Programación con ASP.NET. Visual Basic.NET

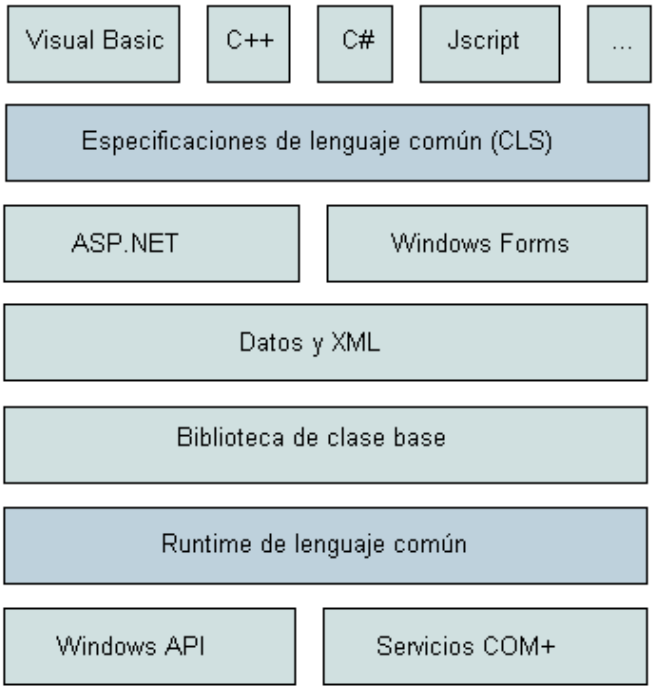
Indice

3. Programación con ASP.NET. Visual Basic.NET

1. Programación en ASP.NET

Volvemos otra vez a repasar los conceptos de .NET. Microsoft en un intento por unificar el mundo del desarrollo diseño el modelo llamado .NET Framework. Éste es el núcleo que proporciona la tecnología para el desarrollo de aplicaciones para Windows y de aplicaciones para web (ASP.NET)

Recuerda este esquema:



Analicémoslo de abajo hacia arriba:

- Tenemos que el núcleo de .NET Framework se sitúa en el "Runtime de lenguaje común". Es el responsable de ejecutar el código que se realiza sobre las propias "tripas" de Windows que están en la última fila. Se encarga de decirle a Windows lo que debe hacer con las instrucciones que le hemos proporcionado.
- En el siguiente nivel tenemos la "biblioteca de clase base". Windows está basado en objetos: formularios, botones, vídeos, iconos... todo son objetos manipulables. Esta biblioteca pone a nuestra disposición una ingente cantidad de objetos que podemos utilizar para nuestras aplicaciones.
- El siguiente nivel hace referencia a la capa de manejo de datos. La capa de datos y XML contienen las clases .NET que trabajan con bases de datos y con XML
- Subiendo nos encontramos con dos formas de crear aplicaciones: con "Windows Forms" para crear programas Windows como conocemos y programas con ASP.NET que generan páginas web (nuestro curso)
- Saltamos ahora a la primera línea. Una de las tremendas ventajas del mundo .NET es que podemos utilizar diferentes lenguajes para generar el código y obtener el mismo resultado. Por ejemplo uno de

los lenguajes soportados es "C#". Podemos crear un programa con C# para crear un programa Windows y que utilice los objetos de .NET. Igualmente podemos escribir un programa con Visual Basic.NET trabajando sobre los mismos objetos para conseguir el mismo resultado.
Esto es de tremenda utilidad porque podemos utilizar el lenguaje que mas nos guste para programar. como todos trabajan sobre la base de .NET todos utiliza los mismos objetos.
Nosotros hemos decidido que por sencillez vamos a utilizar Visual Basic.NET para construir nuestras páginas ASP.NET. En muchos sitios web de programación verás como utilizan este lenguaje o C# para realizar lo mismo. La elección es a gusto del consumidor pero para comenzar es mas sencillo VB.NET así que es el que seleccionamos

- Finalmente la segunda capa traduce lo que hayamos escrito en el lenguaje elegido en un conjunto de instrucciones iguales para .NET. Así utilicemos el lenguaje que sea este parte lo traduce a un lenguaje interno estándar

¡Importante! Como ves podemos elegir varios lenguajes para escribir nuestras páginas ASP.NET. Seleccionamos VB.NET por sencillez pero podríamos obtener los mismos resultados con los otros lenguajes.

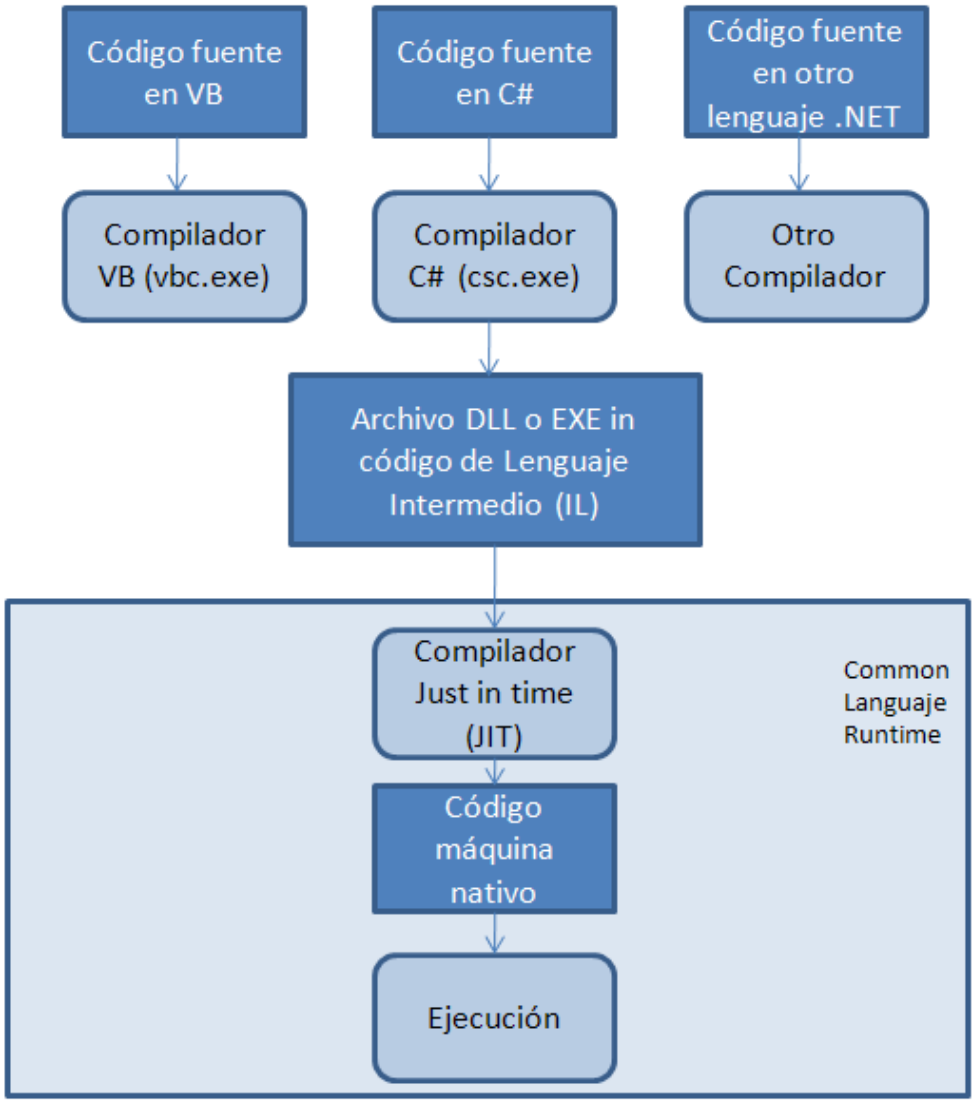
Y ahora un pequeño resumen de la evolución de las distintas versiones de ASP.NET

- ASP.NET 1.0. Primera versión de la implementación de .net framework para páginas web. Es muy superior al antiguo ASP y permite utilizar un modelo de objeto mucho mas amplio
- ASP.NET 1.1. Segunda versión que corregía muchas limitaciones y mejora general del rendimiento
- ASP.NET 2.0. La tercera versión amplió sustancialmente la cantidad de objetos e incorporó por primera vez un entorno de desarrollo integrado potente y eficaz. Incorporó muchas funciones para la realización de sitios web completos. Antes se centraba mas en las páginas y en este versión mejoró mucho en el tratamiento de los estilos, páginas maestras, facilidad de navegación...
- ASP.NET 3.5. Mantiene básicamente la potencia de la versión 2.0 pero añade dos grandes e importantes mejora. La principal es la incorporación del kit de herramientas AJAX que proporciona al desarrollador potentes herramientas para elaborar mejores páginas y un control total sobre los elementos que hay en ellas, por ejemplo, la posibilidad de "arrastrar y soltar", tan utilizadas en los programas normales con los formularios pero que no se podían implementar en las páginas web. La segunda mejora es el soporte de LINQ, un conjunto de mejoras del lenguaje incluidas en .NET 3.5 que permite consultar datos de la misma forma que se realiza en una base de datos.

Nota No, nos hemos olvidado de la versión 3.0 porque simplemente no existe. Esto es porque Microsoft liberó un conjunto de herramientas para otros entorno bajo una versión 3.0, como éstas no eran para ASP.NET no quiso poner 3.0 para que los usuarios no pensasen que estas herramientas se incorporaban en ASP.NET. Por tanto diferenció las tecnologías llamando a la siguiente versión 3.5

1.1 El lenguaje de programación. Desde nuestro código hasta el código máquina

Hemos dicho que bajo .NET framework podemos utilizar distintos lenguajes que utilizarán todos los objetos de éste. Los dos principales lenguajes de Microsoft son C# y VB. ¿Cual es mejor? Pues los dos, al trabajar en el mismo entorno tienen la misma potencia porque los dos tienen el mismo entorno de desarrollo y los mismos objetos o componentes para trabajar. ¿por cual me decido? Pues nos decidiremos por Visual Basic (VB) ya que es de sintaxis mas sencilla. C# lo acogerán con gusto los programadores del lenguaje C, por que es la misma sintaxis, mas compleja pero algo mas potente que la de VB, en el sentido de que se pueden realizar operaciones mas complejas en una sola instrucción, pero nada que no se pueda hacer con VB. Al final el resultado es el mismo... repasemos que pasa con esa página que escribimos en el lenguaje que hayamos decidido utilizar:



Estas son las fases por las que pasa nuestro código antes de ejecutarse, son distintas etapas en las que primero se traducen al lenguaje común y por fin realiza la ejecución. Luego el CLR (common languaje runtime) es el motor que soporta todos los lenguajes de .NET. Las bondades de esta arquitectura ya las hemos comentado así que vamos a pasar a ver unos detalles mas del código y enseguida comenzaremos a trabajar con nuestro Visual Basic.NET

Como ya sabes los ordenadores sólo entienden un lenguaje: el de 111 y 000, es decir el binario. Estas instrucciones son el llamado código máquina porque precisamente es el que entiende la máquina. Como es absolutamente imposible escribir un lenguaje en binario utilizaremos un lenguaje mas sencillo que luego traduzca esta instrucciones en código máquina.

El proceso para esta conversión se llama **compilación** que literalmente convierte unas instrucciones en un lenguaje sencillo a las instrucciones equivalentes en código máquina. Existen dos tipos de compilación:

Código interpretado

El código interpretado hace lo siguiente: Windows extrae una línea del código la traduce a lenguaje máquina y la ejecuta, luego coge la siguiente la traduce y la ejecuta...

Ventajas	Inconvenientes
El código es cómodo para depurar, ya que no es necesario volver a compilar tras un cambio.	La ejecución se ralentiza, al ser necesaria la interpretación línea a línea cada vez.
No es necesario disponer de un compilador, ya que el intérprete (que forma parte del navegador) ejecuta el script.	El código es visible y puede ser objeto de plagio por parte de otras personas.
El mantenimiento es fácil y rápido, por parte del autor o de otro programador.	El usuario tiene acceso al código y puede modificarlo, estropeando alguna operación.

Los intérpretes son mas lentos y no forman un proyecto compacto ya que se ejecuta a instrucción a instrucción. Ejemplos son el antiguo Visual Basic y las antiguas páginas ASP.

Código pre-compilado

El código pre-compilado o compilado presenta grandes ventajas al traducir completamente el programa y generar un fichero ejecutable mucho mas depurado y coherente.

Ventajas	Inconvenientes
El código compilado se ejecuta muy rápido, al no ser necesaria una traducción cada vez.	Es necesario disponer de un compilador-linkador para el proceso de la compilación.
El código compilado no puede ser "abierto" por otras personas. No es necesario transmitir el código fuente.	El código compilado suele ocupar bastante en disco, ya que incorpora en el propio código algunas librerías del sistema.
El código compilado puede estar, íntegramente, incluido en un solo fichero.	Depurar un programa implica volver a compilar tras los cambios.

Compilados son los mejores lenguajes: .NET, Java, ...

1.2 Los objetos

Esta palabra puede producir un pequeño desconcierto al plantear que debemos realizar una programación "orientada a objetos". Pues no te preocupes, es todo los contrario, al no tener que programar con algo concreto sino un poco mas abstracto nos va a permitir una mayor flexibilidad y potencia. Veamos una pequeña definición de los objetos y de las clases

Las clases

Todo lo que tiene .NET Framework son clases. Una clase no es ni más ni menos que código. ¿Pero no acabo de decir que son todo objetos? ¿Que es esto de las clases?

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.

Un coche tiene unas propiedades: color, marca, modelo, ... y unos métodos para trabajar con él: arrancar, frenar, cambiar de marcha. La definición de estas partes es lo que llamamos clase, sólo la definición, para trabajar con el coche lo veremos luego. Luego es algo muy sencillo que no debe parecernos ni complejo, ni técnico. Los antiguos programadores de VB son los que menos acostumbrados están a trabajar con clases porque se podían prescindir de ellas pero en VB.NET no. Por eso es una ventaja para los novatos porque aprenderéis de 0 los conceptos y os quedarán mas claros.

Los Objetos

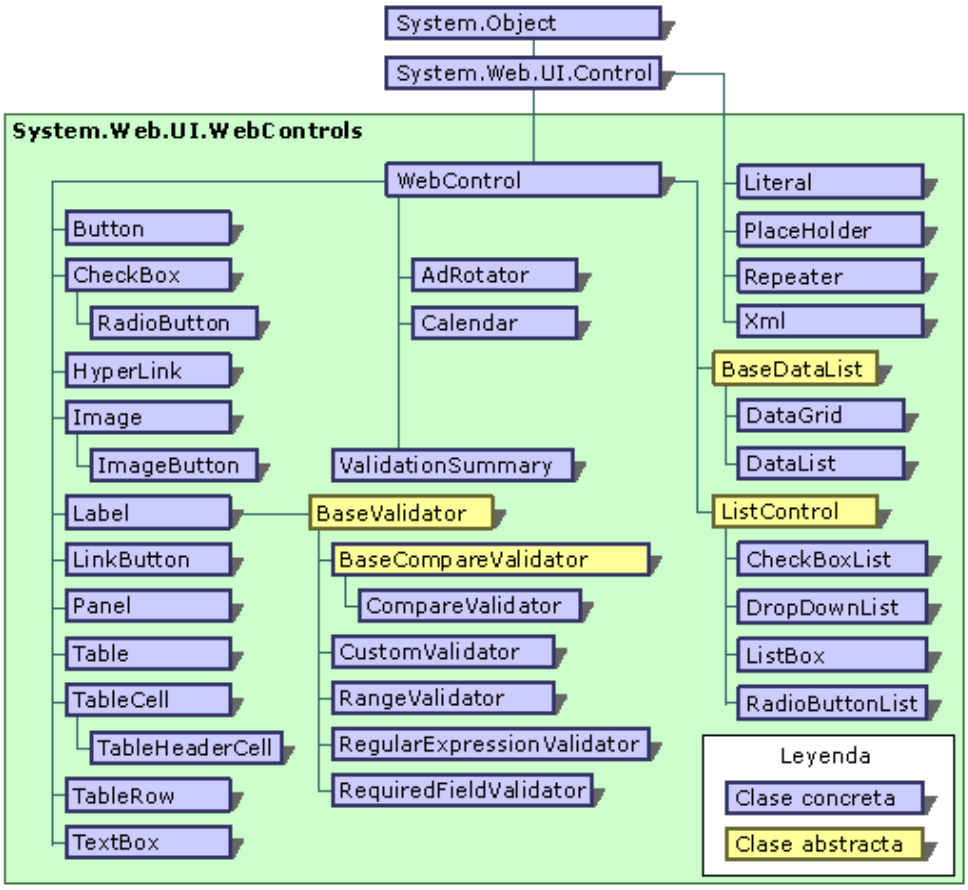
Por un lado tenemos una clase que es la que define un "algo" con lo que podemos trabajar. Pero para que ese "algo" no sea un "nada", tendremos que poder convertirlo en "algo tangible", es decir, tendremos que tener la posibilidad de que exista. Aquí es cuando entran en juego los objetos, ya que un objeto es una clase que tiene información real. Por fin podemos crear un coche del cual ya tenemos su definición en la clase Coche

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.

En nuestros formularios: tenemos 10 botones y ... han sido creados a partir de la clase "Botón". Otro ejemplo es que tenemos una clase que se llama "Coches" donde describe que es y cómo funciona un coche pues bien, podemos crear diferentes coches a partir de la clase "Coches", cada uno puede tener sus propias propiedades: color, ... pero funcionan todos igual

¿Quedan claros estos dos conceptos? Vale pues ahora sólo comentar que existen objetos de mas categoría que otros y que esta relación se llama jerarquía de objetos. Por ejemplo un objeto Coche puede tener a su vez varios objetos mas pequeños: motor, carrocería, ...

En .NET tenemos entonces varias Clases principales y debajo de ellas todas las instrucciones del lenguaje. Por ejemplo habrá una clase para los formularios (ventanas de Windows) que a su vez tendrán otras clases dentro: botones, textos, imágenes. Otra clase sería la colección de funciones matemáticas que podemos utilizar. Para que hagas una idea, esta es la jerarquía de objetos para desarrollo en Web:



No te preocupes, veremos esto con mas detalle a lo largo del curso, de momento nos quedamos con la idea de que todo lo que tenemos en Windows (navegador incluido) son objetos creados a partir de sus plantillas o "clases"

1.3 La clase base .NET

Una de las ventajas de las clases que nos ofrece .NET es que tenemos una enorme cantidad e material para trabajar con él en la "biblioteca de clase base". Esta contiene una enorme cantidad e funciones disponibles para nuestros programas y que nos simplificará las tareas mas comunes.

La clase base contiene muchas funciones diferentes. Por ejemplo necesitamos mostrar un texto pero ¿qué pasa si en lugar de hacerlo de la forma básica lo hacemos a través de unas operaciones mas especializadas y versátiles? por ejemplo para incluir o incrustarlo automáticamente con rectángulos, gráficos animados... Estas funciones están agrupadas en un "spacename" o espacio de nombres llamado "System.Drawing"

Un **spacename** o espacio de nombres es simplemente una forma de agrupar las distintas clases del mismo tipo. Por ejemplo System.Drawing agrupa todas las clases para trabajar con gráficos, otros como veremos mas adelante agrupan acceso a ficheros, a bases de datos...

Pero de momento no te líes, simplemente que podemos utilizar como una especie de grupos de instrucciones y otros componentes añadiendo o utilizando "namespaces" a nuestros programas. De esta forma mantenemos los recursos de las aplicaciones en diferentes sitios reduciendo los conflictos.

Hay espacios de nombres específicos por ejemplo para los gráficos y otros para trabajar con bases de datos, así los tenemos organizados de una forma mas cómoda para localizarlo y referirnos a ellos en el código.

Para que nuestras páginas reconozcan este grupo de clases incluiremos una "directiva" en la parte superior del código. Por ejemplo si vamos a utilizar el spacename System.Drawing escribiremos esto en la parte superior de la página:

```
import Namespace=System.Drawing
```

Las directivas son unas instrucciones especiales que se aplican para todo el programa o toda la página. Al incluir la directiva al comienzo de ella se reconocerá todo el código que hagamos referencia a System.Drawing. De no haberla puesto nos habría dado error al intentar utilizar estos objetos.

Hay varios spacenames pero no te preocupes son siempre los mismos y con varios ejemplos verás que se repiten continuamente. Pero ¿porque hace falta esto y .NET no reconoce ya todo al comienzo? La razón principal es el rendimiento, al manejar todas clases de golpe el programa es mas grande y complejo, si le indicamos los grupos de instrucciones que necesitaremos pues mejor.

2. Visual Basic .NET

2.1 Las variables y los tipos de datos.

Las variables como ya conoces son partes fundamentales de los programas y sirven para almacenar datos en memoria bajo un nombre descriptivo. Existen muchos tipos de variables dependiendo de lo que queramos almacenar: fechas, cadenas de caracteres, enteros, reales ... y en nuestros programas debemos definir cuantas y de que tipo utilizaremos.

Para los que venís de ASP buenas y malas noticias. La buena es que por fin hay declaraciones de variables de verdad y además están fuertemente tipificadas (adiós a los variant) y la mala es que hay que abandonar el vicio de no declarar variables.

VB.NET es un lenguaje "fuertemente tipificado" es decir, debemos declarar todas las variables que vayamos a utilizar y además decirle de que tipo de datos es. Esta obviedad no existía hasta ahora ya que en ASP no existían las variables y en el antiguo Visual Basic (hasta la versión 6) era un poco caótico la utilización correcta de las variables por la supuesta ventaja de no decir que de tipo son las variables al declararlas. Tranquilo iremos despacio con este tema...

Esto del tipificado significa por ejemplo que definíamos una variable "a" y no hacía falta decirle que tipo de datos iba a contener, así que podía admitir un número entero, con decimales, una fecha, una cada de caracteres... esto hacía que si el resultado de una operación o asignación no eran correcto el programa ni se enteraba.

El primer paso será aprender qué son, cómo se definen y cómo funcionan las variables

2.2 Variables, constantes y otros conceptos relacionados

Una variable es un identificador que guarda un valor que puede ser modificado a lo largo del programa, por ejemplo asignarle a la palabra "edad" el valor "39". El concepto de constante, es algo que permanece inalterable a lo largo del programa. Por ejemplo podemos asignarle a la palabra "euro" el valor 166,386.

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, una variable es una dirección de memoria en la que se guarda un valor o 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), pero en cualquier caso, la forma de hacerlo siempre es de la misma, 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.

Al ser una variable 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 modificará y el que se almacenará será el nuevo.

También podemos aplicar expresiones al asignar una variable, una expresión es 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.

Cuando se asignan valores de cadenas de caracteres ó alfanuméricos (string) el contenido de la variable debe ponerse entre comillas "". Para asignar una cadena 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 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 esto utilizaremos las declaraciones de variables. La declaración de una variable es el proceso por el que le decimos s VB.NET que cree una variable y le indicaremos su nombre y su tipo. Es necesario, aunque no obligatorio, declarar las variables según el tipo de datos que va a almacenar. Esto es una de las cosas que mas nos pueden chocar al principio de VB.NET (en general de VB): las variables podemos declararlas o no.

Ya hablaremos de esto mas adelante pero si no hace falta declararlas... ¿para qué tanta teoría e instrucciones? Bueno pues porque existen las dos posibilidades y la de no declarar variables puede ser muy cómodo al principio pero muy complejo mas adelante. Las normas del buen programador obligan a declarar todas las variables que utilicemos en el programa. Esto puede dar un poco mas de trabajo pero es imprescindible para escribir un buen código.

Para declarar una variable utilizaremos las palabras clave: DIM, PRIVATE, PUBLIC ó STATIC, dependiendo de cómo queremos que se comporte.

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.

Vale, existe un tipo de datos para los valores numéricos y otro para los alfanuméricos... ahora veamos en una tabla todos los tipos existentes y comentaremos cuales debemos utilizar:

2.3 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. Valores: **True** o **False.** 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. Dim b As Boolean = True
Byte	**System.Byte**	1 byte	Un valor positivo, sin signo, para contener datos binarios. Valores: de 0 a 255 Puede convertirse a: **Short**, **Integer**, **Long**, **Single**, **Double** o **Decimal** sin recibir overflow Dim b As Byte = 129
Char	**System.Char**	2 bytes	Un carácter Unicode. Valores: de 0 a 65535 (sin signo). No se puede convertir directamente a tipo numérico. Para indicar que una constante de cadena, realmente es un Char, usar la letra C después de la cadena: Dim c As Char = "N"c
Date	**System.DateTime**	8 bytes	Una fecha. Valores: desde las 0:00:00 del 1 de Enero del 0001 hasta las 23:59:59 del 31 de Diciembre del 9999. Las fechas deben representarse entre almohadillas # y por lo habitual usando el formato norteamericano: #m-d-yyyy# Dim d As Date = #10-27-2001#
Decimal	**System.Decimal**	16 bytes	Un número decimal. Valores: de 0 a +/-79,228,162,514,264,337,593,543,950,335 sin decimales; de 0 a +/-7.9228162514264337593543950335 con 28 lugares a la derecha del decimal; el número más pequeño es: +/-0.0000000000000000000000000001 (+/-1E-28). En los literales se puede usar la letra D o el signo @ para indicar que el valor es Decimal. Dim unDecimal As Decimal = 922337203685477580 8D Dim unDecimal2 As Decimal = 987654321.125@
Double	**System.Double**	8 bytes	

			Un número de coma flotante de doble precisión. Valores: de -1.79769313486231570E+308 a -4.94065645841246544E-324 para valores negativos; de 4.94065645841246544E-324 a 1.79769313486231570E+308 para valores positivos. Se puede convertir a **Decimal** sin recibir un overflow. Se puede usar como sufijo el signo almohadilla # o la letra R para representar un valor de doble precisión: Dim unDoble As Double = 125897.0235R Dim unDoble2 As Double = 987456.0125#
Integer	**System.Int32**	4 bytes	Un número entero (sin decimales) Valores: de -2,147,483,648 a 2,147,483,647. Se puede convertir a **Long**, **Single**, **Double** o **Decimal** sin producir overflow. Se puede usar la letra I o el signo % para indicar que es un número entero: Dim unEntero As Integer = 250009I Dim unEntero2 As Integer = 652000%
Long (entero largo)	**System.Int64**	8 bytes	Un entero largo (o grande) Valores: de -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807. Se puede convertir a **Single**, **Double** o **Decimal** sin producir overflow. Se puede usar la letra L o el signo & para indicar que es un número Long: Dim unLong As Long = 12345678L Dim unLong2 As Long = 1234567890&
Object	**System.Object** (class)	4 bytes	Cualquier tipo se puede almacenar en una variable de tipo **Object**. Todos los datos que se manejan en .NET están basados en el tipo Object.
Short (entero corto)	**System.Int16**	2 bytes	Un entero corto (sin decimales) Valores: de -32,768 a 32,767. Se puede convertir a: **Integer**, **Long**, **Single**, **Double** o **Decimal** sin producir un overflow. Se puede usar la letra S para indicar que es un número entero corto: Dim unShort As Short = 32000S
Single	**System.Single**	4 bytes	Número de coma flotante de precisión simple. Valores: de -3.4028235E+38 a -1.401298E-45 para valores negativos; de 1.401298E-45 a 3.4028235E+38 para valores positivos. Se puede convertir a: **Double** o **Decimal** sin producir overflow. Se pueden usar la letra F y el símbolo ! para indicar que es un número Single: Dim unSingle As Single = 987.125F Dim unSingle2 As Single = 65478.6547!
String (cadenas de longitud variable)	**System.String** (clase)	Depende de la plataforma	Una cadena de caracteres Unicode. Valores: de 0 to aproximadamente 2 billones (2^31) de caracteres Unicode.

			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.

Esta tabla muestra algunos de los mas importantes de ellos y los valores mínimos y máximos que puede contener, así como el tamaño que ocupa en memoria. Tenemos 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 variable As Short
```

Después podemos asignar el valor correspondiente:

```
variable = 15000
```

Cuando asignemos valores a las variables podemos hacer esta asignaciones dependiendo del tipo que es, es decir, según el tipo de dato de la variable puede ser necesario el uso de delimitadores para encerrar el valor que vamos a asignar:

- Tipos numéricos. Las variables no necesitan delimitadores, se asignan directamente. Si es una valor real los decimales se separan con el punto.
- Tipos alfanuméricos o cadenas de caracteres. Las variables se encierran entre comillas: "pepe", "casado"
- Fecha. Podemos encerrar la fecha con los signos #. Por ejemplo #01/01/2004#. o con comillas dobles. La diferencia (y muy importante) es que si utilizamos las almohadillas # el formato de la fecha es mes/día/año y si es la comilla el formato es día/mes/año (por el valor de la configuración regional de nuestro equipo).
- Tipos lógicos (boolean). Las variables de este tipo sólo pueden tener los valores Verdadero (True) o falso (False)

Las variables las podemos declarar en cualquier parte del código pero por norma utilizaremos lo mas lógico que es declararlas al principio de nuestras rutinas o procedimientos. También podemos asignar un valor a la variable en el momento de crearla por ejemplo:

```
Dim valor as string="mesa"

Dim edad as long="23"
```

Que sería lo mismo que hacer:

```
Dim valor as string
Dim edad as long

valor="mesa"
edad=23
```

Por regla general no haremos esa asignación en la declaración simplemente porque en muchos casos no sabremos su valor predeterminado ya que estamos declarando esas variables para realizar cálculos en el código.

2.4 Detalles sobre los tipos de datos

¡Horror, qué es esto! ¿Quiere decir que antes de escribir programas debo saber qué variables debo utilizar y además de qué tipo? Pues si, y además esto conlleva cosas tremendamente importantes en la programación:

- Nunca se debe comenzar a programar sin haber hecho un diseño de lo que queremos realizar
- Es obligatorio que se definan todas las variables que se van a utilizar y su tipo.

El segundo punto me encargaré mas adelante de explicaros. Del primero pues nada, está claro: papel y lápiz y a pensar en el programa, luego pasar a limpio la idea y los procesos o algoritmos mas importantes que vamos a realizar y por fin lo plasmaremos en un programa. Si esto se hace correctamente la declaración de variables mas que un engorro es otro paso mas en el desarrollo de nuestro programa y que nos va a asegurar que lo demás va a funcionar bien, en cuando a tipos de datos se refiere.

Es un vicio generalizado en los programadores de Visual Basic que cojan el ratón y un formulario y empiecen a dibujar el resultado de la pantalla y luego, lo último a escribir código. ¡NO! La diferencia entre un programador y un analista es que el analista primero piensa, plasma las ideas y realiza el diseño sobre el papel. El programador no debe realizar la típica acción de "prueba y ensayo", es decir escribo código y lo pruebo, si no funciona, lo corrijo, lo vuelvo a probar... y así hasta que funciona.

Esto, obviamente, os pasará al principio, pero luego lo interesante es hacer un análisis del programa para detectar los problemas y el flujo correcto que debe seguir. Finalmente escribir el código que lógicamente no funcionará a la primera pero estará bastante cerca. (Una ley de Murphy dice que "si algo funciona a la primera es que está mal hecho"). Paciencia, la programación es algo que no se debe aprender rápido, hay que asimilar, el entorno, la herramienta y nuestra capacidad, limitada al principio por el propio conocimiento del lenguaje. Si no lo conocemos no sabremos muy bien que se puede hacer con él. Pero esto es temporal, la práctica y sobre todo, el buen estilo ayudarán mucho a que el proceso de aprendizaje sea corto.

Y dicho este "paquete" sigamos con nuestras variables. Estábamos con que nos hemos asustado un poco porque tenemos un montón de tipos de datos y no sabremos muy bien cual utilizar. Atento a este supuesto que le ha pasado a todo programador en sus inicios:

Resulta que voy a realizar unas operaciones con números enteros y elijo el tipo "short" que me dice el manual que es para números enteros. Ahora mi programa opera con este valor y por razones del programa debe asignarle el valor de 87000... sigo con el programa y resulta que no funciona nada, que una pantalla sencilla me devuelve un valor imposible. Pero... ¡si sólo he hecho una multiplicación!. Bien, el problema está en que el tipo de datos elegido para la variable ha sido el "short" (equivalente al Int de VB6) y en este tipo de datos sólo le caben números hasta el valor 32767. ¡Uf! para encontrar este fallo me he tirado horas. Para resolverlo debo utilizar un tipo de datos que admita números mas grandes (Long o Integer).

Como disponemos de mucha memoria en nuestros equipos y no queremos ser rácanos vamos a utilizar siempre los tipos de datos mas grandes para que no se nos den estos problemas, total, gastar 2 bytes mas cuando tengo 1.000 millones (1 GB de memoria) no es mucho ¿no? . Mas adelante, cuando tengamos mas práctica escogeremos el tipo exacto... Veamos entonces que tipo de datos son los que te recomiendo que debemos utilizar en nuestro aprendizaje:

- INT/LONG, para valores numéricos enteros
- DOUBLE, para valores reales o con decimales
- DATE, para valores de fecha
- BOOLEAN, para valores boleanos, es decir los que pueden ser sólo cierto/falso (true/false). Por

ejemplo: "Servicio militar cumplido: True"
- STRING, para cadenas de caracteres.

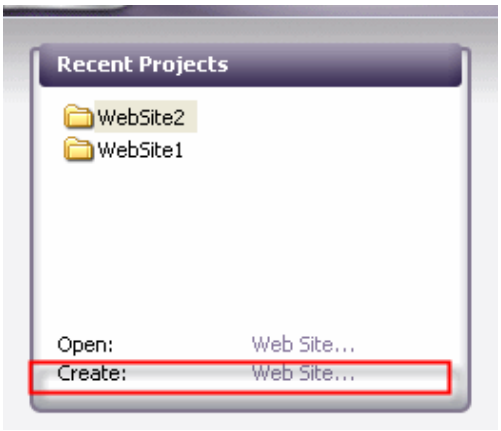
Importante Recuerda esto último, sólo utiliza estos tipos de datos, los demás no se utilizan o no nos sirven para nuestras páginas

Con estos 5 tipos de datos tendré prácticamente resuelto todo el tema de las variables en cuanto al tipo de datos que debo utilizar se refiere. Ya los ampliaremos a lo largo del curso, ahora vamos a ver si declaramos o no las variables y cómo se hace...

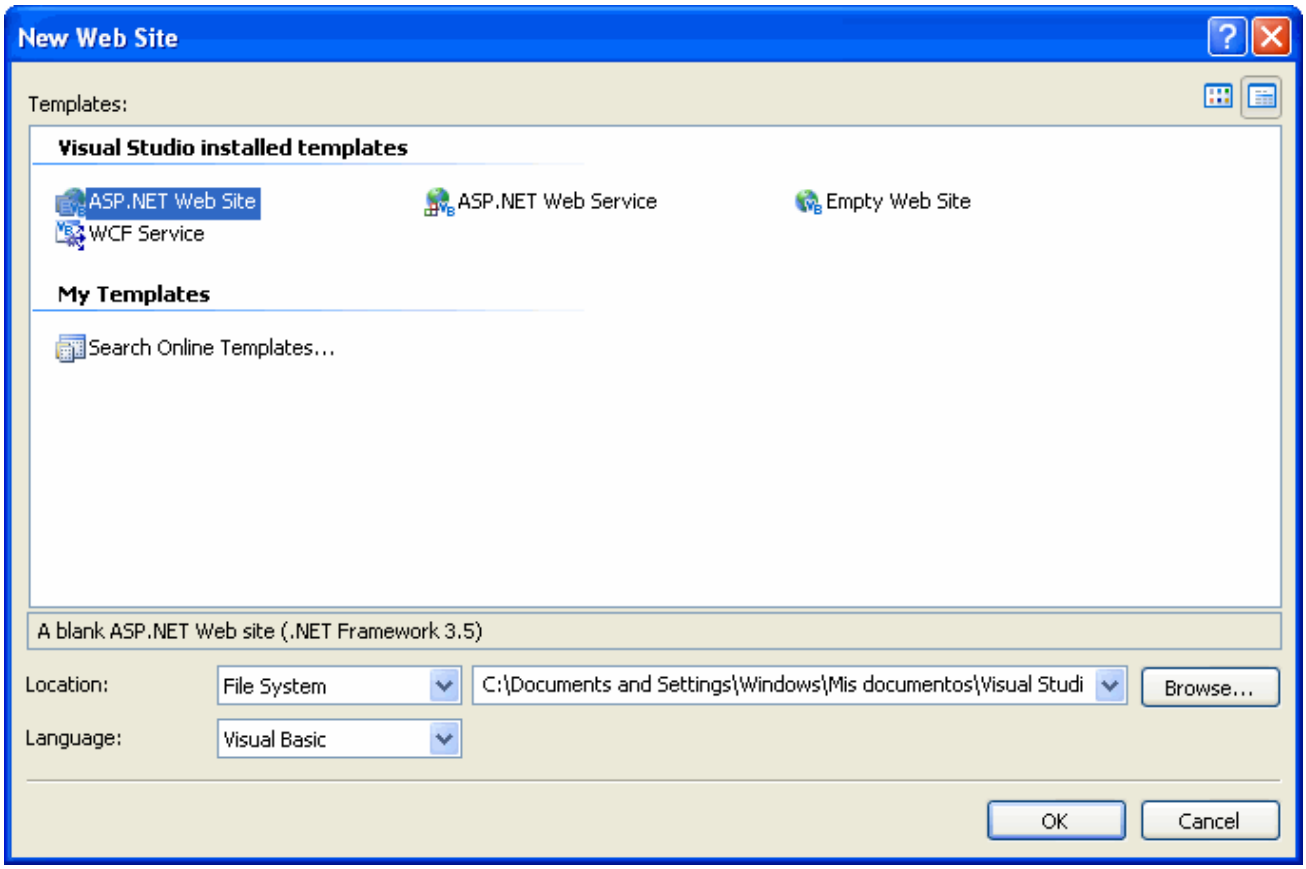
2.5 Ejemplo.

Preparar un entorno de pruebas antes de hacer ejemplos

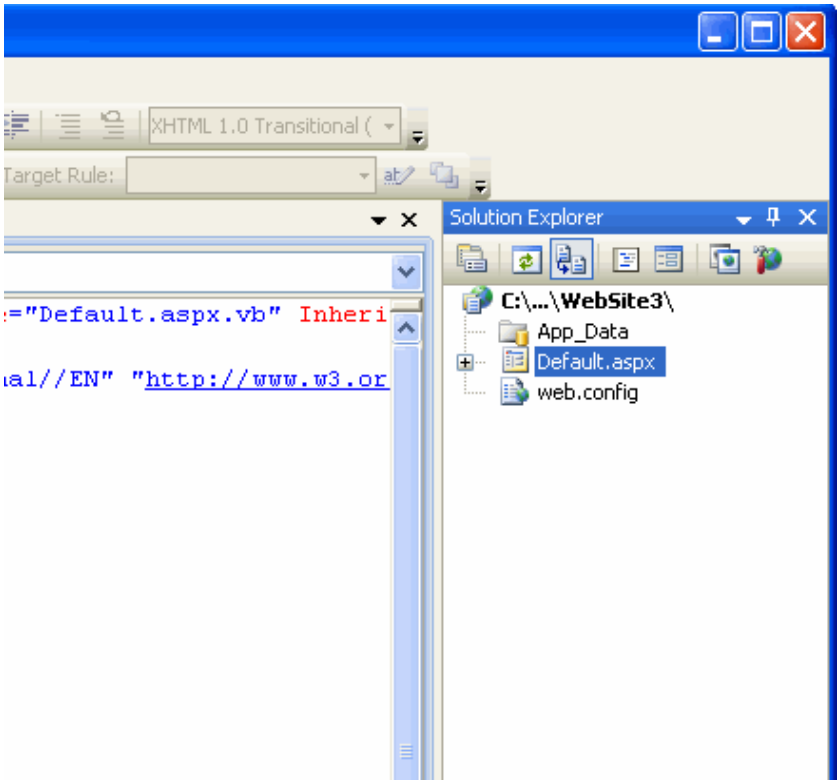
Antes de seguir vamos a realizar una página de ejemplo para ver la salida por pantalla. Luego seguiremos declarando... tranquilo nos cansaremos de declarar variables... Vamos a crear una página de ejemplo en nuestro web developer. Vamos a configurarlo rápido para que podamos comenzar a trabar. No te preocupes si no conocemos todavía el entorno, esto lo veremos mas adelante. Ahora quiero que practiquemos con variables. Así que pon en marcha el Visual Web Developer 2008 y le decimos "Crear nuevo Web":



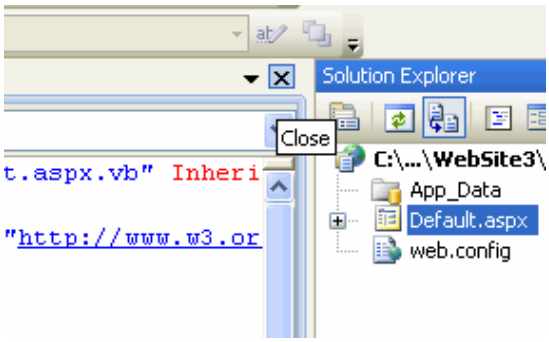
Le indicamos la ruta predeterminada:



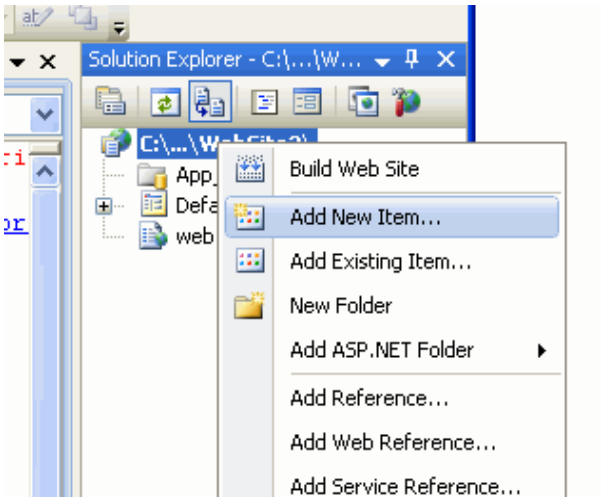
Con las opciones que aparecen debajo de "Location" y "Language" que ves. Una vez terminado tendremos un pantalla como esta. Fíjate a la derecha arriba como tendremos nuestro explorador de proyectos, es decir, las páginas web y otros ficheros que componen nuestra aplicación Web:



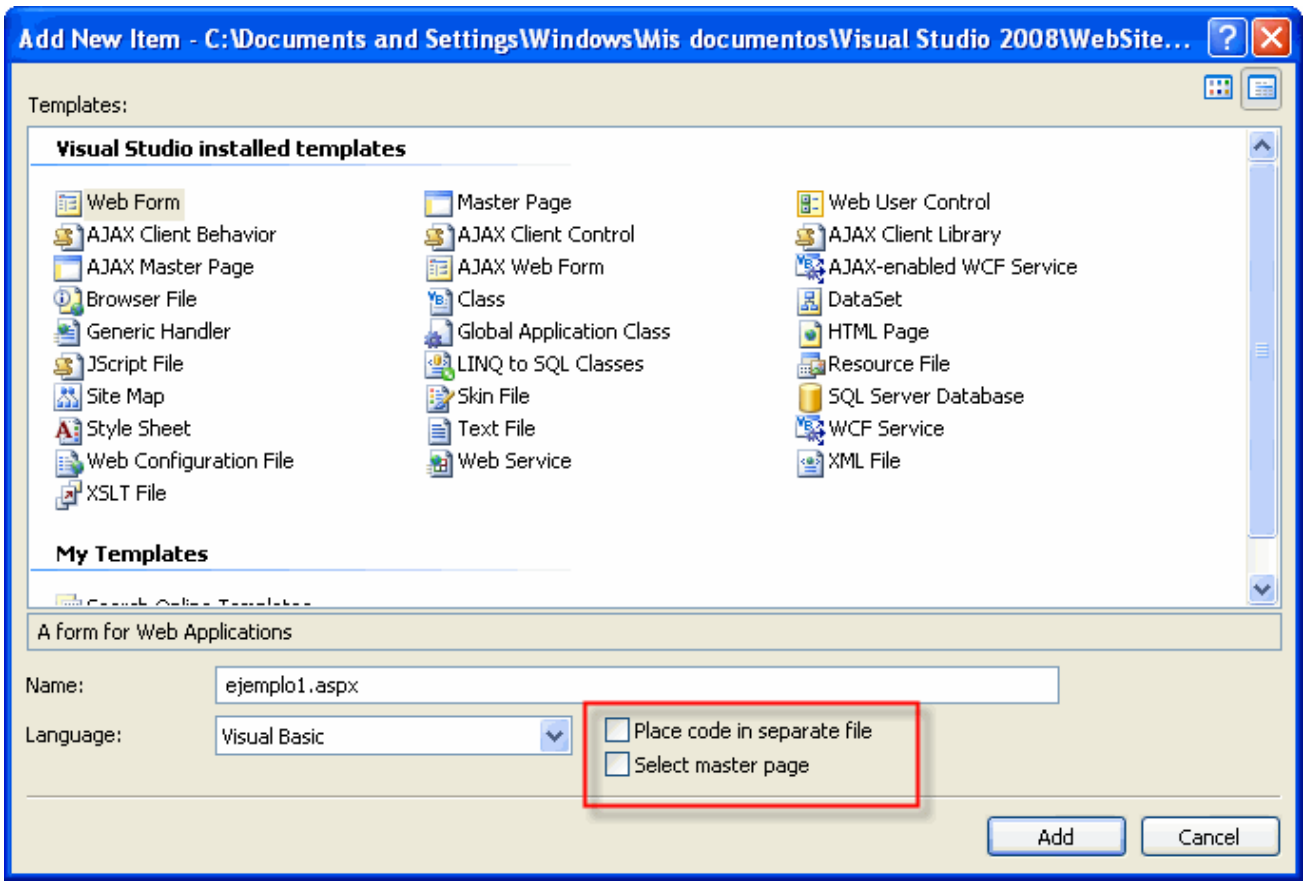
Por defecto nos aparece en pantalla el código de una página predeterminada que este entorno crea siempre y que se llama "default.aspx". La cerraremos pulsando en la "x" del título:



Y cerraremos también la de bienvenida para quedarnos con la pantalla vacía. Ahora pulsaremos con el botón derecho en el título de nuestro web en el explorador de proyectos:



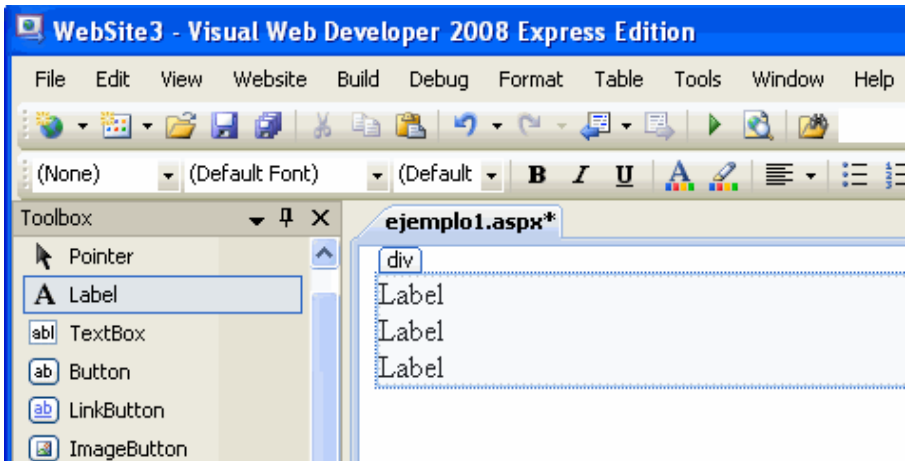
Para indicarle que queremos añadir un nuevo elemento, que será una página web para realizar nuestros ejemplos. Fíjate en esta pantalla:



Le indicaremos que es un "Web Form", es decir un formulario web que son las páginas estándar de ASP.NET y en las opciones que ves abajo a la derecha deben estar desmarcadas las dos. Mas adelante ya veremos estas opciones, de momento nos van a facilitar la labor de escribir estos sencillos ejemplos. Abre el ejemplo1.aspx y veremos en el editor lo siguiente:

```
1  <%@ Page Language="VB" %>
2
3  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http
4
5  <script runat="server">
6
7  </script>
8
9  <html xmlns="http://www.w3.org/1999/xhtml">
10 <head runat="server">
11     <title>Untitled Page</title>
12 </head>
13 <body>
14     <form id="form1" runat="server">
15     <div>
16
17     </div>
18     </form>
19 </body>
20 </html>
```

Debajo verás pestañas en las que podemos cambiar de la vista de código a la de diseño. Pulsa en la de diseño para que tengas la ventana y dentro del recuadro que hay arriba arrastra tres controles de tipo "label" que tienes a la izquierda:



Estos son componentes de ASP.NET que nos van a ayudar a ejecutar nuestros ejemplos. De momento la idea es que los hagamos funcionar, la explicación detallada la veremos mas adelante. Abre ahora la vista de diseño:

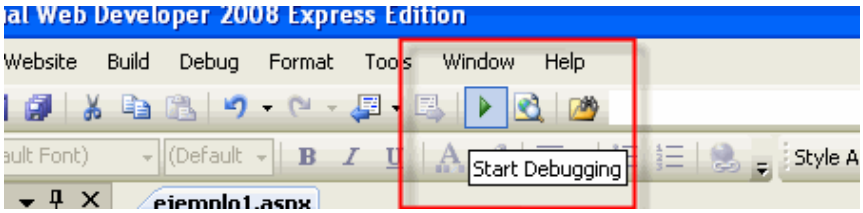
```
<%@ Page Language="VB" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.c

<script runat="server">

</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

        <asp:Label ID="Label1" runat="server" Text="Label"></asp:Label>
        <br />
        <asp:Label ID="Label2" runat="server" Text="Label"></asp:Label>
        <br />
        <asp:Label ID="Label3" runat="server" Text="Label"></asp:Label>

    </div>
    </form>
</body>
</html>

```

Fíjate en la parte de abajo como ha escrito estos tres controles de ASP.NET de tipo label. Son simplemente literales en los que podré escribir unos datos en pantalla. Es lo que utilizaremos en estos ejemplos. Ahora otro detalle que te cuento y en otro capítulo explicaremos. Cada vez que se carga una página se produce un evento especial de carga de página. Técnicamente quiero decir que cada vez que se carga una página se ejecuta un procedimiento que se llama "Page_Load" donde podremos escribir código que queremos ejecutar antes de mostrar la página al usuario. Por ejemplo vamos a escribir en las etiquetas del ejemplo, para esto le asignaremos a la propiedad "Text" de estas etiquetas un texto de prueba. ¿Como se llaman estas etiquetas? Pues si te fijas en el código tienen un valor que empieza por "ID" que las identifica: <asp:Label ID="Label1"....

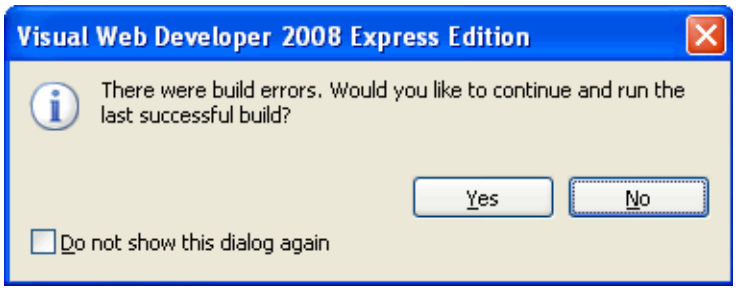
Por tanto para asignarles un texto pondremos Label1.text="Hola" y esto hará que en pantalla ese control de etiqueta tome ese valor y lo escriba en pantalla. Escribe entonces este código justo donde está puesto y con todo el código que ves:

```
1  <%@ Page Language="VB" %>
2
3  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.or
4
5  <script language="VB" runat="server">
6          Sub Page_Load()
7           Label1.Text = "Hola, este es mi primer mensaje en una etiqueta"
8           Label2.Text = "Este sería otro control Label"
9           Label3.Text = "Y este otro"
10         End Sub
11     </script>
12
13 <html xmlns="http://www.w3.org/1999/xhtml">
14 <head runat="server">
15     <title>Untitled Page</title>
16 </head>
```

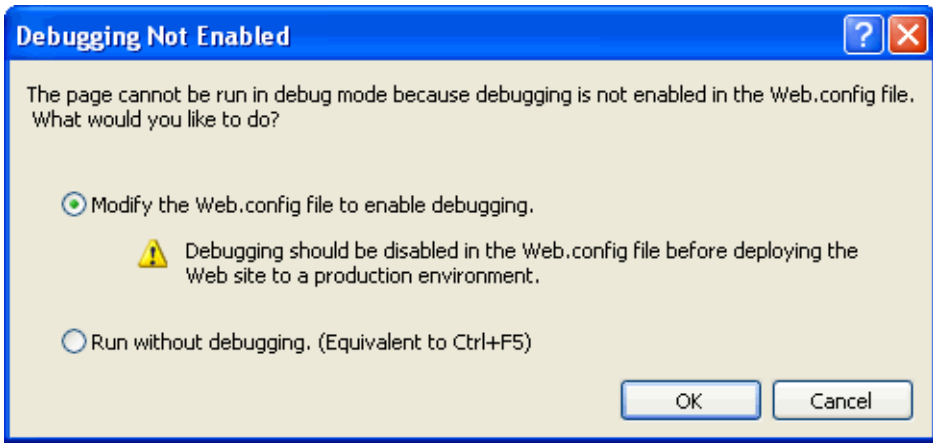
Solo he puesto la parte de arriba de la página, la parte de abajo es igual. Pulsaremos ahora en el botón "Depuración" o ·"Debug" para ejecutar nuestra página:



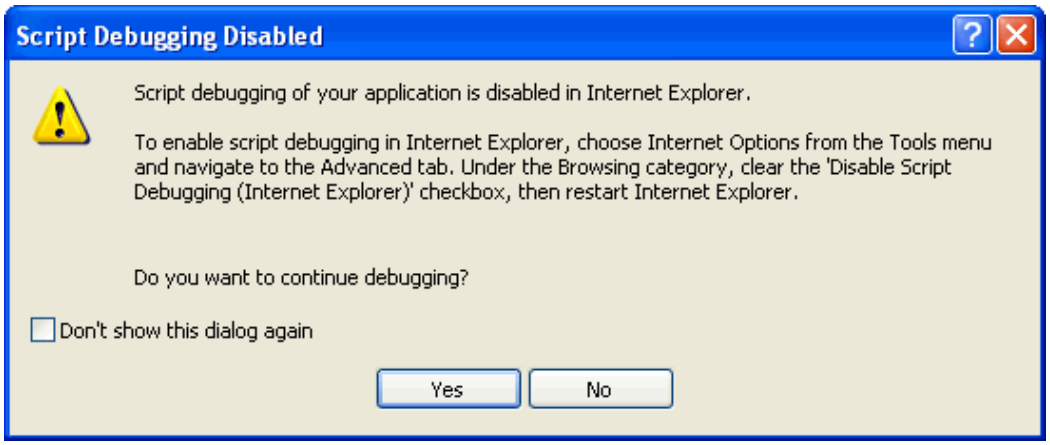
Si no hemos tenido ningún error se ejecutará la página, pero si lo hemos tenido nos lo avisará con un mensaje:



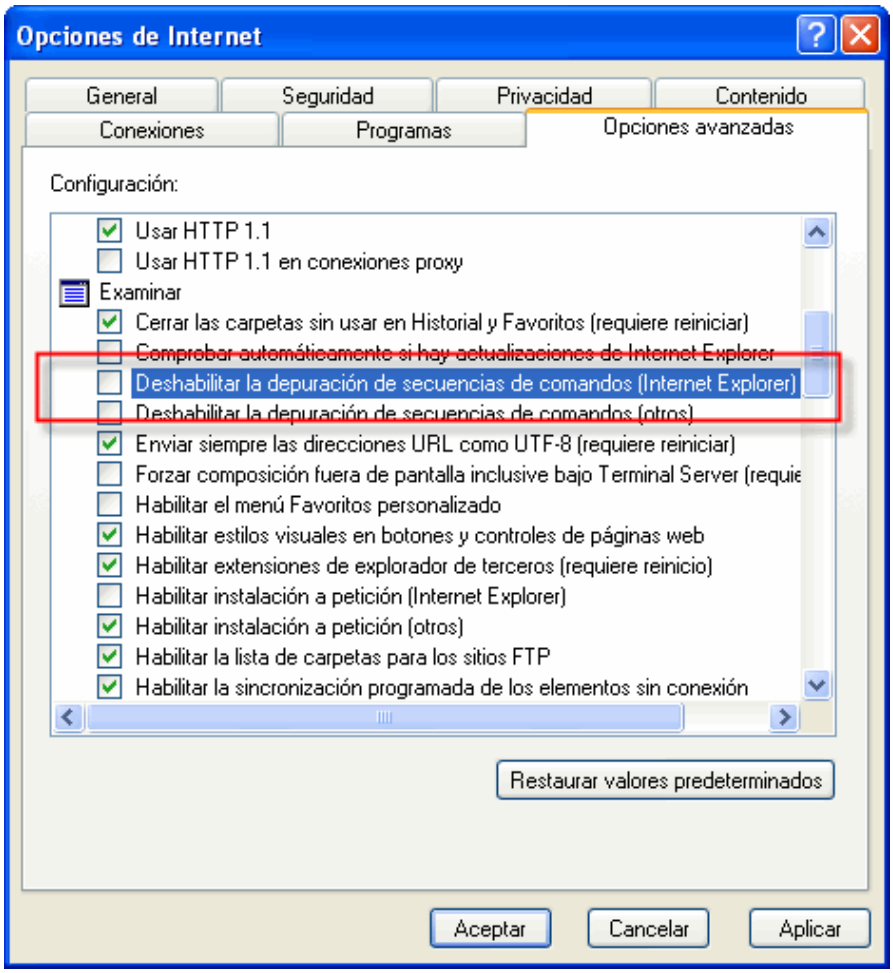
Nos dice que tenemos errores y si queremos continuar, le diremos que no porque vamos a corregirlo. Ahora nos dice lo siguiente:



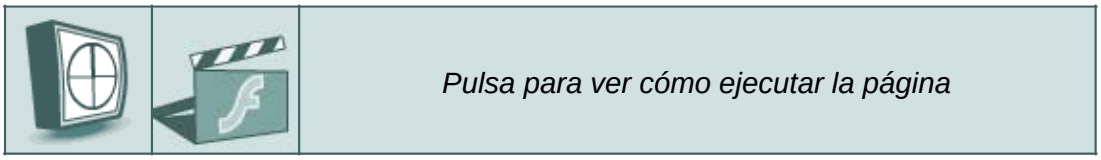
Que es un mensaje para que activemos la depuración ya que por defecto está desactivada. Ahora ya debería estar todo pero no, falta un último detalle... por defecto Internet Explorer no muestra datos sobre la ejecución de páginas ni depuraciones, ya que esto es solo para los desarrolladores, así que me avisa que está desactivada:



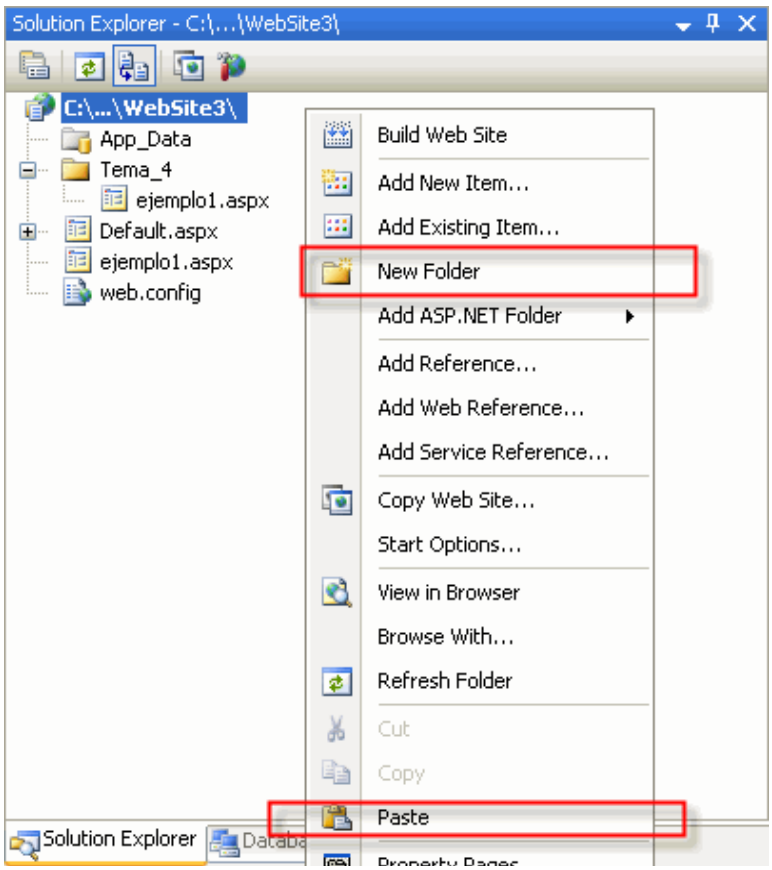
Y que tendremos que irnos al Internet Explorer y en su página de propiedades desactivar esta opción:



Y por fin! todo está listo. Como te digo, todo esto lo veremos mas adelante, ahora es para ponernos a practicar realizando sencillos ejemplos de variables. Le decimos OK y nos mostrará una lista con los errores que hemos realizado en la página. SI todo ha ido correcto nos mostrará el resultado:



Te recomiendo que hagas todos los ejemplos para practicar aunque te voy a proporcionar todos los que vayamos haciendo. Para descargarte este ejemplo pulsa aquí. Los pongo comprimidos para que no den problemas al tener extensión .aspx, ya que son ficheros especiales. Los descomprimes en tu equipo y luego en el explorador de soluciones le haces un copiar, pegar. Así lo incorporas a tu sitio web de ejemplo. Te recomiendo que crees una carpeta con el número del tema:



Con el botón derecho tendrás todas estas opciones. Así las tendrás mas organizadas. Como ves he creado una carpeta llamada "Tema_4" y he "copiado" y "pegado" el fichero de ejemplo.

Ya tenemos todo listo así que a empezar con nuestros ejemplos de verdad.

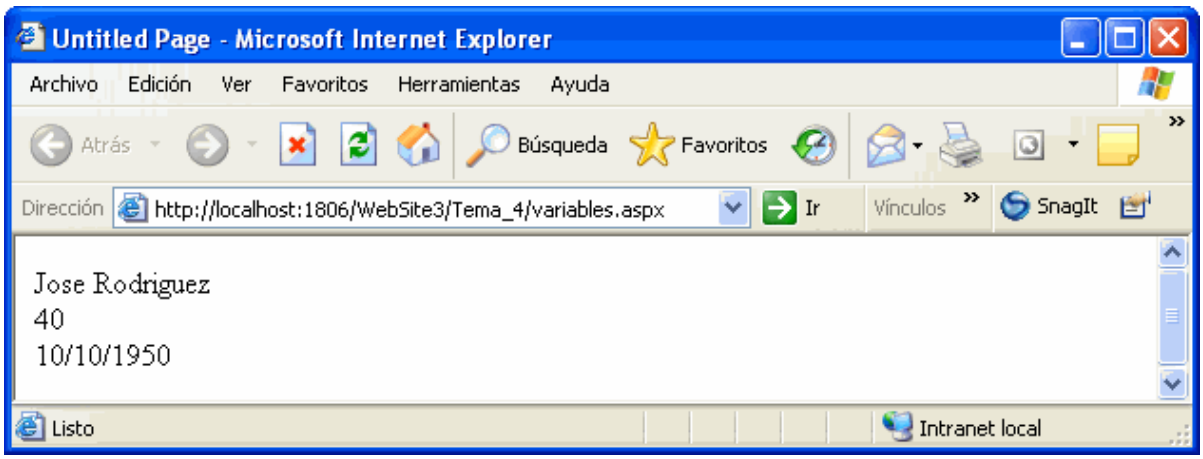
Ejemplo

1. Crea una nueva página que se llame "variables.aspx" similar al anterior con tres controles label (puedes copiar y pegar el anterior y luego renombrarlo) y en la parte de la carga de la página ponemos:

```
<script language="VB" runat="server">
    Sub Page_Load()
        'Declaración de variables
        Dim nombre As String
        Dim edad As Integer
        Dim fecha As Date

        'Asignamos los datos:
        nombre = "Jose Rodriguez"
        edad = 40
        fecha = #10/10/1950#

        'Asigamos los valores a los controles label
        Label1.Text = nombre
        Label2.Text = edad
        Label3.Text = fecha
    End Sub
   </script>
```

Es decir, declaramos tres variables de tres tipos distintos, le asignamos unos valores y se lo asignamos a los controles label.

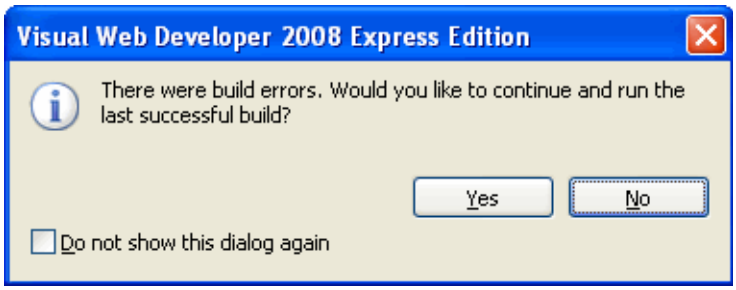
2. Ejecutamos la página:



3. Ahora vamos al código y añadimos esta asignación:

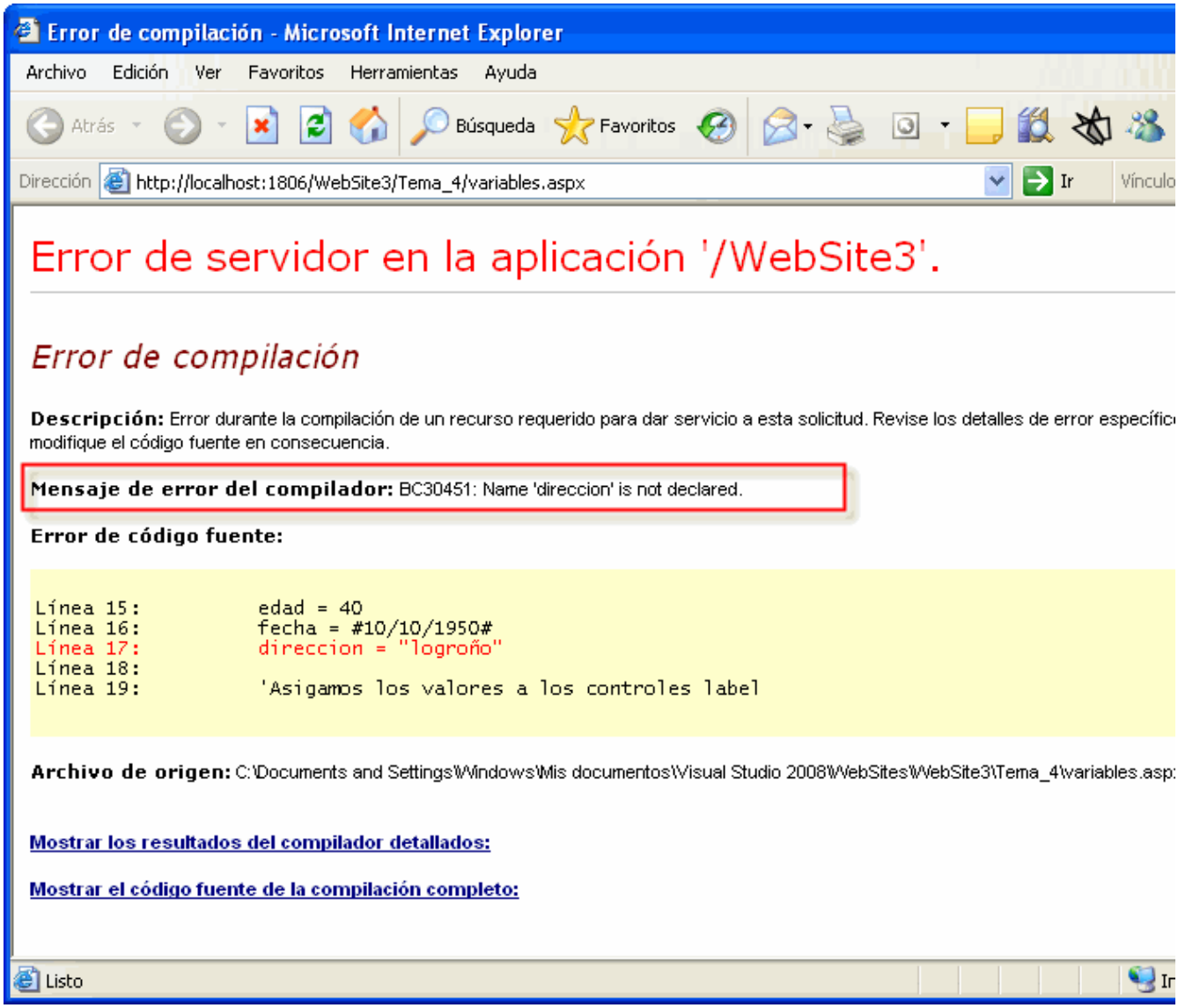
```
'Asignamos datos:
nombre="Jose Rodriguez"
edad=40
fecha=#10/10/1950#
dirección="Logroño"
```

Es decir añadimos la variable "dirección" y le ponemos un valor. Ahora intentamos ejecutar la página y:



Nos avisa de que tenemos un error. Aun así le decimos que la queremos ejecutar y pulsamos en "si":

4. El resultado es un error:



a partir de ahora prestaremos mas atención a estas páginas de error. Son muy descriptivas y nos proporcionan información completa de lo que ha sucedido en la página. Por un lado ya nos está diciendo que:

Mensaje de error del compilador: BC30451:Name 'dirección' is not

declared.

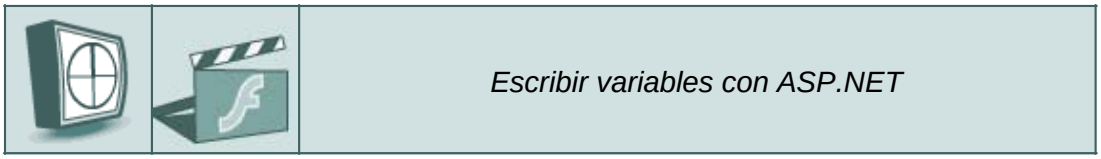
Así que ya sabemos cual es el error falta saber el origen. Si miras mas abajo:

```
Línea 10:    edad=40
Línea 11:    fecha=#10/10/1950#
Línea 12:    dirección="Logroño"
Línea 13:
Línea 14:    'Asignamos sus valores a los controles label
```

Perfecto, tenemos en amarillo nuestro código y nos ha destacado en rojo dónde tenemos el problema. La verdad es que esto ha mejorado muchísimo, la depuración en web ahora es un proceso verdaderamente fácil por fin. Además cuando estábamos editando el código y justo antes de darle a ejecutar, nuestro editor ya nos avisaba de algo, ya que siempre que alguna palabra quede subrayada debemos poner el cursor encima para ver que está pasando:

```
'Asignamos los datos:
nombre = "Jose Rodriguez"
edad = 40
Name 'direccion' is not declared.
direccion = "logroño"
```

Así que ya sabemos que no la hemos declarado y debemos hacerlo.



Los ejemplos para descargar los pongo todos al final de cada tema, así que si lo quieres ahora vete al final del este capítulo que tendrás el enlace...

2.6 Declarar variables

Ya estamos preparados entonces para declarar variables, éstas se pueden declarar de dos formas, aunque básicamente es lo mismo:

1. Declarar la variable y dejar que VB asigne el valor por defecto
2. Declarar la variable y asignarle 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 programadores de VB), que la forma mostrada en el punto 1. es la forma clásica, (la única que se puede usar en las versiones anteriores de VB). Y es la que mas veremos en los ejemplos de código realizado por nosotros y por terceros.

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
```

¿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.

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

```
Const s As String = "Hola"
Const euro As Single = 166,386
Const pi As Single = 3,1416
```

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.

Otra definición puede ser por ejemplo definir una constante que se llame "pi" y asignarle el valor "3.1416", de esta forma en el código haré referencia siempre al valor "pi" y el programa lo entenderá bien y además será mas legible.

Podemos usar cualquier constante o variable en las expresiones, 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 a 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. Es decir lo leemos de derecha a izquierda: calcula i + 15 y el resultado se lo asignas a 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.

Aunque parezca que esta sintaxis no simplifica la escritura de código (i+=15) por mi parte prefiero la tradicional y mas directa de i=i+15 por razones de claridad en nuestros ejemplos. De todas formas los que vengáis del lenguaje C que ya admitía esta sintaxis os vendrá bien para no cambiar de estilo.

Sigamos... no vamos a profundizar en lo anterior 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")

Es decir la función VAL nos asigna a una variable numérica el resultado de convertir una cadena de caracteres. Por ejemplo, cuando trabajemos con nuestra interfaz crearemos muchos cuadros de texto donde escribiremos los datos para interactuar con el programa. Bueno, pues los datos que recogen estos cuadros de texto se comportan como cadenas de texto, así que si un campo es por ejemplo, la edad, luego tendremos que convertir ese valor a numérico:

```
edad=VAL(cuadro_de_texto.texto)
```

Así en la variable "edad" que declaramos como de tipo entero tendrá el valor adecuado... ¿se entiende mejor? Así que en general procuraremos asignar a las variables los valores con el formato adecuado. Vemos unos ejemplos de la función VAL:

```
edad=VAL ("23")        --> Devuelve el valor numérico 23
edad=VAL ("hola")      --> Devuelve el valor numérico 0
edad=VAL ("23jose")   --> Devuelve el valor numérico 23
```

Es decir convierte las letras en números hasta que se encuentra algo que no es numérico. En el ejemplo que teníamos un poco mas arriba ("10 * 25") al estar entre comillas entiende que es una cadena de caracteres y comienza a convertirlo... cuando llega al "*" ve que no es un número y detiene la conversión, de ahí que nos devuelva sólo el valor numérico 10.

Detalles de las asignaciones y tipos de datos

Recuerda que hay que tener cuidado con definir el tipo de datos de las variables, aunque con el tiempo esto se convertirá en una tarea mecánica, verás... Pero pueden parte cosas como esta.

Sin querer definimos dos variables como string:

```
dim valor1 as string

dim valor2 as string
```

Durante el programa le asignamos unos valores:

```
dato1="20"

dato2="40"
```

Y realizamos una sencilla operación con ellos:

```
resultado=dato1+dato2
```

El resultado no es el valor 60 si no que es la concatenación de las dos variables "20" + "40" = "2040". Las variables de tipo string siempre son entre comillas.

Para los tipos fecha recuerda que si utilizamos la asignación con los caracteres # el formato interno es #mes-dia-año# que es el americano, sino utilizará el que tengamos en nuestra configuración regional. Por lo tanto:

```
#10/01/2005# Es el 1 de Octubre de 2005 y no el 10 de enero.
```

Las variables de tipo boolean (true-false) se utilizan mas de lo que supones así que tenlas presentes porque facilitan la lectura del código siempre que se puedan utilizar...

2.7 Convenciones de nombre

La guía de buen estilo de los programadores intenta marcar unas pautas en cuanto a los nombres que se deben poner a las variables y otras partes de los programas. Esto que aparece a continuación es una recomendación

que intentaremos seguir a lo largo del curso. La ventaja es que el tipo de datos se lo decimos en el nombre de la variable, así cuando leamos el código podemos saber de que tipo son cada una de ellas sin tener que acudir a la parte donde se declararon.

Tipo de datos	Prefijo	Ejemplo
Boolean	bln	blnestado
Date	dat	dathoy
Double	dbl	dblnomina
Integer	int	intedad
Long	lng	lnglargo
String	str	strnombre

Como normalmente tenemos poco tiempo para escribir código no solemos detenernos para seguir una guía de estilo pero te aconsejo que lo intentes. Imagina que un día te toca retocar el código de otra persona y ha seguido un mínimo las normas de estilo... seguramente inviertas la mitad del tiempo en resolver el problema...

3. Operadores y comparadores

Una vez que ya conocemos las variables veamos que operadores tenemos para trabajar con estos datos. Seguro que ya conoces muchos de ellos pero los recordamos en esta tabla:

Operador	Símbolo	Ejemplo
Suma	+	4 + 3
Resta	-	4 - 3
División real	/	56 / 45. Da como resultado un decimal, single o double
División entera	\	56 \ 45. Da como resultado un entero
Multiplicación	*	12 * 34
Potencia	^	2 ^ 3 | (-4) ^2
Resto	**Mod**	resultado=10 mod 3. Devuelve el resto de la división, en este caso resultado=1
Concatenar	**& +**	resultado="Jose" & " Angel". Unen dos cadena para crear una: "Jose Angel"

3.1 Comparación

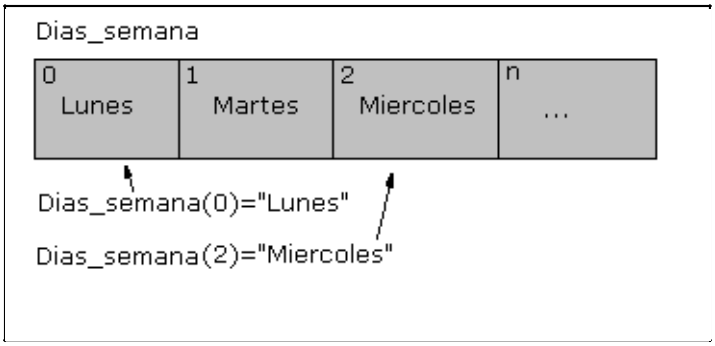
Estos operadores permiten compara dos operandos, el resultado será un valor boolean (lógico): True (verdadero) ó False (falso).

Comparador	Devuelve True	Devuelve False
< Menor que	A<B	A>=B

> Mayor que	A>B	A<=B
<= Menor o igual que	A<=B	A>B
>= Mayor o igual que	A>=B	A<B
= Igual a	A=B	A<>B
<> Distinto de	A<>B	A=B

4. Matrices

Las matrices o arrays son una lista de valores asociados a un sólo identificador. Es decir podemos asignarle a la variable "dias_semana" siete valores alfanuméricos o de cadenas de caracteres. Para acceder a cada uno de ellos se lo indicaremos por un índice. Las matrices pueden ser de mas de una dimensión, cuando son de una sola dimensión se le suele llamar también vector. Mira este gráfico:



Es decir, vamos a definir una matriz o un array de una dimensión que se va a llamar "dias_semana" y luego le voy a asignar los valores que quiera. Una vez creada para acceder a cada uno de sus elementos se lo indicaré por un índice entre paréntesis. Como detalle saber que las matrices comienzan por el elemento 0. Es decir si quiero que tenga 5 elementos la declararé con el valor 4: elementos 0,1,2,3 y 4. Vamos a declarar una matriz:

```
dim dias_semana (6) as string
```

Esta declaración define una matriz de 7 elementos de tipo string.

¡Atención! Las matrices son colecciones del mismo tipo de datos, es decir, INT,LONG, STRING, ... pero siempre del mismo tipo. Si queremos que sean diferentes debemos declarar el array como de tipo Object. Este tipo es el genérico de VS.NET y permite almacenar distintos tipos pero lo veremos mas adelante.

Veamos ahora cómo asignamos los valores al array:

```
dias_semana(0)="Lunes"
dias_semana(1)="Martes"
dias_semana(2)="Miercoles"
dias_semana(3)...
```

Lógicamente recibiremos un error si queremos asignar el valor octavo: dias_semana (7) ya que definimos la matriz con 7 elementos.

Para definir una matriz de mas de una dimensión será muy parecido, simplemente le indicamos el número de elementos de cada dimensión:

```
dim mi_matriz(4,5) as long
dim otra_matriz(6,6,6) as string
```

En el primer caso la matriz es de 30 elementos (0 al 4 y 0 al 5) y para referirnos a un elemento en concreto le tendremos que indicar fila,columna. En el segundo caso es de tres dimensiones y tiene hasta 7*7*7 elementos, vemos cómo tendremos que referirnos a sus elementos:

```
mi_matriz (2,2)=345
mi_matriz(0,3)=12
otra_matriz(0,0,0)="primer elemento"
tra_metriz(2,4,4)="otro elemento"
```

Nota Las matrices pertenecen a la jerarquía de clases del sistema, es decir, está incluida en el espacio de nombre System. Por lo tanto podremos utilizar las matrices con toda la capacidad de la programación orientada a objeto ya que las matrices son objetos. De todas formas seguiremos con la notación tradicional y no utilizaremos la POO con matrices de momento.

No me acuerdo hasta cuantas dimensiones permite VS.NET pero te puedo asegurar que muchas mas de las que necesitaremos en nuestros programas.

¿Defectos? Pues si, ya lo habréis pensado: tenemos que decirle el número de elementos que debe tener nuestra matriz (array)... pero tranquilo esto tiene solución VB.NET permite modificar el tamaño de las matrices (arrays) en tiempo de ejecución, es decir mientras nuestro programa se está ejecutando. Para esto tengo dos instrucciones: REDIM y REDIM PRESERVE. La primera redimensiona una matriz (array), cambia su tamaño al indicado en el parámetro pero borra su contenido. La segunda instrucción le cambia su tamaño pero conservando los valores actuales, esta opción es la mas utilizada porque normalmente querremos asignar un nuevo valor y al ver que hemos llegado al último elemento ampliaremos en mas elementos pero conservando lógicamente los actuales. Mira estas instrucciones:

- redim matriz1(10). Cambia de tamaño a matriz1 pasando a tener 11 elementos y borrando su contenido.
- redim preserve matriz2 (19). Cambia el tamaño de matgriz2 a 20 elementos pero conservando los elementos anteriores.

4.1 Formas de declarar una matriz

Además de la más básica vista arriba vamos a ver mas formas de declarar una matriz:

1. Estableciendo el número de elementos:

```
dim dias_semana (6) as string
```

2. Indicando el tipo de datos pero no el número de elementos

```
dim numeros () as longh
```

3. Iniciando los valores

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:

```
a()   mInteger = {1, 42, 15, 90, 2}
```

También podemos hacerlo de esta otra forma:

```
a AsDInteger() = {1, 42, 15, 90, 2}
```

4.2 Recorrer una matriz. Utilizar bucles For Next y For Each para recorrer los elementos de un array

Mas adelante veremos las estructuras de flujo de los programas, como avance veamos dos tipos de bucles que nos ayudarán a recorrer las variables.

El tipo For Next es muy sencillo de utilizar y se adecuado para las matrices porque sabemos cuantos elementos tenemos que es fundamental para un bucle de este tipo.

```
Dim a() As Integer = {1, 42, 15, 90, 2}
'
Escribe ("Elementos del array a()= {0}", a.Length)
'
Dim i As Integer
For i=0 to 4
    Escribe (i & "<br>")
Next
```

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}

'

Escribe ("Elementos del array a() " & a.Length)

'

Dim i As Integer

For Each i In a

    Escribe(i)

Next

'
```

Me he adelantado un poco porque los bucles los veremos en el siguiente capítulo, pero te puedes hacer una idea de cómo funcionan... ahora vamos a hacer un pequeño ejemplo de matrices

4.3 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 clasifica. 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

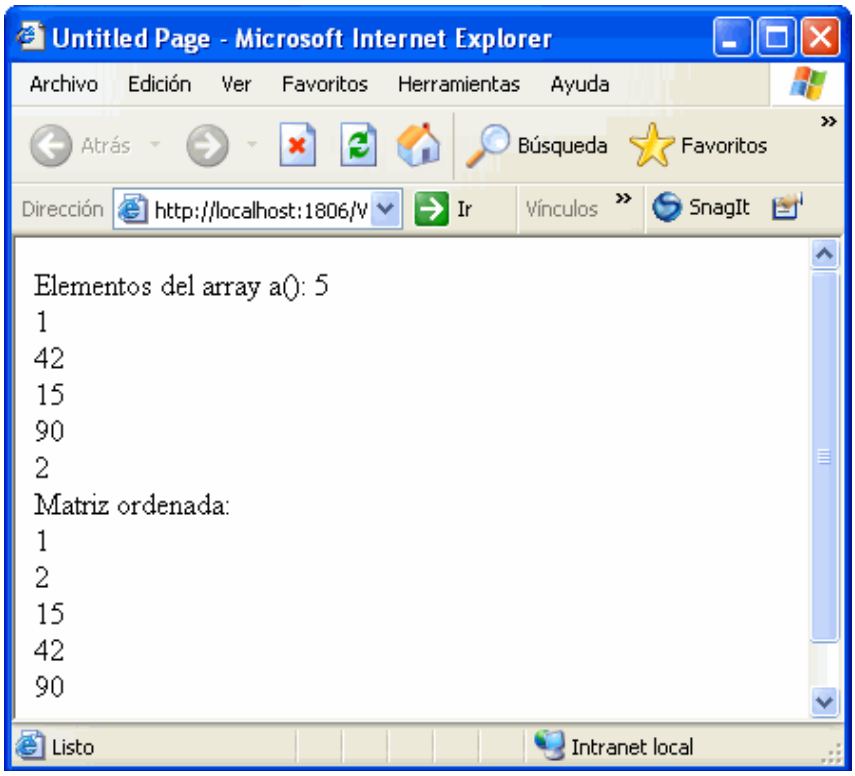
Nota Para nuestros ejemplos vamos a utilizar una instrucción nueva llamada "response.write" que nos escribirá en el navegador lo que le indiquemos ahí. Si en nuestro "Page_Load" le ponemos un "response.write ("Hola", le mandará ese texto al navegador. Con esto nos evitamos utilizar de momento los controles label ya que lo escribimos directamente. Veámoslo en este ejemplo...

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. Crea una página nueva y le pones este código en la parte inicial

```
Sub Page_Load()
    Dim a() As Integer = {1, 42, 15, 90, 2}
    'Escribimos en pantalla el número de elementos
    Response.write ("Elementos del array a(): " & a.Length & "<br>")
    '
    Dim i As Integer
    For Each i In a
        'Escribimos cada elemento (<br> es una línea en blanco)
        Response.write(i & "<br>")
    Next
    '
    Array.Sort(a)
    Response.write("Matriz ordenada:" & "<br>")
    '
    For i = 0 To a.Length - 1
'Escribimos cada elementos de la matriz ordenada (<br> es una línea en blanco)
        Response.write(a(i) & "<br>")
```

```
        Next
        '
    End Sub
```

Resultado:



4.4 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
    '
    Response.write("m = " & m & ", n = " & 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

    Response.Write("a(i) = " & a (i) & " , b(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**.

4.5 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
```

```
        Response.Write("a(i) = " & a(i) & ", b(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.

Límites de las matrices

Para manejarnos con las matrices tenemos de momento dos instrucciones muy interesantes:

- Ubound. Devuelve el mayor subíndice disponible para la dimensión indicada de una matriz
- Lbound. Devuelve el menor subíndice disponible para la dimensión indicada de una matriz

Es decir me dice cuantos elementos puede contener la matriz porque me dice el índice mayor disponible. Luego Ubound (mimatriz) me devolverá 7 por ejemplo si es ese su número de elementos. Mira este ejemplo de Ubound para una matriz de dos dimensiones:

```
Dim A(100, 5, 4) As Byte
```

Llamada a UBound	Valor devuelto
`UBound(A, 1)`	100
`UBound(A, 2)`	5
`UBound(A, 3)`	4

5. 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. A menudo a los arrays de una sola dimensión se les llama vectores. 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...

5.1 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:

```
'Una dimensión sin indicar número de elementos
Dim a1() As Integer
'Dos dimensiones sin indicar número de elementos
Dim a2(,) As Integer
'Tres dimensiones sin indicar número de elementos
Dim a3(,,) As Integer
'Una dimensión indicando número de elementos
Dim b1(2) As Integer
'Dos dimensiones indicando número de elementos
Dim b2(1, 6) As Integer
'Cuatro dimensiones indicando número de elementos
Dim b3(3, 1, 5, 2) As Integer
'Una dimensión, sin indicar cuantos elementos y declarando
Dim c1() As Integer = {1, 2, 3, 4}
'Dos dimensiones, sin indicar cuantos elementos y declarando
Dim c2(,) As Integer = {{1, 2, 3}, {4, 5, 6}}
'Tres dimensiones, sin indicar cuantos elementos y declarando
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, tres y cuatro dimensiones.

En el último caso, he usado el continuador de líneas para que sea más fácil leer la asignación 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.

5.2 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.

Veamos cómo haríamos para averiguar cuantos elementos tiene cada una de las tres dimensiones del array **c3**:

```
Response.Write("Las dimensiones de c3 son: " & c3.GetUpperBound(0)
        & " - " & c3.GetUpperBound(1) & " - " & c3.GetUpperBound(2))
```

5.3 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:

```
Response.Write("El array c3 tiene " & c3.Rank & "dimensiones.")
```

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... al 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 esto de momento y veamos el ejemplo:

```
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)
            Response.Write("El valor de c3(" i & "," & j & "," & k & ") es:" & c3(i, j, k)
        Next
    Next
```

```
Next
```

5.4 Cambiar el tamaño de un array y mantener los elementos que tuviera.

Para poder cambiar el tamaño de un array manteniendo sus datos 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*.

Redimensionar un array multidimensional.

Acabamos de ver 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. 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 un subrayado 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 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.
```

Resumiendo, 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.

5.5 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.

5.6 ¿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.

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.

5.7 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 antes para copiar el contenido de un array, la respuesta es NO, simplemente porque CopyTo sólo se puede usar con arrays unidimensionales.

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.

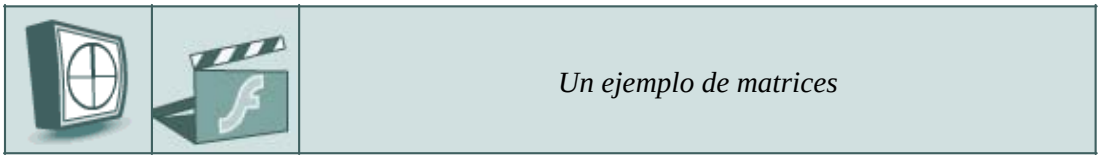
Para entenderlo mejor, veamos varios casos que se podrían dar. Usaremos el contenido del array **c3** que, como sabemos, está definido de esta forma: c3(3, 2, 1) y como destino un array llamado **c31**. Para copiar los elementos haremos algo así:

```
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.

Y con esto finalizamos una buena introducción a las matrices, seguramente no necesitemos saber mas que todo esto en mucho tiempo, así que tómalo como referencia.



Un ejemplo de matrices

5.8 El Arraylist

Los "arraylist" son matrices dinámicas en las que no hace falta declarar el número de elementos que va a tener. Por ejemplo, observa este código:

```
'Declaramos la matriz dinámica, sin indicarle un número de elementos fijo:

Dim Lista_dinamica as New ArrayList


'Añadimos elementos a esa matriz:

Lista_dinamica.Add ("uno")

Lista_dinamica.Add ("dos")

Lista_dinamica.Add ("tres")

'Recuperamos un valor de la matriz:

Valor=CType (lista_dinamica (0), String)
```

Es muy útil porque en ocasiones tendremos que crear una matriz de elementos sin saber a priori cuantos valores tienes. Por ejemplo, leer líneas de un fichero texto, no sabemos cuantas líneas hay así que tendremos que ir leyendo el fichero hasta que se termine, en ese caso iremos creando tantos elementos con el método "Add" mientras leamos líneas del fichero.

La función CType la veremos mas adelante, estamos realizando una conversión de tipos para asegurarnos que metemos un valor de tipo "String" en la variable.

6. Estructuras

Hasta ahora hemos visto los tipos de variables que nos proporciona Visual Basic .NET, pero ¿puedo yo crear algo mas complejo para adaptarlo a mi programa? La respuesta está en las estructuras.

Las estructuras permiten crear nuestro propios tipos de datos. Una estructura contiene uno o mas miembros que pueden ser del mismo o diferente tipo de datos. Cada miembro tiene un nombre que permite referenciarlo de forma única en la estructura, es decir, no puede haber dos elementos con el mismo nombre en la estructura. En anteriores versiones de Visual Basic se definían con la palabra clave "Type", ahora utilizaremos "Structure" con esta sintaxis:

```
Structure nombre

    declaraciones

End Structure
```

En el siguiente ejemplo veremos cómo se declaraban antes en VB6 y cómo se declaran ahora.

En VB.NET	En VB 6.0
Structure Empleado Dim Numero as long Dim Nombre as String Dim Direccion as String Dim Puesto as String End Structure	Type Empleado Numero as long Nombre as String Direccion as String Puesto as String End Type

Una vez definida la estructura ya podemos utilizarla con normalidad en el programa, para referirnos a cada uno de los miembros de la estructura simplemente se lo indicaremos con: estructura.miembro:

```
Dim emp1 As Empleado
dim emp2 As Empleado

emp1.Numero=10
emp1.Nombre="Jose"
emp1.Direccion="Jorge Vigon"
emp1.Puesto="Muy Jefe"

emp2.Numero=11
emp2.Nombre="Maria"
emp2.Direccion="Club Deportivo"
emp2.Puesto="Currante"
```

¿Y podría crear una matriz de estos elementos? Pues también, sólo indicando el tipo de datos:

```
Dim matriz_empleados (10) as Empleado

matriz_empleados (0).Numero=10
matriz_empleados (0).Nombre="Jose"
matriz_empleados (0).Direccion="Jorge Vigon"
```

```
matriz_empleados (0).Puesto="Muy Jefe"
```

Es un tipo de datos muy utilizado cuando manejamos muchos datos en los formularios. Si son coherentes, es decir, fichas de empleados, datos de materiales, podemos declarar una variable de este tipo y quedará mucho mas legible y elegante el trabajo con estos datos.

7. Las enumeraciones (Enum)

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.

Las enumeraciones proporcionan una forma cómoda de trabajar con conjuntos de constantes relacionadas y de asociar valores de constantes con nombres. Por ejemplo, se puede declarar una enumeración para un conjunto de constantes de tipo entero asociadas con los días de la semana, y después utilizar los nombres de los días en el código en lugar de sus valores enteros.

Por ejemplo, si queremos tener una variable llamada color y queremos que contenga un valor numérico que haga referencia a un color en particular, así 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 (ya llegaremos a estas partes de Clases y Módulos), pero también pueden estar declarados dentro de un espacio de nombres (recuerda los spacenames), todo dependerá del alcance (o amplitud de acceso) que queramos darle.

Los valores que pueden tener los miembros de una enumeración, pueden ser cualquiera del tipo entero o real. 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
```

En este caso el 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.

Otra cosa que tendremos que tener en cuenta es que una variable declarada del tipo de una enumeración, en teoría no debería admitir ningún valor que no esté incluido en dicha enumeración. Veamos un ejemplo:

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

Aunque el valor 1, sea un valor correcto nos indicaría que no se permite la conversión implícita entre Integer y el tipo colores.

```
  unCT  e( , colores)
```

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

Es decir, primero creamos la enumeración con "Enum" y luego vamos a utilizarla. Escribimos un procedimiento cualquiera y al escribir "Dim micolor as ..." se desplegará una lista con los tipos posibles y veremos que aparece nuestro "colores" luego vamos bien. El segundo paso es utilizarlo:

```
enum colores as long
    azul
    verde
    rojo
end enum
```

```
sub miprograma

    dim micolor as colores

    micolor=colores.azul

end sub
```

Ya lo tenemos declarado ahora vamos a utilizarlo. Como "micolor" es una variable de tipo la enumeración anterior sólo me debemos seleccionar uno de los colores que he definido antes.

Nota 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.

Ahora veremos algunas funciones o métodos de las enumeraciones que nos será muy útiles... observa esta línea de código

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

Tranquilo no es tan complicada, sólo queremos saber si existe el color número 12. Supón que en el programa necesitamos un elemento de la enumeración, normalmente no tendremos problemas pero deberíamos controlar los errores para que no termine de forma anómala o simplemente no funcione bien nuestro programa.

Utilizaremos entonces las herramientas que nos proporciona VB.NET para manejar las enumeraciones. El ejemplo de antes devolverá un valor falso, ya que 12 no es miembro de la enumeración colores. En el primer caso usamos la variable del tipo colores (**miColor**) y usamos el método **GetType** para saber que tipo de datos es esa variable.

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 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 ser útiles:

GetName, indica el nombre con el que se ha declarado el miembro de la enumeración, por ejemplo esta llamada devolvería "azul":

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

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

GetNames, devuelve un array (recuerda las matrices) de tipo String con los nombres de todos los miembros de la enumeración.

```
a()   mString = System.Enum.GetNames(unColor.GetType)

i AsDInteger

i =  oTo ubound (a)

Response.Write("el valor " & i & " es: " &  a(i))

    Next
```

Curioso ejemplo, además puedes entenderlo. Por un lado definimos una matriz que no le ponemos un tamaño fijo y le asignamos en la propia declaración un valor y ese valor es la lista de la enumeración definida. Así que

hemos hecho tres cosas en una sólo línea. Luego hacemos un bucle desde el primer elemento hasta el último (o hasta a.length-1)

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 ubound(a)

    Response.Write("el valor " & i & " es: " &  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 estos que son los más interesantes, de todas formas puedes practicar con la ayuda para ver con mas detalles estos ejemplos y ver que mas posibilidades existen,

Ya que hemos comentado la el método Parse veamos algún detalle mas sobre él. 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:

```
s As String = "123"

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:

```
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. Vale 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)
```

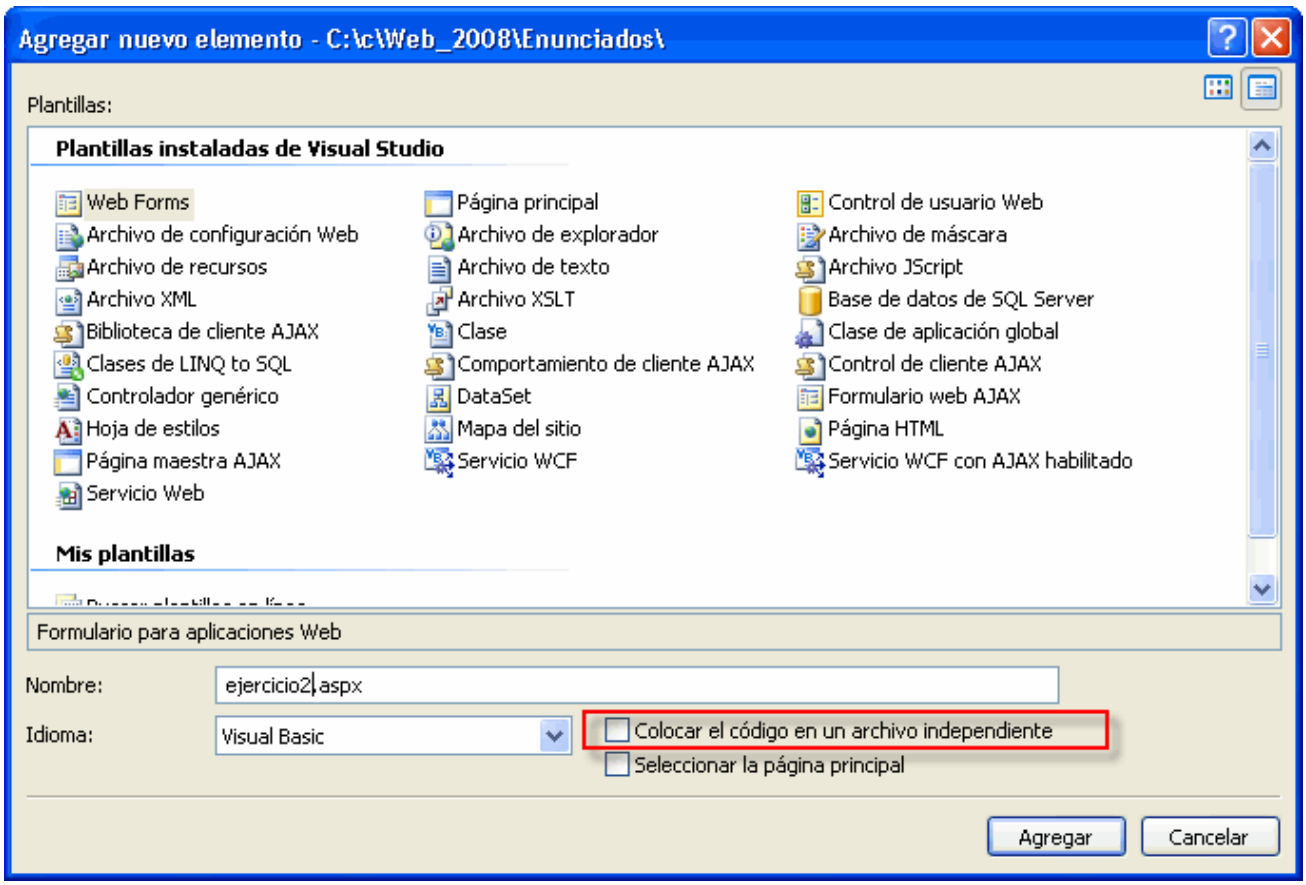
En este caso, el error se produce porque 129.33 no es un número entero válido, por tanto, cuando usemos Parse, o cualquiera de las funciones de conversión, tendremos que tener cuidado de que el valor sea el correcto para el tipo al que queramos asignar el valor...

Ya siento meteros toda esta teoría pero debéis conocer todo lo que VB.NET pone a nuestro alcance... Cuando queráis hacer páginas mas complejas necesitaréis cada vez mas estructuras mas complejas para facilitar la escritura del código, aunque las importante es la que vamos a vez a continuación: la estructuración del código mediante procedimientos y funciones

Pulsa aquí para descargar los ejemplos de este tema. Son los que hicimos antes, estos últimos fragmentos no los incluyo.

Ejercicios

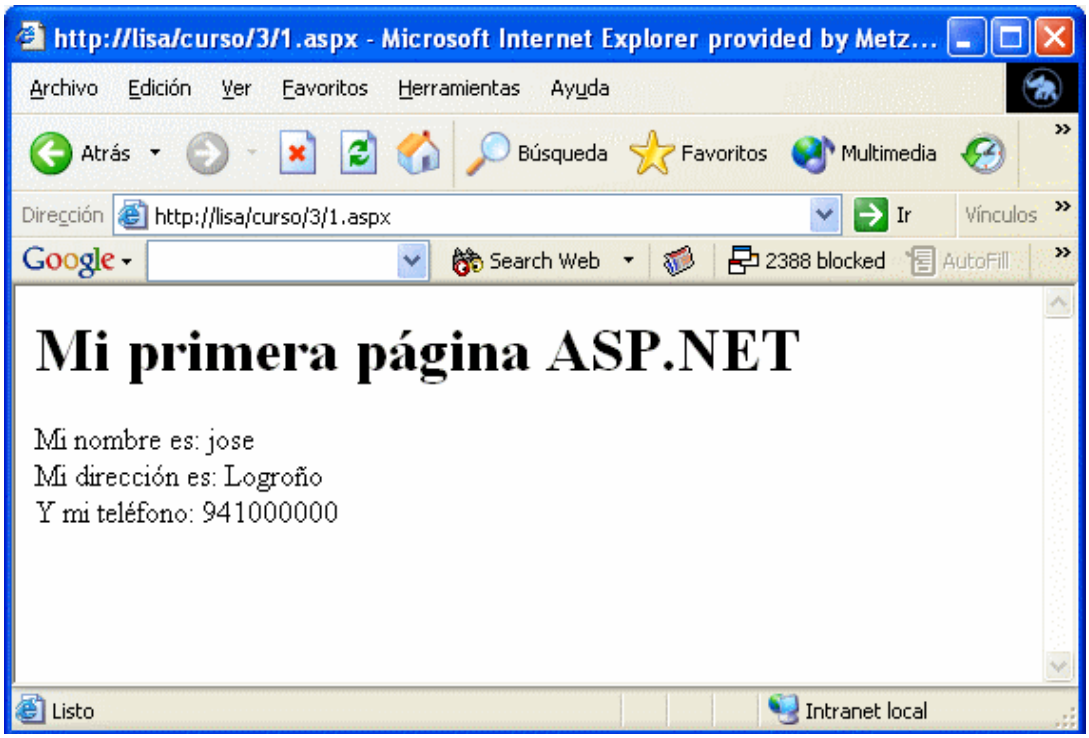
Npta: Si no lo has creado ya, crea un sitio web y dentro de él crea una carpeta llamada "tema_3" para que hagas los ejercicios dentro de esa carpeta. Todos los ejercicios los haremos desmarcando:



Ejercicio 1

Crea una página con tres etiquetas de tipo "Label" y que ponga los textos de tres variables que declares en el código: "nombre", "direccion", "telefono"

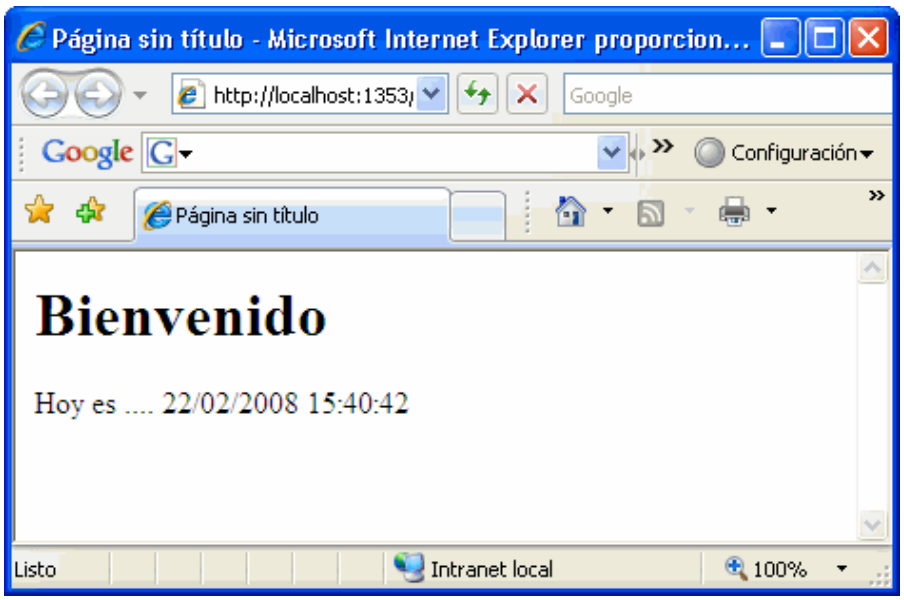
El código lo ejecutará en el evento "Load" de la página



Recuerda que la asignación del texto de las etiquetas es así --- lb_nombre.text=variable

Ejercicio 2

Haz una página que ponga el día y la hora del sistema::



EL día y hora del sistema lo obtenemos de la función Now, por tanto eso es lo que le tienes que asignar a la etiqueta de tipo label que pongas.

Ejercicio 3

Crea una página que defina una matriz de cuatro elementos de tipo string. Los asignas con valores de prueba y luego haces un bucle escribiendo el contenido de la matriz: