una ventana de configuración, con varias opciones, selecciona Build del panel izquierdo y en el derecho te mostrará lo mismo que en la figura 1:

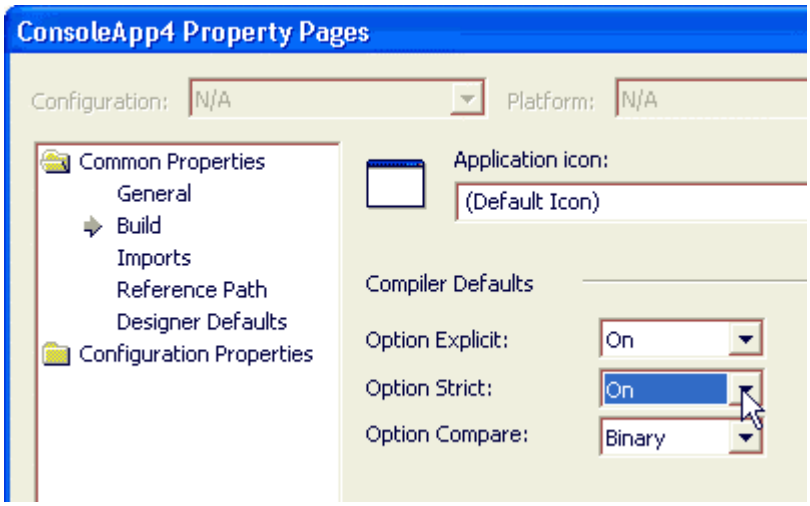



**Figura 1**

Estos son los valores que yo te recomiendo. También te los recomiendan otros muchos programadores, a los cuales parece ser que Microsoft no tiene tan en cuenta como dice... en fin...

Lo del **Option Compare** dependerá, de si quieres que las cadenas se comparen diferenciando las mayúsculas de las minúsculas o no.

Con el valor **Binary** se diferencian las mayúsculas de las minúsculas y con el otro valor: **Text**, no se hace ningún tipo de distinción, cosa que en algunas ocasiones puede resultar útil, pero que mediante código se puede solventar.

Por tanto, te pido, te ruego, te suplico, (hasta me pongo de rodillas si es necesario, para que me hagas caso), que siempre tengas asignado el valor ON tanto en Option Explicit como en Option Strict.

Gracias.

Después de las recomendaciones y súplicas, a las que espero que hagas caso, sigamos con esto de las declaraciones de las variables, e incluso de las constantes, ya que también podemos declarar constantes.

Las variables se pueden declarar de dos formas, aunque básicamente es lo mismo:

1- Declarando la variable y dejando que VB asigne el valor por defecto.

2- Declarando la variable y asignándole el valor inicial que queramos que tenga.

Por defecto, cuando no se asigna un valor a una variable, éstas contendrán los siguientes valores, dependiendo del tipo de datos que sea:

- Las variables numéricas tendrán un valor CERO.
- Las cadenas de caracteres una cadena vacía: ""
- Las variables Boolean un valor False (recuerda que False y CERO es lo mismo)
- Las variable de tipo Objeto tendrán un valor Nothing, es decir nada, un valor nulo.

Por ejemplo:

```
Dim i As Integer
```

Tendrá un valor inicial de 0

Pero si queremos que inicialmente valga 15, podemos hacerlo de cualquiera de estas dos formas:

1.)Dim i As Integer

i = 15

2.) Dim i As Integer = 15

Esta segunda forma es exclusiva de la versión .NET de Visual Basic, (también de otros lenguajes, pero es algo nuevo para los que tratamos con VB).

Mientras que la forma mostrada en el punto 1.) Es la forma clásica, (la única que se puede usar en las versiones anteriores de VB)

Las constantes se declaran de la misma forma que la indicada en el punto 2.), ya que no se podrían declarar como lo mostrado en el punto 1.), por la sencilla razón de que a una constante no se le puede volver a asignar ningún otro valor, ya que si no, no serían constantes, sino variables.

Por ejemplo:

Const n As Integer = 15

## 4.4. ¿Qué ventajas tiene usar constantes en lugar de usar el valor directamente?

Pues que, hay ocasiones en las que dicho valor se repite en un montón de sitios, y si por una casualidad decidimos que en lugar de tener el valor 15, queremos que tenga el 22, por ejemplo, siempre será más fácil cambiar el valor que se le asigna a la constante en la declaración, que tener que buscar los sitios en los que usamos dicho valor y cambiarlos, con la posibilidad de que se nos olvide o pasemos por alto alguno y entonces habremos "*metido la pata*", (por no decir *jodido la marrana*, por aquello de que... ¡Guille!, que ya sabemos que no quieres herir la sensibilidad de nadie, así que... lo mejor es que evites decir esas palabras malsonantes...)

Para declarar una constante de tipo String, lo haremos de esta forma:

Const s As String = "Hola"

De igual manera, para declarar una variable de tipo String y que contenga un valor, lo haremos de esta forma:

Dim Nombre As String = "Guillermo"

Es decir, en las variables usaremos la palabra **DIM**, mientras que en las constantes usaremos **CONST**.

Después veremos algunas variantes de esto, aunque para declarar constantes, siempre hay que usar Const.

Podemos usar cualquier constante o variable en las expresiones, (¿recuerdas lo que es una expresión?), e incluso, podemos usar el resultado de esa expresión para asignar un valor a una variable.

Por ejemplo:

Dim x As Integer = 25

```
Dim i As Integer

i = x * 2
```

En este caso, se evalúa el resultado de la expresión, (lo que hay a la derecha del signo igual), y el resultado de la misma, se asigna a la variable que estará a la izquierda del signo igual.

Incluso podemos hacer cosas como esta:

```
i = i + 15
```

Con esto, estamos indicándoles al VB que: calcula lo que actualmente vale la variable i, súmale el valor 15 y el resultado de esa suma, lo guardas en la variable i.

Por tanto, suponiendo que i valiese 50, después de esta asignación, su valor será 65, (es decir 50 que valía antes más 15 que le sumamos).

Esto último se llama incrementar una variable, y el vb.NET tiene su propio operador para estos casos, es decir cuando lo que asignamos a una variable es lo que ya había antes más el resultado de una expresión:

```
i += 15
```

Aunque también se pueden usar: *=, /=, -=, etcétera, dependiendo de la operación que queramos hacer con el valor que ya tuviera la variable.

Por tanto i = i * 2, es lo mismo que i *= 2

Por supuesto, podemos usar cualquier tipo de expresión, siempre y cuando el resultado esté dentro de los soportados por esa variable:

```
i += 25 + (n * 2)
```

Es decir, no podemos asignar a una variable de tipo numérico el resultado de una expresión alfanumérica:

```
i += "10 * 25"
```

Ya que **"10 * 25"** es una constante de tipo cadena, no una expresión que multiplica 10 por 25.

Al estar entre comillas dobles se convierte *automáticamente* en una constante de cadena y deja de ser una expresión numérica.

Y si tenemos Option Stric On, tampoco podríamos usar números que no fuesen del tipo Integer:

```
i += 25 * 3.1416
```

Ya que el VB se quejará... aunque para solventar estos inconvenientes existen unas funciones de conversión, que sirven para pasar datos de un tipo a otro.

No vamos a profundizar, pero para que sepas que haciendo las cosas como se deben hacer... casi todo es posible, aunque lo que esté escrito dentro de comillas dobles o esté contenido en una variable de cadena no se evalúa... lo más que podemos hacer es convertir esa cadena en un valor numérico, en el caso de "10 * 25", el resultado de convertirlo en valor numérico será 10, ya que todo lo que hay después del 10, no se evalúa... simplemente ¡porque no es un número! son letras, que tienen el "aspecto" de operadores, pero que no es el operador de multiplicar, sino el símbolo *.

Por tanto, esto: i = **Val**("10 * 25")

es lo mismo que esto otro: i = Val("10")

En este caso, usamos la función Val para convertir una cadena en un número, pero ese número es del tipo Double y si tenemos Option Strict On, no nos dejará convertirlo en un Integer... así de "estricto" es el Option Strict.

Para solucionarlo, usaremos la función CType:

i = **CType**(Val("10 * 25"), Integer)

Con esto le estamos diciendo al VB que primero convierta la cadena en un número mediante la función Val, (que devuelve un número de tipo Double), después le decimos que ese número Double lo convierta en un valor Integer.

También podríamos hacerlo de esta otra forma:

i = **CInt**(Val("10 * 25"))

Pero cuidado con los valores que se evalúan, ya que si el valor que se quiere asignar no "cabe" en la variable a la que lo asignamos, nos dará un error de **overflow**... es decir que el número que queremos asignar es más grande de los que ese tipo de datos puede soportar... para solucionar esto, habrá que usar un tipo de datos que soporte valores mayores... a eso es a lo que me refería con lo de la conversión a otros tipos sin producir overflow de la tabla anterior.

Por ejemplo:

i = CInt(Val("25987278547875"))

dará error, porque el número ese que está dentro de las comillas es demasiado grande para almacenarlo en una variable de tipo Integer.

Veamos un resumen de las distintas funciones de conversión de tipos y algunos ejemplos:

(estos están tomados de la ayuda de Visual Basic .NET)

| Nombre de la función | Tipo de datos que devuelve | Valores del argumento "expresion" |
|---|---|---|
| **CBool(*expresion*)** | **Boolean** | Cualquier valor de cadena o expresión numérica. |
| **CByte(*expresion*)** | **Byte** | de 0 a 255; las fracciones se redondean. |
| **CChar(*expresion*)** | **Char** | Cualquier expresión de cadena; los valores deben ser de 0 a 65535. |
| **CDate(*expresion*)** | **Date** | Cualquier representación válida de una fecha o una hora. |
| **CDbl(*expresion*)** | **Double** | Cualquier valor **Duoble**, ver la tabla anterior para los valores posibles. |
| **CDec(*expresion*)** | **Decimal** | Cualquier valor **Decimal**, ver la tabla anterior para los valores posibles. |
| **CInt(*expresion*)** | **Integer** | Cualquier valor **Integer**, ver la tabla anterior para los valores posibles, las fracciones se redondean. |
| **CLng(*expresion*)** | **Long** | Cualquier valor **Long**, ver la tabla anterior para los valores posibles, las fracciones se redondean. |
| **CObj(*expresion*)** | **Object** | Cualquier expresión válida. |
| **CShort(*expresion*)** | **Short** | Cualquier valor **Short**, ver la tabla anterior para los valores posibles, las fracciones se redondean. |
| **CSng(*expresion*)** | **Single** | Cualquier valor **Single**, ver la tabla anterior para los valores posibles. |

| | | |
|---|---|---|
| **CStr(*expresion*)** | **String** | Depende del tipo de datos de la expresión. |
| | | **Nota:** Todos los objetos de vb.NET tienen unos métodos para realizar conversiones a otros tipos, al menos de número a cadena, ya que tienen la propiedad .**ToString** que devuelve una representación en formato cadena del número en cuestión (igual que CStr). |
| **CType(*expresion*, Tipo)** | **El indicado en el segundo parámetro** | Cualquier tipo de datos |
| | | |
| **Val(*expresion*)** | **Double** | Una cadena de caracteres. |
| **Fix(*expresion*)** | **Depende del tipo de datos de la expresión** | Cualquier tipo de datos |
| **Int(*expresion*)** | **Depende del tipo de datos de la expresión** | Cualquier tipo de datos |

Quiero hacer hincapié en las dos últimas funciones, sobre todo si ya has usado anteriormente el Visual Basic e incluso el Basic de MS-DOS, ya que por tradición esas dos funciones devolvían valores enteros de tipo Integer.

Ahora, a pesar de lo que la ayuda de VB.NET pueda decir, ya que en un sitio dice una cosa y en otro dice otra, **Int y Fix devuelve un valor del *mismo tipo* que el que se indica en el parámetro o expresión, pero sin decimales**.

Aunque si esos números son negativos, Fix devuelve el siguiente valor igual o mayor que el número indicado, mientras que Int lo hace con el primer número menor o igual...

Por ejemplo: Fix(-8.4) devuelve -8, mientras que Int(-8.4) devolverá -9.

En caso de que sean positivos, las dos funciones devuelven el mismo valor: Int(8.4) devuelve 8, lo mismo que Fix(8.4).

Haz pruebas por tu cuenta usando Option Strict On y Off, para que veas porqué algunas veces es conveniente dejarlo en On para que nos avise de que algunas operaciones que hacemos pueden no dar los resultados esperados.

## 4.5. Evaluar expresiones lógicas.

El siguiente tema del que vamos a tratar son las expresiones lógicas.

Es decir evaluar expresiones cuyo resultado pueda ser un valor verdadero o falso.

En la entrega anterior, vimos, aunque no te lo expliqué a fondo, las instrucciones IF / THEN...

Ahora si que te lo voy a explicar, aunque estas instrucciones se usan muy a menudo, lo que ahora no entiendas, lo comprenderás después de que lo repitamos hasta la saciedad, pero al menos te voy a dar una pequeña explicación para que sepas para que se usan y sobre todo **cómo** se usan.

Hay ocasiones en las que necesitaremos decidir que hacer dependiendo de algún condicionante, por ejemplo, en la entrega anterior teníamos que comprobar si la tecla

que se había pulsado era la tecla Suprimir y si era así, hacer algo, en aquella ocasión, eliminar elementos de un ListBox.

Por tanto podemos decir que para tomar decisiones usaremos:

**If** <expresión a evaluar> **Then** <Lo que haya que hacer si la expresión evaluada devuelve Verdadero>

Esta es la forma más simple, ya que aquí lo que se hace es evaluar la expresión que se indica después de IF y si esa expresión devuelve un valor verdadero, (es decir es verdad), se ejecutan los comandos que haya después de THEN y si esa expresión no es cierta, se ejecuta lo que haya en la siguiente línea.

Eso mismo también se suele usar de esta otra forma:

**If** <expresión a evaluar> **Then**

    <Lo que haya que hacer si la expresión evaluada devuelve Verdadero>

**End If**

Que para el caso es lo mismo, con la diferencia de que resulta más claro de leer y que podemos usar más de una línea de código, con lo cual resulta más evidente el que podamos hacer más cosas...

Pero si también queremos hacer algo cuando la expresión NO se cumpla, podemos usar la palabra **ELSE** y a continuación el código que queremos usar cuando la expresión no se cumpla.

**If** <expresión a evaluar> **Then** <Lo que haya que hacer si la expresión evaluada devuelve Verdadero> **Else** <Lo que haya que hacer si no se cumple> (todo en una misma línea)

O mejor aún de esta otra forma, que además queda más claro y evidente lo que queremos hacer:

**If** <expresión a evaluar> **Then**

    <Lo que haya que hacer si la expresión evaluada devuelve Verdadero>

**Else**

    <Lo que haya que hacer si no se cumple>

**End If**

Después de Else podemos usar otro IF si así lo creemos conveniente, esto es útil cuando queremos comprobar más de una cosa y dependiendo del valor, hacer una cosa u otra:

```
If a = 10 Then
    ' Lo que sea que haya que hacer cuando a vale 10
ElseIf a = 15 Then
    ‘ Lo que haya que hacer cuando a vale 15
    Else
    ' Lo que haya que hacer en caso de que a no valga ni 10 ni 15
    End If
```

```
' Esto se ejecuta siempre después de haberse comprobado todo lo anterior.
```

Fíjate que en medio de cada If / Then he usado lo que se llama un **comentario**.

Los comentarios empiezan por una comilla simple (apóstrofe), en los comentarios podemos poner lo que queramos, con la seguridad de que no será tenido en cuenta por el Visual Basic.

Los comentarios sólo pueden ocupar una línea, salvo que dicha línea al final tenga el signo _ (subrayado bajo), lo cual indica al IDE que se quiere continuar en la siguiente línea.

Ese símbolo se puede llamar "continuador de línea" y lo podemos usar siempre que queramos, no sólo para los comentarios.

Los comentarios también se pueden hacer con la palabra reservada **Rem**, pero eso es algo que ya nadie usa.

Sigamos con el If.

Si tenemos el Option Strict On, la expresión que se use después de If debe devolver un valor del tipo Boolean, es decir, debe dar como resultado un valor True o False.

Si Option Strict está en Off, el VB lo "convertirá" en un valor True o False, pero no te acostumbres a que VB haga las cosas medio-automáticas, ya que en ocasiones puede ser que ese automatismo no de como resultado lo que nosotros "creíamos" que iba a dar...

De todas formas, cuando el Visual Basic se encuentra con algo como esto:

```
If i > 25 Then
```

Lo que hace es evaluar la expresión y al comprobar si el valor de i es mayor de 25 y en caso de que así sea, devolverá un valor True y si resulta que i no es mayor de 25, devolverá False.

A continuación se comprueba ese valor devuelto por la expresión y si es verdadero (True) se hace lo que esté después del Then y si es falso (False), se hará lo que esté después del Else, (si es que hay algún Else...)

La expresión que se indica después de IF puede ser una expresión "compuesta", es decir se pueden indicar más de una expresión, pero para ello hay que usar algunos de los **operadores lógicos**, tales como **AND**, **OR** o **NOT**.

Por ejemplo si queremos comprobar si el valor de i es mayor que 200 o es igual a 150, haríamos algo así:

```
If i > 200 Or i = 150 Then
```

Pero si lo que queremos es que el valor de i sea mayor que 200 y menor de 500, habría que usar AND:

```
If i > 200 And i < 500 Then
```

Por último, si queremos que la condición se cumpla cuando i NO sea igual a 100:

```
If Not i = 100 Then
```

Aunque esto mismo podríamos haberlo hecho de esta otra forma:

```
If i <> 100 Then
```

Con AND se cumple la verdad si las dos expresiones son verdaderas.

Con Or se cumple si cualquiera de las expresiones es verdadera.

Por supuesto, podemos usar expresiones en las que se mezclen AND y OR, aunque en estos casos es recomendable el uso de paréntesis para separar las expresiones "dudosas".

Por ejemplo:

```
If A = 100 Or B > 50 And x = n * 2 Then
```

¿Que quiere decir esto?

¿Que pasa si A es igual a 100 pero B es menor de 50, y x es igual a n * 2?

Que se cumple, igual que si x no fuese igual a n * 2, pero si A no vale 100, sólo se cumpliría si B fuese mayor de 50.

Es decir, la última expresión sólo se tiene en cuenta si A no vale 100 y B es mayor de 50.

Por tanto quedaría más claro de esta otra forma:

```
If A = 100 Or (B > 50 And x = n * 2) Then
```

Aunque si nuestra intención era otra, podíamos haberlo escrito de esta otra forma:

```
If (A = 100 Or B > 50) And x = n * 2 Then
```

En cuyo caso sólo se cumplirá cuando A sea 100 o B mayor de 50, pero SIEMPRE x debe ser igual a n * 2

Es decir, usa los paréntesis dependiendo de lo que realmente quieras comprobar...

De todas formas, he de aclararte, sobre todo si has usado versiones anteriores de Basic, que en vb.NET las expresiones se van evaluando de izquierda a derecha y se van descartando según se van encontrando cosas que "cumplan" lo que allí se comprueba.

Antes se evaluaban todas las expresiones, (con las posibles consecuencias que podían tener si alguna de ellas contenía una función que hacía algún tipo de cálculo largo y al final resultaba que no era necesario haberlo calculado...) y después se empezaba a descartar posibilidades...

Ya que estamos, decirte que los símbolos que podemos usar para efectuar comparaciones, son estos:

= igual

< menor que

> mayor que

<= menor o igual

>= mayor o igual

<> distinto

Seguramente ya lo sabías, pero... nunca está de más.

Para terminar, decirte que las expresiones que se pueden usar con el IF pueden ser tanto numéricas como alfanuméricas o de cadena, ya que también puede ser conveniente saber si el contendido de la cadena s es mayor que la palabra "hola", aunque en este tipo de expresiones, se evalúa tal y como si se fuese a clasificar... es decir "ahora" será

menor que "hola", ya que si lo clasificáramos, tendríamos que la letra A está antes que la H.

Aquí también juega un poco el Option Compare Binary ya que, como te dije, se hacen distinciones de mayúsculas y minúsculas, aunque a la hora de clasificar (u ordenar), las minúsculas están después de las mayúsculas, por tanto "Amigo" será menor que "amigo" si es que tenemos puesto el Option Compare Binary.

Pero serán iguales si está puesto el Option Compare Text.

Si quieres hacer algunas pruebas, recuerda que puedes crear un proyecto del tipo Consola y usar el Console.WriteLine para mostrar cosas en la pantalla, aunque debes acordarte al final de poner un Console.ReadLine para que la pantalla se quede visible hasta que pulses Intro.

Por ejemplo podrías hacer estas comprobaciones y antes de ejecutar el programa intentar saber que es lo que haría. (aunque también te dejo las pruebecillas que he hecho para explicarte todo esto... no fuese que metiera la pata...)

```
'
Dim unByte As Byte = 129
Dim unBoolean As Boolean = True
Dim unChar As Char = "N"c
'Dim unChar2 As Char = "B" ' (Con Option Strict On da error)
Dim unaFecha As Date = #10/27/2001#
Dim unDecimal As Decimal = 99912581258.125D
Dim unDecimal2 As Decimal = 98765432101234456@
Dim unDoble As Double = 125897.12045R
Dim unDoble2 As Double = 2457998778745.4512#
'Dim unInt As Integer = 24579987787456 ' (Con Option Strict On da error)
Dim unEntero As Integer = 250009I
Dim unEntero2 As Integer = 652000%
Dim unLong As Long = 123456789L
Dim unLong2 As Long = 987654&
Dim unShort As Short = 32000S
'
Const n As Integer = 15
'
Dim i As Integer

i = 10
i += 25 + (n * 2)
Console.WriteLine("(i=10, n=15), 'i += 25 + (n * 2)' es igual a: " &
CStr(i))
i += CType(Val("10 * 25"), Integer)
'i = CInt(Val("25987278547875")) ' (dará error)
i = CInt(Val("25000"))
Console.WriteLine(i)
'
unDoble = Fix(unDoble2)
Console.WriteLine("unDoble = Fix(" & unDoble2.ToString & ") : " &
unDoble.ToString)
unDoble2 = 2457998778745.665#
unDoble = CInt(unDoble2)
Console.WriteLine("unDoble = Int(" & unDoble2.ToString & ") : " &
unDoble.ToString)
'unDoble = CInt(unDoble2)
'Console.WriteLine("unDoble = CInt(" & unDoble2.ToString & ") : " &
unDoble.ToString)
'
unDoble = Fix(8.9)
Console.WriteLine("unDoble = Fix(8.9) : " & unDoble.ToString)
'
```

```
unDecimal = Fix(8.9D)
Console.WriteLine("unDecimal = Fix(8.9D) : " & unDecimal.ToString)
'
Console.WriteLine("i vale: " & CStr(i))
If i > 1500 Then
    Console.WriteLine("i es mayor de 1500")
End If
'
i = 200
Console.WriteLine("Se asigna el valor " & CStr(i) & " a i")
If i > 15 + n Then
    Console.WriteLine("i es mayor de 15 + n")
Else
    Console.WriteLine("i NO es mayor de 15 + n")
End If
'
If i > 200 Or i < 500 Then
    Console.WriteLine("el valor de i es mayor de 200 O menor de 500")
End If
If i > 100 And i < 500 Then
    Console.WriteLine("el valor de i es mayor de 100 Y menor de 500")
End If
If Not i = 100 Then
    Console.WriteLine("i NO vale 100")
End If

Dim a As Integer = 100
Dim b As Integer = 50
Dim x As Integer = n * 2 + 10
If a = 100 Or (b > 50 And x = n * 2) Then
    Console.WriteLine("SI: If a = 100 Or b > 50 And x = n * 2 Then")
End If

If "amigo" > "Amigo" Then
    Console.WriteLine("amigo > Amigo")
Else
    Console.WriteLine("amigo no es > Amigo")
End If

Console.ReadLine()
```

Y hasta aquí hemos llegado.

Espero que ya sepas qué es una variable, y que también sepas declararlas y asignarles valores "predeterminados" o lo que es lo mismo iniciarlas con un valor al declararlas, en lugar del valor que por defecto le asigna el VB.

También sabrás cómo tomar decisiones... aunque sea dentro del código de un programa...

Ahora estás preparado para "toparte" con cualquier tipo de datos de vb.NET y saber cuales son los valores máximos y mínimos que se les puede asignar, también sabes o al menos deberías saber qué funciones se pueden usar para convertir una variable de un tipo en otro distinto.

Igualmente sabrás que el Guille prefiere que uses el **Option Strict On** y que declares todas las variables con el tipo de datos que van a tener.

En fin... un montón de cosillas que a la larga te resultarán de utilidad.

En la próxima entrega veremos cómo declarar varias variables en una misma línea y veremos algo que no todo el mundo acaba por comprender... la "visibilidad" o ámbito de las variables... y algunas cosas más...

# 5. Quinta entrega

Año nuevo, entrega nueva... je, je, no te preocupes que no será así en las próximas... por tanto no tendrás que esperar un año para ver el contenido de la entrega 6... (o casi, ya que desde la anterior han pasado casi TRES meses... que se dice pronto... con razón se hace uno viejo y ni te enteras... en fin...)

Te recomiendo que le eches un vistacillo al Glosario, ya que he añadido cosas nuevas y ampliado otras.

Como te comenté en la entrega anterior, vamos a seguir con el tema de las variables.

Empezaremos con la parte más sencilla de lo que te tengo preparado para esta quinta entrega:

## 5.1. Declarar varias variables en una misma línea:

¿Que significa eso de declarar varias variables en una misma línea?

Pues eso... declarar varias variables en una misma línea, o lo que es lo mismo, con una misma instrucción **Dim**.

Por ejemplo, esta línea declara dos variables del tipo Integer:

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

Pero esta otra, también:

```
Dim c As Integer, d As Integer
```

¿Cual es la diferencia?

Evidentemente que en la segunda línea indicamos dos veces el tipo de datos de cada una de las variables, mientras que en la primera sólo lo especificamos una vez.

Antes de entrar en detalles, apúntate esto:

Podemos declarar más de una variable en una instrucción DIM, si las variables las separamos con una coma.

Es decir, hay que poner una coma entre cada variable que declaremos en la misma línea y opcionalmente el tipo de datos... bueno, opcionalmente, opcionalmente... no… ya que después de la última variable si que debemos indicar el tipo de datos que tendrán todas las variables de esa línea... o casi, según veremos un poco más abajo.

## 5.2. Declarar varios tipos de variables en una misma línea:

Por supuesto que esta no es la única forma de declarar varias variables en una misma línea, ya que puede ser que queramos declarar variables de distintos tipos. En ese caso, hay que indicar junto a cada variable el tipo de datos que queramos que tenga.

Mejor lo veremos con un ejemplo:

```
Dim i As Integer, s As String
```

En este caso, tenemos dos variables de dos tipos distintos, cada una con su **As tipo** correspondiente, pero separadas por una coma.

Pero si lo que pretendemos es que el que lea nuestro código se "caliente" el coco intentando entenderlo... (por decir algo), podemos complicar aún más la cosa:

```
Dim j, k As Integer, s1, Nombre As String, d1 As Decimal
```

En esta ocasión, las variables **j** y **k** son del tipo **Integer**, las variables **s1** y **Nombre** del tipo **String** y por último la variable **d1** es de tipo **Decimal**.

**Tipo de dato por defecto de las variables:**

Si has tratado con alguna versión anterior de Visual Basic, incluso del BASIC de MS-DOS, también estaba permitido hacer las declaraciones de esa forma, pero en el caso de que no se indicara explícitamente el tipo de datos, el Basic le asignaba el tipo de datos por defecto.

En el caso de las versiones de MS-DOS y el VB1, el tipo de datos era Single, en las versiones anteriores a VB.NET el tipo de datos predeterminados era Variant y, si la cosa siguiera igual que en las versiones anteriores, (por suerte ya no es así), en VB.NET, el tipo de datos sería Object. Aunque... si sólo declaras una variable con Dim y no especificas el tipo (y tampoco sigues mis recomendaciones), el tipo asignado será ese: Object.

En las declaraciones con las versiones anteriores al vb.NET existían unas excepciones para que el tipo de datos predeterminado no fuese el que el "compilador" imponía, pero no vamos a entrar en detalles, ya que no vienen al caso y lo único que conseguiría sería complicarte la vida... aunque si quieres saber a qué me refiero... échale un vistazo al Curso Básico de VB que tengo en mis páginas (el Guille siempre aprovecha cualquier ocasión para hacer publicidad).

Sigamos viendo cómo actuaría el vb.NET en el caso de que hagamos algo como esto:

```
Dim z
```

Si no me has hecho caso... o se te ha olvidado poner lo de **Option Strict On**, esa declaración sería válida y el tipo de datos de la variable **z** sería **Object**, es decir, seria lo mismo que hacer esto otro:

```
Dim z As Object
```

Pero si, por otro lado, tienes puesto el Option Strict en On, el vb.NET te diría que "nones", que eso no está permitido.

Realmente nos mostraría un mensaje como este:

**Option Strict requiere que todas las declaraciones de variables tengan una cláusula 'As'**.

Con lo cual tendríamos que especificar el tipo de datos que "realmente" queremos que tenga esa variable.

Como puedes comprobar... nos puede venir bien que de vez en cuando nos avisen de que no estamos haciendo las cosas como debiéramos...

Una cosa que no está permitida al declarar varias variables usando sólo un As Tipo, es la asignación de un valor predeterminado.

Ya vimos en la entrega anterior de que podíamos hacer esto para asignar el valor 15 a la variable N:

```
Dim N As Integer = 15
```

Pero lo que no podemos hacer es declarar, por ejemplo, dos variables de tipo Integer y "pretender" asignarle a una de ellas un valor predeterminado (o inicial), por ejemplo:

```
Dim p, q As Integer = 1
```

Eso daría el error: **No se permite la inicialización explícita con varios declaradores**.

Por tanto deberíamos hacerlo de esta otra forma:

```
Dim p As Integer, q As Integer = 1
```

O de esta otra:

```
Dim p1 As Integer = 12, q1 As Integer = 1
```

Aunque esto otro si que podemos hacerlo:

```
Dim n1 As Integer = 12, n2, n3 As Integer
```

Es decir, si asignamos un valor al declarar una variable, éste debe estar "explícitamente" declarado con un **As Tipo = valor**.

Por tanto, esto otro también se puede hacer:

```
Dim n4 As Integer = 12, n5, n6 As Integer, n7 As Integer = 9
```

Ya que las variables n5 y n6 se declaran con un tipo de datos, pero no se asigna un valor inicial.

Por supuesto los tipos usados no tienen porque ser del mismo tipo:

```
Dim h1 As Integer = 25, m1, m2 As Long, s3 As String = "Hola", d2, d3 As Decimal
```

Pero... la recomendación es que no compliques las declaraciones de las variables de esa forma...

Procura usar **Dim**s diferentes para diferentes declaraciones, si no quieres declarar cada variable con un Dim, al menos usa un Dim para cada tipo de variable.

Por ejemplo el último ejemplo quedaría más legible de esta otra forma:

```
Dim h1 As Integer = 25

Dim m1, m2 As Long

Dim s3 As String = "Hola"

Dim d2, d3 As Decimal
```

Además de que es más "legible", es más fácil de comprobar...

Ahora vamos a pasar a otro de los temas propuestos para esta entrega:

## 5.3. La visibilidad (o alcance) de las variables:

¿Que significa eso de visibilidad (o alcance) de una variable?

¿Quiere decir que al escribir una variable, ésta desaparezca de la vista?

A la última pregunta, la respuesta es: NO... salvo que escribas con tinta invisible... 8-)

A la primera pregunta... es a la que debo dar respuesta, ya que de eso se trata... ¿o no? pues eso...

Aunque la verdad es que a estas alturas del curso, no sé si debería explicarlo... pero bueno, ya que está dicho... apliquémonos el parche y vamos a ver cómo me defiendo... (eso, que algunas veces por hablar de más, te pasan estas cosas...)

Si tienes el Visual Studio y has creado algún proyecto, por ejemplo una aplicación de consola. Habrás visto que el código que se crea es el siguiente:

```
Module Module1
    Sub Main()

    End Sub
End Module
```

Es decir, se crea un módulo llamado Module1 y un procedimiento llamado Sub Main, que por otro lado es el que sirve como punto de entrada al programa... aunque de esto aún no debes preocuparte y dejarlo estar... ya te explicaré de que va todo esto... Por ahora sólo debes saber que los procedimientos Sub son como instrucciones y cuando se usan en otras partes del programa, se ejecuta el código que haya en su interior...

Lo que nos interesa es saber que hay dos "espacios" diferentes en los que poder insertar las declaraciones de las variables:

Después de Module Module1 o después de Sub Main.

El código insertado entre **Module** y **End Module**, se dice que es código a nivel de módulo, en este caso el único código posible es el de declaraciones de variables o procedimientos.

El código insertado dentro del **Sub** y **End Sub**, se dice que es código a nivel de procedimiento, en esta ocasión podemos poner lo que queramos, además de declaraciones de variables.

Pues bien. Si declaramos una variable a nivel de módulo, dicha variable estará disponible en "todo" el módulo, incluido los procedimientos que pudiera haber dentro del módulo.

Por otro lado, las variables declaradas dentro de un procedimiento, sólo serán **visibles** dentro de ese procedimiento, es decir que fuera del procedimiento no se tiene conocimiento de la existencia de dichas variables.

Veamos un pequeño ejemplo con dos procedimientos:

```
Option Strict On

Module Module1
    ' Variable declarada a nivel de módulo
    Dim n As Integer = 15

    Sub Main()
        ' Variable declarada a nivel de procedimiento
        Dim i As Long = 10
        '
        ' Esto mostrará que n vale 15
        Console.WriteLine("El valor de n es: {0}", n)
        Console.WriteLine("El valor de i es: {0}", i)

        Console.ReadLine()
    End Sub

    Sub Prueba()
        ' Esto mostrará que n vale 15
```

```
        Console.WriteLine("El valor de n es: {0}", n)

        ' Esta línea dará error, ya que la variable i no está declarada
        Console.WriteLine("El valor de i es: {0}", i)

        Console.ReadLine()
    End Sub
End Module
```

Te explico un poco:

Este módulo contiene dos procedimientos de tipo Sub, (los SUBs son como las instrucciones, realizan una tarea, pero no devuelven un valor como lo harían las funciones y las propiedades) (Guille, eso ya está explicado en el Glosario, ¿es que pretendes que la entrega sea más larga?).

La variable **n** la hemos declarado a nivel de módulo, por tanto estará visible en todo el módulo y por todos los procedimientos de ese módulo.

Dentro del procedimiento **Main**, hemos declarado la variable **i**, ésta sólo estará disponible dentro de ese procedimiento. Por eso al intentar usarla en el procedimiento **Prueba**, el vb.NET nos da un error diciendo que la variable no está declarada.

Y a pesar de que está declarada, lo está pero sólo para lo que haya dentro del procedimiento Main, por tanto en Prueba no se sabe que existe una variable llamada **i**.

(je, je... seguro que ni tú, Guille, te has enterado de lo que has explicado... Que sí, que es mu fácil... si la variable i... *¡VALEEEE!* que yo si que me he enterado... así que calla y sigue, que ya queda poco)

**Variables que ocultan a otras variables:**

Ahora vamos a complicar la cosa un poquito más:

Veamos el código y después te explico.

```
Option Strict On

Module Module1
    ' Variable declarada a nivel de módulo
    Dim n As Integer = 15

    Sub Main()
        '
        Console.WriteLine("El valor de n Main es: {0}", n)
        '
        Console.ReadLine()
    End Sub

    Sub Prueba()
        Dim n As Long = 9547
        '
        Console.WriteLine("El valor de n en Prueba es: {0}", n)
        '
        Console.ReadLine()
    End Sub

End Module
```

Este ejemplo me sirve para que compruebes que una variable de nivel "superior" puede ser eclipsada por otra de un nivel "inferior".

La variable n declarada a nivel de módulo estará visible en todos los procedimientos del módulo, pero al declarar otra variable, también llamada n, dentro del procedimiento

Prueba, ésta última "eclipsa" a la variable que se declaró a nivel de módulo y se usa la variable "local" en lugar de la "global". Aunque sólo dentro del procedimiento Prueba, ya que en el procedimiento Main si que se ve el valor de la variable declarada a nivel de módulo, por la sencilla razón de que no hay ninguna variable declarada dentro de Main con el mismo nombre. (elemental mi querido Guille, algunas veces me sorprende tu lógica aplastante)

Una vez visto esto, podemos afirmar que:

Cuando se declara una variable dentro de un procedimiento, (Sub, función o propiedad), esa variable "eclipsa" (u oculta) a otras variables que, teniendo el mismo nombre, pudieran existir a nivel de módulo o a un nivel "superior".

Por supuesto, si declaramos una variable dentro de un procedimiento y no existe ninguna otra variable que tenga ese mismo nombre, no ocultará nada... se supone que entiendes que esto es aplicable sólo cuando existen variables con el mismo nombre. (lo dicho: este Guille tiene una lógica aplastante... en fin...)

Las variables declaradas dentro de un procedimiento se dicen que son "locales" a ese procedimiento y por tanto sólo visibles (o accesibles) dentro del procedimiento en el que se ha declarado.

En otra ocasión veremos más cosas sobre la visibilidad de las variables y cómo podemos ampliar la "cobertura" o el campo de visibilidad, para que puedan ser vistas en otros sitios... pero eso en otra ocasión. (sí, déjalo ya, que es tarde y los Reyes Magos nos van a pillar levantados...)

En la próxima entrega veremos más cosas, pero será referente a lo que ya vimos en la entrega anterior, respecto a las decisiones lógicas con IF/THEN y seguramente veremos cómo hacer bucles (o lo que es lo mismo, hacer que una variable tome valores de forma automática y por un número determinado de veces) y otras cosillas que a la larga nos serán de utilidad en la mayoría de los programas.

Sólo me queda hacerte, (como espero que sea la norma en todas las entregas), un "pequeño" resumen de lo que hemos visto en esta entrega:

- Definición de varias variables de un mismo tipo con un sólo AS TIPO
- Cómo definir más de una variable con una misma instrucción DIM y poder asignarle un valor inicial.
- Cómo definir más de una variable de diferentes tipos con una misma instrucción DIM.
- Alcance o visibilidad de las variables a nivel de módulo o procedimiento y
- Cómo se pueden "ocultar" variables con el mismo nombre declaradas en distintas partes del módulo.

Y esto es todo por hoy... mañana más... (aunque viniendo del Guille, es una forma de hablar, no te lo tomes al pie de la letra...)

# 6. Sexta entrega

En la entrega anterior te dije que no iba a dejar que pasara un año para publicar la siguiente, pero... casi, casi... si me descuido, si que pasa un año... en fin... Aunque tengo que decirte que ha sido por causas ajenas y por otras en las que no voy a entrar en detalles... lo importante, al menos para mí, es que aquí tienes la sexta entrega de este curso de iniciación a Visual Basic .NET

Lo que si es importante, es que en las fechas en las que estoy escribiendo estas líneas, la versión definitiva de Visual Basic .NET ya es algo real... y también tengo que decirte que todo lo dicho anteriormente es aplicable a esta versión, que según Microsoft es Visual Studio .NET 2002, (te recuerdo que anteriormente estaba trabajando con la Beta 2 de Visual Studio .NET)

Bien, empecemos.

Según te comenté hace casi un año... en esta entrega vamos a seguir viendo cosas relacionadas con las comparaciones (IF / THEN) y también veremos cómo hacer bucles, etc.

Aunque esto de los bucles ya vimos algo en la tercera entrega, aquí lo veremos con algo de más detalle, es decir, te lo voy a explicar.

Pero empecemos con If Then y los operadores lógicos.

Como vimos en la cuarta entrega podíamos usar AND, OR y NOT, pero en esta nueva versión de Visual Basic los dos primeros tienen su versión especial, la cual es la más recomendable, ahora veremos porqué.

Cuando hacemos una comparación usando AND, nuestro querido VB comprueba si las dos expresiones usadas con ese operador son ciertas, esto está claro, por ejemplo si tenemos lo siguiente:

```
IF N = 3 AND X > 10 THEN
```

Se comprueba si el contenido de N es igual a 3 *y también* si el contenido de X es mayor de 10.

Y tu dirás... vale, muy bien, está claro, ¿cual es el problema?

Pues ninguno... bueno, si, si hay un problema.

¿Qué pasa si el contenido de N no es 3?

Tu dirás: Que no se cumple la condición.

Y estarás en lo cierto, pero... aunque el contenido de N no sea igual a 3, también se comprueba si el contenido de X es mayor de 10.

Si lo piensas durante unos segundos... (ya) comprobarás que realmente no sería necesario hacer la segunda comprobación, ya que si la primera no se cumple... ¿qué necesidad hay de comprobar la segunda?

Como sabemos, AND "necesita" que las dos expresiones sean ciertas para dar por cierta la expresión completa.

Esto en teoría no sería problemático, ya que Visual Basic no tarda mucho tiempo en comprobar la segunda expresión, al menos en este ejemplo, ya que si en lugar de ser una "simple" comparación, en la segunda parte del AND tuviéramos algún tipo de

expresión que se tomara su tiempo... sería un desperdicio inútil tener que esperar ese tiempo, sobre todo sabiendo que no es necesario hacer esa segunda comprobación.

En las versiones anteriores de Visual Basic, e incluso en la época del BASIC de MS-DOS, este tipo de expresiones se solían "partir" en dos comparaciones IF, ya que, si la primera no se cumple, no es necesario hacer la segunda, por tanto esa comparación se podría hacer de la siguiente forma:

```
IF N = 3 THEN

    IF X > 10 THEN
```

Es decir, sólo comprobamos la segunda expresión si se cumple la primera.

Ahora, con Visual Basic .NET no es necesario tener que llegar a este "desdoblamiento" y podemos hacer la comparación "casi" como lo que hemos visto, pero usando una instrucción (u operador) especial: **AndAlso**.

**Cuando usamos AndAlso, Visual Basic .NET sólo comprobará la segunda parte, si se cumple la primera.**

Es decir, si N no vale 3, no se comprobará si X es mayor que 10, ya que no lo necesita para saber que TODA la expresión no se cumple. Por tanto, la comparación anterior podemos hacerla de esta forma:

```
IF N = 3 ANDALSO X > 10 THEN
```

Lo mismo, (o casi), ocurre con OR, aunque en este caso, sabemos que si el resultado de cualquiera de las expresiones usadas con ese operador es verdadero, la expresión completa se da por buena.

Por ejemplo, si tenemos:

```
IF N = 3 OR X > 10 THEN
```

En el caso de que N valga 3, la expresión (N = 3 OR X > 10) sería verdadera, pero, al igual de lo que pasaba con AND, aunque N sea igual a 3, también se comprobará si X vale más de 10, aunque no sea estrictamente necesario.

Para que estos casos no se den, Visual Basic .NET pone a nuestra disposición del operador **OrElse**.

**Cuando usamos OrElse, Visual Basic .NET sólo comprobará la segunda parte si no se cumple la primera.**

Es decir, si N es igual a 3, no se comprobará la segunda parte, ya que no es necesario hacerlo. En este caso la comparación quedaría de esta forma:

```
IF N = 3 ORELSE X > 10 THEN
```

A estos dos operadores se les llama operadores de cortocircuito (shortcircuit operators) y a las expresiones en las que participan se llaman expresiones cortocircuitadas.

Otra cosa que debemos tener en cuenta cuando evaluamos expresiones lógicas, o cuando simplemente evaluamos expresiones, por ejemplo para asignar el resultado a una variable, hay que tener presente que Visual Basic .NET sigue un orden a la hora de evaluar las distintas partes de la expresión.

Esto es lo que se llama:

## 6.1. Prioridad de los operadores

Tenemos que tener en cuenta que los operadores pueden ser aritméticos, de comparación y lógicos. Veamos cada grupo por separado, te advierto que algunos de estos operadores aún no los hemos visto, pero... creo que es conveniente saberlo y ya tendremos ocasión de aprender para que sirven.

### 6.1.1. Prioridad de los operadores aritméticos y de concatenación:

Exponenciación (^)

Negación (-)

Multiplicación y división (*, /)

División de números enteros (\)

Módulo aritmético (Mod)

Suma y resta (+, -)

Concatenación de cadenas (&)

### 6.1.2. Operadores de comparación:

Igualdad (=)

Desigualdad (<>)

Menor o mayor que (<, >)

Mayor o igual que (>=)

Menor o igual que (<=)

### 6.1.3. Operadores lógicos:

Negación (Not) Conjunción (And, AndAlso)

Disyunción (Or, OrElse, Xor)

Cuando en una misma expresión hay sumas y restas o multiplicación y división, es decir operadores que tienen un mismo nivel de prioridad, éstos se evalúan de izquierda a derecha.
Cuando queramos alterar este orden de prioridad, deberíamos usar paréntesis, de forma que primero se evaluarán las expresiones que estén dentro de paréntesis.

Por ejemplo, si tenemos esta expresión:

X = 100 - 25 – 3

El resultado será diferente de esta otra:

X = 100 - (25 - 3)

En el primer caso el resultado será 72, mientras que en el segundo, el resultado será 78, ya que primero se evalúa lo que está entre paréntesis y el resultado (22) se le resta a 100.

## 6.2. Bucles en Visual Basic .NET

Cada vez que queramos "reiterar" o repetir un mismo código un número determinado de veces, e incluso un número indeterminado de veces, tendremos que echar mano de los bucles.

En Visual Basic .NET existen diferente instrucciones para hacer bucles (o reiteraciones), veamos esas instrucciones y después las veremos con más detalle:

### For / Next, con este tipo de bucle podemos repetir un código un número determinado de veces.

La forma de usarlo sería:

**For** <variable numérica> **=** <valor inicial> **To** <valor final> [**Step** <incremento>]

    ' contenido del bucle, lo que se va a repetir

**Next**
La *variable numérica* tomará valores que van desde el *valor inicial* hasta el *valor final*, si no se especifica el valor del *incremento*, éste será 1.

Pero si nuestra intención es que el valor del incremento sea diferente a 1, habrá que indicar un valor de incremento; lo mismo tendremos que hacer si queremos que el valor inicial sea mayor que el final, con idea de que "cuente" de mayor a menor, aunque en este caso el incremento en realidad será un "decremento" ya que el valor de incremento será negativo.

Lo entenderemos mejor con un algunos ejemplos:

```
Dim i As Integer
'
For i = 1 To 10
    ' contará de 1 hasta 10
    ' la variable i tomará los valores 1, 2, 3, etc.
Next
'
For i = 1 To 100 Step 2
    ' contará desde 1 hasta 100 (realmente 99) de 2 en 2
    ' la variable i tomará los valores 1, 3, 5, etc.
Next
'
For i = 10 To 1 Step -1
    ' contará desde 10 hasta 1
    ' la variable i tomará los valores 10, 9, 8, etc.
Next
'
For i = 100 To 1 Step -10
    ' contará desde 100 hasta 1, (realmente hasta 10)
    ' la variable i tomará los valores 100, 90, 80, etc.
Next
'
For i = 10 To 1
    ' este bucle no se repetirá ninguna vez
Next
'
For i = 1 To 20 Step 50
```

```
    ' esto sólo se repetirá una vez
Next
```

En algunos casos, hay que tener en cuenta que el valor final del bucle puede que no sea el indicado, todo dependerá del incremento que hayamos especificado. Por ejemplo, en el tercer bucle, le indicamos que cuente desde 1 hasta 100 de dos en dos, el valor final será 99.
En otros casos puede incluso que no se repita ninguna vez... este es el caso del penúltimo bucle, ya que le decimos que cuente de 10 a 1, pero al no indicar Step con un valor diferente, Visual Basic "supone" que será 1 y en cuanto le suma uno a 10, se da cuenta de que 11 es mayor que 1 y como le decimos que queremos contar desde 10 hasta 1, pues... sabe que no debe continuar.

## For Each, este bucle repetirá o iterará por cada uno de los elementos contenidos en una colección.

La forma de usarlo es:

**For Each** <variable> **In** <colección del tipo de la variable>

    ` lo que se hará mientras se repita el bucle

**Next**

Este tipo de bucle lo veremos con más detalle en otras entregas, pero aquí veremos un par de ejemplos.

Sólo decirte que podemos usar este tipo de bucle para recorrer cada uno de los caracteres de una cadena, este será el ejemplo que veremos, ya que, como tendrás la oportunidad de comprobar, será un tipo de bucle que usaremos con bastante frecuencia.

```
Dim s As String
'
For Each s In "Hola Mundo"
    Console.WriteLine(s)
Next
Console.ReadLine()
```

## While / End While, se repetirá mientras se cumpla la expresión lógica que se indicará después de While.

La forma de usarlo es:

**While** <expresión>

    ' lo que haya que hacer mientras se cumpla la expresión

**End While**

Con este tipo de bucles, se evalúa la expresión y si el resultado es un valor verdadero, se ejecutará el código que esté dentro del bucle, es decir, entre While y End While.

La expresión será una expresión lógica que se evaluará para conseguir un valor verdadero o falso.

Veamos algunos ejemplos:

```
Dim i As Integer
'
While i < 10
    Console.WriteLine(i)
    i = i + 1
End While
```

```
Console.ReadLine()
'
'
Dim n As Integer = 3
i = 1
While i = 10 * n
    ' no se repetirá ninguna vez
End While
```

En el primer caso, el bucle se repetirá mientras i sea menor que 10, fíjate que el valor de i se incrementa después de mostrarlo, por tanto se mostrarán los valores desde 0 hasta 9, ya que cuando i vale 10, no se cumple la condición y se sale del bucle.

En el segundo ejemplo no se repetirá ninguna vez, ya que la condición es que i sea igual a 10 multiplicado por n, cosa que no ocurre, ya que i vale 1 y n vale 3 y como sabemos 1 no es igual a 30.

## Do / Loop, este tipo de bucle es muy parecido al anterior, aunque algo más flexible.

Si se utiliza **sólo** con esas dos instrucciones, este tipo de bucle no acabará nunca y repetirá todo lo que haya entre Do y Loop.

Pero la flexibilidad a la que me refería es que este tipo de bucle se puede usar con dos instrucciones que nos permitirán evaluar expresiones lógicas:

**While** y **Until**

Pero no debemos confundir este While con el While/End While que acabamos de ver anteriormente, aunque la forma de usarlo es prácticamente como acabamos de ver.

La ventaja de usar While o Until con los bucles Do/Loop es que estas dos instrucciones podemos usarlas tanto junto a Do como junto a Loop, la diferencia está en que si los usamos con Do, la evaluación se hará antes de empezar el bucle, mientras que si se usan con Loop, la evaluación se hará después de que el bucle se repita al menos una vez.

Veamos cómo usar este tipo de bucle con las dos "variantes":

**Do While** <expresión>

**Loop**

**Do**

    '

    **Loop While** <expresión>

    **Do Until** <expresión>    '

    **Loop**

**Do**

**Loop Until** <expresión>

Usando Do con While no se diferencia en nada con lo que vimos anteriormente, con la excepción de que se use junto a Loop, pero como te he comentado antes, en ese caso la evaluación de la expresión se hace después de que se repita como mínimo una vez.

Until, a diferencia de While, la expresión se evalúa cuando no se cumple la condición, es como si negáramos la expresión con While (Not <expresión>)

Una expresión usadas con Until podríamos leerla de esta forma: hasta que la expresión se cumpla:

Do Until X > 10 (repite hasta que X sea mayor que 10)

Veamos un ejemplo para entenderlo mejor:

```
i = 0
Do Until i > 9
    Console.WriteLine(i)
    i = i + 1
Loop
```

Este bucle se repetirá para valores de i desde 0 hasta 9 (ambos inclusive).

Y este también:

```
i = 0
Do While Not (i > 9)
    Console.WriteLine(i)
    i = i + 1
Loop
```

Con esto entenderás mejor a lo que me refería con negar la expresión con While.

En el caso de que Until se use junto a Loop, la expresión se comprobará después de repetir al menos una vez el bucle.

Y esto son todos los bucles que podemos usar.

Pero aún no hemos acabado esta entrega... (después de lo que se ha hecho esperar... ¡que menos!)

Lo que nos queda por ver, es que cuando estamos dentro de un bucle, podemos abandonarlo de dos formas:

1- Esperando a que el bucle termine.

2- Saliendo "abruptamente" o abandonándolo antes de que termine de forma lógica o por los valores que le hemos indicado (al menos en el caso de los bucles For)

Para poder abandonar un bucle (esto veremos que es ampliable a otros temas) hay que usar la instrucción **Exit** seguida del tipo de bucle que queremos abandonar:

**Exit For**

**Exit While**

**Exit Do**

Esto es útil si necesitamos abandonar el bucle por medio de una condición, normalmente se utiliza con un If / Then.

En otras ocasiones veremos estas instrucciones en plena acción, pero eso será en otra ocasión, ya que esta entrega la vamos a dejar aquí.

En la próxima entrega veremos otra forma de tomar decisiones, hasta ahora sólo hemos visto que podemos tomarlas con If / Then.

Sólo me queda hacer el resumen de lo que hemos visto en esta sexta entrega:

- Otra forma de evaluar expresiones lógicas de una forma más "lógica", usando AndAlso y OrElse en lugar de And y Or.

- Los diferentes tipos de bucles que podemos usar con Visual Basic .NET
- Cómo abandonar un bucle de forma explícita.

Lo dicho, espero que la próxima entrega no se haga tanto de rogar y podamos verla antes de que acabe este año.

# 7. Séptima entrega

En esta entrega veremos otra forma con la que podemos escoger entre varias opciones. Hasta ahora hemos usado las instrucciones IF / Then / Else, pero Visual Basic pone a nuestra disposición la instrucción **Select Case** con la que podemos elegir entre varias opciones y en la versión .NET nos facilita un poco las cosas, al menos con respecto a como se usaba en las versiones anteriores de Visual Basic, esto lo comprobaremos en cuanto veamos una nueva forma de crear constantes.

Empecemos con la instrucción Select Case.

Esta instrucción se usa de la siguiente forma:

Select Case <expresión a evaluar>

Case <lista de expresiones>

' ...

Case <otra lista de expresiones>

' ...

Case Else

' si no se cumple ninguna de las listas de expresiones

End Select

Después de Select Case se pondrá una expresión a evaluar, es decir lo que queremos comprobar si se cumple o no, esto es lo mismo que hacemos con el If <expresión a evaluar> Then; lo que diferencia al Select Case del If... Then es que con el If... Then si queremos hacer varias comprobaciones, tendremos que usar diferentes If... Then con varios ElseIf..., mientras que con el Select Case, cada una de las cosas que queremos comprobar lo ponemos en los distintos Case... ¿cómo? ¿que no te enteras?, vale, veámoslo con un ejemplo.

Supongamos que queremos comprobar si el contenido de la variable **i** tiene distintos valores y según esos valores tendremos que hacer una cosa u otra.

Si lo hiciéramos usando If... Then, tendríamos que escribir algo como esto:

```
If i = 3 Then
    '
ElseIf i > 5 AndAlso i < 12 Then
    '
ElseIf i = 14 OrElse i = 17 Then
    '
ElseIf i > 25 Then
    '
Else
    '
End If
```

Esto mismo, con Select Case lo haríamos de esta forma:

```
Select Case i
    Case 3
        '
    Case 6 To 11
        '
    Case 14, 17
```

```
        '
    Case Is > 25
        '
    Case Else
        '
End Select
```

Como podemos comprobar, después de Select Case ponemos lo que queremos tener en cuenta, en este caso es el contenido de la variable i, cuando queremos comprobar si es igual a 3, simplemente ponemos el 3, lo mismo hacemos cuando queremos hacer algo para el caso de que el valor sea 14 o 17, pero en este caso simplemente especificamos los valores que queremos comprobar separados por comas, podemos poner tantos como necesitemos.
Si queremos comprobar un rango de valores, por ejemplo que el valor de **i** estuviera entre 5 y 12 (mayor que 5 y menor que 12), podemos hacerlo de esta forma: usamos 6 To 11, es decir queremos que esa condición se cumpla cuando el valor de **i** tenga un valor de *6 a 11*.

Cuando queremos comprobar si el valor **es** mayor (o cualquier otra comprobación), usaremos **Is**, como en el caso de Is > 25, esto es lo mismo que comprobar si **i** es mayor que 25.

Por último, si ninguno de esos casos se cumple, se ejecutará lo que esté a continuación de Case Else, que funciona igual que el Else último que tenemos en el bloque If... Then.

Como vemos, funciona casi igual que con If... Then, pero de una forma algo más ordenada, la única pega es que sólo podemos evaluar una expresión, mientras que con If podemos usar tantas como queramos... precisamente por eso existen estas dos posibilidades... cuando una no es válida, podemos usar la otra.

Además de expresiones simples, en Select Case podemos indicar cualquier expresión válida, siempre que el resultado de esa expresión sea comparable con un valor, el cual, a su vez, producirá un valor verdadero o falso.

Estas expresiones, pueden ser tanto numéricas como de cadenas de caracteres.

No voy a dar ahora más ejemplos, ya que a lo largo de este curso de iniciación a la programación de Visual Basic .NET tendremos ocasión de ver más de un ejemplo del uso de Select Case (e incluso de If... Then), así que no te impacientes y si quieres ver ejemplos de Select Case, te invito a que le eches una ojeada a la documentación de Visual Studio .NET, que es bastante amplia y en algunos casos, hasta fácil de comprender... (bueno, sólo en algunos casos, ya que si siempre fuese tan "clara", ¿para que escribir nada que lo aclare...?)

Bien, sigamos nuestro aprendizaje, ahora vamos a ver esa otra forma de crear constantes que mencioné al principio.

Me estoy refiriendo a:

## 7.1. Las enumeraciones (Enum)

Como habrás podido ver en la descripción del glosario (si es que has seguido el link o enlace de la línea anterior), una enumeración es un tipo especial de variable numérica en la que los valores que dicha variable puede tomar, son constantes simbólicas, es decir que en lugar de usar un número, se usa una palabra (constante) que hace referencia a un número.

Por ejemplo, (si mejor pon un ejemplo, que si no, no hay quien te entienda), si queremos tener una variable llamada color la cual queremos que contenga un valor numérico que

haga referencia a un color en particular, es que en lugar de usar el valor 1, 2 ó 3, queremos usar la constante rojo, azul, verde, etc. Esto lo haríamos de esta forma:

```
Enum colores
    rojo = 1
    azul
    verde
End Enum
```

Las declaraciones de las enumeraciones hay que hacerla fuera de cualquier procedimiento, por ejemplo dentro de una clase o un módulo, pero también pueden estar declarados dentro de un espacio de nombres, todo dependerá de la cobertura (o amplitud de acceso) que queramos darle. Además podemos poder darle las propiedades pública, privada, etc., como siempre, esto influirá en los sitios desde los que podemos usar esa enumeración.

Los valores que pueden tener los miembros de una enumeración, pueden ser cualquiera de los que un tipo numérico de tipo entero pueda tener. Por defecto el tipo es Integer, pero las enumeraciones también pueden ser de tipo Byte, Long o Short, para poder especificar un tipo diferente a Integer, lo indicaremos usando As Tipo después del nombre de la enumeración.

Por defecto, el primer valor que tendrá un elemento de una enumeración será cero y los siguientes elementos, salvo que se indique lo contrario, tendrán uno más que el anterior.

En el ejemplo mostrado, el elemento rojo, valdrá 1, azul valdrá 2 y verde tendrá un valor 3.

Para poder cambiar esos valores automáticos, podemos indicarlo usando una asignación como la usada para indicar que rojo vale 1: rojo = 1

En caso de que no indiquemos ningún valor, el primero será cero y los siguientes valdrán uno más que el anterior, por ejemplo, si la declaración anterior la hacemos de esta forma:

```
Enum colores
    rojo
    azul
    verde
End Enum
```

rojo valdrá 0, azul será igual a 1 y verde tendrá el valor 2.

La asignación podemos hacerla en cualquier momento, en el siguiente caso, rojo valdrá cero, azul tendrá el valor 3 y verde uno más que azul, es decir 4.

```
Enum colores
    rojo
    azul = 3
    verde
End Enum
```

Por supuesto, los valores que podemos asignar a los elementos (o miembros) de una enumeración serán valores que estén de acuerdo con el tipo de datos, recordemos que si no indicamos nada, serán de tipo Integer, pero si especificamos el tipo de datos, por ejemplo, de tipo Byte, el cual si recordamos la tabla vista en la cuarta entrega, sólo podrá contener valores enteros comprendidos entre 1 y 255. Sabiendo esto, no podríamos declarar la siguiente enumeración sin recibir un mensaje de error:

```
Enum colores As Byte
    azul = 255
    rojo
    verde
End Enum
```

¿Por qué? te preguntarás, ¿si el valor está dentro de los valores permitidos?

Por la sencilla razón de que azul tiene un valor adecuado, pero tanto rojo como verde, tendrán un valor 256 y 257 respectivamente, los cuales están fuera del rango permitido por el tipo Byte.

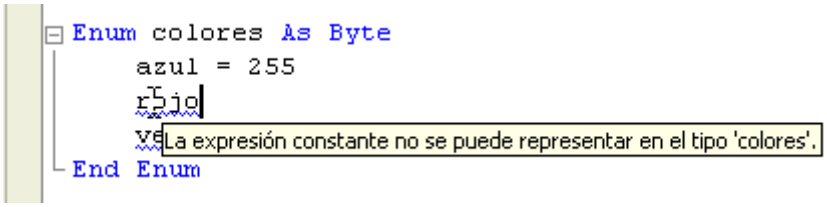



**Figura 1**. Los valores de los miembros de las enumeraciones deben estar comprendidos en el rango del tipo usado.

Otra cosa con la que tendremos que andarnos con cuidado al usar las enumeraciones, es que una variable declarada del tipo de una enumeración, en teoría no debería admitir ningún valor que no esté incluido en dicha enumeración, aunque esta restricción es sólo "recomendable", no es algo "obligatorio", aunque si tenemos activado Option Strict On, el IDE de Visual Studio .NET nos lo recordará, con lo cual nos obligará a hacer una conversión de datos entre la variable (o el valor usado) y el tipo "correcto" al que corresponde la enumeración.

Para entenderlo mejor, veamos un ejemplo:

```
Dim unColor As colores
'
unColor = 1
```

Aunque el valor 1, sea un valor correcto, si tenemos Option Strict On, nos indicaría que no se permite la conversión implícita entre Integer y el tipo colores. Si no tuviéramos activada la comprobación estricta, (cosa que yo te recomiendo y si pudiera, hasta te obligaría a usar, más que nada porque así aprenderás a hacer las cosas bien, que siempre habrá tiempo de hacerlas mal), a lo que iba, si no tuvieras esa opción activada, no te mostraría ningún error, pero si sabemos que para un buen uso de los tipos de datos, deberíamos hacer la conversión correspondiente, ¿por qué no hacerla?

Si te portas como el Guille quiere, tendrías que hacer lo siguiente:

```
unColor = CType(1, colores)
```

Es decir, usamos CType para hacer la conversión entre el número y el tipo correcto de datos.

De todas formas, te diré que si usas el IDE de Visual Studio .NET, éste te mostrará los valores que puedes asignarle a la variable declarada del tipo colores:

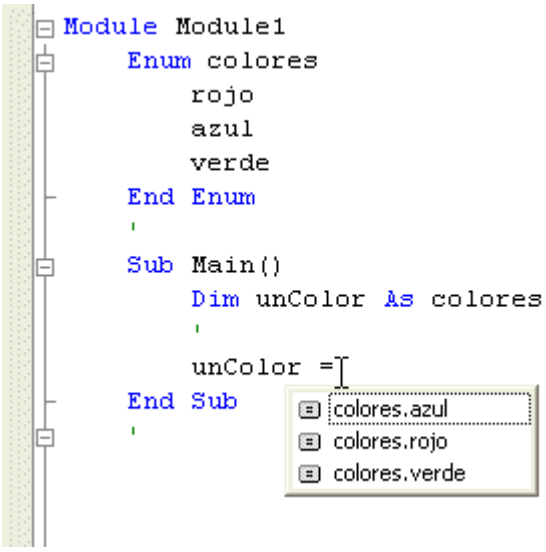
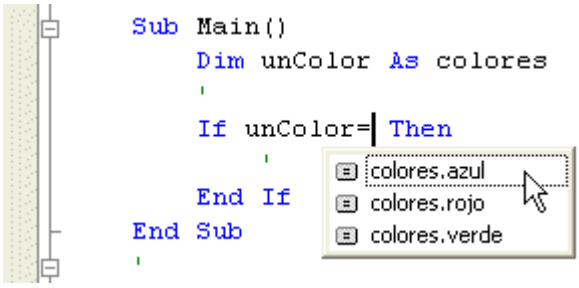


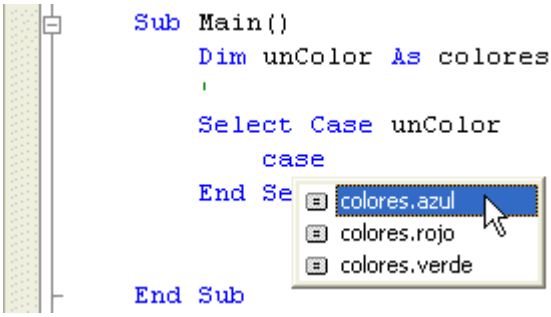

**Figura 2.** Las variables del tipo de una enumeración sólo deberían tener los valores de dicha enumeración

Al igual que ocurre con las variables de tipo Boolean, (que sólo pueden tomar los valores True o False), cuando vamos a asignar un valor, se nos muestra los valores posibles, esto mismo también ocurre si queremos usar esa variable en una comparación o en un Select Case, a esto es a lo que me refería al principio de esta entrega.

Veamos lo que nos mostraría el IDE al usar la variable **unColor** en los casos antes mencionados:




Una enumeración en una comparación If




Una variable enumerada en un Select Case

En las versiones anteriores de Visual Basic, podíamos usar los miembros de las enumeraciones sin indicar la enumeración a la que pertenecen, por ejemplo, podíamos asignar el valor azul a una variable del tipo colores, pero en Visual Basic .NET esto ya no es posible, si queremos usar el miembro azul, tenemos que indicar la enumeración a la que pertenece, por ejemplo con **unColor = colores.azul**, esto más que un inconveniente es una ventaja, ya que así no habrá posibilidades de "mal interpretación" o mal uso de los miembros de una enumeración.

Como he mencionado antes, deberíamos hacer la comprobación correspondiente para saber si el valor que estamos asignando a una variable "enumerada" es el adecuado o no, por ejemplo, si el parámetro de un procedimiento recibe un tipo enumerado, puede ser que el valor con el que se llame a ese procedimiento no sea un valor válido, ¿cómo podemos evitar estos casos?

En principio podríamos hacer una comprobación, bien usando comparaciones If o, mejor aún, usando Select Case, para el caso de la enumeración colores no habría mucho problema, ya que sólo tiene tres valores y esa comprobación podría quedar de esta forma:

```
Private Sub pruebaColores(ByVal elColor As colores)
    Select Case elColor
        Case colores.azul, colores.rojo, colores.verde
            ' OK
        Case Else
            ' si no es ninguno de los válidos,
            ' asignamos uno por defecto
            elColor = colores.azul
    End Select
    ' resto del código...
End Sub
```

Esto está bien, ya que, a pesar de que usemos CType para convertir un valor en la enumeración, no evita que que se "cuele" un valor erróneo:

```
pruebaColores(CType(12, colores))
```

Esta llamada sería correcta, pero el valor no estaría dentro de los valores válidos, aunque no produciría ningún tipo de error, ya que, aunque 12 no sea un valor válido del tipo colores, si que es un valor válido del tipo Integer, que al fin y al cabo es el tipo de valores que admiten las variables declaradas como colores.

En el procedimiento, se comprueba si el valor pasado en el parámetro es del tipo adecuado y se actuaría de la forma que nosotros quisiéramos. El problema vendría si la cantidad de miembros de la enumeración fuese muy extensa y esos valores no fuesen consecutivos, ya que nos obligaría a hacer muchas "comparaciones" para poder saber si el valor indicado es uno de los valores correctos.

Para poder solventar este "problemilla", tendremos que echar mano del propio .NET Framework, en particular de la clase **Enum**, esta clase tiene una serie de métodos "estáticos" que podemos usar para distintas circunstancias, en nuestro caso particular, podemos usar el método **IsDefined** que devolverá un valor verdadero o falso, según el valor indicado esté o no definido en la enumeración.

Por ejemplo si hacemos esto:

```
If System.Enum.IsDefined(unColor.GetType, 12) Then
```

o esto otro, que para el caso es lo mismo, ya que el primer parámetro de esta función espera un tipo:

```
If System.Enum.IsDefined(GetType(colores), 12) Then
```

Se devolverá un valor falso, ya que 12 no es miembro de la enumeración colores.

En el primer caso usamos la variable del tipo colores (**unColor**) y usamos el método **GetType** para saber que tipo de datos es esa variable y en el segundo ejemplo, usamos la función GetType para averiguar el tipo del parámetro indicado.

Veamos el código del procedimiento **pruebaColores** usando este nuevo sistema de averiguar si un valor es o no correcto (o válido) para la enumeración indicada.

```
Private Sub pruebaColores(ByVal elColor As colores)
```

```
    If System.Enum.IsDefined(elColor.GetType, elColor) = False Then
        elColor = colores.azul
        Console.WriteLine("el valor no es correcto")
    Else
        Console.WriteLine("valor correcto")
    End If
    ' resto del código...
End Sub
```

Aunque no debes preocuparte demasiado, al menos por ahora, si no llegas a entender todo esto que estamos viendo, simplemente "acepta" que funciona y más adelante te das una vueltecita por esta entrega y seguro que lo comprenderás mejor. De todas formas, te voy a explicar con más o menos detalle que es lo que se hace en este procedimiento.

La función IsDefined espera dos parámetros, el primero es el tipo de una enumeración, todos los objetos de .NET Framework tienen el método GetType que indica ese tipo de datos, en este caso usamos la variable elColor (que es del tipo colores) y mediante el método GetType, indicamos el dato que esa función espera. En el segundo parámetro, hay que indicar un valor que será el que se compruebe si está o no definido en dicha enumeración, si ese valor es uno de los incluidos en la enumeración indicada por GetType, la función devolverá True, en caso de que no sea así, devolverá un valor False.

Además de IsDefined, la clase Enum tiene otros métodos que pueden sernos útiles. Pero en lugar de enumerártelos aquí, dejaré que le eches un vistazo a la documentación del .NET Framework (o de Visual Studio .NET), para que practiques en el uso de dicha documentación...

(Guille, no seas malo, anda, dale aunque sea un par de ejemplos, bueno, pero es que si no, no leen la documentación y así... pues no aprenden...)

Valeee, está bien, veamos algunos de los métodos de la clase Enum: (que hay que usar con el System. delante para que el VB no se confunda con el "intento" de una declaración del tipo Enum)

**GetName**, indica el nombre con el que se ha declarado el miembro de la enumeración, por ejemplo, en el ejemplo de la enumeración colores mostrada en la figura 2, el valor 1, sería azul:

System.Enum.GetName(unColor.GetType, 1)

En caso de que el valor indicado no pertenezca a la enumeración, devolverá una cadena vacía.

También se pude obtener el "nombre" del miembro usando ToString: unColor.ToString

**GetNames**, devuelve un array de tipo String con los nombres de todos los miembros de la enumeración.

Lo que es un array lo veremos en detalle en otra ocasión, pero espero que puedas comprender qué es lo que hace el siguiente código sin que te de un "dolor" de cabeza.

```
Dim a() As String = System.Enum.GetNames(unColor.GetType)
Dim i As Integer
For i = 0 To a.Length - 1
    Console.WriteLine("el valor {0} es {1}", i, a(i))
Next
```

**GetValues**, devuelve un array con los valores de los miembros de la enumeración.

El tipo devuelto es del tipo Array, que en realidad no es un array (o matriz) de un tipo de datos específico, sino más bien es el tipo de datos en el que se basan los arrays o matrices.

```
Dim a As Array = System.Enum.GetValues(unColor.GetType)
```

```
For i = 0 To a.Length - 1
    Console.WriteLine("el valor {0} es {1}", i, a.GetValue(i))
Next
```

Por último vamos a ver un método que, casi con toda seguridad veremos en más de una ocasión:

**Parse**, devuelve un valor de tipo Object con el valor de la representación de la cadena indicada en el segundo parámetro. Esa cadena puede ser un valor numérico o una cadena que representa a un miembro de la enumeración.

```
System.Enum.Parse(unColor.GetType, "1")
System.Enum.Parse(unColor.GetType, "azul")
```

Hay más métodos, pero creo que estos que acabo de enumerar son los más interesantes, de todas formas, te invito a seguir investigando por tu cuenta... cosa que, aunque yo no te lo dijera, deberías acostumbrarte a hacer.

Pero ya que estamos con esto de Parse y para ir terminando esta entrega, veamos cómo podemos usar ese método para los tipos de datos que podemos usar en .NET Framework (los cuales vimos en la cuarta entrega).

El método Parse se utiliza para convertir una cadena en un valor numérico, el tipo de número devuelto dependerá del tipo desde el que hemos usado ese método, por ejemplo si hacemos lo siguiente:

```
Dim s As String = "123"
Dim i As Integer = Integer.Parse(s)
```

El valor asignado a la variable numérica **i**, sería el valor 123 que es un número entero válido.
Pero si hacemos esto otro:

```
Dim b As Byte = Byte.Parse(s)
```

También se asignaría el valor 123 a la variable b, que es de tipo Byte, pero el número 123 ya no es un número entero, sino del tipo byte... esto es claro, dirás, pero, si el valor guardado en la variable **s** no estuviese dentro del "rango" de valores aceptados por el tipo Byte, esto produciría una excepción (o error).

```
s = "129.33"
i = Integer.Parse(s)
```

En este caso, el error se produce porque 129.33 no es un número entero válido, por tanto, cuando usemos Parse, o cualquiera de las funciones de conversión, tendremos que tener cuidado de que el valor sea el correcto para el tipo al que queramos asignar el valor...

¿Cómo solucionar este pequeño inconveniente?

No perdiéndote la próxima entrega de este Curso de Iniciación a la Programación con Visual Basic .NET, ya que esta entrega se acaba aquí, así que... a esperar... ¡que remedio!

Pero no te preocupes que no tendrás que esperar mucho... (al menos intentaré que sea antes de que acabe este "pedazo" de puente de la Constitución que empezó hoy día 6 y que dura hasta el lunes día 9)

Veamos las cosillas que hemos visto en esta séptima entrega:

Hemos visto cómo elegir entre varias opciones mediante Select Case, también sabemos cómo crear enumeraciones o constantes simbólicas que están relacionadas de alguna forma, también hemos visto algunos métodos de la clase Enum con los que podemos saber si un valor pertenece a los definidos en la enumeración o con los que podemos saber los nombres de los miembros de dicha enumeración, además de saber cómo

podemos convertir cadenas de caracteres en un valor numérico, mediante el método Parse.

Para la próxima entrega veremos cómo detectar o interceptar errores y algunas otras cosillas, como el tema que también hemos tratado, aunque sólo haya sido de pasada, sobre los arrays o matrices.

# 8. Octava entrega

Como prometí en la entrega anterior, (pero que no sirva de precedente, no sea que te acostumbres), en esta ocasión veremos cómo se interceptan los errores en Visual Basic .NET.

Cuando en el código de nuestra aplicación se produce un error sintáctico, es decir, porque hayamos escrito mal alguna instrucción de Visual Basic .NET, será el propio entorno de desarrollo el que nos avise de que hay algo que no es correcto; a este tipo de errores se suele llamar errores sintáctico o en tiempo de diseño. Pero si lo que ocurre es que hemos asignado un valor erróneo a una variable o hemos realizado una división por cero o estamos intentado acceder a un archivo que no existe, entonces, se producirá un error en tiempo de ejecución, es decir sólo sabremos que hay algo mal cuando el ejecutable esté funcionando.

## 8.1. Dos formas de interceptar errores

### 8.1.1. Control no estructurado de errores

Si conoces las versiones anteriores de Visual Basic, sabrás que la captura de errores se hacía con las instrucciones **On Error.** Si no has trabajado con ninguna versión anterior de Visual Basic, no te preocupes, ya que no tendrás porqué conocer esa forma de interceptar errores, ya que aquí no te lo voy a explicar... pero si aún así, quieres saber más sobre esas instrucciones para detectar errores, tendrás que acudir al curso Básico de VB (de mis páginas) o a otros artículos (también en este mismo sitio, en el que he tratado ese tema).

Seguramente te preguntarás por qué no quiero tratar esa forma de tratar errores... pues, simplemente porque no... (¡guille!) Bueno, seré algo más explícito... porque esa forma de tratar o interceptar errores no es la recomendable y es una forma, que si no se usa correctamente, nos puede ser más perjudicial que beneficiosa; y el uso de los tratamientos de errores es para que nos ayude, no para que nos cree más problemas... siempre te quedará el recurso de poder aprenderlo por tu cuenta, cosa que podrás hacer si es que realmente no te convence la nueva forma de tratar los errores, ya que a algunos puede que nos parezca menos útil que la forma "antigua", también llamada control no estructurado de errores.

No me enrollo más, otra cosa que debes saber, por si quieres indagar por tu cuenta, es que no se pueden mezclar en un mismo procedimiento las dos formas de tratar los errores.

### 8.1.2. Control estructurado de errores

El método recomendado de capturar errores en Visual Basic .NET, es usando la estructura Try Catch Finally.

La forma de usar esta estructura será algo así:

```
Try
        ' el código que puede producir error
Catch [tipo de error a capturar]
        ' código cuando se produzca un error
Finally
        ' código se produzca o no un error
End Try
```

En el bloque Try pondremos el código que puede que produzca un error.

Los bloques Catch y Finally no son los dos obligatorios, pero al menos hay que usar uno de ellos, es decir o usamos Catch o usamos Finally o, usamos los dos, pero como mínimo uno de ellos.

Una vez aclarado que, además de Try, debemos usar o Catch o Finally, veamos para que sirve cada uno de ellos:

Si usamos Catch, esa parte se ejecutará si se produce un error, es la parte que "capturará" el error.

Después de Catch podemos indicar el tipo de error que queremos capturar, incluso podemos usar más de un bloque Catch, si es que nuestra intención es detectar diferentes tipos de errores.

En el caso de que sólo usemos Finally, tendremos que tener en cuenta de que si se produce un error, el programa se detendrá de la misma forma que si no hubiésemos usado la detección de errores, por tanto, aunque usemos Finally (y no estemos obligados a usar Catch), es más que recomendable que siempre usemos una cláusula Catch, aunque en ese bloque no hagamos nada, pero aunque no "tratemos" correctamente el error, al menos no se detendrá porque se haya producido el error.

Aclaremos este punto, ya que puede parecer extraño.

Si decidimos "prevenir" que se produzca un error, pero simplemente queremos que el programa continúe su ejecución, podemos usar un bloque Catch que esté vacío, con lo cual el error simplemente se ignorará... si has usado o has leído sobre cómo funciona On Error Resume Next, pensarás que esto es algo parecido... si se produce un error, se ignora y se continúa el programa como si nada. Pero no te confundas que aunque lo parezca... no es igual. (guille, no te enrolles más y explícalo)

Si tenemos el siguiente código, se producirá una excepción (o error), ya que al dividir i por j, se producirá un error de división por cero.

```
Dim i, j As Integer
Try
    i = 10
    j = 0
    i = i \ j
Catch
        ' nada que hacer si se produce un error
End Try
' se continúa después del bloque de detección de errores
```

Pero cuando se produzca ese error, no se ejecutará ningún código de "tratamiento" de errores, ya que dentro del bloque Catch no hay ningún código.

Seguro que pensarás (o al menos así deberías hacerlo), que eso es claro y evidente.

Si usáramos On Error Resume Next, el código podría ser algo como esto:

```
Dim i, j As Integer
On Error Resume Next
i = 10
j = 0
i = i \ j
' se continúa después del bloque de detección de errores
```

Bueno, si nos basamos en estos dos ejemplos, ambos hacen lo mismo: Si se produce un error se continúa justo por donde está el comentario ese que dice ' se continúa... pero... (¿de que te extrañas? ya deberías saber que "casi" siempre hay un pero), cuando se usan los bloques Try... Catch... el tratamiento de errores es diferente a cuando se usa el

"antiguo" On Error Resume Next, para que nos entendamos, talvez este otro ejemplo aclare un poco las cosas:

```
Dim i, j As Integer
Try
    i = 10
    j = 0
    i = i \ j
    Console.WriteLine("el nuevo valor de i es: {0}", i)
Catch
    ' nada que hacer si se produce un error
End Try
' se continúa después del bloque de detección de errores
Console.WriteLine("después del bloque de detección de errores")
```

Aquí tenemos prácticamente el mismo código que antes, con la diferencia de que tenemos dos "instrucciones" nuevas, una se ejecuta después de la línea que "sabemos" que produce el error, (sí, la línea `i = i \ j` es la que produce el error, pero el que sepamos qué línea es la que produce el error no es lo habitual, y si en este caso lo sabemos con total certeza, es sólo es para que comprendamos mejor todo esto...), y la otra después del bloque Try... Catch.

Cuando se produce el error, el Visual Basic .NET, o mejor dicho, el runtime del .NET Framework, deja de ejecutar las líneas de código que hay en el bloque Try, (las que hay después de que se produzca la excepción), y continúa por el código del bloque Catch, que en este caso no hay nada, así que, busca un bloque Finally, y si lo hubiera, ejecutaría ese código, pero como no hay bloque Finally, continúa por lo que haya después de End Try.

¿Te ha quedado claro?

Te lo repito y te muestro lo que ocurriría en un bloque Try... Catch si se produce o no un error:

**Si se produce una excepción** o error en un bloque Try, el código que siga a la línea que ha producido el error, deja de ejecutarse, para pasar a ejecutar el código que haya en el bloque Catch, se seguiría (si lo hubiera) por el bloque Finally y por último con lo que haya después de End Try.

**Si no se produce ningún error**, se continuaría con todo el código que haya en el bloque Try y después se seguiría, (si lo hubiera), con el bloque Finally y por último con lo que haya después de End Try.

Ahora veamos el código equivalente si se usara On Error Resume Next:

```
Dim i, j As Integer
On Error Resume Next
i = 10
j = 0
i = i \ j
Console.WriteLine("el nuevo valor de i es: {0}", i)
' se continúa después del bloque de detección de errores
Console.WriteLine("después del bloque de detección de errores")
```

En este caso, cuando se produce un error, se continúa ejecutando el código, por tanto "se ejecutarán todas las líneas", entre otras cosas porque no hay ninguna distinción del código que se está "chequeando" del que no se comprueba.

Que te parece mejor solución, ya que así no te preocupas de nada cuando se produzca un error... ¡pues vale!

Pero... piénsatelo antes de usar un código como este... en el que al fin y al cabo no ocurriría nada extraño... pero si en lugar de ser un código simple, dependiéramos de que

no se debe producir un error para continuar procesando el código, entonces lo tendríamos más complicado.

Veamos un ejemplo, que no será con código real, sino con "pseudo código", el que abrimos un archivo (o fichero) de texto y en el que vamos a escribir el contenido de una o varias variables. Si no se produce error, todo irá bien, pero si se produce un error justamente cuando vamos a abrir el archivo, (por ejemplo, porque el disco esté lleno o no tengamos acceso de escritura o cualquier otra cosa), el archivo no estará abierto y por tanto lo que guardemos en él, realmente no se guardará.

El "pseudo código" sería algo como esto:

Abrir el archivo

    Guardar el contenido de la variable

    repetir la línea anterior mientras haya algo que guardar

Cerrar el archivo

Avisar al usuario de que todo ha ido bien, que ya se puede ir a su casa a descansar porque todo se ha guardado.

Si usamos On Error Resume Next, se producirá un error al abrir el archivo, pero como le hemos dicho que queremos continuar, seguirá "palante"  e intentará guardar información en dicho archivo, pero como no puede guardar esa información, se volverá a producir un error, pero como continúa aunque haya errores, ni nos enteramos, pero se estarán produciendo errores de continuo y "no pasará nada" porque así lo hemos querido.

Y tu te preguntarás... ¿y que pasa? esa era la intención de usar lo de Resume Next. Pues sí, para que engañarnos, pero si lo que teníamos que guardar en el fichero ese eran 10000 líneas de texto, seguramente habremos desperdiciado un tiempo "precioso", a demás de que puede que el usuario ni se haya percatado de que ha estado esperando un buen rato y al final no se ha guardado la información.

Pero... como eres una persona que no se conforma con lo que yo digo y te has leído más cosas de "la otra forma" de detectar errores, seguramente me dirás que podrías haber usado On Error Goto nnn y haber mostrado un mensaje de aviso al usuario.

Pues ¡muy bien! Eso estaría bien, pero se sale de lo que aquí estamos, tratando, así que... ¡cállate! y ¡sigue leyendo!

Si en lugar de usar On Error Resume Next (e incluso On Error Goto nnn), hemos optado por usar Try... Catch, todo el seudo código ese, lo escribiríamos dentro del bloque Try y si se produce un error, avisaríamos al usuario de que se ha olvidado de poner el disquete, (por ejemplo), y le podríamos dar la oportunidad de volver a intentarlo...

Sí, ya se que eso mismo podríamos hacerlo con el On Error... pero hacerlo con Try resulta más elegante y no tendríamos la necesidad de usar un salto a una línea anterior usando una instrucción que... en fin... no sé si debería decirte que instrucción usarías... pero sí, es GoTo, la terrible instrucción que ha hecho del lenguaje Basic un lenguaje al que nadie ha tratado con "seriedad" y por culpa de la que este querido lenguaje de programación ha sido tan "blasfemado"... o casi, que tampoco es "pa" tanto, pero en fin...

De todas formas, hasta aquí hemos llegado con esto del Goto, que ni te voy a explicar para que sirve y también hemos llegado al final de todo lo que a tratamiento de errores no estructurados se refiere... por tanto, si quieres saber más de esas instrucciones... ya sabes, o te das una vuelta por algún "viejo" curso del VB o te lees lo que la documentación de Visual Basic .NET te diga, porque yo, no te voy a decir nada más de esas instrucciones...

Sigamos con el tratamiento de errores estructurados.

Como te he comentado antes, el bloque Catch sirve para detectar errores, incluso para detectar distintos tipos de errores, con idea de que el "runtime" de .NET Framework (el CLR), pueda ejecutar el que convenga según el error que se produzca.

Esto es así, porque es posible que un bloque Try se produzcan errores de diferente tipo y si tenemos la "previsión" de que se puede producir algún que otro error, puede que queramos tener la certeza de que estamos detectando distintas posibilidades, por ejemplo, en el "pseudo código" mostrado anteriormente, es posible que el error se produzca porque el disco está lleno, porque no tenemos acceso de escritura, porque no haya disco en la disquetera, etc. Y podría sernos interesante dar un aviso correcto al usuario de nuestra aplicación, según el tipo de error que se produzca.

Cuando queremos hacerlo de esta forma, lo más lógico es que usemos un Catch para cada uno de los errores que queremos interceptar, y lo haríamos de la siguiente forma:

```
Dim i, j As Integer
Dim s As String
'
Try
    Console.Write("Escribe un número (y pulsa Intro) ")
    s = Console.ReadLine
    i = CInt(s)
    Console.Write("Escribe otro número ")
    s = Console.ReadLine
    j = CInt(s)
    '
    Console.WriteLine("El resultado de dividir {0} por {1} es {2}", i, j, i
\ j)
    '
Catch ex As DivideByZeroException
    Console.WriteLine("ERROR: división por cero")
Catch ex As OverflowException
    Console.WriteLine("ERROR: de desbordamiento (número demasiado grande)")
Catch ex As Exception
    Console.WriteLine("Se ha producido el error: {0}", ex.Message)
End Try
'
Console.ReadLine()
```

Aquí estamos detectando tres tipos de errores:

- El primero si se produce una división por cero.
- El segundo si se produce un desbordamiento, el número introducido es más grande de lo esperado.
- Y por último, un tratamiento "genérico" de errores, el cual interceptará cualquier error que no sea uno de los dos anteriores.

Si usamos esta forma de detectar varios errores, te comentaré que debes tener cuidado de poner el tipo genérico al final, (o el que no tenga ningún tipo de "error a capturar" después de Catch), ya que el CLR siempre evalúa los tipos de errores a detectar empezando por el primer Catch y si no se amolda al error producido, comprueba el siguiente, así hasta que llegue a uno que sea adecuado al error producido, y si da la casualidad de que el primer Catch es de tipo genérico, el resto no se comprobará, ya que ese tipo es adecuado al error que se produzca, por la sencilla razón de que **Exception** es el tipo de error más genérico que puede haber, por tanto se adecua a cualquier error.

**Nota:** Realmente el tipo Exception es la clase de la que se derivan (o en la que se basan) todas las clases que manejan algún tipo de excepción o error.

Si te fijas, verás que todos los tipos de excepciones que podemos usar con Catch, terminan con la palabra Exception, esto, además de ser una "norma" o recomendación nos sirve para saber que ese objeto es válido para su uso con Catch. Esto lo deberíamos tener en cuenta cuando avancemos en nuestro aprendizaje y sepamos crear nuestras propias excepciones.

Por otro lado, si sólo usamos tipos específicos de excepciones y se produce un error que no es adecuado a los tipos que queremos interceptar, se producirá una excepción "no interceptada" y el programa finalizará.

Para poder comprobarlo, puedes usar el siguiente código y si simplemente pulsas intro, sin escribir nada o escribes algo que no sea un número, se producirá un error que no está detectado por ninguno de los Catch:

```
Dim i, j As Integer
Dim s As String
'
Try
    Console.Write("Escribe un número (y pulsa Intro) ")
    s = Console.ReadLine
    i = CInt(s)
    Console.Write("Escribe otro número ")
    s = Console.ReadLine
    j = CInt(s)
    '
    Console.WriteLine("El resultado de dividir {0} por {1} es {2}", i, j, i
\ j)
    '
Catch ex As DivideByZeroException
    Console.WriteLine("ERROR: división por cero")
Catch ex As OverflowException
    Console.WriteLine("ERROR: de desbordamiento (número demasiado grande)")
End Try
```

Sabiendo esto, mi recomendación es que siempre uses un "capturador" genérico de errores, es decir un bloque Catch con el tipo de excepción genérica: Catch variable As Exception.

Después de Catch puedes usar el nombre de variable que quieras, la recomendada es **ex**, que por cierto, es el nombre que de forma predeterminada muestra el Visual Studio .NET 2003.

Cuando escribimos Try y pulsamos Intro, en la versión 2002 (la primera versión definitiva de Visual Studio .NET), cuando escribimos Try, sólo se muestra el bloque Try y el final: End Try junto a un aviso (dientes de sierra) de que Try necesita un bloque Catch o Finally, pero en la futura versión de VS .NET (futura en la fecha en que escribo esta entrega, ya que ahora mismo sólo es una beta), cuando se escribe Try y se pulsa Intro, además del End Try, se muestra el bloque Catc ex As Exception.

Bien, esto no se ha acabado aún, ya que hay más cosas que contar sobre Try... Catch, como por ejemplo:

- que podemos "lanzar" excepciones, sin necesidad de que se produzca una explícitamente,
- cómo se comporta el Visual Basic .NET cuando ejecutamos un código que produce error y no hay un bloque específico de tratamiento para ese error, pero existe un bloque Try que aunque esté en otra parte del código sigue activo,
- o que podemos crear nuestras propios tipos de excepciones, pero eso lo veremos cuando tratemos el tema de las clases un poco más a fondo.

Ahora empezaremos viendo

## 8.2. Cómo hacer que se produzca una excepción:

Para lanzar (o crear) excepciones tendremos que usar la instrucción **Throw** seguida de un objeto derivado del tipo Exception.

Normalmente se hace de la siguiente forma:

```
Throw New Exception("Esto es un error personalizado")
```

Con este código estamos indicándole al Visual Basic .NET que queremos "lanzar" (Throw) una nueva excepción (New Exception) y que el mensaje que se mostrará, será el que le indiquemos dentro de los paréntesis.

Cuando nosotros lanzamos (o creamos) una excepción, el error lo interceptará un bloque Try (o una instrucción On Error) que esté activo, si no hay ningún bloque Try activo, será el CLR (runtime de .NET Framework) el que se encargará de interceptar esa excepción, pero deteniendo la ejecución del programa... y esto último me recuerda que tengo que explicarte lo que ocurre cuando hay un bloque Try "activo"...

Supongamos que tenemos un bloque Try desde el cual llamamos a algún procedimiento (Sub, Function, etc.), que puede que a su vez llame a otro procedimiento y resulta que en alguno de esos procedimientos se produce una excepción "no controlada", por ejemplo, si tenemos el siguiente código:

```
Sub Main()
    Dim n, m As Integer
    '
    Try
        n = 10
        m = 15
        Dim k As Integer = n + m
        '
        ' llamanos a un procedimiento
        Prueba()
        '
        '... más código...
        '
        '
    Catch ex As DivideByZeroException
        Console.WriteLine("ERROR: división por cero")
    Catch ex As OverflowException
        Console.WriteLine("ERROR:  de  desbordamiento  (número  demasiado
grande)")
    Catch ex As InvalidCastException
        Console.WriteLine("ERROR: lo escrito no es un número.")
    Catch ex As Exception
        Console.WriteLine("Se  ha  producido  el  error:  {0}  {1}  {2}",
ex.Message, vbCrLf, ex.ToString)
    End Try
    '
    Console.WriteLine("Pulsa Intro para terminar")
    Console.ReadLine()
End Sub

Sub Prueba()
    Dim i, j As Integer
    Dim s As String
    Console.Write("Escribe un número (y pulsa Intro) ")
    s = Console.ReadLine
    i = CInt(s)
    Console.Write("Escribe otro número ")
```

```
    s = Console.ReadLine
    j = CInt(s)
    Console.WriteLine("El resultado de dividir {0} por {1} es {2}", i, j, i
\ j)
End Sub
```

En el procedimiento **Main** tenemos cierto código, en el que hemos usado un bloque Try... Catch, dentro del bloque Try además de otras cosas, llamamos al procedimiento **Prueba**, en el cual se piden dos números y se realizan unas operaciones con esos números, pero en ese procedimiento no tenemos ningún bloque Try que pueda "interceptar" errores.

Tal como vimos anteriormente, si simplemente pulsamos Intro cuando nos pide alguno de esos números o cuando escribimos en el segundo un cero o si alguno de esos dos números que introducimos es más grande que lo que un tipo Integer pude soportar, se producirá un error (o excepción), pero, resulta que dentro del procedimiento Prueba no tenemos nada que "intercepte" los posibles errores que se puedan producir. Si ese código fuese el único que tuviéramos en el programa y se produjera una excepción, sería el CLR o runtime del .NET Framework el que se encargaría de avisarnos de que algo va mal (deteniendo la ejecución del programa). Pero, cuando se llama al procedimiento Prueba desde Main, hay un bloque Try "activo" y el CLR se da cuenta de ese detalle y en lugar de detener el programa y mostrar el error, lo que hace es "pasar" esa excepción a dicho bloque Try (porque está activo) y "confiar" que el error sea "tratado" por dicho bloque.

¿Te enteras?

¿No?

Pues no te preocupes, que algún día puede que te resulte claro... je, je... por ahora, simplemente compruébalo y experimenta... y sigue leyendo, que aún queda un poco más...

A ver si con este otro ejemplo lo entiendes mejor.

Es parecido al anterior, pero con un "pequeño" detalle que lo diferencia:

```
Sub Main()
    Dim n, m As Integer
    '
    Try
        n = 10
        m = 15
        Dim k As Integer = n + m
        '
        '... más código...
        '
        '
    Catch ex As DivideByZeroException
        Console.WriteLine("ERROR: división por cero")
    Catch ex As OverflowException
        Console.WriteLine("ERROR:  de  desbordamiento  (número  demasiado
grande)")
    Catch ex As InvalidCastException
        Console.WriteLine("ERROR: lo escrito no es un número.")
    Catch ex As Exception
        Console.WriteLine("Se  ha  producido  el  error:  {0}  {1}  {2}",
ex.Message, vbCrLf, ex.ToString)
    End Try
    '
    ' llamanos a un procedimiento
    Prueba()
    '
    '
    Console.WriteLine("Pulsa Intro para terminar")
```

```
    Console.ReadLine()
End Sub

Sub Prueba()
    Dim i, j As Integer
    Dim s As String
    Console.Write("Escribe un número (y pulsa Intro) ")
    s = Console.ReadLine
    i = CInt(s)
    Console.Write("Escribe otro número ")
    s = Console.ReadLine
    j = CInt(s)
    Console.WriteLine("El resultado de dividir {0} por {1} es {2}", i, j, i
\ j)
End Sub
```

El código del procedimiento Prueba es exactamente igual que el anterior, pero en el procedimiento Main, la llamada a dicho procedimiento no está "dentro" del bloque Try; por tanto si ejecutas el código y cuando te pida alguno de esos dos números escribes algo "incorrecto", se producirá, como bien supones, una excepción, pero en esta ocasión no hay ningún bloque Try que intercepte ese error, por tanto, será el CLR el que se encargue de avisarnos de que algo va mal, pero el CLR nos lo dice de una forma "brusca" y poco educada... ya que detiene el programa y se acabó lo que se daba...

Pues eso... no quiero parecer "mal educado", pero..., se acabó lo que se daba y hasta aquí hemos llegado en esta octava entrega dedicada exclusivamente al tratamiento de errores... al final no he tratado de los arrays... esperemos que sea en la próxima entrega... cosa que descubrirás en cuanto esté publicada, así que, ahora dedícate a "indagar" por tu cuenta y riesgo y sigue probando con esto de las excepciones y los bloques Try... Catch.

# 9. Novena entrega

En esta novena entrega del curso de iniciación a la programación con Visual Basic .NET vamos a ver un tipo especial de datos, al menos eso será lo que pienses cuando te diga lo que es pero, (si has trabajado con las versiones anteriores de Visual Basic), a lo largo de esta entrega descubrirás que no todo lo que creías sigue siendo igual que antes... no, no te preocupes, no hablaré de **Grandes Temas** (en mayúsculas), ni de lo que puede que estés pensando al ver la fecha en la que escribo esta entrega... aquí sólo se tratan temas relacionados con la programación, así que descansa, relájate y no te preocupes, que esta noche podrás ir a la Misa del Gallo... (¡Guille! te van a tachar de ateo y blasfemo, así que *cuidadín* con lo que dices...)

Que el personal esté tranquilo, que me refería al tema de los Arrays.

Si no sabes nada de Arrays, (*o matrices o arreglos, como prefieras llamarlo, en la documentación en español de Visual Studio .NET se llaman matrices, pero yo voy a usar el nombre en inglés, ya que es el que estoy acostumbrado a usar.*), lo que voy a comentar en este párrafo te será indiferente, pero aún así, podrías leerlo para que tengas más oportunidades de saber de qué estoy hablando...

Los arrays en las versiones anteriores de Visual Basic eran tipos de datos de los llamados por valor, al igual que lo son los tipos Integer, Double, etcétera. Pero en Visual Basic .NET, los arrays realmente son tipos por referencia.

Antes de entrar en detalles sobre los arrays, creo que, ya que lo he mencionado, deberíamos ver en qué se diferencian los tipos de datos por valor y los tipos de datos por referencia.

## 9.1. Tipos de datos por valor

Cuando dimensionamos una variable, por ejemplo, de tipo Integer, el CLR reserva el espacio de memoria que necesita para almacenar un valor del tipo indicado; cuando a esa variable le asignamos un valor, el CLR almacena dicho valor en la posición de memoria que reservó para esa variable. Si posteriormente asignamos un nuevo valor a dicha variable, ese valor se almacena también en la memoria, sustituyendo el anterior, eso lo tenemos claro, ya que así es como funcionan las variables. Si a continuación creamos una segunda variable y le asignamos el valor que contenía la primera, tendremos dos posiciones de memoria con dos valores, que por pura casualidad, resulta que son iguales, entre otras cosas porque a la segunda hemos asignado un valor "igual" al que tenía la primera.

Vale, seguramente pensarás que eso es así y así es como te lo he explicado o al menos, así es como lo habías entendido.

Pero, espera a leer lo siguiente y después seguimos hablando... a ver si te parece que lo tenías "aprendido".

## 9.2. Tipos de datos por referencia

En Visual Basic .NET también podemos crear objetos, los objetos se crean a partir de una clase, (del tema de las clases hablaremos largo y tendido en entregas posteriores). Cuando dimensionamos una variable cuyo tipo es una clase, simplemente estamos creando una variable que es capaz de manipular un objeto de ese tipo, en el momento de la declaración, el CLR no reserva espacio para el objeto que contendrá esa variable, ya que esto sólo lo hace cuando usamos la instrucción New. En el momento en que creamos el objeto, (mediante New), es cuando el CLR reserva la memoria para dicho objeto y le dice a la variable en que parte de la memoria está almacenado, de forma que la variable

pueda acceder al objeto que se ha creado. Si posteriormente declaramos otra variable del mismo tipo que la primera, tendremos dos variables que saben "manejar" datos de ese tipo, pero si a la segunda variable le asignamos el contenido de la primera, en la memoria no existirán dos copias de ese objeto, sólo existirá un objeto que estará referenciado por dos variables. Por tanto, cualquier cambio que se haga en dicho objeto se reflejará en ambas variables.

¿Qué? ¿Te aclaras? A que no... pues no me extraña... porque esto no es para aclararse... pero, confía en mí y aunque no te enteres, sigue leyendo, no abandones ahora... (lo que no te dice el Guille es que todavía quedan cosas más complicadas y difíciles de digerir... je, je, ¡no te quea ná!)

Vamos a ver un par de ejemplos para aclarar esto de los tipos por valor y los tipos por referencia.

Crea un nuevo proyecto de tipo consola y añade el siguiente código:

```
Sub Main()
    ' creamos una variable de tipo Integer
    Dim i As Integer
    ' le asignamos un valor
    i = 15
    ' mostramos el valor de i
    Console.WriteLine("i vale {0}", i)
    '
    ' creamos otra variable
    Dim j As Integer
    ' le asignamos el valor de i
    j = i
    Console.WriteLine("hacemos esta asignación: j = i")
    '
    ' mostramos cuanto contienen las variables
    Console.WriteLine("i vale {0} y j vale {1}", i, j)
    '
    ' cambiamos el valor de i
    i = 25
    Console.WriteLine("hacemos esta asignación: i = 25")
    ' mostramos nuevamente los valores
    Console.WriteLine("i vale {0} y j vale {1}", i, j)
    '
    Console.WriteLine("Pulsa Intro para finalizar")
    Console.ReadLine()
End Sub
```

Como puedes comprobar, cada variable tiene un valor independiente del otro.

Esto está claro.

Ahora vamos a ver qué es lo que pasa con los tipos por referencia.

Para el siguiente ejemplo, vamos a crear una clase con una sola propiedad, ya que las clases a diferencia de los tipos por valor, deben tener propiedades a las que asignarles algún valor.

No te preocupes si no te enteras, que de todo esto hablaremos en otra ocasión, pero por ahora sólo es para ver, prácticamente, esto de los tipos por referencia.

```
Class prueba
    Public Nombre As String
End Class

Sub Main()
    ' creamos una variable de tipo prueba
    Dim a As prueba
```

```
    ' creamos (instanciamos) el objeto en memoria
    a = New prueba()
    ' le asignamos un valor
    a.Nombre = "hola"
    ' mostramos el contenido de a
    Console.WriteLine("a vale {0}", a.Nombre)
    '
    ' dimensionamos otra variable
    Dim b As prueba
    '
    ' asignamos a la nueva el valor de a
    b = a
    Console.WriteLine("hacemos esta asignación: b = a")
    '
    ' mostramos el contenido de las dos
    Console.WriteLine("a vale {0} y b vale {1}", a.Nombre, b.Nombre)
    ' cambiamos el valor de la anterior
    a.Nombre = "adios"
    '
    Console.WriteLine("hacemos una nueva asignación a a.Nombre")
    '
    ' mostramos nuevamente los valores
    Console.WriteLine("a vale {0} y b vale {1}", a.Nombre, b.Nombre)
    '
    Console.WriteLine("Pulsa Intro para finalizar")
    Console.ReadLine()
End Sub
```

La clase **prueba** es una clase muy simple, pero como para tratar de los tipos por referencia necesitamos una clase, he preferido usar una creada por nosotros que cualquiera de las clases que el .NET Framework nos ofrece.

Dimensionamos una variable de ese tipo y después creamos un nuevo objeto del tipo **prueba**, el cual asignamos a la variable **a**.

Una vez que tenemos "instanciado" (o creado) el objeto al que hace referencia la variable **a**, le asignamos a la propiedad **Nombre** de dicho objeto un valor.

Lo siguiente que hacemos es declarar otra variable del tipo **prueba** y le asignamos lo que contiene la primera variable.

Hasta aquí, es casi lo mismo que hicimos anteriormente con las variables de tipo Integer. La única diferencia es la forma de manipular las clases, ya que no podemos usarlas "directamente", porque tenemos que crearlas (mediante New) y asignar el valor a una de las propiedades que dicha clase contenga. Esta es la primera diferencia entre los tipos por valor y los tipos por referencia, pero no es lo que queríamos comprobar, así que sigamos con la explicación del código mostrado.

Cuando mostramos el contenido de la propiedad **Nombre** de ambas variables, las dos muestran lo mismo, que es lo esperado, pero cuando asignamos un nuevo valor a la variable **a**, al volver a mostrar los valores de las dos variables, ¡las dos siguen mostrando lo mismo!

Y esto no es lo que ocurría en el primer ejemplo.

¿Por qué ocurre? Porque las dos variables, apuntan al mismo objeto que está creado en la memoria.

**¡Las dos variables hacen referencia al mismo objeto!** y cuando se realiza un cambio mediante una variable, ese cambio afecta también a la otra, por la sencilla razón que se está modificando el único objeto de ese tipo que hay creado en la memoria.

Para simplificar, los tipos por valor son los tipos de datos que para usarlos no tenemos que usar New, mientras que los tipos por referencia son tipos (clases) que para crear un nuevo objeto hay que usar la instrucción New.

Los tipos por valor son los tipos "básicos" (o elementales), tales como Integer, Double, Decimal, Boolean, etcétera. En la primera tabla de la entrega cuatro tienes una relación de los tipos por valor.

Los tipos por referencia son todos los demás tipos.

Sin querer parecer que quiero confundirte más, te diré que en .NET Framework, realmente todos los tipos son clases, aunque esos que están relacionados en la tabla mencionada, son los que actúan como tipos por valor y el .NET Framework los trata de una forma especial... ¡vale! lo dejo, no voy a "calentarte" más la cabeza, que seguramente ya la tendrás en ebullición...

Ahora sí, ahora vamos a seguir complicándonos la existencia.

## 9.3. Los Arrays

Las variables que hemos estado usando hasta ahora, eran de tipo escalar: sólo pueden contener un valor a la vez, (no te asustes por la palabra esa que ha usado el Guille: *escalar*, es que lo ha leído en la ayuda del VS y se las quiere dar de *entendido*). Pero resulta que en ocasiones nos podemos ver en la necesidad de querer tener en una misma variable, valores que de alguna forma están relacionados. Por ejemplo, si tenemos una colección de discos, nos podría interesar tener esa discografía incluida en una misma variable para poder acceder a cualquiera de esos discos sin necesidad de tener que crear una variable distinta para cada uno de los discos, ya que sería totalmente ineficiente si, por ejemplo, quisiéramos imprimir una relación de los mismos.

Realmente para el ejemplo este que estoy poniendo, hay otros tipos de datos que serían más prácticos, pero... es eso, sólo un ejemplo, y cuando veamos esos otros tipos de datos, serás tú el que decida cual utilizar.

Una de las formas en las que podemos agrupar varios datos es mediante los arrays (o matrices).
Usando un array, podemos acceder a cualquiera de los valores que tenemos almacenado mediante un índice numérico. Por ejemplo, si tenemos la variable **discografía** y queremos acceder al tercer disco, podríamos hacerlo de la siguiente forma: **discografía(3)**.

Sabiendo esto, podemos comprobar que sería fácil recorrer el contenido de los arrays mediante un bucle For.

## 9.4. ¿Qué tipos de datos se pueden usar para crear arrays?

Los tipos de datos de las variables usadas como array, pueden ser de cualquier tipo, dependiendo de lo que queramos guardar. Por ejemplo, si queremos guardar los nombres de los discos que tenemos, podemos usar un array del tipo String, que lo que nos interesa es saber el porcentaje de goles por partido de nuestro equipo de fútbol favorito, el tipo de datos del array podía ser Decimal. Incluso si queremos, también podemos crear un array de un tipo que nosotros hayamos definido o de cualquier clase que exista en el .NET Framework.

Vale, muy bien, pero ¿cómo narices le digo al Visual Basic que una variable es un array?

## 9.5. Declarar variables como arrays

Para poder indicarle al VB que nuestra intención es crear un array podemos hacerlo de dos formas distintas, para este ejemplo crearemos un array de tipo Integer:

1- La clásica (la usada en versiones anteriores)

```
Dim a() As Integer
```

2- La nueva forma introducida en .NET:

```
Dim a As Integer()
```

De cualquiera de estas dos formas estaríamos creando un array de tipo Integer llamada **a**.

Cuando declaramos una variable de esta forma, sólo le estamos indicando al VB que nuestra intención es que la variable **a** sea un array de tipo Integer, pero ese array no tiene reservado ningún espacio de memoria.

## 9.6. Reservar memoria para un array

Para poder hacerlo tenemos que usar la instrucción **ReDim**:

```
ReDim a(5)
```

Al ejecutarse este código, tendremos un array con capacidad para 6 elementos.

Y son seis y no cinco, (como por lógica habría que esperar), porque en .NET Framework el índice menor de un array siempre es cero y en Visual Basic, el índice superior es el indicado entre paréntesis. Por tanto el array **a** tendrá reservada memoria para 6 valores de tipo Integer, los índices serían desde 0 hasta 5 ambos inclusive.

Además de usar ReDim, que realmente sirve para "*redim*ensionar" el contenido de un array, es decir, para volver a dimensionarlo o cambiarlo por un nuevo valor. Si sabemos con antelación el tamaño que contendrá el array, podemos hacerlo de esta forma:

```
Dim a(5) As Integer
```

Con este código estaríamos declarando la variable **a** como un array de 6 elementos (de 0 a 5) del tipo Integer.

Cuando indicamos la cantidad de elementos que contendrá el array no podemos usar la segunda forma de declaración que te mostré anteriormente: `Dim a As Integer(5)` ya que esto produciría un error sintáctico

.

**Nota:** Aunque este curso es de Visual Basic .NET te diré que en C#, cuando declaramos un array con el equivalente a `Dim a(5) As Integer`, que sería algo como:
`int[] a = new int[5]`, lo que estamos creando es un array de tipo int (el Integer de C#) con 5 elementos, de 0 a 4.

El que en Visual Basic tenga este tratamiento diferente en la declaración de los arrays, según nos dicen, es por compatibilidad con las versiones anteriores... ¿compatibilidad? ¡JA!

La verdad es que muchas veces parece como si quisieran seguir marginando a los programadores de Visual Basic. Ahora que con VB.NET tenemos un lenguaje "decente", nos lo enfangan con una pretendida compatibilidad "hacia atrás"... es que estos diseñadores de lenguajes no saben eso de: "para atrás, ni para tomar carrerilla" (el Guille hubiese dicho: patrás, ni pa coger carrerilla).

Pero esto es lo que hay, así que... tendremos que adaptarnos.

## 9.7. Asignar valores a un array

Para asignar un valor a un elemento de un array, se hace de la misma forma que con las variables normales, pero indicando el índice (o posición) en el que guardará el valor.

Por ejemplo, para almacenar el valor 15 en la posición 3 del array a, haríamos lo siguiente:

a(3) = 15

## 9.8. Acceder a un elemento de un array

De igual forma, si queremos utilizar ese elemento que está en la posición 3 para una operación, podemos hacerlo como con el resto de las variables, pero siempre usando el paréntesis y el número de elemento al que queremos acceder:

i = b * a(3)

El índice para poder acceder al elemento del array puede ser cualquier expresión numérica, una variable o como hemos estado viendo en estos ejemplos, una constante. La única condición es que el resultado sea un número entero.

Por tanto, podríamos acceder a la posición indicada entre paréntesis, siempre que el resultado sea un valor entero y, por supuesto, esté dentro del rango que hayamos dado al declarar el array:

x = a(i + k)

**Nota aclaratoria:** Cuando digo el elemento que está en la posición 3, no me refiero al tercer elemento del array, ya que si un array empieza por el elemento 0, el tercer elemento será el que esté en la posición 2, ya que el primer elemento será el que ocupe la posición cero.

## 9.9. Los límites de los índices de un array

Como ya he comentado antes, el índice inferior de un array, **siempre** es cero, esto es invariable, todos los arrays de .NET Framework empiezan a contar por cero.

Pero el índice superior puede ser el que nosotros queramos, aunque sin pasarnos, que la memoria disponible se puede agotar si pretendemos usar un valor exageradamente alto. Realmente el índice superior de un array es 2^64 -1 (el valor máximo de un tipo Long)

## 9.10. Saber el tamaño de un array

Cuando tenemos un array declarado y asignado, podemos acceder a los elementos de ese array mediante un índice, esto ya lo hemos visto; pero si no queremos "pasarnos" cuando queramos acceder a esos elementos, nos será de utilidad saber cuantos elementos tiene el array, para ello podemos usar la propiedad **Length**, la cual devuelve el número total de elementos, por tanto, esos elementos estarán comprendidos entre 0 y Length - 1.

Esto es útil si queremos acceder mediante un bucle For, en el siguiente código se mostrarían todos los elementos del array **a**:

```
For i = 0 To a.Length - 1
    Console.WriteLine(a(i))
```

```
Next
```

## 9.11. Inicializar un array al declararla

Al igual que las variables normales se pueden declarar y al mismo tiempo asignarle un valor inicial, con los arrays también podemos hacerlo, pero de una forma diferente, ya que no es lo mismo asignar un valor que varios.

Aunque hay que tener presente que si inicializamos un array al declararla, no podemos indicar el número de elementos que tendrá, ya que el número de elementos estará supeditado a los valores asignados.

Para inicializar un array debemos declarar ese array sin indicar el número de elementos que contendrá, seguida de un signo igual y a continuación los valores encerrados en llaves. Veamos un ejemplo:

```
Dim a() As Integer = {1, 42, 15, 90, 2}
```

También podemos hacerlo de esta otra forma:

```
Dim a As Integer() = {1, 42, 15, 90, 2}
```

Usando cualquiera de estas dos formas mostradas, el número de elementos será 5, por tanto los índices irán desde 0 hasta 4.

## 9.12. Los arrays pueden ser de cualquier tipo

En todos estos ejemplos estamos usando valores de tipo Integer, pero podríamos hacer lo mismo si fuesen de tipo String o cualquier otro.

En el caso de que sean datos de tipo String, los valores a asignar deberán estar entre comillas dobles o ser variables de tipo String. Por ejemplo:

```
Dim s As String() = {"Hola", "Mundo, ", "te", "saludo"}

s(3) = "saludamos"

Dim i As Integer
For i = 0 To s.Length - 1
    Console.WriteLine(s(i))
Next
```

## 9.13. Usar un bucle For Each para recorrer los elementos de un array

El tipo de bucle For Each es muy útil para recorrer los elementos de un array, además de ser una de las pocas, por no decir la única, formas de poder acceder a un elemento de un array sin indicar el índice.

```
Dim a() As Integer = {1, 42, 15, 90, 2}
'
Console.WriteLine("Elementos del array a()= {0}", a.Length)
'
Dim i As Integer
For Each i In a
    Console.WriteLine(i)
Next
'
Console.WriteLine("Pulsa Intro para finalizar")
Console.ReadLine()
```

## 9.14. Clasificar el contenido de un array

Todos los arrays están basados realmente en una clase del .NET Framework, (recuerda que TODO en .NET Framework son clases, aunque algunas con un tratamiento especial)

La clase en las que se basan los arrays, es precisamente una llamada **Array**.

Y esta clase tiene una serie de métodos y propiedades, entre los cuales está el método **Sort**, el cual sirve para clasificar el contenido de un array.

Para clasificar el array **a**, podemos hacerlo de dos formas:

a.Sort(a)

La variable **a** es un array, por tanto es del tipo Array y como tal, tiene el método Sort, el cual se usa pasando como parámetro el array que queremos clasificar.

Pero esto puede parecer una redundancia, así que es preferible, por claridad, usar el segundo método:

Array.Sort(a)

En el que usamos el método Sort de la clase Array. Este método es lo que se llama un método "compartido" y por tanto se puede usar directamente... no te doy más explicaciones, ya que esto de los métodos compartidos y de instancia lo veremos cuando tratemos el tema de las clases.

Veamos ahora un ejemplo completo en el que se crea y asigna un array al declararla, se muestra el contenido del array usando un bucle For Each, se clasifica y se vuelve a mostrar usando un bucle For normal.

```
Sub Main()
    Dim a() As Integer = {1, 42, 15, 90, 2}
    '
    Console.WriteLine("Elementos del array a(): {0}", a.Length)
    '
    Dim i As Integer
    For Each i In a
        Console.WriteLine(i)
    Next
    '
    Console.WriteLine()
    '
    Array.Sort(a)
    '
    For i = 0 To a.Length - 1
        Console.WriteLine(a(i))
    Next
    '
    Console.WriteLine("Pulsa Intro para finalizar")
    Console.ReadLine()
End Sub
```

Si quieres saber más cosas de los arrays, te recomiendo que le eches un vistazo a la documentación de Visual Studio .NET, pero no te preocupes, que no es que quiera "quitarte" de en medio, simplemente es para que te acostumbres a usar la documentación, ya que allí encontrarás más cosas y explicadas de una forma algo más extensa y sobre todo formal... que aquí parece que nos lo tomamos a "cachondeo"... je, je.

Para terminar con esta entrega, vamos a ver lo que te comentaba al principio: que los arrays son tipos por referencia en lugar de tipos por valor.

## 9.15. El contenido de los arrays son tipos por referencia

Cuando tenemos el siguiente código:

```
Dim m As Integer = 7
Dim n As Integer
'
n = m
'
m = 9
'
Console.WriteLine("m = {0}, n = {1}", m, n)
```

El contenido de **n** será 7 y el de **m** será 9, es decir cada variable contiene y mantiene de forma independiente el valor que se le ha asignado y a pesar de haber hecho la asignación **n = m**, y posteriormente haber cambiado el valore de **m**, la variable **n** no cambia de valor.

Sin embargo, si hacemos algo parecido con dos arrays, veremos que la cosa no es igual:

```
Dim a() As Integer = {1, 42, 15, 90, 2}
Dim b() As Integer
Dim i As Integer
'
b = a
'
a(3) = 55
'
For i = 0 To a.Length - 1
    Console.WriteLine("a(i) = {0}, b(i) = {1}", a(i), b(i))
Next
```

En este caso, al cambiar el contenido del índice 3 del array **a**, también cambiamos el contenido del mismo índice del array **b**, esto es así porque sólo existe una copia en la memoria del array creado y cuando asignamos al array **b** el contenido de **a**, realmente le estamos asignando la dirección de memoria en la que se encuentran los valores, no estamos haciendo una nueva copia de esos valores, por tanto, al modificar el elemento 3 del array **a**, estamos modificando lo que tenemos "guardado" en la memoria y como resulta que el array **b** está apuntando (o hace referencia) a los mismos valores... pues pasa lo que pasa... que tanto **a(3)** como **b(3)** devuelven el mismo valor.

Para poder tener arrays con valores independientes, tendríamos que realizar una copia de todos los elementos del array **a** en el array **b**.

## 9.16. Copiar los elementos de un array en otro array

La única forma de tener copias independientes de dos arrays que contengan los mismos elementos es haciendo una copia de un array a otro.

Esto lo podemos hacer mediante el método **CopyTo**, al cual habrá que indicarle el array de destino y el índice de inicio a partir del cual se hará la copia. Sólo aclarar que el destino debe tener espacio suficiente para recibir los elementos indicados, por tanto deberá estar inicializado con los índices necesarios. Aclaremos todo esto con un ejemplo:

```
Dim a() As Integer = {1, 42, 15, 90, 2}
Dim b(a.Length - 1) As Integer
'
a.CopyTo(b, 0)
'
a(3) = 55
'
```

```
Dim i As Integer
For i = 0 To a.Length - 1
    Console.WriteLine("a(i) = {0}, b(i)= {1}", a(i), b(i))
Next
```

En este ejemplo, inicializamos un array, declaramos otro con el mismo número de elementos, utilizamos el método CopyTo del array con los valores, en el parámetro le decimos qué array será el que recibirá una copia de esos datos y la posición (o índice) a partir de la que se copiarán los datos, (indicando cero se copiarán todos los elementos); después cambiamos el contenido de uno de los elementos del array original y al mostrar el contenido de ambos arrays, comprobamos que cada uno es independiente del otro.

Con esto, no está todo dicho sobre los arrays, aún quedan más cosas, pero habrá que dejarlo para otra entrega, que esta ya ha dado bastante de sí.

Así que, disfruta, si puedes o te apetece, de estos días de fiesta y prepara el cuerpo para lo que seguirá, casi con toda seguridad el próximo año. Si no quieres esperar tanto, ya sabes, abres el Visual Studio .NET, y busca en la ayuda arrays o matrices (esto último si la ayuda está en español) y verás que es lo que te encontrarás en la próxima entrega... bueno, todo no, que es mucho, sólo una parte... lo que yo considere más necesario, que del resto ya te encargarás tú de investigarlo por tu cuenta (eso espero).

Lo dicho, que disfrutes y a pasarlo bien: **¡Feliz Navidad!** (y si lees esto después del día 25 de Diciembre, pues... ¡Feliz loquesea!)

# 10. Décima entrega

En esta primera entrega del recién estrenado año 2003 del curso de iniciación a la programación con Visual Basic .NET seguiremos tratando el tema de los arrays o matrices.

En la entrega anterior vimos los arrays de una sola dimensión, que, por cierto, son las que con mayor asiduidad usaremos en nuestras aplicaciones, pero el tema que vamos a tratar hoy sigue siendo también los arrays, pero con un matiz especial, ya que son las llamadas arrays (o matrices) multidimensionales, es decir arrays que no sólo tienen un índice simple, sino que serán como tablas, es decir, arrays que se pueden representarse por medio de filas y columnas, además de, si se le echa un poco de imaginación, algún otro nivel dimensional... vale, espera y sabrás a que me refiero.

Tal y como vimos en la entrega anterior, la diferencia entre las variables "normales" y los arrays era que se utilizaba un índice para acceder al contenido de cada una de las variables normales contenidas en un array. Por ejemplo, para acceder al tercer elemento de un array llamado **nombres**, tendríamos que hacer algo como esto: **nombres(2)**. Recuerda que los arrays empiezan con un índice cero, por tanto el índice 2 será el tercero.

## 10.1. Los arrays multidimensionales

La diferencia entre los arrays unidimensionales y los arrays multidimensionales es precisamente eso, que los arrays unidimensionales, (o normales, por llamarlos de alguna forma), sólo tienen una dimensión: los elementos se referencian por medio de un solo índice. Por otro lado, los arrays multidimensionales tienen más de una dimensión. Para acceder a uno de los valores que contengan habrá que usar más de un índice. La forma de acceder, por ejemplo en el caso de que sea de dos dimensiones, sería algo como esto: **multiDimensional(x, y)**.

Es decir, usaremos una coma para separar cada una de las dimensiones que tenga. El número máximo de dimensiones que podemos tener es de 32, aunque no es recomendable usar tantas, (según la documentación de Visual Studio .NET no deberían usarse más de tres o al menos no debería ser el caso habitual), en caso de que pensemos que debe ser así, (que tengamos que usar más de tres), deberíamos plantearnos usar otro tipo de datos en lugar de una matriz. Entre otras cosas, porque el Visual Basic reservará espacio de memoria para cada uno de los elementos que reservemos al dimensionar un array, con lo cual, algo que puede parecernos pequeño no lo será tanto.

Hay que tener en cuenta que cuando usamos arrays de más de una dimensión cada dimensión declarada más a la izquierda tendrá los elementos de cada una de las dimensiones declaradas a su derecha. De esto te enterarás mejor en cuanto te explique cómo crear los arrays multidimensionales... que me pongo a explicar cosas y lo mismo ni te estás enterando... (menos mal que el Guille se ha dado cuenta y no le he tenido que llamar la atención... la verdad es que estaba despistadillo escuchando música y... en fin... espero estar más al tanto de los deslices del Guille para que no te confunda o líe más de la cuenta...)

## 10.2. Declarar arrays multidimensionales

Para declarar un array multidimensional, lo podemos hacer, (al igual que con las unidimensionales), de varias formas, dependiendo de que simplemente declaremos el array, que le indiquemos (o reservemos) el número de elementos que tendrá o de que le

asignemos los valores al mismo tiempo que la declaramos, veamos ejemplos de estos tres casos:

```
Dim a1() As Integer
Dim a2(,) As Integer
Dim a3(,,) As Integer
Dim b1(2) As Integer
Dim b2(1, 6) As Integer
Dim b3(3, 1, 5, 2) As Integer
Dim c1() As Integer = {1, 2, 3, 4}
Dim c2(,) As Integer = {{1, 2, 3}, {4, 5, 6}}
' este array se declararía como c3(3, 2, 1)
Dim c3(,,) As Integer = { _
                         {{1, 2}, {3, 4}, {5, 6}}, _
                         {{7, 8}, {9, 10}, {11, 12}}, _
                         {{13, 14}, {15, 16}, {17, 18}}, _
                         {{19, 20}, {21, 22}, {23, 24}} _
                        }
```

En estos ejemplos he usado arrays con una, dos y tres dimensiones.

En el último caso, he usado el continuador de líneas para que sea más fácil "deducir" el contenido de cada una de las dimensiones... imagínate que en lugar de tres dimensiones hubiese usado más... sería prácticamente imposible saber cuantos elementos tiene cada una de las dimensiones. Recuerda que cuando declaramos con a(1) realmente tenemos dos elementos el cero y el uno.

## 10.3. El tamaño de un array multidimensional

En la entrega anterior utilizamos la propiedad Length para averiguar el tamaño de un array de una sola dimensión, en el caso de los arrays de varias dimensiones, se puede seguir usando Length para saber el número total de elementos que contiene, (será la suma de todos los elementos en su totalidad), ya que Length representa la longitud o tamaño, pero para saber cuantos elementos hay en cada una de las dimensiones tendremos que usar otra de las propiedades que exponen los arrays: **GetUpperBound(dimensión)**.

Esta propiedad se usará indicando como parámetro la dimensión de la que queremos averiguar el índice superior.

Por ejemplo, en el caso del array **c3**, podemos usar c3.GetUpperBound(2) para saber cuantos elementos hay en la tercera dimensión (la del array, no la de las novelas y pelis de SCI-FI).

Veamos cómo haríamos para averiguar cuantos elementos tiene cada una de las tres dimensiones del array **c3**:

```
Console.WriteLine("Las dimensiones de c3 son: (?,,)= {0}, (,?,)= {1}, (,,?)= {2}", _
                  c3.GetUpperBound(0), c3.GetUpperBound(1), c3.GetUpperBound(2))
```

## 10.4. El número de dimensiones de un array multidimensional.

Una cosa es saber cuantos elementos tiene un array (o una de las dimensiones del array) y otra cosa es saber cuantas dimensiones tiene dicho array.

Para saber el número de dimensiones del array, usaremos la propiedad **Rank**.

Por ejemplo, (si usamos la declaración hecha anteriormente), el siguiente código nos indicará que el array **c3** tiene tres dimensiones:

**Console.WriteLine("El array c3 tiene {0} dimensiones.", c3.Rank)**

Como siempre, el valor devuelto por **Rank** será el número total de dimensiones del array, pero ese número de dimensiones será desde cero hasta Rank - 1.

Veamos ahora un ejemplo de cómo recorrer todos los elementos del array **c3**, los cuales se mostrarán en la consola. Para saber cuantos elementos hay en cada una de las dimensiones, utilizaremos la propiedad GetUpperBound para indicar hasta qué valor debe contar el bucle For, el valor o índice menor sabemos que *siempre* será cero, aunque se podría usar la propiedad **GetLowerBound**.

**Nota:** Esto es curioso, si todos los arrays empiezan por cero, ¿qué sentido tiene poder averiguar el valor del índice menor? ya que, según sabemos **siempre** debe ser cero. Lo mismo es que han dejado la puerta abierta a un posible cambio en esta "concepción" del índice menor de los arrays... en fin... el tiempo (y las nuevas versiones de .NET Framework) lo dirá.

Si has trabajado anteriormente con Visual Basic clásico, sabrás que en VB6 podemos indicar "libremente" el valor inferior así como el superior de un array y el uso del equivalente a GetLowerBound si que tenía sentido.

Dejemos estas cábalas y veamos el ejemplo prometido:

```
Dim i, j, k As Integer

For i = 0 To c3.GetUpperBound(0)
   For j = 0 To c3.GetUpperBound(1)
     For k = 0 To c3.GetUpperBound(2)
        Console.WriteLine("El valor de c3({0}, {1}, {2}) es {3}", i, j, k, c3(i, j, k))
     Next
   Next
Next
```

Seguramente te estarás preguntando qué son todos esas llaves con "numericos" que está usando el Guille en los ejemplos... no te preocupes que dentro de poco te lo explico con un poco de detalle, aunque espero que sepas "deducir" para qué sirven... al menos si estás probando estos "trozos" de ejemplos.

## 10.5. Cambiar el tamaño de un array y mantener los elementos que tuviera.

Esto es algo que debería haber explicado en la entrega anterior, pero seguramente se me "escapó", (venga va, perdonemos al despistado del Guille, que demasiado hace... *pa* lo *despistao* que es...), así que, para que la próxima sección no te suene a "chino", voy a explicarte cómo poder cambiar el tamaño de un array sin perder los valores que tuviese antes.

Para poder conseguirlo, debemos usar **ReDim** seguida de la palabra clave **Preserve**, por tanto si tenemos la siguiente declaración:

**Dim a() As Integer = {1, 2, 3, 4, 5}**

Y queremos que en lugar de 5 elementos (de 0 a 4) tenga, por ejemplo 10 y no perder los otros valores, usaremos la siguiente instrucción:

**ReDim Preserve a(10)**

A partir de ese momento, el array tendrá 11 elementos (de 0 a 10), los 5 primeros con los valores que antes tenía y los nuevos elementos tendrán un valor cero, que es el valor por defecto de los valores numéricos.

Si sólo usamos **ReDim a(10)**, también tendremos once elementos en el array, pero todos tendrán un valor cero, es decir, *si no se usa Preserve, se pierden los valores contenidos en el array*.

## 10.6. Redimensionar un array multidimensional.

En la entrega anterior vimos que usando **ReDim** podemos cambiar el número de elementos de un array, e incluso que usando **ReDim Preserve** podemos cambiar el número de elementos y mantener los que hubiese anteriormente en el array, (esto último no lo busques en la entrega anterior, ya que se ha aclarado hace unas pocas líneas). Con los arrays multidimensionales también podemos usar esas instrucciones con el mismo propósito que en los arrays unidimensionales.

El único problema con el que nos podemos encontrar, al menos si queremos usar Preserve para conservar los valores previos, es que sólo podemos cambiar el número de elementos de la última dimensión del array. Si has usado el VB6 esto es algo que te "sonará", ya que con el VB clásico tenemos el mismo inconveniente, pero a diferencia de aquél, con ReDim podemos cambiar el número del resto de las dimensiones, al menos la cantidad de elementos de cada dimensión, ya que si un array es de 3 dimensiones siempre será de tres dimensiones.

Con esto último hay que tener cuidado, ya que si bien será el propio IDE el que nos avise de que no podemos cambiar "la cantidad" de dimensiones de un array, es decir, si tiene tres dimensiones, siempre debería tener tres dimensiones, por ejemplo, siguiendo con el ejemplo del array c3, si hacemos esto:

ReDim c3(1, 4)

Será el IDE de Visual Studio .NET el que nos avise indicando con los dientes de sierra que hay algo que no está bien.

Pero si cambiamos el número de elementos de las dimensiones (usando Preserve), hasta que no estemos en tiempo de ejecución, es decir, cuando el programa llegue a la línea que cambia el número de elementos de cada dimensión, no se nos avisará de que no podemos hacerlo.

Veamos qué podemos hacer sin problemas y que daría error:

Si tenemos c3 dimensionada con tres dimensiones, al estilo de Dim c3(3, 2, 1):

ReDim c3(3, 2) dará error en tiempo de diseño.

ReDim c3(2, 3, 4) funcionará bien.

ReDim Preserve c3(3, 3, 1) en tiempo de ejecución nos dirá que nones... que no se puede.
ReDim Preserve c3(3, 2, 4) será correcto y los nuevos elementos tendrán el valor por defecto.
ReDim Preserve c3(3, 2, 0) será correcto, hemos reducido el número de elementos de la última dimensión.

**Aclarando temas:** Podemos usar ReDim para cambiar el número de elementos de cada una de las dimensiones, pero no podemos cambiar el número de dimensiones.

Podemos usar ReDim Preserve para cambiar el número de elementos de la última dimensión sin perder los valores que previamente hubiera.

En ningún caso podemos cambiar el número de dimensiones de un array.

## 10.7. Eliminar un array de la memoria.

Si en algún momento del programa queremos eliminar el contenido de un array, por ejemplo para que no siga ocupando memoria, ya que es posible que no siga ocupando memoria, podemos usar **Erase** seguida del array que queremos "limpiar", por ejemplo:

**Erase a**

Esto eliminará el contenido del array a.

Si después de eliminar el contenido de un array queremos volver a usarlo, tendremos que ***ReDim***ensionarlo con el mismo número de dimensiones que tenía, ya que Erase sólo borra el contenido, no la definición del array.

## 10.8. ¿Podemos clasificar un array multidimensional?

Pues la respuesta es: No.

El método **Sort** sólo permite clasificar un array unidimensional, (de una sola dimensión).

La única forma de clasificar un array multidimensional sería haciéndolo de forma manual, pero esto es algo que, sintiéndolo mucho, no te voy a explicar.

De igual forma que no podemos clasificar un array multidimensional, al menos de forma "automática", tampoco podemos usar el resto de métodos de la clase Array que se suelen usar con arrays unidimensionales, como puede ser Reverse.

## 10.9. Copiar un array multidimensional en otro.

Para copiar el contenido de un array, sea o no multidimensional, podemos usar el método **Copy** de la clase **Array**.

Seguramente te preguntarás si se puede usar **CopyTo**, que es la forma que vimos en la entrega anterior para copiar el contenido de un array, la respuesta es NO, simplemente porque CopyTo sólo se puede usar con arrays unidimensionales.

**Nota:** Que no te sorprenda que, en el IDE, al mostrar los miembros (métodos y propiedades) de un array multidimensional se muestren métodos no válidos para los arrays multidimensionales, así que, acostúmbrate a "leer" el ToolTip (mensaje de ayuda emergente) que se muestra o, mejor aún, leer la documentación de Visual Studio .NET.

Para usar el método Copy de la clase Array, debemos indicar el array de origen, el de destino y el número de elementos a copiar. Siguiendo con el ejemplo del array **c3**, podríamos copiar el contenido en otro array de la siguiente forma:

```
Dim c31(,,) As Integer
'
ReDim c31(c3.GetUpperBound(0), c3.GetUpperBound(1), c3.GetUpperBound(2))
Array.Copy(c3, c31, c3.Length)
```

Fíjate que no se indica qué dimensión queremos copiar, ya que se copia "todo" el contenido del array, además de que el array de destino debe tener como mínimo el mismo número de elementos.

Otra condición para poder usar Copy es que los dos arrays deben tener el mismo número de dimensiones, es decir, si el array origen tiene 3 dimensiones, el de destino también debe tener el mismo número de dimensiones.

Si bien, el número de dimensiones debe ser el mismo en los dos arrays, el número de elementos de cada una de las dimensiones no tiene porqué serlo. Sea como fuere, el

número máximo de elementos a copiar tendrá que ser el del array que menos elementos tenga... sino, tendremos una excepción en tiempo de ejecución.

Para entenderlo mejor, veamos varios casos que se podrían dar. Usaremos el contenido del array **c3** que, como sabemos, está definido de esta forma: c3(3, 2, 1) y como destino un array llamado **c31**.

Para copiar los elementos haremos algo así:

**Array.Copy(c3, c31, c3.Length)**

Ahora veamos ejemplos de cómo estaría dimensionado el array de destino y si son o no correctos:

**Dim c31(3, 2, 1)**, correcto ya que tiene el mismo número de dimensiones y elementos.

**Dim c31(3, 3, 2)**, correcto porque tiene el mismo número de dimensiones y más elementos.
**Dim c31(2, 1, 0)**, correcto, tiene el mismo número de dimensiones aunque tiene menos elementos, por tanto la cantidad de elementos a copiar debe ser el del array de destino: **Array.Copy(c3, c31, c31.Length)**.

**Dim c31(3, 2)**, no es correcto porque tiene un número diferente de dimensiones.

Espero que todo lo que hemos tratado sobre los arrays o matrices que hemos visto en esta dos últimas entregas sea suficiente para que sepas cómo usarlas, pero te repito y a pesar de que pienses que pueda ser un "pesado", te recomiendo que practiques, practiques y sigas practicando, eso unido a que leas la documentación de Visual Studio .NET, hará que te enteres bien de cómo usarlas.

También te digo que, seguramente, te encontrarás con algunas cosas en la ayuda que, salvo que lo hayas aprendido por tu cuenta, te resultarán extrañas y puede que no las entiendas... paciencia te pido, ya que aún quedan muchos conceptos que debemos aprender y, como es natural no se pueden resumir en 10 entregas.

No te voy a decir que es lo que puedes encontrarte en la documentación que no entiendas... puede que sean muchas... pero, ciñéndonos en el tema de los arrays, hay algunos conceptos referidos a la herencia y otras cosas relacionadas con la programación orientada a objetos que de seguro no te serán familiares.

Así que, puede ser que en la próxima entrega veamos el tema de la programación orientada a objetos, así que... ve preparando el cuerpo y, sobre todo, la mente para que cuando llegue ese día, que con total seguridad será el próximo mes de febrero (siempre que estés leyendo esto cuando se publica), no te de ningún tipo de "espasmo" o ataque "cerebral".

De todas formas, aún puedes descansar la mente unos minutos para que veamos de que va eso de las llaves con los números ({0}) que he usado en algunos de los ejemplos de esta y otras entregas anteriores.

## 10.10. Los formatos a usar con las cadenas de Console.Write y WriteLine.

Aunque seguramente lo usarás en pocas ocasiones, ya que me imagino que la mayoría de las aplicaciones que hagas serán de tipo "gráficas" en lugar de aplicaciones de consola, te voy a explicar cómo poder dar "formato" a las cadenas usadas con los métodos Write y WriteLine de la clase Console.

Con esos métodos podemos indicar, entre otras cosas, una cadena que se mostrará en la ventana de consola, dentro de esa cadena podemos usar unos indicadores o marcadores en los que se mostrarán los parámetros indicados en esos dos métodos de la clase Console, los cuales también están presentes en otras clases del .NET Framework.

**Nota:** El formato especificado en la cadena usada en los métodos Write y WriteLine será el mismo que se puede usar con el método Format de la clase String, por tanto, lo aquí explicado será aplicable a cualquier cadena.

Los marcadores se indican con un número encerrado entre llaves, los números indicarán el número u orden del parámetro que siga a la cadena en el que está ese marcador. Como es habitual, el primer parámetro estará representado por el cero y así sucesivamente.

Por ejemplo, para mostrar los valores de dos parámetros usaremos algo como esto:

**Console.WriteLine("{0} {1}", i, j)**

En este ejemplo, el {0} se sustituirá por el valor de la variable **i**, y el {1} por el valor de la variable **j**.

Cuando se indican marcadores, estos deben coincidir con el número de parámetros, es decir, si usamos el marcador {3}, debería existir ese parámetro, en caso de que haya menos de 4 parámetros (de 0 a 3), se producirá una excepción.

**Nota:** Los parámetros pueden ser numéricos, de cadena, e incluso objetos basados en una clase. En todos los casos se usará la "versión" de tipo cadena de esos parámetros, de esto sabrás un poco más cuando toquemos el tema de las clases... sólo te lo comento para que no te extrañe ver algo "raro" si haces tus propias pruebas.

Con uno de estos marcadores, además de indicar el número de parámetro que se usará, podemos indicar la forma en que se mostrará: cuantos caracteres se mostrarán y si se formatearán a la derecha o la izquierda; también se pueden indicar otros valores de formato.

La forma de usar los marcadores o las especificaciones de formato será la siguiente:

{ N [, M ][: Formato ]}

**N** será el número del parámetro, empezando por cero.

**M** será el ancho usado para mostrar el parámetro, el cual se rellenará con espacios. Si M es negativo, se justificará a la izquierda, y si es positivo, se justificará a la derecha.

**Formato** será una cadena que indicará un formato extra a usar con ese parámetro.

Veamos algunos ejemplos para aclarar todo esto:

```
Console.WriteLine("---------- ----------")
Console.WriteLine("{0,10} {1,-10} {2}", 10, 15, 23)
'
Console.WriteLine("---------- ----------")
Console.WriteLine("{0,10:#,###.00} {1,10}", 10.476, 15.355)
```

El resultado de este ejemplo sería este:

```
---------- ----------
        10 15         23
---------- ----------
     10,48     15,355
```

En el primer caso, el primer número se muestra con 10 posiciones, justificado a la derecha, el segundo se justifica a la izquierda.

En el segundo caso, el número 10.476 se justifica también con 10 posiciones a la derecha, pero el número se muestra con un formato numérico en el que sólo se muestran dos decimales, por eso se ha redondeado el resultado, sin embargo en el segundo valor se muestran todos los decimales, ya que no se ha especificado ningún formato especial.

Estas especificaciones de formato es aplicable a números, pero también lo es a cadenas de caracteres y fechas. Veamos algunos ejemplos más:

```
Console.WriteLine("Hoy es: {0:G}", DateTime.Now)
Console.WriteLine("Hoy es: {0:dd/MM/yy}", DateTime.Now)
Console.WriteLine("Hoy es: {0:ddd, dd/MMM/yyyy}", DateTime.Now)
```

El resultado de estas líneas será el siguiente (al menos usando la configuración regional que tengo en mi equipo):

```
Hoy es: 19/01/2003 02:51:35
Hoy es: 19/01/03
Hoy es: dom, 19/ene/2003
```

Como puedes comprobar, en esta ocasión no se indica cuantos caracteres se usan para mostrar los datos, (ya que sólo se indica el parámetro y el formato a usar), pero en el caso de que se especifiquen, si la cantidad de caracteres a mostrar es mayor que los indicados, no se truncará la salida, es decir, no se quitarán los caracteres que sobren. Esto lo podemos ver en el siguiente ejemplo:

```
Dim s As String = "Esta es una cadena larga de más de 20 caracteres"
Dim s1 As String = "Cadena más corta"
'
Console.WriteLine("--------------------")
Console.WriteLine("{0,20}", s)
Console.WriteLine("{0,20}", s1)
Console.WriteLine("--------------------")
```

La salida de este código es la siguiente:

```
--------------------
Esta es una cadena larga de más de 20 caracteres
    Cadena más corta
--------------------
```

Tal como podemos comprobar, la primera cadena excede los 20 caracteres (los indicados con las rayas), por tanto se muestra todo el contenido, sin embargo la segunda cadena mostrada si se posiciona correctamente con espacios a la izquierda hasta completar los 20 especificados.

En *formato* se pueden usar muchos caracteres diferentes según el tipo de datos a "formatear", la lista sería bastante larga y ya que está disponible en Visual Studio .NET no voy a repetirla, de todas formas, con toda seguridad, veremos algunos ejemplos a lo largo de este curso de iniciación.

Para ver una lista detallada y bastante extensa, puedes buscar en la ayuda la función **Format**, que es propia de Visual Basic, también encontrarás más información sobre los formatos en otras partes de la documentación, pero es en esa función donde están más detallados los diferentes formatos a usar.

Pues creo que con esto acabamos esta décima entrega en la que además de ver cómo se trabaja con arrays multidimensionales, hemos visto algunos de los métodos usados con la clase Array, así como la forma de dar formatos a las cadenas usadas con los métodos Write y WriteLine de la clase Console.

En la próxima entrega, salvo que cambie de opinión a última hora, cuando vaya a escribirla, veremos o empezaremos a ver temas relacionados con la programación orientada a objetos, al menos en sus conceptos más básicos y elementales, los cuales iremos ampliando en otras entregas.

Creo que es muy importante saber sobre esos temas, ya que todo en .NET Framework y por tanto en Visual Basic .NET, está estrechamente relacionado con la programación orientada a objetos, así que... lo que te dije, ve dejando espacio libre en tu "cabecita" para que pueda entrar todo lo referente a ese tema. Aunque no te asustes si te parece que puede que sea algo demasiado "sofisticado" o avanzado, ya que, como lo trataremos bastante en todo este curso de iniciación, al final acabarás aprendiéndolo, ¡aunque no quieras!

# 11. Undécima entrega

Bueno, aún no ha pasado un mes completo desde la entrega anterior, así que... no deberías quejarte demasiado, ya que me estoy comportando como un tío serio y formal... ¡pero no te lo vayas a creer! que eso sólo son las apariencias... je, je...

Como te comenté en la entrega anterior, en esta ocasión vamos a ver un tema del que acabarás hasta la coronilla, entre otras cosas porque es algo que "tienes" que saber y si lo dominas, cosa que pretendo y espero conseguir, te será mucho más fácil hacer cualquier tipo de programa o utilidad con Visual Basic .NET o con cualquier otro lenguaje de la familia .NET, como puede ser C#.

Lo que si te pido es paciencia y mucha mente "en blanco", es decir, déjate llevar y no desesperes si de algo no te enteras, ya que intentaré que todo esto que empezaremos en esta entrega llegue a formar parte de ti... no, no te voy a implantar ningún chip, así que no te asustes... pero este tema es algo de lo que los programadores de Visual Basic debemos estar "orgullosos" (por decir algo) de que al fin... se nos permita tener, de forma más o menos completa, ya que hasta el momento sólo teníamos acceso a ello de forma limitada... Bueno, menos rollo y vamos al tema, que es ni más ni menos que:

## 11.1. La programación orientada a objetos

Todo .NET Framework está basado en clases (u objetos). A diferencia de las versiones anteriores de Visual Basic, la versión .NET de este lenguaje basa su funcionamiento casi exclusivamente en las clases contenidas en .NET Framework, además casi sin ningún tipo de limitaciones. Debido a esta dependencia en las clases del .NET Framework y sobre todo a la forma "hereditaria" de usarlas, Visual Basic .NET tenía que ofrecer esta característica sin ningún tipo de restricciones. Y, por suerte para nosotros, esto es así... por tanto, creo conveniente que veamos de que va todo esto de la Programación Orientada a Objetos.

**Nota:** Seguramente, en esta y en las próximas entregas, utilice las siglas **OOP** cuando me refiera a la programación orientada a objetos, esto es así porque éstas son las siglas de ***O****bject* ***O****riented* ***P****rogramming*, que es como se dice en inglés y como estoy acostumbrado a usar OOP, pues...

Aunque no soy muy dado a esto de las cosas teóricas, vamos a ver de que va todo esto de la programación orientada a objetos y cuales son las "condiciones" que cualquier lenguaje que se precie de serlo debe cumplir.

## 11.2. Los tres pilares de la Programación Orientada a Objetos

Según se dice por ahí, cualquier lenguaje basado en objetos debe cumplir estos tres requisitos:

1. **Herencia**
2. **Encapsulación**
3. **Polimorfismo**

**Nota:** Algunos autores añaden un cuarto requisito: **la abstracción**, pero este último está estrechamente ligado con la encapsulación, ya que, de echo, es prácticamente lo mismo, cuando te lo explique, lo comprobarás y podrás decidir por tu cuenta si son tres o cuatro los pilares de la OOP...

Veamos que es lo que significa cada uno de ellos:

### 11.2.1. Herencia

Esto es lo que la documentación de Visual Studio .NET nos dice de la herencia:

*Una relación de herencia es una relación en la que un tipo (el tipo derivado) se deriva de otro (el tipo base), de tal forma que el espacio de declaración del tipo derivado contiene implícitamente todos los miembros de tipo no constructor del tipo base.*

*(ms-help://MS.VSCC.2003/MS.MSDNVS.3082/vbls7/html/vblrfVBSpec4_2.htm)*

Está claro ¿verdad?

¿Cómo? ¿Que no te has enterado? A ver, a ver... y si probamos con esto otro:

*La herencia es la capacidad de una clase de obtener la interfaz y comportamiento de una clase existente.*

Ya nos vamos entendiendo.

Para que te quede un poco más claro, pero no esperes milagros, aquí tienes otra definición de herencia:

*La herencia es la cualidad de crear clases que estén basadas en otras clases. La nueva clase heredará todas las propiedades y métodos de la clase de la que está derivada, además de poder modificar el comportamiento de los procedimientos que ha heredado, así como añadir otros nuevos.*

Resumiendo: Gracias a la herencia podemos ampliar cualquier clase existente, además de aprovecharnos de todo lo que esa clase haga... (sí, ya se que aún no hemos profundizado en lo que son las clases, pero... es que esto no es fácil de explicar, al menos para que se entienda a la primera, así que *muuucha* paciencia...)

Para que lo entiendas mejor, veamos un ejemplo *clásico*:

Supongamos que tenemos una clase Gato que está derivada de la clase Animal.

El Gato hereda de Animal todas las características comunes a los animales, además de añadirle algunas características particulares a su condición felina.

Podemos decir que un Gato es un Animal, lo mismo que un Perro es un Animal, ambos están derivados (han heredado) de la clase Animal, pero cada uno de ellos es diferente, aunque en el *fondo* los dos son animales.

Esto es herencia: usar una clase base (Animal) y poder ampliarla sin perder nada de lo heredado, pudiendo ampliar la clase de la que se ha derivado (o heredado).

¿Qué? Sigues sin enterarte, ¿verdad?

Bueno, que le vamos a hacer... pero no desanimes y sigue leyendo... que si bien esto no es cosa fácil, tampoco es algo que sea imposible de entender, (hasta el Guille se ha enterado de que va todo esto de la herencia, así que... paciencia).

### 11.2.2. Encapsulación

A ver si con la encapsulación tenemos más suerte:

*La encapsulación es la capacidad de contener y controlar el acceso a un grupo de elementos asociados. Las clases proporcionan una de las formas más comunes de encapsular elementos.*

*(ms-help://MS.VSCC.2003/MS.MSDNVS.3082/vbcn7/html/vbconClassModulesPuttingDataTypesProceduresTogether.htm)*

Está crudo ¿verdad?

La encapsulación es la capacidad de separar la implementación de la interfaz de una clase del código que hace posible esa implementación. Esto realmente sería una especie de abstracción, ya que no nos importa cómo esté codificado el funcionamiento de una clase, lo único que nos debe interesar es cómo funciona...

Para que nos vayamos entendiendo, cuando digo: la implementación de la interfaz de una clase, me refiero a los miembros de esa clase: métodos, propiedades, eventos, etc. Es decir, lo que la clase es capaz de hacer.

Cuando usamos las clases, éstas tienen una serie de características (los datos que manipula) así como una serie de comportamientos (las acciones a realizar con esos datos). Pues la encapsulación es esa capacidad de la clase de ocultarnos sus interioridades para que sólo veamos lo que tenemos que ver, sin tener que preocuparnos de cómo está codificada para que haga lo que hace... simplemente nos debe importar que lo hace.

Si tomamos el ejemplo de la clase Gato, sabemos que araña, come, se mueve, etc., pero el cómo lo hace no es algo que deba preocuparnos, salvo que se lance sobre nosotros... aunque, en ese caso, lo que deberíamos tener es una clase "espanta-gatos" para quitárnoslo de encima lo antes posible...

### 11.2.3. Polimorfismo

Pues no sé que decirte... si en los dos casos anteriores la cosa estaba complicada... En fin... a ver que es lo que nos dice la documentación de VS.NET sobre el polimorfismo:

*El polimorfismo se refiere a la posibilidad de definir múltiples clases con funcionalidad diferente, pero con métodos o propiedades denominados de forma idéntica, que pueden utilizarse de manera intercambiable mediante código cliente en tiempo de ejecución.*

*(ms-help://MS.VSCC.2003/MS.MSDNVS.3082/vbcn7/html/vbconInheritancePolymorphismAllThat.htm)*

Mira tu por dónde, esta definición si que me ha gustado...

Vale, que a ti te ha dejado igual que hace un rato... pues... no se yo que decirte... es que, si no lo entiendes, la verdad es que podrías dedicarte a otra cosa... je, je, *tranqui colega,* que ya te he comentado antes que había que tener la mente en blanco... y lo que no te dije es que seguramente la ibas a seguir teniendo en blanco...

Ya en serio, el Polimorfismo nos permite usar miembros de distintas clases de forma genérica sin tener que preocuparnos si pertenece a una clase o a otra.

Siguiendo con el ejemplo de los animales, si el Gato y el Perro pueden morder podríamos tener un "animal" que muerda sin importarnos que sea el Gato o el Perro, simplemente podríamos usar el método Morder ya que ambos animales tienen esa característica "animal mordedor".

Seguramente estarás pensando: *Guille, te podrías haber ahorrado toda estas parrafadas, ya que me he quedado igual o peor que antes...* y estarás en lo cierto... pero... ya te comenté que esto de la teoría no es lo mío, así que... cuando lo veamos con ejemplos "reales" seguro que lo entenderás mejor. Así que, como te dije antes: paciencia... mucha paciencia...

Realmente todo este rollo sobre las características de la Programación Orientada a Objetos es para que sepas que ahora con Visual Basic es posible hacer todo esto; cómo hacerlo, será lo que realmente tenga yo que enseñarte, así que... sigue leyendo que aún no hemos terminado.

Y para que podamos ver ejemplos antes debemos tener claro una serie de conceptos, como por ejemplo:

## 11.3. Las clases

Ya hemos visto antes lo que son las clases. Ya que todo lo que tiene el .NET Framework, en realidad son clases.

Una clase no es ni más ni menos que código. Aunque dicho de esta forma, cualquier programa sería una clase.

Cuando definimos una clase, realmente estamos definiendo dos cosas diferentes: los datos que dicha clase puede manipular o contener y la forma de acceder a esos datos.

Por ejemplo, si tenemos una clase de tipo Cliente, por un lado tendremos los datos de dicho cliente y por otro la forma de acceder o modificar esos datos. En el primer caso, los datos del Cliente, como por ejemplo el nombre, domicilio etc., estarán representados por una serie de campos o propiedades, mientras que la forma de modificar o acceder a esa información del Cliente se hará por medio de métodos.

Esas propiedades o características y las acciones a realizar son las que definen a una clase.

## 11.4. Los Objetos

Por un lado tenemos una clase que es la que define un "algo" con lo que podemos trabajar. Pero para que ese "algo" no sea un "nada", tendremos que poder convertirlo en "algo tangible", es decir, tendremos que tener la posibilidad de que exista. Aquí es cuando entran en juego los objetos, ya que un objeto es una clase que tiene información real.

Digamos que la clase es la "plantilla" a partir de la cual podemos crear un objeto en la memoria.
Por ejemplo, podemos tener varios objetos del tipo Cliente, uno por cada cliente que tengamos en nuestra cartera de clientes, pero la clase sólo será una.

Dicho de otra forma: podemos tener varias instancias en memoria de una clase. Una instancia es un objeto (los datos) creado a partir de una clase (la plantilla o el código).

Para entender mejor todo este galimatías, desglosemos las clases:

## 11.5. Los miembros de una clase

Las clases contienen datos, esos datos suelen estar contenidos en variables. A esas variables cuando pertenecen a una clase, se les llama: campos o propiedades.

Por ejemplo, el nombre de un cliente sería una propiedad de la clase Cliente. Ese nombre lo almacenaremos en una variable de tipo String, de dicha variable podemos decir que es el "campo" de la clase que representa al nombre del cliente.

Por otro lado, si queremos mostrar el contenido de los campos que contiene la clase Cliente, usaremos un procedimiento que nos permita mostrarlos, ese procedimiento será un método de la clase Cliente.

Por tanto, los miembros de una clase son las propiedades (los datos) y los métodos las acciones a realizar con esos datos.

Como te he comentado antes, el código que internamente usemos para almacenar esos datos o para, por ejemplo, mostrarlos, es algo que no debe preocuparnos mucho,

simplemente sabemos que podemos almacenar esa información (en las propiedades de la clase) y que tenemos formas de acceder a ella, (mediante los métodos de dicha clase), eso es "abstracción" o encapsulación.

Seguro que dirás: Vale, todo esto está muy bien, pero... ¿cómo hago para que todo eso sea posible?

¿Cómo creo o defino una clase? y ya que estamos, ¿cómo defino las propiedades y los métodos de una clase?

Aquí quería yo llegar... y te lo explico ahora mismo.

## 11.6. Crear o definir una clase

Al igual que existen instrucciones para declarar o definir una variable o cualquier otro elemento de un programa de Visual Basic, existen instrucciones que nos permiten crear o definir una clase.

Para crear una clase debemos usar la instrucción **Class** seguida del nombre que tendrá dicha clase, por ejemplo:

```
Class Cliente
```

A continuación escribiremos el código que necesitemos para implementar las propiedades y métodos de esa clase, pero para que Visual Basic sepa que ya hemos terminado de definir la clase, usaremos una instrucción de cierre:

```
End Class
```

Por tanto, todo lo que esté entre Class <nombre> y End Class será la definición de dicha clase.

### 11.6.1. Definir los miembros de una clase

Para definir los miembros de una clase, escribiremos dentro del "bloque" de definición de la clase, las declaraciones y procedimientos que creamos convenientes. Veamos un ejemplo:

```
Class Cliente

    Public Nombre As String

    Sub Mostrar()
        Console.WriteLine("El nombre del cliente: {0}", Nombre)
    End Sub

End Class
```

En este caso, la línea ***Public Nombre As String***, estaría definiendo una propiedad o "campo" público de la clase Cliente.

Por otro lado, el procedimiento Mostrar sería un método de dicha clase, en esta caso, nos permitiría mostrar la información contenida en la clase Cliente.

Esta es la forma más simple de definir una clase. Y normalmente lo haremos siempre así, por tanto podemos comprobar que es muy fácil definir una clase, así como los miembros de dicha clase.

Pero no sólo de clases vive el Visual Basic... o lo que es lo mismo, ¿para que nos sirve una clase si no sabemos crear un objeto basado en esa clase...? Así que, sepamos cómo crearlos.

### 11.6.2. Crear un objeto a partir de una clase

Como te he comentado antes, las clases definen las características y la forma de acceder a los datos que contendrá, pero sólo eso: los define.

Para que podamos asignar información a una clase y poder usar los métodos de la misma, tenemos que crear un objeto basado en esa clase, o lo que es lo mismo: tenemos que crear una nueva instancia en la memoria de dicha clase.

Para ello, haremos lo siguiente:

Definimos una variable capaz de contener un objeto del tipo de la clase, esto lo haremos como con cualquier variable:

```
Dim cli As Cliente
```

Pero, a diferencia de las variables basadas en los tipos visto hasta ahora, para poder crear un objeto basado en una clase, necesitamos algo más de código que nos permita "crear" ese objeto en la memoria, ya que con el código usado en la línea anterior, simplemente estaríamos definiendo una variable que es capaz de contener un objeto de ese tipo, pero aún no existe ningún objeto en la memoria, para ello tendremos que usar el siguiente código:

```
cli = New Cliente()
```

Con esto le estamos diciendo al Visual Basic: crea un **nuevo** objeto en la memoria del tipo Cliente.

Estos dos pasos los podemos simplificar de la siguiente forma:

```
Dim cli As New Cliente()
```

A partir de este momento existirá en la memoria un objeto del tipo Cliente.

**Nota:** En las versiones anteriores de Visual Basic no era recomendable usar esta forma de instanciar un nuevo objeto en la memoria, porque, aunque de forma transparente para nosotros, el compilador añadía código extra cada vez que se utilizaba esa variable, pero en la versión .NET no existe ese problema y por tanto no deteriora el rendimiento de la aplicación.

Y ahora nos queda saber cómo

## 11.7. Acceder a los miembros de una clase

Para acceder a los miembros de una clase (propiedades o métodos) usaremos la variable que apunta al objeto creado a partir de esa clase, seguida de un punto y el miembro al que queremos acceder, por ejemplo, para asignar el nombre al objeto **cli**, usaremos este código:

```
cli.Nombre = "Guillermo"
```

Es decir, de la misma forma que haríamos con cualquier otra variable, pero indicando el objeto al que pertenece dicha variable.

Como podrás comprobar, esto ya lo hemos estado usando anteriormente tanto en la clase Console como en las otras clases que hemos usado en entregas anteriores, incluyendo los arrays.

Y para acceder al método Mostrar:

```
cli.Mostrar()
```

Ves que fácil.

Pues así es todo lo demás en .NET Framework, así que... ¿para que seguir explicándote?

Bueno, vale... seguiremos viendo más cosas...

Espero que con lo dicho y lo que ya llevamos visto hasta ahora, tengas claro que son las propiedades y los métodos y si aún tienes dudas, no te preocupes ya que ahondaremos más en estos conceptos y sobre todo los utilizaremos hasta la saciedad, con lo cual puedes tener la tranquilidad de que acabarás enterándote al 100% de que va todo esto...

A pesar de que aún no estés plenamente informado de todo, vamos a ver un par de ejemplos de cómo usar las características de la programación orientada a objetos, después, en próximas entregas, profundizaremos más.

## 11.8. Ejemplo de cómo usar la herencia

Para poder usar la herencia en nuestras clases, debemos indicar al Visual Basic que esa es nuestra intención, para ello disponemos de la instrucción **Inherits**, la cual se usa seguida del nombre de la clase de la que queremos heredar. Veamos un ejemplo.

Empezaremos definiendo una clase "base" la cual será la que heredaremos en otra clase.

Ya sabemos cómo definir una clase, aunque para este ejemplo, usaremos la clase Cliente que vimos anteriormente, después crearemos otra, llamada ClienteMoroso la cual heredará todas las características de la clase Cliente además de añadirle una propiedad a esa clase derivada de Cliente.

Veamos el código de estas dos clases.

Para ello crea un nuevo proyecto del tipo consola y escribe estas líneas al principio o al final del fichero que el IDE añade de forma predeterminada.

```
Class Cliente
    Public Nombre As String

    Sub Mostrar()
        Console.WriteLine("El nombre del cliente: {0}", Nombre)
    End Sub
End Class

Class ClienteMoroso
    Inherits Cliente

    Public Deuda As Decimal

End Class
```

Como puedes comprobar, para que la clase ClienteMoroso herede la clase Cliente, he usado **Inherits Cliente**, con esta línea, (la cual debería ser la primera línea de código después de la declaración de la clase), le estamos indicando al VB que nuestra intención es poder tener todas las características que la clase Cliente tiene.

Haciendo esto, añadiremos a la clase ClienteMoroso la propiedad Nombre y el método Mostrar, aunque también tendremos la nueva propiedad que hemos añadido: Deuda.

Ahora vamos a ver cómo podemos usar estas clases, para ello vamos a añadir código en el procedimiento Main del módulo del proyecto:

```
Module Module1

    Sub Main()
        Dim cli As New Cliente()
        Dim cliM As New ClienteMoroso()
        '
        cli.Nombre = "Guille"
        cliM.Nombre = "Pepe"
        cliM.Deuda = 2000
        '
        Console.WriteLine("Usando Mostrar de la clase Cliente")
        cli.Mostrar()
        '
        Console.WriteLine("Usando Mostrar de la clase ClienteMoroso")
        cliM.Mostrar()
        '
        Console.WriteLine("La deuda del moroso es: {0}", cliM.Deuda)
        '
        Console.ReadLine()
    End Sub

End Module
```

Lo que hemos hecho es crear un objeto basado en la clase Cliente y otro basado en ClienteMoroso.

Le asignamos el nombre a ambos objetos y a la variable **cliM** (la del ClienteMoroso) le asignamos un valor a la propiedad Deuda.

Fíjate que en la clase ClienteMoroso no hemos definido ninguna propiedad llamada Nombre, pero esto es lo que nos permite hacer la herencia: heredar las propiedades y métodos de la clase base. Por tanto podemos usar esa propiedad como si la hubiésemos definido en esa clase. Lo mismo ocurre con los métodos, el método Mostrar no está definido en la clase ClienteMoroso, pero si que lo está en la clase Cliente y como resulta que ClienteMoroso hereda todos los miembros de la clase Cliente, también hereda ese método.

La salida de este programa sería la siguiente:

```
Usando Mostrar de la clase Cliente
El nombre del cliente: Guille
Usando Mostrar de la clase ClienteMoroso
El nombre del cliente: Pepe
La deuda del moroso es: 2000
```

Como puedes ver no es tan complicado esto de la herencia...

Ahora veamos cómo podríamos hacer uso del polimorfismo o al menos una de las formas que nos permite el .NET Framework.

Teniendo ese mismo código que define las dos clases, podríamos hacer lo siguiente:

```
    Sub Main()
        Dim cli As Cliente
        Dim cliM As New ClienteMoroso()
        '
        cliM.Nombre = "Pepe"
        cliM.Deuda = 2000
        cli = cliM
        '
        Console.WriteLine("Usando Mostrar de la clase Cliente")
        cli.Mostrar()
        '
        Console.WriteLine("Usando Mostrar de la clase ClienteMoroso")
```

```
        cliM.Mostrar()
        '
        Console.WriteLine("La deuda del moroso es: {0}", cliM.Deuda)
        '
        Console.ReadLine()
    End Sub
```

En este caso, la variable **cli** simplemente se ha declarado como del tipo Cliente, pero no se ha creado un nuevo objeto, simplemente hemos asignado a esa variable el contenido de la variable **cliM**.

Con esto lo que hacemos es asignar a esa variable el contenido de la clase ClienteMoroso, pero como comprenderás, la clase Cliente "no entiende" nada de las nuevas propiedades implementadas en la clase derivada, por tanto sólo se podrá acceder a la parte que es común a esas dos clases: la parte heredada de la clase Cliente.

Realmente las dos variables apuntan a un mismo objeto, por eso al usar el método Mostrar se muestra lo mismo. Además de que si hacemos cualquier cambio a la propiedad Nombre, al existir sólo un objeto en la memoria, ese cambio afectará a ambas variables.

Para comprobarlo, añade este código antes de la línea Console.ReadLine():

```
        Console.WriteLine()
        cli.Nombre = "Juan"
        Console.WriteLine("Después de asignar un nuevo nombre a cli.Nombre")
        cli.Mostrar()
        cliM.Mostrar()
```

La salida de este nuevo código, sería la siguiente:

```
Usando Mostrar de la clase Cliente
El nombre del cliente: Pepe
Usando Mostrar de la clase ClienteMoroso
El nombre del cliente: Pepe
La deuda del moroso es: 2000

Después de asignar un nuevo nombre a cli.Nombre
El nombre del cliente: Juan
El nombre del cliente: Juan
```

La parte en gris es lo que se mostraría antes de realizar el cambio.

Como puedes comprobar, al cambiar en una de las variables el contenido de la propiedad Nombre, ese cambio afecta a las dos variables, pero eso no es porque haya nada mágico ni ningún fallo, es por lo que te comenté antes: sólo existe un objeto en la memoria y las dos variables acceden al mismo objeto, por tanto, cualquier cambio efectuado en ellas, se reflejará en ambas variables por la sencilla razón de que sólo hay un objeto en memoria.

A este tipo de variables se las llama variables por referencia, ya que hacen referencia o apuntan a un objeto que está en la memoria. A las variables que antes hemos estado viendo se las llama variables por valor, ya que cada una de esas variables tienen asociado un valor que es independiente de los demás.

Recuerda que estas definiciones ya las vimos en la entrega número nueve.

Pues, vamos a dejarlo aquí... no sin resumir un poco lo que hemos tratado en esta entrega, de esta forma, si eres de los que empiezan a leer por el final, lo mismo te decides a leer desde el principio.

Hemos visto las condiciones que un lenguaje debe cumplir para que sea considerado orientado a objetos: herencia, encapsulación y polimorfismo. También hemos aprendido a definir una clase y a derivar una clase a partir de otra por medio de la instrucción

Inherits. Ya sabemos cómo definir variables cuyo tipo sea una clase y cómo crear nuevos objetos en memoria a partir de una clase. También hemos comprobado que el que exista más de una variable del tipo de una clase no significa que exista un objeto independiente para cada una de esas variables y por último hemos comprobado que son las variables por referencia y cómo afectan a los objetos referenciados por ellas.

En la próxima entrega profundizaremos más en estos temas y así comprobarás que las cosas se pueden complicar aún mucho más... je, je.

También veremos, entre otras cosas, cómo podemos modificar o personalizar un método heredado para que se adapte a las nuevas características de la clase derivada, pero eso será en la próxima entrega. Mientras tanto te recomiendo que te leas la documentación de Visual Studio .NET y vayas profundizando en esto de la herencia y las clases.

# 12. Duodécima entrega

Como te comenté en la entrega anterior, lo que vamos a tratar en esta duodécima entrega será una continuación del tema anterior: la herencia y las clases. Aunque en realidad todas las entregas, al menos las que restan, estarán basadas de una forma u otra en esos mismos conceptos, ya que, prácticamente todo lo que hagamos en .NET estará relacionado con las clases y una gran parte tendrá que ver con la herencia, así que... lo mismo hoy te cuento otra cosa distinta...

Pero no adelantemos acontecimientos, ni nos enfrasquemos en discusiones que al final voy a ganar yo, ya que soy el que escribe y de alguna forma tengo "el poder", je, je.

En serio, en esta entrega del curso de iniciación a la programación con Visual Basic .NET vamos a profundizar un poco en el tema de los procedimientos ya que es algo que necesitaremos conocer más o menos a fondo para poder usarlos en las clases.

También veremos con un poco más de detalles conceptos sobre el ámbito de esos procedimientos, así como otras cosas relacionadas con los procedimientos o lo que es lo mismo: los métodos de las clases y algo sobre las propiedades... aunque no se si me alcanzará a ver todas estas cosas en una sola entrega, así que... sigue leyendo para ver hasta dónde llegamos.

## 12.1. Las partes o elementos de un proyecto de Visual Basic .NET

En la primera entrega de este curso de Visual Basic .NET vimos una serie de elementos en los que podemos "repartir" el código de un proyecto. Repasemos un poco para ir enterándonos bien de todo y que no se quede nada colgado...

Empecemos con los ensamblados.

**Los ensamblados (assembly)**

Para simplificar, un ensamblado es el ejecutable o la librería que podemos crear con VB .NET.

En un ensamblado podemos tener clases, módulos y otros elementos tal como los espacios de nombres.

**Los espacios de nombres (namespace)**

Los espacios de nombres se usan para agrupar clases y otros tipos de datos que estén relacionados entre sí.

Para acceder a los tipos incluidos en un espacio de nombres hay que indicar el namespace seguido de un punto y el nombre de ese tipo, por ejemplo una clase. Por ejemplo, para acceder a la clase Console que está en el espacio de nombres system, habría que hacerlo así:

System.Console.

Para poder definir nuestros propios espacios de nombres, tendremos que usar la instrucción **Namespace** seguida del nombre que queramos darle, y para indicar cuando termina ese espacio de nombres, lo indicaremos con **End Namespace**.

Dentro de un espacio de nombres podemos declarar otros espacios de nombres.

**Nota:** Cuando creamos un proyecto de Visual Basic, por defecto se crea un espacio de nombres llamado de la misma forma que el proyecto, aunque si el nombre del proyecto

incluye espacios u otros caracteres "raros", estos serán sustituidos por guiones bajos. Todas las declaraciones que hagamos en dicho proyecto se "supondrán" incluidas en ese espacio de nombres, por tanto, para poder acceder a ellas desde fuera de ese proyecto, habrá que usar ese espacio de nombres.

Aunque esta forma automática de crear espacios de nombres "ocultos" podemos cambiarla indicándole al Visual Basic de que no cree un espacio de nombres predeterminado, para ello, en las propiedades del proyecto deberemos dejar en blanco el valor indicado en "**Espacio de nombres de la raíz:**" de la ficha General, y especificar en el código el nombre que nos interese que tenga el espacio de nombres indicándolo con Namespace y el nombre que queramos.

En en la siguiente figura podemos ver la página de propiedades del proyecto creado en la entrega anterior y el nombre que por defecto le dio Visual Basic al espacio de nombres.

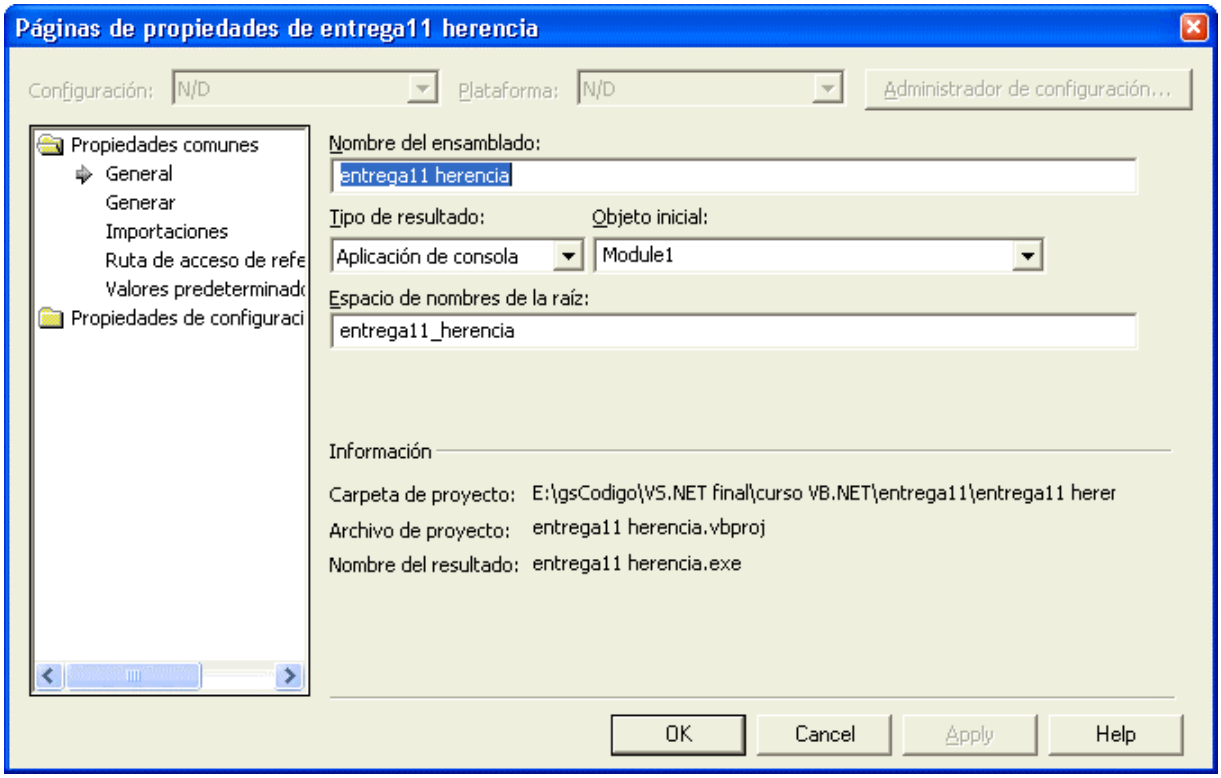



**Figura 1, las propiedades Generales del proyecto**

### Los módulos y las clases

En Visual Basic .NET podemos crear clases de dos formas distintas, usando la instrucción **Module** o usando la instrucción **Class**, en ambos casos, a continuación de esas instrucciones, se indicará el nombre que tendrá ese elemento. Tanto los módulos como las clases, deben estar declarados dentro de un espacio de nombres.

Dentro de una clase podemos definir otras clases, pero no podemos definir módulos dentro de otros módulos.

La diferencia entre un módulo y una clase, es que un módulo define todos sus miembros como compartidos (**Shared**), esto lo veremos con más detalle en otra ocasión, pero te lo explicaré de forma simple, para que no te quedes con la duda:

Como sabemos, cuando queremos crear un objeto basado en una clase, debemos usar **New** para crear una nueva instancia en la memoria, cada nuevo objeto creado con New será independiente de los otros que estén basados en esa clase. Por otro lado, para usar los elementos contenidos en un módulo, no necesitamos crear una nueva instancia, los usamos directamente y asunto arreglado; esto es así porque esos elementos están

compartidos (o si lo prefieres) son estáticos, es decir, siempre existen en la memoria y por tanto no es necesario crear un nuevo objeto. Al estar siempre disponible, sólo existe una copia en la memoria.

**Las enumeraciones**

Como te expliqué en la entrega 7, las enumeraciones son constantes que están relacionadas entre sí.

Las enumeraciones podemos declararlas a nivel de espacios de nombres o a nivel de clases (y/o módulos).

**Las estructuras (Structure)**

Las estructuras o tipos definidos por el usuario, son un tipo especial de datos que veremos en otra ocasión con más detalle, pero que se comportan casi como las clases, permitiendo tener métodos, propiedades, etc. (el etcétera lo veremos en esta misma entrega), la diferencia principal entre las clases y las estructuras es que éstas últimas son tipos por valor, mientras que las clases son tipos por referencia.

Las estructuras, al igual que las clases, las podemos declarar a nivel de espacios de nombres y también dentro de otras estructuras e incluso dentro de clases y módulos.

Bien, tal como hemos podido comprobar, tenemos un amplio abanico de posibilidades entre las que podemos escoger a la hora de crear nuestros proyectos, pero básicamente tenemos tres partes bien distintas:

- los ensamblados,
- los espacios de nombres y
- el resto de elementos de un proyecto.

En realidad, tanto los espacios de nombres como las clases, estructuras y enumeraciones estarán dentro de los ensamblados. Pero las clases, estructuras y enumeraciones suelen estar incluidas dentro de los espacios de nombres, aunque, como te he comentado antes, no tiene porqué existir un espacio de nombres para que podamos declarar las clases, estructuras y enumeraciones, aunque lo habitual es que siempre exista un espacio de nombres, el cual es definido de forma explícita al crear un nuevo proyecto.

En el caso de que no definamos un espacio de nombres, y eliminemos el que Visual Basic crea para nosotros (ver figura 1), no habrá ningún espacio de nombres y por tanto las clases o miembros que definamos en el proyecto estarán accesibles a nivel de ensamblado, sin necesidad de usar ningún tipo de "referencia" extra... esto los veremos con más detalle dentro de un momento...

Además de los elementos que te he relacionado, dentro de las clases, módulos y estructuras, podemos tener otros elementos (o miembros).

## 12.2. Las partes o elementos de una clase

Tal y como hemos tenido la oportunidad de ver en las entregas anteriores, podemos declarar variables y usar procedimientos. Cuando esas variables y procedimientos forman parte de una clase, módulo o estructura tienen un comportamiento "especial", (realmente se comportan de la misma forma que en cualquier otro lado, pero...), ya que dejan de ser variables y procedimientos para convertirse en "miembros" de la clase, módulo o estructura (e incluso de la enumeración) que lo declare.

Te resumiré los distintos miembros de las clases y por extensión de los módulos y las estructuras:

En las clases podemos tener: campos, propiedades, métodos y eventos.

**Los métodos** son procedimientos de tipo Sub o Function que realizan una acción.

**Los campos** son variables usadas a nivel de la clase, es decir son variables normales y corrientes, pero que son accesibles desde cualquier parte dentro de la clase e incluso fuera de ella.

**Las propiedades** podemos decir que son procedimientos especiales, que al igual que los campos, representan una característica de las clases, pero a diferencia de los campos nos permiten hacer validaciones o acciones extras que un campo nunca podrá hacer.

**Los eventos** son mensajes que utilizará la clase para informar de un hecho que ha ocurrido, es decir, se podrán usar para comunicar al que utilice la clase de que se ha producido algo digno de notificar.

La diferencia entre los campos y propiedades lo veremos en otra ocasión posterior, así como el tema de los eventos, ya que ahora lo que nos interesa es saber algo más de:

## 12.3. Los procedimientos: métodos de las clases.

Ya hemos estado usando procedimientos en varias ocasiones. De hecho, en toda aplicación de consola existe un procedimiento de tipo Sub llamado Main que es el que se utiliza como punto de entrada del ejecutable.

Los métodos de una clase pueden ser de dos tipos: Sub o Function.

Los procedimientos Sub son como las instrucciones o palabras clave de Visual Basic: realizan una tarea.

Los procedimientos Function además de realizar una tarea, devuelven un valor, el cual suele ser el resultado de la tarea que realizan. Debido a que las funciones devuelven un valor, esos valores se pueden usar para asignarlos a una variable además de poder usarlos en cualquier expresión.

Para que veamos claras las diferencias entre estos dos tipos de procedimientos, en el siguiente ejemplo vamos a usar tanto un procedimiento de tipo Sub y otro de tipo Function:

```
Module Module1
    Sub Main()
        MostrarS()
        Dim s As String = MostrarF()
        Console.WriteLine(s)
        '
        Console.ReadLine()
    End Sub
    '
    Sub MostrarS()
        Console.WriteLine("Este es el procedimiento MostrarS")
    End Sub
    '
    Function MostrarF() As String
        Return "Esta es la función MostrarF"
    End Function
End Module
```

La salida producida por este código será la siguiente:

```
Este es el procedimiento MostrarS
Esta es la función MostrarF
```

En este módulo tenemos tres procedimientos, dos de tipo Sub y uno de tipo Function, el Sub Main ya lo conocemos de entregas anteriores, pero al fin y al cabo es un procedimiento de tipo Sub y como ya hemos comprobado ejecuta el código que esté entre la definición del procedimiento, el cual empieza con la declaración del procedimiento, el cual siempre es de la misma forma, es decir: usando Sub seguido del nombre del procedimiento y termina con End Sub.

Por otro lado, los procedimientos de tipo Function empiezan con la instrucción Function seguido del nombre de la función y el tipo de dato que devolverá la función, ya que, debido a que las funciones siempre devuelven un valor, lo lógico es que podamos indicar el tipo que devolverá. El final de la función viene indicado por End Function.

Pero como te he comentado, las funciones devuelven un valor, el valor que una función devuelve se indica con la instrucción **Return** seguido del valor a devolver. En este ejemplo, el valor devuelto por la función **MostrarF** es el texto que está entrecomillado.

En el procedimiento Main utilizamos el procedimiento Sub usando simplemente el nombre del mismo: **MostrarS**. Ese procedimiento se usa en una línea independiente; cuando la ejecución del código llegue a esa línea, se procesará el contenido del mismo, el cual simplemente muestra un mensaje en la consola.

Por otro lado, el resultado devuelto por la función **MostrarF** se asigna a la variable **s**. Cuando Visual Basic se encuentra con este tipo de asignación, procesa el código de la función y asigna el valor devuelto, por tanto la variable **s** contendrá la cadena "**Esta es la función MostrarF**" y tal como podemos comprobar por la salida producida al ejecutar este proyecto, eso será lo que se muestre en la consola.

Cuando los procedimientos de tipo Sub o las funciones (Function) pertenecen a una clase se dicen que son métodos de esa clase. Los métodos siempre ejecutan una acción, y en el caso de las funciones, esa acción suele reportar algún valor, el cual se podrá usar para asignarlo a una variable o para usarlo en una expresión, es decir, el valor devuelto por una función se puede usar en cualquier contexto en el que se podría usar una variable o una constante. Por otro lado, los procedimientos de tipo Sub sólo ejecutan la acción y nada más. (Sí, ya se que todo esto ya lo sabes, sobre todo porque acabo de explicarlo, pero lo repito para que no te queden dudas).

Cuando los procedimientos se convierten en métodos, (porque están declarados en una clase), estos suelen representar lo que la clase (o método o estructura) es capaz de hacer. Es decir, siempre representarán una acción de dicha clase.

¿Te ha quedado claro? Espero que sí.

## 12.4. Parámetros o argumentos de los procedimientos

Cuando necesitemos que un procedimiento realice una tarea, es posible que necesitemos indicarle alguna información adicional. Esa información se suele indicar mediante parámetros o argumentos. Los argumentos pasados a los procedimientos se indican a continuación del nombre del procedimiento y deben estar incluidos dentro de los paréntesis que siempre hay que usar con los procedimientos.

Por ejemplo, el método WriteLine de la clase Console permite que se indiquen mediante parámetros (o argumentos) los datos a mostrar en la consola.

Para indicar que un procedimiento acepta argumentos estos se indicarán de la siguiente forma:

<tipo procedimiento> <nombre del procedimiento>([<parámetro>][,<parámetro>])

Para verlo de forma clara, supongamos que tenemos un procedimiento llamado Saludar, al cual hay que pasarle un parámetro de tipo cadena. Dicho procedimiento usará ese

parámetro como parte de un mensaje que tendrá que mostrar por la consola. Sería algo como esto:

```
Sub Saludar(ByVal nombre As String)
    Console.WriteLine("Hola " & nombre)
End Sub
```

En este ejemplo, **nombre** sería el parámetro o argumento del método **Saludar**.

Para usar este procedimiento lo podríamos hacer de esta forma:

```
Saludar("Guillermo")
```

Si necesitamos que el procedimiento reciba más de un parámetro, se podrán indicar separándolos unos de otros con una coma. Veamos el método anterior en el que se indica además del nombre el tipo de saludo a realizar:

```
Sub Saludar(ByVal tipoSaludo As String, ByVal nombre As String)
    Console.WriteLine(tipoSaludo & " " & nombre)
End Sub
```

Este procedimiento con dos parámetros lo usaríamos de la siguiente forma:

```
Saludar("Hello", "Guille")
```

## 12.5. Parámetros por valor y parámetros por referencia

Normalmente, cuando se pasa un parámetro a un procedimiento, éste se suele pasar o indicar lo que se llama por valor, es decir, el parámetro será una copia del valor indicado, en el caso de Saludar("Guillermo") la constante "Guillermo" se copiará en la variable nombre. Cualquier cambio que se realice dentro del procedimiento a la variable nombre no afectará al parámetro. Seguramente dirás que sería imposible cambiar el contenido de una constante, que es al fin y al cabo lo que se le pasa al procedimiento Saluda en el ejemplo que te he mostrado, y estarías en lo cierto... (se ve que vas aprendiendo). Pero... (je, je, ¿te creías que no iba a haber un pero...), ¿que ocurre si ese parámetro es una variable en lugar de una constante? Por ejemplo, si tenemos el siguiente procedimiento:

```
Sub Saludar2(ByVal nombre As String)
    nombre = "Hola " & nombre
    Console.WriteLine(nombre)
End Sub
```

Al que llamamos con este otro código:

```
Sub PruebaSaludar2()
    Dim elNombre As String
    elNombre = "Guillermo"
    Saludar2(elNombre)
    '
    ' ¿qué valor mostraría esta línea?
    Console.WriteLine(elNombre)
End Sub
```

¿Que valor tendrá la variable **elNombre** después de llamar al procedimiento **Saludar2**?

La respuesta es: el que tenía antes de llamar al procedimiento.

¿Por qué? Si el valor se ha modificado...

Porque se ha pasado por valor (**ByVal**) y por tanto, lo que se ha pasado al procedimiento es una copia del contenido de la variable **elNombre**, con lo cual, cualquier cambio que se realice en la variable **nombre** sólo afectará a la copia, no al original.

Pero si queremos que el procedimiento pueda modificar el valor recibido como parámetro, tendremos que indicarle al Visual Basic .NET de que lo pase por referencia, para ello habrá que usar la instrucción **ByRef** en lugar de ByVal.

Veámoslo con un ejemplo:

```
Sub Saludar3(ByRef nombre As String)
    nombre = "Hola " & nombre
    Console.WriteLine(nombre)
End Sub

Sub PruebaSaludar3()
    Dim elNombre As String
    elNombre = "Guillermo"
    Saludar3(elNombre)
    '
    ' ¿qué valor mostraría esta línea?
    Console.WriteLine(elNombre)
End Sub
```

En esta ocasión la variable **elNombre** contendrá "Hola Guillermo".

La explicación es que al pasar la variable por referencia (**ByRef**), el VB lo que ha hecho es asignar a la variable nombre del procedimiento la misma dirección de memoria que tiene la variable **elNombre**, de forma que cualquier cambio realizado en **nombre** afectará a **elNombre**.

¿Te acuerdas de lo que hablamos sobre los tipos por valor y los tipos por referencia? Pues esto sería algo como lo que ocurre con los tipos por referencia.

**Nota:** En Visual Basic .NET, de forma predeterminada, los parámetros serán ByVal (por valor), a diferencia de lo que ocurría con las versiones anteriores de Visual Basic que eran por referencia (ByRef). Es decir, si declaras un parámetro sin indicar si es ByVal o ByRef, el VB.NET lo interpretará como si fuera ByVal.

**Resumiendo:**

Las variables indicadas con ByVal se pasan por valor, es decir se hace una copia del contenido de la variable o constante y es esa copia la que se pasa al procedimiento.

Por otro lado, los parámetros indicados con ByRef se pasan por referencia, es decir se pasa al procedimiento una referencia a la posición de memoria en la que está el contenido de la variable en cuestión, por tanto cualquier cambio efectuado a la variable dentro del procedimiento afectará a la variable indicada al llamar al procedimiento.

Por supuesto, todo esto es aplicable tanto a los procedimientos de tipo Sub como a los de tipo Function. En el caso de las funciones, el utilizar parámetros ByRef nos permiten devolver más de un valor: el que devuelve la función más los que se puedan devolver en los parámetros declarados con ByRef.

Ni que decir tiene que en un procedimiento se pueden usar indistintamente parámetros por valor como por referencia, es decir, podemos tener tanto parámetros declarados con ByVal como con ByRef, y, por supuesto, sólo los indicados con ByRef podrán cambiar el contenido de las variables indicadas al llamar al procedimiento.

Vamos a dejar la cosa aquí.

En la próxima entrega seguiremos con el tema de los parámetros, en esa ocasión veremos cómo indicar parámetros opcionales, es decir, parámetros que no son obligatorios indicar al llamar al procedimiento y si te preguntas porqué dejo ese tema que en realidad estaría relacionado con lo que acabamos de ver, te diré que es porque también te voy a explicar otras cosas relacionadas con la forma de declarar los procedimientos y cómo podemos tener varias "versiones" de un mismo procedimiento que tenga distinto número de parámetros... y como el tema ese será algo extenso... prefiero dejarlo para otra entrega... que esta ya ha dado mucho de sí y no quiero que tu cabecita termine por explotar... je, je... así que, paciencia y a esperar a la Semana Santa

que será la fecha en la que seguramente publicaré la entrega número trece... espero que no seas una persona supersticiosa y no te la saltes, ya que seguro que te perderás cosas interesantes...

No quiero que te vayas sin que veamos un pequeño resumen de lo que te he explicado en esta entrega número doce, aunque haya sido por encima o de pasada:

Los elementos de un proyecto de .NET: ensamblados, espacios de nombres, clases, módulos, estructuras; además de los miembros de las clases: campos, propiedades y métodos. Y lo que seguro que no ha sido de pasada es el tema de los procedimientos o métodos, así como los parámetros o argumentos de dichos procedimientos y la forma en que los podemos declarar: ByVal y ByRef.

# 13. Treceava entrega

Pues no, no ha sido para Semana Santa y por poco ni para el mes de abril... pero, lo importante es que ya está publicada, aunque no haya sido en el plazo que yo tenía planeado... y no es por justificarme, ya que al fin y al cabo lo mismo a ti te da igual... o puede que no te de igual, sobre todo si lees esto algún tiempo después de que esté publicado y ni siquiera sepas a que va todo este rollo que me está contando el Guille, pero... te lo cuento por si eres de los lectores fieles que está pendientes de que publique una nueva entrega y sigues esta serie de entregas en el momento (o casi) en que se publica, que también hay algunos que se incorporan pasado un tiempo desde su publicación y después hasta se quejan...

Pues eso, que si ves que las entregas no las publico muy a menudo, piensa que el Guille está "liado" con algunas cosillas que no le dejan tiempo para escribir...

Y ya, después de este rollo introductorio, vamos a ver qué tendremos en esta entrega número trece, que puede que a algunos les hubiese gustado que se llamara doce bis o algo así, (por aquello de que para los supersticiosos el trece trae mala suerte).

Como te comenté en la entrega anterior, en esta entrega veremos más cosas relacionadas con los procedimientos, así que... despeja tu mente y ¡empecemos!

## 13.1. Parámetros opcionales

Como vimos, en un procedimiento podemos indicar parámetros, en el caso de que el procedimiento necesite más de un parámetro, tendremos que indicarlos todos y cada uno de ellos, además hay que indicarlos en el mismo orden en el que se ha indicado en dicho parámetro. Por ejemplo, si tenemos la siguiente declaración de un procedimiento:

```
Sub Prueba(ByVal uno As Integer, ByVal dos As Integer)
```

Para poder usarlo hay que hacerlo de esta forma:

```
Prueba(10, 20)
```

El parámetro "uno" recibirá el valor 10, mientras que el parámetro "dos" recibirá el valor 20. Es decir, el primer valor indicado para el primer parámetro, el segundo valor indicado será para el segundo parámetro y así sucesivamente.

Por otro lado, si sólo quisiéramos indicar el primer parámetro, pero no el segundo, podríamos intentar hacer algo como esto:

```
Prueba(10)
```

Pero... ¡no se puede hacer! ya que produciría un error por la sencilla razón de que hay que indicar los dos parámetros "forzosamente".

Entonces... ¿cómo lo hago si en ocasiones quiero que sólo se indique un parámetro?

La respuesta es: indicando parámetros opcionales.

Para poder indicarle al Visual Basic .NET que un parámetro es opcional debemos indicarlo usando la instrucción **Optional** antes de la declaración del parámetro en cuestión, aunque la cosa no acaba aquí, ya que además tenemos que indicar el valor que tendrá por defecto, es decir, si no se indica ese parámetro, éste debe tener un valor predeterminado. Pero tampoco debemos creernos que haciendo esto de la forma que queramos podemos indicar que un parámetro es opcional. Bueno, si, lo dicho hasta ahora es válido, pero también debemos tener en cuenta que sólo podemos especificar parámetros opcionales después de todos los parámetros obligatorios.

Dicho esto, veamos cómo declarar el procedimiento Prueba para indicar que el segundo parámetro es opcional y que el valor predeterminado (si no se indica) es cinco:

```
Sub Prueba(ByVal uno As Integer, Optional ByVal dos As Integer = 5)
```

Con esta declaración podemos usar este procedimiento de estas dos formas:

**Prueba(10, 20)**

En este caso se indicará un 10 para el parámetro **uno** y 20 para el parámetro **dos**.

**Prueba(10)**

Si no indicamos el segundo parámetro, el valor que se usará dentro del procedimiento será el valor indicado en la declaración, es decir: 5.

Si quisiéramos que los dos parámetros fueran opcionales, tendríamos que declarar el procedimiento de esta forma:

```
Sub Prueba(Optional ByVal uno As Integer = 3, Optional ByVal dos As Integer = 5)
```

En este caso podemos llamar al procedimiento de estas formas:

1- 1Prueba()

2- Prueba(10)

3- Prueba(10, 20)

4- Prueba(,20)

5- Prueba(uno:=4)

6- Prueba(dos:=7)

7- Prueba(dos:=8, uno:=6)

8- Prueba(6, dos:=8)

Las tres primeras formas creo que son fácilmente compresibles.

En la cuarta sólo se indica el segundo parámetro, en este caso para "no" indicar el primero, dejamos el hueco, por eso se pone la coma y a continuación el segundo parámetro.

Esto está bien, y es la forma que "antes" se solía usar, pero en el caso de que haya muchos parámetros opcionales, puede ser algo "lioso" eso de tener que usar la coma para separar los parámetros que no se indican, ya que en el caso de que alguno de los parámetros opcionales no se vaya a indicar hay que "dejar" el hueco... imagínate que tenemos un procedimiento con 5 parámetros opcionales y sólo queremos indicar el tercero y el quinto, usando este mismo sistema lo haríamos así:

Prueba5(,,3,,5)

Y como puedes comprobar, es fácil olvidarse de alguna coma intermedia.

En las tres últimas formas de usar el procedimiento **Prueba**, usamos las variables de los parámetros opcionales para indicar el que queramos y el valor que le vamos a asignar, en estos casos usamos el nombre de la variable del parámetro opcional seguida de **:=** y el valor a asignar.

Usando este sistema, podemos indicar los parámetros que queramos y además en el orden que queramos, ya que no habrá ningún problema de confusión, por la sencilla razón de que se especifica el nombre de la variable y el valor que tendrá. Fíjate que incluso se puede cambiar el orden de los parámetros... ¡efectivamente! por la sencilla razón de que se indica el nombre de la variable del parámetro.

No, no es que pienses que no te enteras, es que ¡quiero que no te quede ninguna duda!

La octava forma es como para rizar el rizo, ya que sería lo mismo que: Prueba(6, 8), pero nos sirve para que sepamos que también podemos hacerlo así, aunque esto sólo sería efectivo si en lugar de sólo dos parámetros tenemos más de dos parámetros. En estos casos en los que se mezclan valores "por posición" con valores "por variable" hay que tener presente que si no se indica el nombre de la variable del parámetro, los parámetros indicados explícitamente deben estar en el mismo orden en el que están declarados en el procedimiento. Por ejemplo, esto daría error: **Prueba(uno:=6, 8)** por la sencilla razón de que si se indica un valor usando el nombre de uno de los parámetros los siguientes deben también indicarse usando el nombre del parámetro.

Si quieres saber más sobre parámetros opcionales, puedes usar este link de la ayuda de Visual Studio .NET:

ms-help://**MS.VSCC/MS.MSDNVS.3082**/vbcn7/html/vaconOptionalArgs.htm

## 13.2. Sobrecarga de procedimientos

Pero aunque esto de usar **Optional** está muy bien, hay que tener en cuenta de que esta instrucción existe "por compatibilidad" con las versiones anteriores de Visual Basic... o casi.

¿Por qué digo esto? Porque ahora, gracias a que Visual Basic .NET permite la sobrecarga de procedimientos, podemos hacer lo mismo de una forma más intuitiva y menos "liante" al menos de cara al que vea nuestro código (en ese "alguien" también nos incluimos nosotros, los autores del código).

Veamos lo que nos dice la documentación de Visual Studio .NET sobre la sobrecarga de propiedades y métodos (o lo que es lo mismo, sobre la sobrecarga de procedimientos):

*La sobrecarga consiste en crear más de un procedimiento, constructor de instancias o propiedad en una clase con el mismo nombre y distintos tipos de argumento.*

*La sobrecarga es especialmente útil cuando un modelo de objeto exige el uso de nombres idénticos para procedimientos que operan en diferentes tipos de datos.*

Es decir, que si necesitamos un procedimiento que utilice distinto número de parámetros o parámetros de distintos tipos, podemos usar la sobrecarga de procedimientos.

Sabiendo esto, podríamos hacer lo mismo que con el procedimiento **Prueba** mostrado anteriormente con estas declaraciones:

```
Prueba()

Prueba(ByVal uno As Integer)

Prueba(ByVal uno As Integer, ByVal dos As Integer)
```

Lo único es que no podríamos usar las formas 4 a 8. Pero esto no sería un inconveniente, ya que esas formas no son "lógicas" o intuitivas, como prefieras llamarlo.

La pregunta que seguramente te harás (o deberías hacerte) es:

Si escribo tres procedimientos con diferente número de parámetros ¿debo repetir el código tres veces, uno para cada uno de los procedimientos?

La respuesta es: si o no... depende. Es decir, si quieres, puedes escribir tres veces o tres códigos distintos, uno para cada procedimiento. Pero si no quieres no...

Vale... muy bien, pero ¿cómo puedo hacer que los tres procedimientos sean operativos sin tener que repetir prácticamente lo mismo en los tres procedimientos?

Pues... imagínate que quieres que funcione como en el primer ejemplo, el que usaba los parámetros opcionales. Es decir, si no se indicaba uno de los parámetros tomara un valor por defecto.

Podríamos hacerlo de la siguiente forma:

```
Sub Prueba2()
    ' Si no se indica ninguno de los dos parámetros,
    ' usar los valores "por defecto"
    prueba2(3, 5)
End Sub
Sub Prueba2(ByVal uno As Integer)
    ' Si no se indica el segundo parámetro,
    ' usar el valor "por defecto"
    Prueba(uno, 5)
End Sub
Sub Prueba2(ByVal uno As Integer, ByVal dos As Integer)
    Console.WriteLine("uno = {0}, dos = {1}", uno, dos)
    Console.WriteLine()
End Sub
```

Es decir, si no se indica ningún parámetro, se usará la primera declaración, desde la que se llama a la tercera declaración que recibe los dos valores que nosotros queremos que tenga por defecto.

Si se usa la segunda declaración, se usará como primer valor el que se ha indicado y como segundo el predeterminado.

Y por último, si indicamos los dos parámetros se llamará a la tercera declaración.

**Nota:** Cuando se usan procedimientos sobrecargados, es el propio compilador de Visual Basic .NET el que decide cual es el procedimiento que mejor se adecua a los parámetros que se han indicado al llamar a ese procedimiento.

Otra de las ventajas de la sobrecarga de procedimientos, es que además de poder indicar un número diferente de parámetros, es que podemos indicar parámetros de distintos tipos.

Esto es útil si queremos tener procedimientos que, por ejemplo, reciban parámetros de tipo **Integer** o que reciba parámetros de tipo **Double**, por poner sólo dos ejemplos.

Incluso podemos hacer que una función devuelva valores de tipos diferentes, aunque en este caso el número o tipo de los parámetros debe ser diferente, ya que *no se pueden sobrecargar procedimientos si sólo se diferencian en el tipo de datos devuelto*.

Tampoco se pueden sobrecargar Propiedades con métodos (Sub o Function), es decir, sólo podemos sobrecargar propiedades con otras propiedades o procedimientos (Sub o Function) con otros procedimientos (Sub o Function).

Para más información, puedes consultar la ayuda de Visual Studio .NET:

ms-help://MS.VSCC/MS.MSDNVS.3082/vbcn7/html/vaconOverloadingInVisualBasicNET70.htm

ms-help://MS.VSCC/MS.MSDNVS.3082/vbcn7/html/vaconoverridingexistingmethods.htm

## 13.3. Sobrecargar el constructor de las clases

Una de las utilidades más prácticas de la sobrecarga de procedimientos es sobrecargar el constructor de una clase.

Sí, ya se que no te he explicado de que va eso del constructor de una clase, así que aquí tienes un pequeño adelanto:

**El constructor de una clase es un procedimiento de tipo Sub llamado New**, dicho procedimiento se ejecuta cada vez que creamos un nuevo objeto basado en una clase.

Si al declarar una clase no escribimos el "constructor", será el compilador de Visual Basic .NET el que se encargará de escribir uno genérico.

**Nota:** De esto del constructor de las clases hablaremos más en otra ocasión, por ahora sólo quiero que sepas que existe y para que se usa.

Como te decía, esto es útil si queremos que al crear un objeto (o instancia) de una clase podamos hacerlo de varias formas, por ejemplo, sin indicar ningún parámetro o bien indicando algo que nuestra clase necesite a la hora de crear una nueva instancia de dicha clase.

Por ejemplo, si tenemos una clase llamada Cliente, puede sernos útil crear nuevos objetos indicando el nombre del cliente que contendrá dicha clase.

Veámoslo con un ejemplo:

```
Class Cliente
    Public Nombre As String
    Public email As String
    '
    Sub New()
        '
    End Sub

    Sub New(ByVal elNombre As String)
        Nombre = elNombre
    End Sub
End Class
```

Esta clase nos permite crear nuevos objetos del tipo Cliente de dos formas.

Por ejemplo si tenemos una variable llamada cli, declarada de esta forma:

**Dim cli As Cliente**

podemos crear nuevas instancias sin indicar ningún parámetro:

**cli = New Cliente()**

o indicando un parámetro, el cual se asignará a la propiedad Nombre de la clase:

**cli = New Cliente("Guillermo")**

**Nota:** Como te dije en la nota anterior, de esto del constructor hablaremos en otra ocasión con algo de más detalle o profundidad.

## 13.4. Array de parámetros opcionales

También puede ser que necesitemos que un procedimiento reciba una cantidad no predeterminada de parámetros, en esos casos podemos usar un array (o matriz) de parámetros opcionales.

Para poder indicar esta clase de parámetros, debemos usar la instrucción **ParamArray** seguida de una matriz, por ejemplo si tenemos un procedimiento declarado de esta forma:

```
Sub Prueba3(ByVal uno As Integer, ByVal ParamArray otros() As Integer)
```

Podemos llamarlo usando uno o más parámetros:

Prueba3(1)

Prueba3(1, 2, 3, 4, 5, 6, 7)

Prueba3(1, 2)

Lo único que debemos tener presente es que el array de parámetros indicados con **ParamArray** debe ser el último parámetro indicado en la declaración del procedimiento y que no es necesario usar la palabra **Optional**, ya que estos parámetros son opcionales de forma predeterminada.

Para más información, consulta la ayuda de Visual Studio .NET:

ms-help://MS.VSCC/MS.MSDNVS.3082/vbcn7/html/vaconUnderstandingParamArrays.htm

Bueno, creo que vamos a dejarlo aquí, para que digieras todo lo que hemos visto, que aunque no haya sido muy extenso es algo que deberías probar, además de consultar la documentación de Visual Studio .NET, aunque sea en los temas relacionados con esto de la sobrecarga de procedimientos y los parámetros opcionales... que tampoco es plan de que te lo leas todo, ya que entonces... no tendré que contarte nuevas cosas...

En la siguiente entrega seguiremos tratando el tema de los procedimientos, entre las cosas que tengo previstas explicarte está la forma de indicar el ámbito o "cobertura" de los procedimientos y seguramente entraremos en detalle de cómo declarar y usar los procedimientos de tipo Property.

Pero eso será en la próxima entrega, mientras tanto... que disfrutes.

# 14. Glosario

Algunos conceptos que usaremos con bastante frecuencia en el resto de las entregas:

| Palabra | Descripción |
|---|---|
| **Clases / Objetos** | Prácticamente todo lo que manejemos en el entorno .NET es una clase u objeto, de hecho todas las clases derivan de una clase u objeto básico: la clase System.Object |
| **Programación Orientada a Objetos (OOP / PPO)** | Una forma de programar basada en la reutilización de código mediante herencia, encapsulación y polimorfismo. |
| **Herencia** | La posibilidad de que una clase herede las propiedades y métodos de otra clase de forma que se puedan usar con la nueva clase de igual forma que si se hubiesen escrito directamente en ella. |
| **Encapsulación** | La posibilidad de ocultar el código usado para implementar un método o cualquier otro procedimiento o función de forma que lo único que interese sea el interface expuesto por la clase u objeto. |
| **Polimorfismo** | La posibilidad de usar en clases diferentes propiedades o métodos con el mismo nombre de forma que cuando se usen no nos preocupe a que clase pertenece.<br>Por ejemplo el objeto básico del que derivan todas las clases de .NET tiene una propiedad llamada ToString, ésta propiedad estará implementada de forma diferente en diferentes clases, pero nosotros la usaremos de la misma forma, sin importarnos que objeto estemos usando. |
| **Interface** | Se dice que las propiedades y métodos expuestos por una clase forman el interface de la misma. |
| **Clases abstractas** | Son clases que exponen un interface el cual hay que usar en las clases que se hereden de dicha clase abstracta. |
| **Interface / Implements** | Los interfaces a diferencia de las clases es que no hay que escribir código para los métodos o propiedades que expone, simplemente se indica la "declaración".<br>Usando Implements, se pueden usar esas interfaces en las clases, aunque hay que escribir el código de cada método o propiedad implementado. |
| **Procedimiento** | Un método, función o propiedad de una clase o módulo. |
| **Método** | Un procedimiento (Sub, Function -función) que se usa para realizar una tarea específica en la clase o módulo. |
| **Sub** | Un procedimiento SUB es como una instrucción, es decir, realiza una tarea (ejecuta el código que haya en su interior), pero no devuelve un resultado. |
| **Function** (Función) | Los procedimientos FUNCTION son como las funciones del vb.NET, es decir, realizan una tarea, al igual que un Sub, pero siempre suelen devolver un valor, resultado del código que se ha ejecutado en su interior.<br>A las funciones no se les puede asignar valores, a diferencia de las Propiedades. |
| **Property** (Propiedad) | A diferencia de los métodos, las propiedades se usan para "configurar" la forma que tendrá la clase.<br>Algunas veces es difícil diferenciar un método de una propiedad, pero por convención los métodos realizan tareas. Por ejemplo, el ancho de un objeto es una propiedad, mientras que mostrar el objeto se realizaría con un método.<br>A las Propiedades se les puede asignar valores y pueden |

| | |
|---|---|
| | devolverlos, (como las funciones). Aunque también pueden existir propiedades de solo lectura, (solamente devuelven valores), o de solo escritura, (sólo se les puede asignar valores, pero no los devuelven). |
| **Parámetro** | Los métodos o propiedades pueden tener parámetros, (uno o varios), los cuales le indicarán los valores que deben usar para la tarea que debe realizar.<br>Por ejemplo, un método Contar podría recibir un parámetro con el valor de las veces que tiene que contar. |
| **Parámetros opcionales** | Algunos procedimientos que aceptan parámetros, pueden tener también parámetros opcionales, los cuales, como su nombre indica, pueden o no ser incluidos en la llamada al procedimiento.<br>Los parámetros opcionales tienen unos valores por defecto, el cual se usará en caso de que no se especifique. |
| **Variable** | Son "espacios" de memoria en la que se almacena un valor. Se usarán para guardar en memoria los valores numéricos o de cadena de caracteres que nuestro programa necesite. Usa este link para ver los distintos tipos de datos. |
| **Constante** | Valores numéricos o de cadena que permanecen constantes, sin posibilidad de cambiar el valor que tienen.<br>En caso de que necesitemos cambiar el valor, usaremos las variables. |
| **Evento** | Los eventos son procedimientos (SUB) que se ejecutan normalmente cuando el sistema Windows los provoca, por ejemplo, al hacer click en una ventana o en cualquier objeto de la ventana, cuando cambiamos el tamaño de una ventana, cuando escribimos en una caja de textos, etc. |
| **Handles** | En VB.NET se usa Handles, seguido del nombre del evento, para indicar qué evento es el que se maneja en el procedimiento indicado.<br>El formato suele ser:<br>Sub Nombre(parámetros) Handles Objeto.Evento |
| **Sobrecarga (Overload)** | Se dice que un método está sobrecargado cuando existen distintas versiones de dicho método en la clase.<br>Por ejemplo métodos con el mismo nombre que reciban parámetros de distintos tipos. |
| **Formulario (ventana)** | Un formulario es una ventana de Windows la cual usaremos para interactuar con el usuario, ya que en dicha ventana o formulario, estarán los controles y demás objetos gráficos que mostraremos al usuario de nuestra aplicación.<br>Los formularios también son llamados "formas" o Forms en su nombre en inglés. |
| **MyBase** | La palabra clave *MyBase* se comporta como la clase de la que ha derivado la clase actual, es decir si una clase deriva de una (o hereda a otra) clase, MyBase se referirá a dicha clase base, de esta forma es posible acceder a los métodos, propiedades y eventos de la clase de la que se deriva (o hereda) la clase actual. |
| **Me (this)** | La palabra clave (o instrucción) *Me* hace referencia a la clase actual.<br>Por ejemplo Me.Width se refiere a la propiedad Width de la clase actual.<br>En C# en lugar de Me es **this**. |
| **Colecciones** | Serie de datos que están guardados en una lista, array (o matriz) o una colección propiamente dicha y que permite interactuar con los elementos de las mismas, pudiendo añadir, |

| | |
|---|---|
| | recuperar, eliminar uno o todos, saber cuantos elementos hay, etc. |
| **Expresiones** | Una expresión es una secuencia de operadores y operandos que describe un cálculo. Normalmente una expresión se evalúa en tiempo de ejecución.<br>Existen expresiones numéricas y alfanuméricas o de caracteres. |
| **Expresiones Lógicas** | Las expresiones lógicas son expresiones pero cuyo resultado es un valor "lógico" (verdadero o falso).<br>Este tipo de expresiones se usan normalmente con instrucciones que normalmente necesitan un valor verdadero (true) o falso (false) |
| **Módulo** | Los módulos, al igual que las clases, son "espacios" en los cuales se incluyen declaraciones de variables, procedimientos, funciones, etc.<br>Pero a diferencia de las clases, el código contenido en un módulo siempre está disponible de forma directa, sin necesidad de crear una "instancia" de dicho módulo. |
| **Instancia** | Para poder usar una clase u objeto, hay que crear una instancia del mismo.<br>Es decir, debemos declarar una variable y a esa variable asignarle el objeto o clase en cuestión para que podamos usarlo.<br>Es como si tuviésemos que darle vida al objeto par poder usarlo. |
| **Enumeraciones (Enum)** | Las enumeraciones son una serie de valores constantes (de tipo numérico), que de alguna forma están relacionadas entre sí.<br>A diferencia de las constantes normales, una variable declarada como una enumeración, puede tomar cualquiera de los valores indicados en la enumeración. |
| **Array (matrices)** | Los arrays (o matrices) son un tipo de variable que permiten tener más de un elemento, (o valor en su interior), a los que se pueden acceder mediante un índice.<br>Un array también es el tipo en el que se basan todas las matrices o arrays. |
| **Common Language Runtime (CLR)** | El CLR (*Common Language Runtime*) es el motor en tiempo de ejecución del .NET Framework, es decir la parte del "entorno" que se encarga de ejecutar el código de los lenguajes del .NET Framework. |