

Bloque 2. Visual Basic .NET  
UT 3. Confección de interfaces de usuario y creación de componentes visuales

## TEMA 4

# Variables y Flujo de programa

## 1. VB.NET: Las variables

Ya tenemos lo básico para trabajar: hemos realizado algún programa sencillo y hemos conocido el entorno de desarrollo IDE. Ahora nos tocan unos temas de aprendizaje del lenguaje de programación.

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

### 1.1. 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 "38". Sin embargo, una **constante** es un "objeto" 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 tanto de tipo numérico, de tipo fecha, cadena de caracteres, objetos,... Estos nombres son convenciones que 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 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`.

Al ser una variable podemos alterar su valor. Por ejemplo, si hacemos esta asignación: `i = 25`, el valor de la variable `i` cambiará, de forma que el valor anterior se modifica y se sustituye por el nuevo valor.

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`, que pasará a tener el valor 75.

Cuando se asignan valores de cadenas de caracteres o alfanuméricos (String) el contenido de la variable debe ponerse entre comillas `"`. Para asignar una cadena de caracteres a una variable, se haría así: `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 a 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 a VB.NET que cree una variable, indicando un nombre y el tipo de datos que va a almacenar. En VB.NET las variables podemos declararlas o no.

Existen dos posibilidades: declarar y no declarar variables. Ésta última puede ser muy cómoda al principio pero muy compleja más adelante. Las normas del buen programador obligan a declarar todas las variables que utilicemos en el programa. Esto puede dar un poco más de trabajo pero es imprescindible para escribir un buen código.

Para declarar una variable utilizaremos las palabras clave: **DIM**, **PRIVATE**, **PUBLIC** o **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 a 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.

Veamos ahora en una tabla todos los tipos existentes y comentaremos cuáles debemos utilizar:

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

Tipo de Visual Basic	Tipo en CLR (Framework)	Espacio de memoria que ocupa	Valores que se pueden almacenar y comentarios
<b>Boolean</b>	<b>System.Boolean</b>	2 bytes	Un valor verdadero o falso. Valores: <b>True</b> o <b>False</b> . 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
<b>Byte</b>	<b>System.Byte</b>	1 byte	Un valor positivo, sin signo, para contener datos binarios. Valores: de 0 a 255 Puede convertirse a: <b>Short, Integer, Long, Single, Double</b> o <b>Decimal</b> sin recibir overflow Dim b As Byte = 129
<b>Char</b>	<b>System.Char</b>	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
<b>Date</b>	<b>System.DateTime</b>	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 = #10-27-2013#
<b>Decimal</b>	<b>System.Decimal</b>	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: +/-0.0000000000000000000000000000001 (+/- 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 = 9223372036854775808D Dim unDecimal2 As Decimal = 987654321.125@

<b>Double</b>	<b>System.Double</b>	8 bytes	<p>Un número de coma flotante de doble precisión.</p> <p>Valores: de -1.79769313486231570E+308 a -4.94065645841246544E-324 para valores negativos; de 4.94065645841246544E-324 a 1.79769313486231570E+308 para valores positivos.</p> <p>Se puede convertir a <b>Decimal</b> sin recibir un overflow.</p> <p>Se puede usar como sufijo el signo almohadilla # o la letra R para representar un valor de doble precisión:</p> <p>Dim unDoble As Double = 125897.0235R Dim unDoble2 As Double = 987456.0125#</p>
<b>Integer</b>	<b>System.Int32</b>	4 bytes	<p>Un número entero (sin decimales)</p> <p>Valores: de -2,147,483,648 a 2,147,483,647.</p> <p>Se puede convertir a <b>Long</b>, <b>Single</b>, <b>Double</b> o <b>Decimal</b> sin producir overflow.</p> <p>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%</p>
<b>Long</b> (entero largo)	<b>System.Int64</b>	8 bytes	<p>Un entero largo (o grande)</p> <p>Valores: de -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807.</p> <p>Se puede convertir a <b>Single</b>, <b>Double</b> o <b>Decimal</b> sin producir overflow.</p> <p>Se puede usar la letra L o el signo &amp; para indicar que es un número Long: Dim unLong As Long = 12345678L Dim unLong2 As Long = 1234567890&amp;</p>
<b>Object</b>	<b>System.Object</b> (class)	4 bytes	<p>Cualquier tipo se puede almacenar en una variable de tipo <b>Object</b>.</p> <p>Todos los datos que se manejan en .NET están basados en el tipo Object.</p>
<b>Short</b> (entero corto)	<b>System.Int16</b>	2 bytes	<p>Un entero corto (sin decimales)</p> <p>Valores: de -32.768 a 32.767.</p> <p>Se puede convertir a: <b>Integer</b>, <b>Long</b>, <b>Single</b>, <b>Double</b> o <b>Decimal</b> sin producir un overflow.</p> <p>Se puede usar la letra S para indicar que es un número entero corto: Dim unShort As Short = 32000S</p>

<b>Single</b>	<b>System.Single</b>	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: <b>Double</b> o <b>Decimal</b> 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!
<b>String</b> (cadenas de longitud variable)	<b>System.String</b> (clase)	Depende de la plataforma	Una cadena de caracteres Unicode. Valores: de 0 a aproximadamente 2 billones ( $2^{31}$ ) de caracteres Unicode.  Se puede usar el símbolo \$ para indicar que una variable es un String.
<b>Tipos definidos por el usuario</b> (estructuras)	(heredada de <b>System.Value Type</b> )	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 más importantes de ellos y los valores mínimos y máximos que puede contener, así como el tamaño que ocupa en memoria. Ya sabemos entonces los tipos de datos que podemos usar en VB.NET y, por tanto, los que podremos utilizar para declarar variables. Por ejemplo, si queremos tener una variable en la que guardaremos números enteros, (sin decimales), los cuales sabemos que no serán mayores de 32767 ni menores de -32768, podemos usar el tipo Short:

```
Dim unShort As Short
```

Después podemos asignar el valor correspondiente:

```
unShort = 15000
```

Cuando asignemos valores a las variables podemos hacer estas 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 un 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, #01/01/2019#. 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. Por norma, utilizaremos lo más 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.

### 1.3. Detalles sobre los tipos de datos

Antes de escribir programas debo saber qué variables debo utilizar y además de qué tipo. 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 se tratará más adelante. El primero está claro: papel y lápiz y a pensar en el programa, luego pasar a limpio la idea y los procesos o algoritmos más importantes que vamos a realizar y, por fin, lo plasmaremos en un programa. Si esto se hace correctamente la declaración de variables más que un engorro es otro paso más en el desarrollo de nuestro programa y que nos va a asegurar que lo demás va a funcionar bien, en cuanto a tipos de datos se refiere.

De momento, vamos a utilizar siempre los tipos de datos más grandes para que no nos den problemas. Cuando tengamos más práctica escogeremos el tipo exacto. Los tipos de datos recomendados son:

- **INT/LONG**, para valores numéricos enteros.
- **DOUBLE**, para valores reales o con decimales.
- **DATE**, para valores de fecha.
- **BOOLEAN**, para valores booleanos, es decir, los que pueden ser sólo cierto/falso (true/false). Por ejemplo: "Servicio militar cumplido: True"
- **STRING**, para cadenas de caracteres.

Con estos 5 tipos de datos tendremos prácticamente resuelto todo el tema de las variables en cuanto al tipo de datos a utilizar se refiere. Los ampliaremos a lo largo del curso. Ahora vamos a ver si declaramos o no las variables y cómo se hace.

#### 1.4. Sobre la necesidad u obligatoriedad de declarar las variables

Como se ha comentado, Visual Basic no "obliga" a que se declaren todas las variables que vayamos a usar, pero es altamente recomendable, y nosotros las declararemos.

Existe una instrucción, **Option Explicit**, que viene activada por defecto en VB.NET. Esta instrucción obliga a que declaremos las variables. Si está desactivada (off) o se omite podemos hacer lo que queramos y declarar opcionalmente las variables. Aunque tengamos esta posibilidad, siempre debemos declarar las variables, e incluso siempre debemos declarar las variables del tipo que queremos que dicha variable contenga. (Además de declararla le tenemos que indicar si es de tipo entero, decimal,...)

Hay un tipo de datos "universal" y que puede almacenar un entero, un real, una cadena de caracteres,... Es decir, podemos declarar variables sin un tipo de datos específico:

```
Dim unaVariable
```

En realidad es como si la hubiésemos declarado del tipo `Object`, (`As Object`), por tanto, aceptará cualquier tipo de datos, pero no es una buena práctica. Lo mejor es indicar siempre el tipo de datos que es.

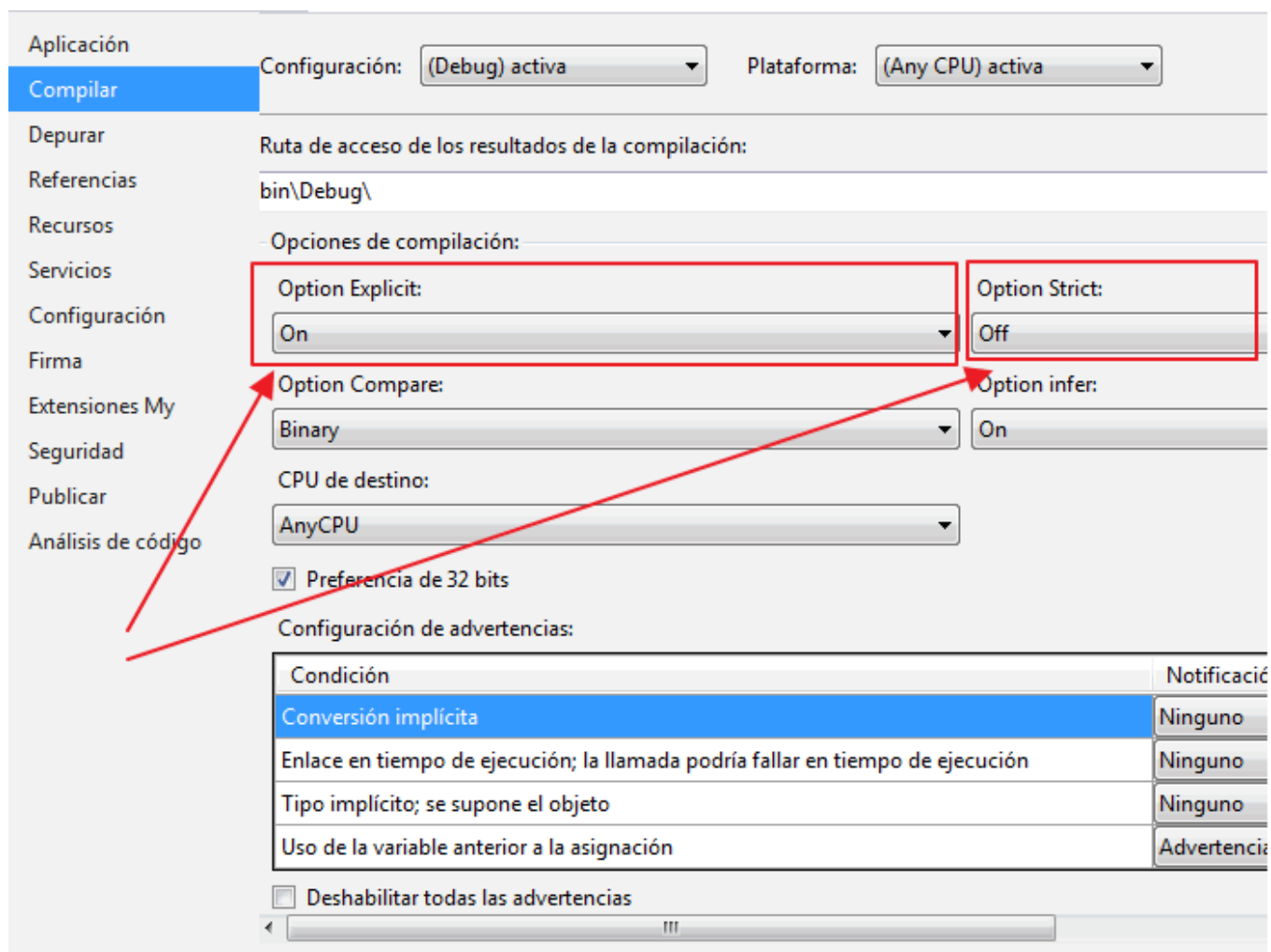
Para preparar nuestro código podemos utilizar la opción: **Option Strict**. Si se activa esta opción (mediante código con "**Option Strict On**" o en las propiedades del proyecto), obligará a que los tipos de datos que utilices sean del tipo adecuado. De esta forma hará que la programación esté más afinada al evitar muchos problemas y errores derivados de algo tan sencillo como es la

declaración de variables. No es indicarle el tipo de datos a utilizar sino que en la "vida" de la variable sólo puede admitir datos del tipo definido.

Así que tenemos dos opciones: "Explicit" para obligar a declarar y "Strict" para asegurarnos de que siempre almacena el tipo de datos correcto. Estas dos opciones las podemos indicar en el código o a nivel global en las propiedades del proyecto.

Para acceder a estas propiedades del proyecto, seleccionaremos el proyecto en la ventana del explorador de soluciones y luego con el botón derecho del ratón seleccionar Propiedades; o bien, en el menú Proyecto, seleccionar Propiedades.

Vamos a crear un proyecto de tipo Windows para ver estos detalles. Crea un proyecto nuevo de tipo aplicación de Windows y luego le das doble clic al icono de la configuración del proyecto "My project" del explorador de soluciones, o lo que es lo mismo, las propiedades del proyecto y luego selecciona la pestaña "Compilar":



Estos son los valores que recomendamos. La otra posibilidad que nos ofrece esta pantalla es **Option Compare** que define si queremos que las cadenas se comparen diferenciando las mayúsculas de las minúsculas o no. Con el valor **Binary** se diferencian las mayúsculas de las



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

**Nota:** Siempre debemos tener activado con el valor On la opción de “Option Explicit”, para que se declaren todas las variables que utilizaremos en los programas.

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 variables 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). La forma mostrada en el punto 1 es la forma clásica.

Las constantes se declaran de la misma forma que la indicada en el punto 2. No se pueden 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
```

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

Hay ocasiones en las que este valor se repite en varios sitios. Si decidimos que en lugar de tener el valor 15, queremos que tenga el 22, por ejemplo, siempre será más fácil cambiar el valor a ese nivel global, 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"
```

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.

Otro ejemplo de uso puede ser 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 más legible.

```
Const PI As Single = 3.1416
```

```
Const EURO As Single = 166.386
```

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.

Lógicamente también podremos modificar una variable con una operación con su propio valor:

```
i = i + 15
```

Con esto, estamos indicándoles a VB que: extrae lo que actualmente vale la variable *i*, súmalo 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 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:  $\ast$  =, /=, -=, etcétera, dependiendo de la operación que queramos hacer con el valor que ya tuviera la variable.

Por tanto  $i = i \ast 2$ , es lo mismo que  $i \ast= 2$

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)
```

Por esto, no podemos asignar a una variable de tipo numérico el resultado de una expresión alfanumérica:

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

Es erróneo, ya que "**10 \* 25**" es una constante de tipo cadena, no una expresión que multiplica 10 por 25. Al estar entre comillas dobles se convierte *automáticamente* en una constante de cadena y deja de ser una expresión numérica.

Y si tenemos "Option Strict On", tampoco podríamos usar números que no fuesen del tipo Single:

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

En este caso el compilador nos avisaría de esta diferencia de tipos de datos en la asignación. Para solventar estos inconvenientes utilizaremos unas funciones de conversión, que sirven para pasar datos de un tipo a otro.

En el caso anterior, la multiplicación imposible de un entero por una cadena de caracteres:  $i = "10 \ast 25"$  la podemos realizar, pero haciendo conversiones de cadena en valores numéricos. Por ejemplo,  $i = \text{Val}("10 \ast 15")$

VB.NET realizará la conversión de esta forma: lee la cadena de caracteres " $10 \ast 25$ " y la analiza letra a letra. Extrae las cifras "1" y "0" pero se encuentra con un carácter no numérico "\*" así que deja de analizar y se queda solo con ese 10, es decir, 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, `i = Val("10 * 25")` es lo mismo que: `i = Val("10")`

Es decir la función VAL nos asigna a una variable numérica el resultado de convertir una cadena de caracteres, pero en este caso de una forma bastante defectuosa. 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.text)
```

Así la variable "edad" que declaramos como de tipo entero tendrá el valor adecuado. 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 más 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.

"Val" no es un buen método para convertir cadenas a números. Es muy inexacta. Según lo que hemos dicho `Val("123abc")` devuelve el valor numérico 123 pero no debería ser así. O se convierte o no, pero no a medias. Utilizaremos otras funciones más potentes que veremos ahora mismo.

## 1.6. Funciones de conversión de tipos

La mayoría de las veces que utilicemos funciones de conversión de tipos será cuando leamos valores de la pantalla y los adaptemos para trabajar con ellos. Recuerda otra vez que los cuadros de texto, cuadros de lista y demás elementos de la interfaz suelen trabajar con cadenas de caracteres y debemos convertir los valores que introduzca el usuario al valor correcto:

- Fecha. Si introducimos una fecha debemos cambiar de tipo "string" a DATE
- Numérico. Debemos convertir de string a LONG/DECIMAL
- Booleano. Debemos convertir de string a BOOLEAN.

En el ejemplo de "10 \* 25", usamos la función *Val* para convertir una cadena en un número, pero ese número es del tipo *Double* y si tenemos "Option Strict On", no nos dejará convertirlo en un *Integer*... Así de "estricto" es el Option Strict.

Por lo tanto, sabiendo que vamos a necesitar funciones para convertir entre tipos de datos, vamos a ver las más interesantes. La anterior *Val* no nos va a servir porque si un usuario escribe en el cuadro de texto "45j" no quiero que me lo convierta a 45 sino que debo mostrarle un mensaje de que el valor introducido para la fecha no es correcto, así que utilizaré otro tipo de función para convertir de tipo de datos.

Para convertir de carácter a valor entero utilizaremos la función "CInt" recuerda que empieza por una letra C que significa de **Carácter** a **INT**. Pero claro si utilizamos esto recuerda lo que comentamos antes sobre los tipos de datos. Si intentamos:

```
i = CInt("25987278547875")
```

dará error, porque el número que está dentro de las comillas es demasiado grande para almacenarlo en una variable de tipo *Integer*. Así que utilizaremos los tipos más grandes, así la variable i sería de tipo *Long* y todo solucionado.

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

	Nombre de la función	Tipo de datos que devuelve	Valores del argumento "expresión"
X	<b>CBool(<i>expresion</i>)</b>	<b>Boolean</b>	Cualquier valor de cadena o expresión numérica.
	<b>CByte(<i>expresion</i>)</b>	<b>Byte</b>	De 0 a 255; las fracciones se redondean.
	<b>CChar(<i>expresion</i>)</b>	<b>Char</b>	Cualquier expresión de cadena; los valores deben ser de 0 a 65535.
X	<b>CDate(<i>expresion</i>)</b>	<b>Date</b>	Cualquier representación válida de una fecha o una hora.
X	<b>CDbl(<i>expresion</i>)</b>	<b>Double</b>	Cualquier valor <b>Double</b> , ver la tabla anterior para los valores posibles.
	<b>CDec(<i>expresion</i>)</b>	<b>Decimal</b>	Cualquier valor <b>Decimal</b> , ver la tabla anterior para los valores posibles.
	<b>CInt(<i>expresion</i>)</b>	<b>Integer</b>	Cualquier valor <b>Integer</b> , ver la tabla anterior para los valores posibles, las fracciones se redondean.
X	<b>CLng(<i>expresion</i>)</b>	<b>Long</b>	Cualquier valor <b>Long</b> , ver la tabla anterior para los valores posibles, las fracciones se redondean.
	<b>CObj(<i>expresion</i>)</b>	<b>Object</b>	Cualquier expresión válida.
	<b>CShort(<i>expresion</i>)</b>	<b>Short</b>	Cualquier valor <b>Short</b> , ver la tabla anterior para los valores posibles, las fracciones se redondean.
	<b>CSng(<i>expresion</i>)</b>	<b>Single</b>	Cualquier valor <b>Single</b> , ver la tabla anterior para los valores posibles.
X	<b>CStr(<i>expresion</i>)</b>	<b>String</b>	Depende del tipo de datos de la expresión.
	<b>Nota:</b> Todos los objetos de VB.NET tienen unos métodos para realizar conversiones a otros tipos, al menos de número a cadena, ya que tienen la propiedad <b>.ToString</b> que devuelve una representación en formato cadena del número en cuestión (igual que CStr).		

	<b>CType(<i>expresion</i>, Tipo)</b>	<b>El indicado en el segundo parámetro</b>	Cualquier tipo de datos
	<b>Val(<i>expresion</i>)</b>	<b>Double</b>	Una cadena de caracteres.
	<b>Fix(<i>expresion</i>)</b>	<b>Devuelven la parte entera de un número</b>	Cualquier tipo de datos. Nota: si <i>number</i> es negativo, <b>Fix</b> devolverá el primer entero negativo mayor o igual que el número. Fix convierte -8,4 en -8.
X	<b>Int(<i>expresion</i>)</b>	<b>Devuelven la parte entera de un número</b>	Cualquier tipo de datos. Nota: si <i>expresión</i> es negativo, <b>Int</b> devolverá el primer entero negativo menor o igual que el número. Int convierte -8,4 en -9

En la columna de la izquierda están marcadas las que utilizaremos con más frecuencia.

Por último, recordar que cuando definimos las variables, éstas toman un valor inicial. Éstos son los valores de los tipos más utilizados:

- Numérico. Cero (0)
- Cadena de caracteres. Cadena vacía ("")
- Fecha. 01/01/0001 0:00:00 (recuerda que el valor fecha también puede almacenar la hora)
- Booleano o lógico. False
- Objeto. Valor nulo (Nothing)

### Resumen de los "Option"

- **Option Explicit On**: hace obligatoria la declaración de variables y es la opción por defecto. Si en nuestro código intentamos asignar un valor a una variable que no esté declarada nos lo avisará antes de ejecutarse
- **Option Explicit Off**. No obliga a declarar las variables que vamos a utilizar en el programa. (Valor desaconsejable)
- **Option Strict On**. Hace obligatoria la definición del tipo de datos (tipificación) de variables y conversión de tipos explícita.
- **Option Strict Off**. No obliga a la definición del tipo de variables.

Como tenemos una función "universal" de conversión de tipos: **CType**, será la que más utilicemos ya que admite cualquier conversión posible. CType, en la ayuda sobre su sintaxis, nos dice:

"Devuelve el resultado de convertir explícitamente una expresión a un tipo de datos, objeto, estructura, clase o interfaz."

## 1.7. 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. 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 qué tipo son cada una de ellas sin tener que acudir a la parte donde se declararon.

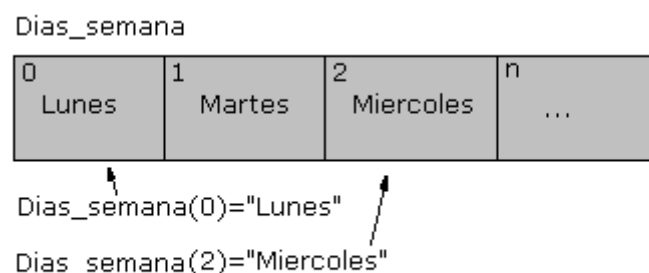
Tipo de datos	Prefijo	Ejemplo
Boolean	bln	bln_estado
Date	dat	dat_hoy
Double	dbl	dbl_nomina
Integer	int	int_edad
Long	lng	lng_largo
String	str	str_nombre

Como normalmente tenemos poco tiempo para escribir código, no solemos detenernos para seguir una guía de estilo, pero es aconsejable intentarlo. Imagina que un día nos toca retocar el código de otra persona y ha seguido un mínimo las normas de estilo... Seguramente invirtamos la mitad del tiempo en resolver el problema.

Una vez vistas las variables, sus tipos, cómo se declaran y cómo se convierten de unos tipos a otros veamos ahora las matrices o "Arrays", donde podremos almacenar bajo una denominación un conjunto de datos del mismo tipo.

## 2. 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 más de una dimensión. Cuando son de una sola dimensión se les suele llamar también **vector**. Observa este gráfico:



Definís una matriz o un array de una dimensión que se va a llamar "dias\_semana". Posteriormente le asignaremos una serie de valores de tipo "string". Una vez creada, podemos acceder a cada uno de sus elementos mediante un índice que escribiremos entre paréntesis. Como detalle importante, debemos tener en cuenta 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.

**Nota:** 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 VB.NET y permite almacenar distintos tipos.

Veamos ahora cómo asignamos los valores al array:

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

Lógicamente recibiremos un error si queremos asignar el valor octavo: dias\_semana (7) ya que definimos la matriz con 7 elementos y estaríamos apuntando al octavo.

La definición de una matriz de más de una dimensión es un proceso 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"  
otra_matriz(2, 4, 4) = "otro elemento"
```

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



El inconveniente de las matrices es que tenemos que decirle el número de elementos que debe tener pero VB.NET permite modificar el tamaño de las matrices (arrays) en tiempo de ejecución, es decir, mientras nuestro programa se está ejecutando. Para esto hay 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 más utilizada porque normalmente queremos asignar un nuevo valor y al ver que hemos llegado al último elemento ampliaremos en más elementos pero conservando lógicamente los actuales. Observa 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 matriz2 a 20 elementos pero conservando los elementos anteriores.

## 2.1. Formas de declarar una matriz

Además de la más básica vista arriba, veamos más 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 Long
```

## Iniciar los valores

En este tema vimos anteriormente que podíamos declarar las variables 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 declararlo, no podemos indicar el número de elementos que tendrá. Esto es porque el número de elementos estará supeditado a los valores asignados. Para inicializar un array debemos declarar esa matriz sin indicar el número de elementos que contendrá, seguida de un signo igual y a continuación los valores encerrados en llaves. Veamos un ejemplo:

```
Dim a() As Integer = {1, 42, 15, 90, 2}
```

También podemos hacerlo de esta otra forma:

```
Dim a As Integer() = {1, 42, 15, 90, 2}
```

## 2.2. Recorrer una matriz. Utilizar bucles para recorrer los elementos.

Para recorrer una matriz podemos hacerlo con los distintos tipos de bucles que nos proporciona VB.NET. Uno de ellos, es el bucle "For/Next" que, aunque lo veremos más adelante, funcionaría de esta forma:

```
Dim i As Integer
For i=0 to 4
    Console.WriteLine(i)
Next
'

Console.WriteLine("Pulsa Intro para finalizar")
Console.ReadLine()
```

Debemos saber el número de elementos que tiene la matriz. Si no conociéramos este número de elementos disponemos de otro tipo de bucle "For Each". Este bucle es muy interesante y permite recorrer colecciones de elementos. En nuestro caso la matriz se compone de una colección de elementos, por tanto, podemos utilizar ese tipo de bucle.

```
Dim a() As Integer = {1, 42, 15, 90, 2}
'
Console.WriteLine("Elementos del array a()= {0}", a.Length)
'
Dim i As Integer
For Each i In a
    Console.WriteLine(i)
Next
'

Console.WriteLine("Pulsa Intro para finalizar")
Console.ReadLine()
```

Los bucles los veremos con detalle en el siguiente tema.

## 2.3. Ordenar 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 la 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**. Este método sirve para ordenar el contenido de un array

Para ordenar el array **a**, usaremos el siguiente método:

```
Array.Sort(a)
```

Se trata del método Sort de la clase Array. Este método es lo que se llama un método "compartido" y, por tanto, se puede usar directamente.

Veamos ahora un ejemplo completo en el que se crea y asigna un array al declararlo. Luego mostramos el contenido del array usando un bucle For Each, lo ordenamos y lo volvemos a mostrar con un bucle For/Next.

```
Sub Main()  
    Dim a() As Integer = {1, 42, 15, 90, 2}  
    '   
    Console.WriteLine("Elementos del array a(): {0}", a.Length)  
    '   
    Dim i As Integer  
    For Each i In a  
        Console.WriteLine(i)  
    Next  
    '   
    Console.WriteLine()  
    '   
    Array.Sort(a)  
    '   
    For i = 0 To a.Length - 1  
        Console.WriteLine(a(i))  
    Next  
    '   
    Console.WriteLine("Pulsa Intro para finalizar")  
    Console.ReadLine()  
End Sub
```

## 2.4. 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  
'   
Console.WriteLine("m = {0}, n = {1}", m, n)
```

El contenido de **n** será 7 y el de **m** será 9, es decir, cada variable contiene y mantiene de forma independiente el valor que se le ha asignado y a pesar de haber hecho la asignación **n = m**, y posteriormente haber cambiado el valor de **m**, la variable **n** no cambia de valor. Sin embargo, si hacemos algo parecido con dos arrays, veremos que no es igual:

```
Dim a() As Integer = {1, 42, 15, 90, 2}
Dim b() As Integer
Dim i As Integer
'
b = a
'
a(3) = 55
'
For i = 0 To a.Length - 1
    Console.WriteLine("a(i) = {0}, b(i) = {1}", a(i), b(i))
Next
```

En este caso, al cambiar el contenido del índice 3 del array **a**, también cambiamos el contenido del mismo índice del array **b**. Esto es así porque sólo existe una copia en la memoria del array creado y cuando asignamos al array **b** el contenido de **a**, realmente le estamos asignando la dirección de memoria en la que se encuentran los valores. No estamos haciendo una nueva copia de esos valores, por tanto, al modificar el elemento 3 del array **a**, estamos modificando lo que tenemos "guardado" en la memoria. Como resulta que el array **b** está apuntando (o hace referencia) a los mismos valores entonces 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**.

## 2.5. 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
    Console.WriteLine("a(i) = {0}, b(i)= {1}", a(i), b(i))
Next
```

En este ejemplo, inicializamos un array, declaramos otro con el mismo número de elementos y 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.

## 2.6. Límites de las matrices

Para trabajar con las matrices tenemos dos instrucciones muy interesantes que nos van a proporcionar tanto el límite inferior como el superior:

- **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

La primera nos indica cuántos elementos puede contener la matriz porque nos está devolviendo el índice mayor disponible. Luego, Ubound(mimatriz) devolverá 7, por ejemplo, si es ese su número de elementos. Veamos 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

Notar que estos valores son perfectos para recorrer las matrices con el bucle "For/Next" ya que sería simplemente indicarle: For i=0 to Ubound(matriz)-1.

## 3. Los arrays multidimensionales

La diferencia entre las matrices unidimensionales y las multidimensionales está en el número de dimensiones que contienen. Los arrays unidimensionales, (o vectores), sólo tienen una dimensión: los elementos se referencian por medio de un solo índice. Por otro lado, los arrays multidimensionales tienen más de una dimensión. Para acceder a uno de los valores que contengan habrá que usar más de un índice. La forma de acceder, por ejemplo en el caso de que sea de dos dimensiones, sería algo como esto: **multiDimensional(x, y)**.

Es decir, utilizaremos una coma para separar cada una de las dimensiones que tenga. El número máximo de dimensiones que podemos tener es de 60, 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 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.

### 3.1. Declarar arrays multidimensionales

Para declarar un array multidimensional, lo podemos hacer de varias formas y son parecidas a las vistas en el caso de una dimensión. Veamos unos ejemplos:

```
'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 los
valores del contenido
Dim c1() As Integer = {1, 2, 3, 4}

'Dos dimensiones, sin indicar cuantos elementos y declarando los
valores del contenido
Dim c2(,) As Integer = {{1, 2, 3}, {4, 5, 6}}

'Tres dimensiones, sin indicar cuantos elementos y declarando los
valores del contenido
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}} _
}
```

Podemos ver definiciones de matrices con una, dos, tres y cuatro dimensiones.

En el último caso, hemos usado el continuador de líneas ( \_ ) para que sea más fácil leer la asignación de cada una de las dimensiones.

### 3.2. Tamaño de un array multidimensional

Podemos utilizar el método *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* ya que representa la longitud o tamaño. Pero para saber cuántos 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 cuántos elementos hay en la tercera dimensión.

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

```
Console.WriteLine("Las dimensiones de c3 son: (?,,)= {0}, (,?,)= _  
{1}, (,,?)= {2}", c3.GetUpperBound(0), c3.GetUpperBound(1), _  
c3.GetUpperBound(2))
```

### 3.3. Número de dimensiones de un array multidimensional.

Una cosa es saber cuántos elementos tiene un array (o una de las dimensiones del array) y otra cosa es saber cuántas dimensiones tiene dicho array.

Para saber el número de dimensiones del array, usaremos la propiedad **Rank**. Por ejemplo, (si usamos la declaración hecha antes), el siguiente código nos indicará que el array **c3** tiene tres dimensiones:

```
Console.WriteLine("El array c3 tiene {0} dimensiones.", c3.Rank)
```

El valor devuelto por **Rank** es el número total de dimensiones del array. Debemos tener en cuenta que el número de dimensiones comienza en cero y va hasta `Rank - 1`.

Veamos ahora un ejemplo de cómo recorrer todos los elementos del array **c3** y cómo mostrar el resultado por consola. Para saber cuántos elementos hay en cada una de las dimensiones, utilizaremos la propiedad *GetUpperBound* y así indicaremos 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**.

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)
            Console.WriteLine("El valor de c3({0}, {1}, {2}) es _ {3}", _
i, j, k, c3(i, j, k))
        Next
    Next
Next
```

### 3.4. Cambiar el tamaño de un array y conservar su contenido

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.

Respecto al número de dimensiones del array, 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 de 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 que 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.

### 3.5. Eliminar un array de la memoria.

Si en algún momento del programa queremos eliminar el contenido de un array, por ejemplo, para liberar la memoria que está ocupando o para restablecerlo a los valores iniciales, podemos utilizar el método **Erase** seguido 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 **ReDimensionarlo** con el mismo número de dimensiones que tenía, ya que Erase sólo borra el contenido, no la definición del array.

### 3.6. ¿Podemos ordenar un array multidimensional?

La respuesta es: No.

El método **Sort** sólo permite ordenar un array unidimensional, (de una sola dimensión). La única forma de ordenar un array multidimensional sería haciéndolo de forma manual.

De igual forma que no podemos ordenar 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**.

### 3.7. 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**.

**CopyTo**, que es la forma que vimos antes para copiar el contenido de un array, NO se puede usar, ya que **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)
```

No se indica qué dimensión queremos copiar, ya que se copia "todo" el contenido del array. Además 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. 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 (error) 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. Se han visto prácticamente todas las opciones que se nos pueden plantear.

### 3.8. 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 cuántos valores tiene. Por ejemplo, leer líneas de un fichero texto, no sabemos cuántas 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 hemos utilizado en una conversión de tipos para asegurarnos que metemos un valor de tipo "String" en la variable.

## 4. Estructuras

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

Las estructuras permiten crear nuestros propios tipos de datos. Una estructura contiene uno o más 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. Utilizaremos la palabra clave "Structure" con esta sintaxis:

```
Structure nombre
    Declaraciones
End Structure
```

En el siguiente ejemplo vemos cómo se declara una estructura.

```
Structure Empleado
    Dim Numero As Long
    Dim Nombre As String
    Dim Direccion As String
    Dim Puesto As String
End Structure
```

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 = "Andrés"
emp1.Direccion = "Jorge Vigón"
emp1.Puesto = "Jefe"

emp2.Numero = 11
emp2.Nombre = "Mario"
emp2.Direccion = "Club Deportivo"
emp2.Puesto = "Técnico"
```

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

```
Dim matriz_empleados(10) As Empleado

matriz_empleados(0).Numero = 10
matriz_empleados(0).Nombre = "Andrés"
matriz_empleados(0).Direccion = "Jorge Vigón"
matriz_empleados(0).Puesto = "Jefe"
```

Es un tipo de datos muy utilizado cuando manejamos muchos datos en los formularios. Para matrices de datos, por ejemplo, con fichas de empleados, datos de materiales, podemos declarar una variable de este tipo y quedará mucho más legible y elegante.

## 5. Operadores y Comparadores

Una vez que ya conocemos las variables y las matrices veamos qué operadores tenemos para trabajar con estos datos.

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 cadenas para crear una: "Jose Angel"

### Comparación

Estos operadores permiten comparar dos operandos, el resultado será un valor booleano (lógico): True (verdadero) o 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

## 6. Flujo de programas. Evaluar expresiones lógicas.

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

Antes vimos por encima las instrucciones IF /THEN. Ahora vamos a verlas en profundidad.

Hay ocasiones en las que necesitaremos decidir qué hacer dependiendo de algún condicionante. Por ejemplo, en los primeros ejemplos, teníamos que comprobar si la tecla que se había pulsado era la tecla *Suprimir* y si era así, ejecutábamos determinadas instrucciones.

Por tanto, podemos decir que para tomar decisiones usaremos:

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

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

Eso mismo también se suele usar de esta otra forma:

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

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

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

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

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

```
If <expresión a evaluar> Then  
    <Lo que haya que hacer si la expresión evaluada devuelve Verdadero>  
Else  
    <Lo que haya que hacer si no se cumple>  
End If
```

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

```
If a = 10 Then
    ' Lo que sea que haya que hacer cuando a vale 10
ElseIf a = 15 Then
    ' Lo que haya que hacer cuando a vale 15
Else
    ' Lo que haya que hacer en caso de que a no valga ni 10 ni 15
End If
' Esto se ejecuta siempre después de haberse comprobado todo lo
anterior.
```

En medio de cada If / Then hay un **comentario** para documentar mejor el código.

**Nota:** Los comentarios sólo pueden ocupar una línea, salvo que dicha línea al final tenga el signo \_ (subrayado bajo). Este símbolo indica al IDE que queremos continuar en la siguiente línea. Ese símbolo se puede llamar "continuador de línea" y lo podemos usar siempre que queramos, no sólo para los comentarios.

Si tenemos el **Option Strict On** (comprobación estricta de tipos de datos), la expresión que se use después de If debe devolver un valor del tipo Boolean, es decir, debe dar como resultado un valor True o False. Si **Option Strict** está en **Off**, VB lo "convertirá" en un valor True o False, pero no debemos acostumbrarnos a que VB haga las cosas medio-automáticas, ya que en ocasiones puede ser que ese automatismo no dé como resultado lo que nosotros "creíamos" que iba a dar...

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

```
If i > 25 Then
```

Lo que hace es evaluar la expresión y comprueba si el valor de i es mayor de 25 y, en caso de que así sea, devolverá un valor True y si resulta que i no es mayor de 25, devolverá False. A continuación se comprueba ese valor devuelto por la expresión y si es verdadero (True) se hace lo que esté después del Then y si es falso (False), se hará lo que esté después del Else, (si es que hay algún Else...)

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

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

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

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

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

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

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

Aunque esto mismo podríamos haberlo hecho de esta otra forma:

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

Detengámonos un poco para ver estos operadores lógicos. Recordemos que los operadores lógicos devuelven un valor de tipo Boolean (true o false). Veámoslos con más detalle:

- **And.** Para que se cumpla deben ser ciertas las dos expresiones: *si es mayor de 30 años AND tiene carné de conducir entonces...* En los demás casos no se cumple la comparación.
- **Or.** Se cumple cuando una de las dos comparaciones es cierta: *Si tiene más de 25 años OR es Varón entonces...*
- **Not.** Esta operación realiza una negación de una expresión.
- **Xor.** Realiza una exclusión entre dos expresiones. Devolverá "false" cuando se cumplan o no ambas expresiones, y true cuando una se cumpla y la otra no.
- **AndAlso.** Esta comparación es muy peculiar y con el tiempo nos será muy útil: en cuanto la primera expresión devuelta sea "false" el resto no se evaluará devolviendo falso como resultado final.
- **OrElse.** Realiza lo siguiente: en cuanto la primera expresión devuelva verdadero el resto no se evaluará devolviendo "true" como resultado final

Los operadores tienen una prioridad cuando existan varios en la misma línea. La prioridad es por este orden: potencia (^), negación (-), multiplicación y división real (\*,/), división entera (\), resto de división (Mod) y Suma y resta (+,-).

Por otra parte, podemos tener entonces varias comparaciones en una misma línea. Por ejemplo:

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

Pero esto queda un poco ambiguo. Le falta algo, ¿verdad?. Pues sí. ¿Qué quiere decir esto? En la primera expresión: ¿Qué pasa si A es igual a 100 pero B es menor de 50, y x es igual a n \* 2? Que se cumple, igual que si x no fuese igual a n \* 2, pero si A no vale 100, sólo se cumpliría si B fuese mayor de 50. Es decir, la última expresión sólo se tiene en cuenta si A no vale 100 y B es mayor de 50.

Por tanto, quedaría más claro de esta otra forma:

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

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

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



En cuyo caso sólo se cumplirá cuando A sea 100 o B mayor de 50, pero SIEMPRE x debe ser igual a  $n * 2$ . Es decir, debemos utilizar los paréntesis dependiendo de lo que realmente queramos comprobar cuando utilicemos más de una comparación

De todas formas, hay que aclarar que en VB.NET las expresiones se van evaluando de izquierda a derecha y se van descartando según se van encontrando cosas que "cumplan" lo que allí se comprueba.

## Ejemplo

Veamos un primer ejemplo completo. En este ejercicio vamos a crear una aplicación de consola, es decir, no va a ser una de Windows, así que la salida nos la va a dar por una ventana de comandos.

Vamos a utilizar dos instrucciones `Console.WriteLine` para escribir textos o variables en pantalla, y `Console.ReadLine` para que la pantalla se quede visible hasta que pulsemos Intro y no termine la aplicación sin poder ver su contenido.

Crea una aplicación de consola y copia este texto.

```
Dim unByte As Byte = 129
Dim unBoolean As Boolean = True
Dim unChar As Char = "N"c
'Dim unChar2 As Char = "B" ' (Con Option Strict On da
error)
Dim unaFecha As Date = #10/27/2019#
Dim unDecimal As Decimal = 99912581258.125D
Dim unDecimal2 As Decimal = 9876543210123456@
Dim unDoble As Double = 125897.12045R
Dim unDoble2 As Double = 2457998778745.4512#

'Dim unInt As Integer = 24579987787456 ' (Con Option
Strict On da error)
Dim unEntero As Integer = 250009I
Dim unEntero2 As Integer = 652000%
Dim unLong As Long = 123456789L
Dim unLong2 As Long = 987654&
Dim unShort As Short = 32000S
'

Const N As Integer = 15
'

Dim i As Integer

'Una sencilla operación
i = 10
i = i + 25 + (N * 2)
Console.WriteLine("(i=10, n=15), 'i = i+ 25 + (N * 2)'"
es igual a: " & CStr(i))
```

```
'Convertir un valor a carácter
i = CInt("25000")
Console.WriteLine(i)

'Devolvemos la parte entera de un numero
unDoble = Fix(unDoble2)
Console.WriteLine("unDoble = Fix(" & unDoble2.ToString
& ") : " & unDoble.ToString)

'Con el tipo de datos real Double
unDoble2 = 2457998778745.665#
unDoble = CLng(unDoble2)
Console.WriteLine("unDoble = Lng(" & unDoble2.ToString
& ") : " & unDoble.ToString)

'Convirtiendo el valor doble a string
unDoble = Fix(8.9)
Console.WriteLine("unDoble = Fix(8.9) : " &
unDoble.ToString)
'
unDecimal = Fix(8.9D)
Console.WriteLine("unDecimal = Fix(8.9D) : " &
unDecimal.ToString)
'
Console.WriteLine("i vale: " & CStr(i))
If i > 1500 Then
Console.WriteLine("i es mayor de 1500")
End If

'Una comparación
i = 200
Console.WriteLine("Se asigna el valor " & CStr(i) & " a
i")
If i > 15 + n Then
Console.WriteLine("i es mayor de 15 + n")
Else
Console.WriteLine("i NO es mayor de 15 + n")
End If
'

'Dos comparaciones
If i > 200 Or i < 500 Then
Console.WriteLine("el valor de i es mayor de 200 0
menor de 500")
End If

If i > 100 And i < 500 Then
Console.WriteLine("el valor de i es mayor de 100 Y
menor de 500")
End If

If Not i = 100 Then
Console.WriteLine("i NO vale 100")
End If
```

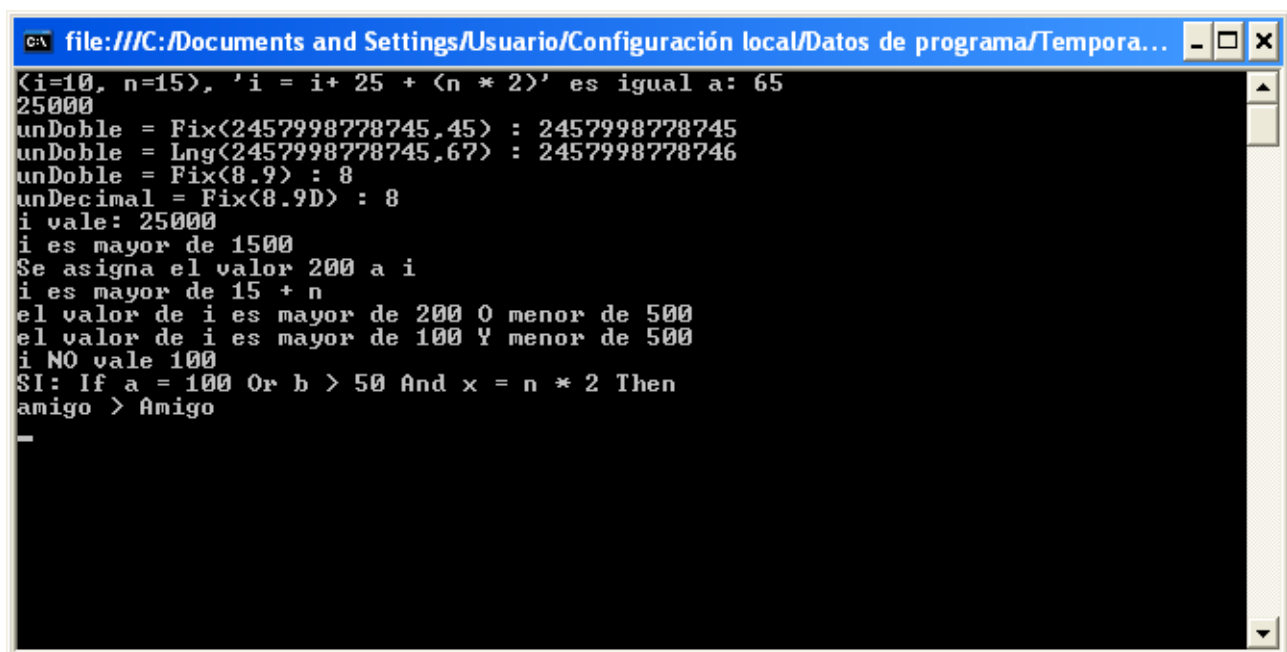
```
'Seguimos con el programa declarando otras variables...
Dim a As Integer = 100
Dim b As Integer = 50
Dim x As Integer = N * 2 + 10

If a = 100 Or (b > 50 And x = N * 2) Then
Console.WriteLine("SI: If a = 100 Or b > 50 And x = N *
2 Then")
End If

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

Console.ReadLine()
```

Que da como resultado:



```
file:///C:/Documents and Settings/Usuario/Configuración local/Datos de programa/Tempora...
<i=10, n=15>, 'i = i+ 25 + <n * 2>' es igual a: 65
25000
unDoble = Fix<2457998778745,45> : 2457998778745
unDoble = Lng<2457998778745,67> : 2457998778746
unDoble = Fix<8.9> : 8
unDecimal = Fix<8.9D> : 8
i vale: 25000
i es mayor de 1500
Se asigna el valor 200 a i
i es mayor de 15 + n
el valor de i es mayor de 200 0 menor de 500
el valor de i es mayor de 100 Y menor de 500
i NO vale 100
SI: If a = 100 Or b > 50 And x = n * 2 Then
amigo > Amigo
-
```

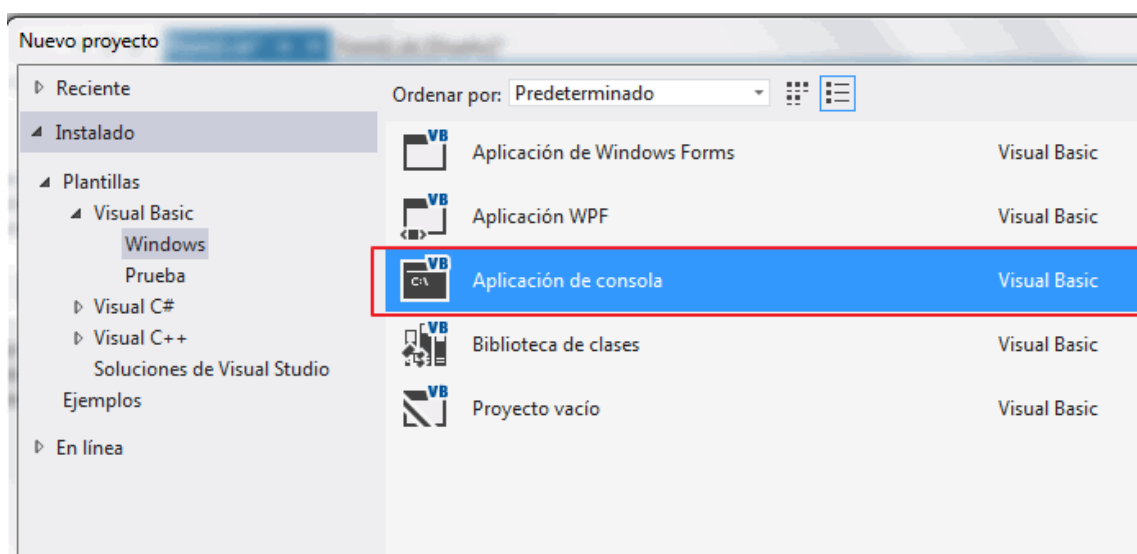
## 7. Aproximación a la depuración: ejecución paso a paso y ventana inmediato.

Antes de continuar con las variables vamos a ver cómo podemos controlar la ejecución de nuestro código en VB.NET. Lo que haremos es conocer las herramientas que nos proporciona el entorno de desarrollo integrado para depurar nuestro programa y poder ejecutarlo línea a línea. Todas las herramientas de depuración que nos ofrece el IDE las veremos más adelante pero esta primera toma de contacto nos será muy útil para ir practicándola ya en los ejemplos que vayamos haciendo.

Cuando un programa no funciona correctamente el IDE nos proporciona muchas formas de poder depurar o vigilar el código para encontrar el error. Vamos a crear un nuevo proyecto para ejecutar unas instrucciones y ver cómo podemos depurarlas o ejecutarlas paso a paso.

### 7.1. Ejemplo con aplicación de consola

En estas pruebas vamos a crear una aplicación de consola, es decir, no es un programa Windows sino que se comportará como si fuese un programa de consola:



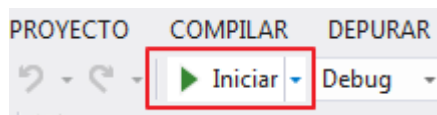
Ahora vamos a escribir este código dentro de las instrucciones “Sub Main” y “End Sub”:

```
Dim edad As Long
Dim nombre As String
Dim apellidos As String
edad = 23
nombre = "José Ma"
apellidos = "Rodríguez"
edad = 87
```

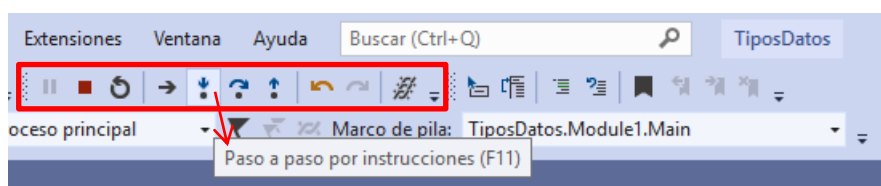
Este programa no hace nada todavía. Sólo declara tres variables y luego les asigna unos valores, pero lo que queremos aprender es cómo se ejecuta el programa paso a paso y cómo ver el valor de las variables a medida que se ejecutan las instrucciones.

El código de ejemplo lo escribiremos entre las instrucciones que indican el comienzo y final del programa Sub\_main y End sub.

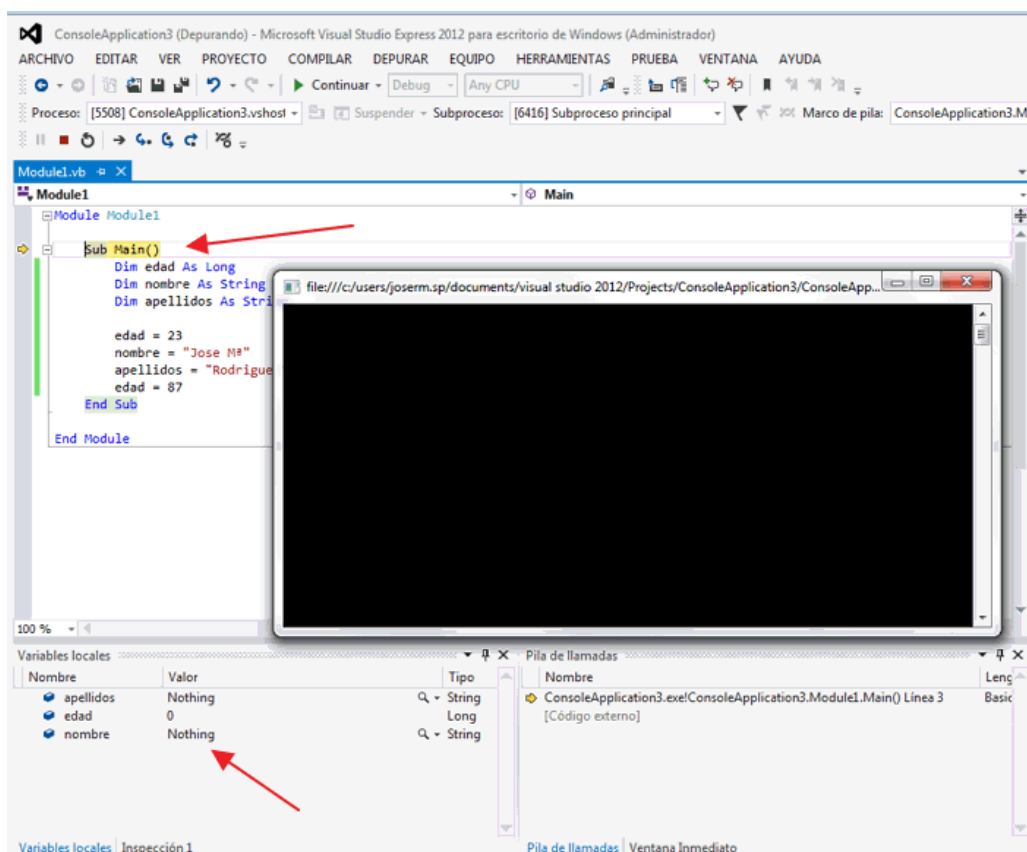
Ahora nos fijaremos en estos iconos de la barra de iconos "Estándar":



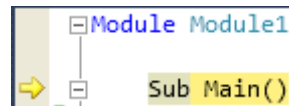
El icono en verde inicia la ejecución del programa. Para iniciar la depuración del programa debemos pulsar F11 (Paso a paso por instrucciones) o F10 (Paso a paso por procedimientos). Ahora veamos la barra de herramientas "Depurar" que debemos tener ya siempre visible:



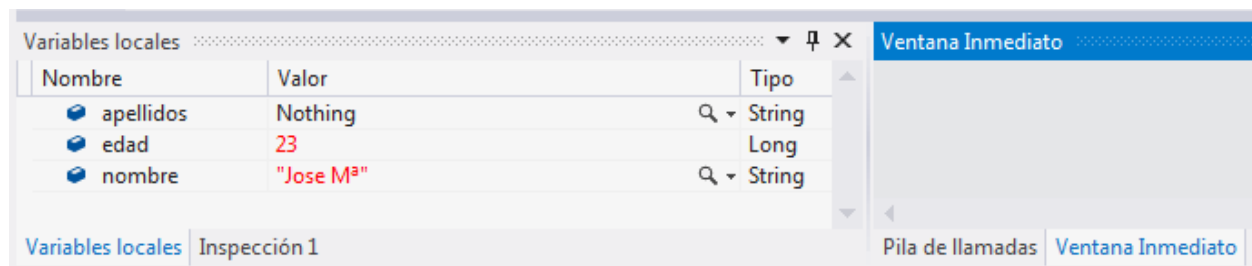
Los tres iconos que vemos en azul se encargan de la depuración, o ejecución controlada de nuestro programa. Pulsamos en el quinto icono de los que he marcado antes y que corresponde con la ejecución "paso a paso por instrucciones". Al pulsar y después de unos segundos veremos varios cambios en pantalla:



Por un lado han desaparecido algunas ventanas de nuestro IDE. Esto es porque ya no estamos en "tiempo de diseño" sino que estamos en "tiempo de ejecución". Puesto que el programa está ahora ejecutándose ya no debe mostrar las barras de controles de diseño sino las ventanas de ejecución. En nuestro caso como además hemos creado una aplicación de consola vemos que se ha abierto una ventana de comandos donde mostrará el resultado del programa (aunque este ejemplo no escriba nada en pantalla). Además vemos como en el código aparece una línea destacada en color amarillo con una flecha a la izquierda:

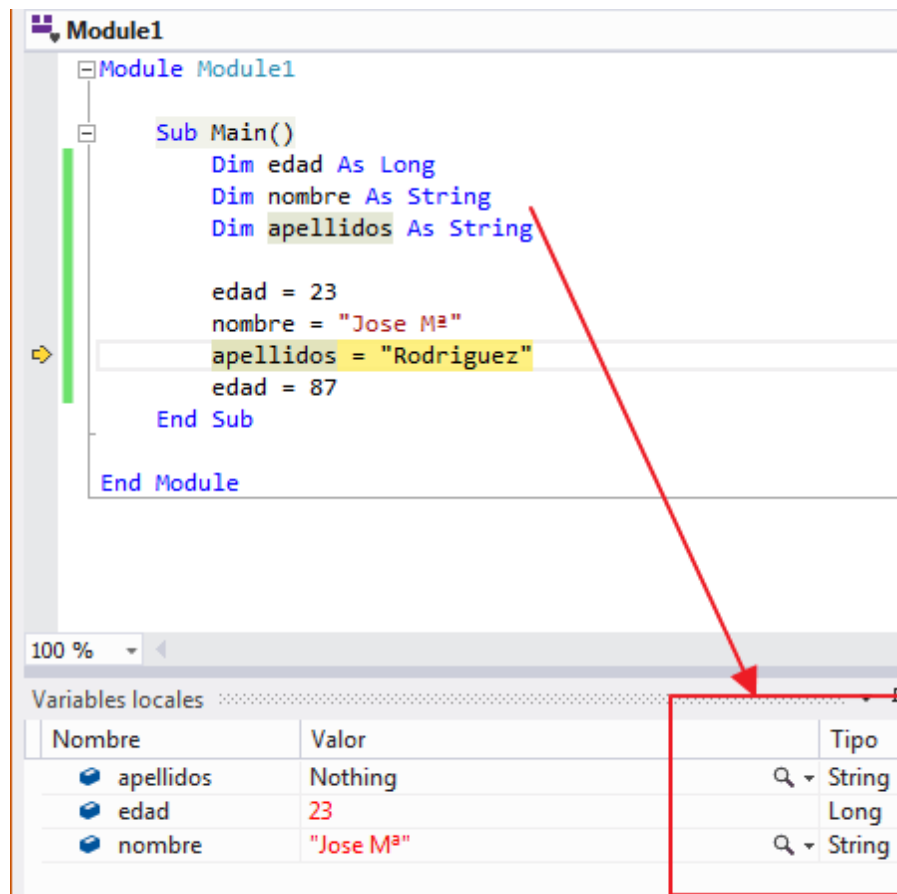


Esto significa que el programa ha comenzado a ejecutarse y me está mostrando exactamente la línea en ejecución, en este caso el comienzo del programa. En la parte inferior de la ventana vemos varias ventanas entre las que encontramos una llamada "Automático" que nos ayudará a ver el contenido de la ejecución del programa. La otra ventana que debes dejar visible es la "Ventana Inmediato". Debes tener las dos ventanas ya que son las que nos ayudarán para estos momentos de la ejecución del programa:

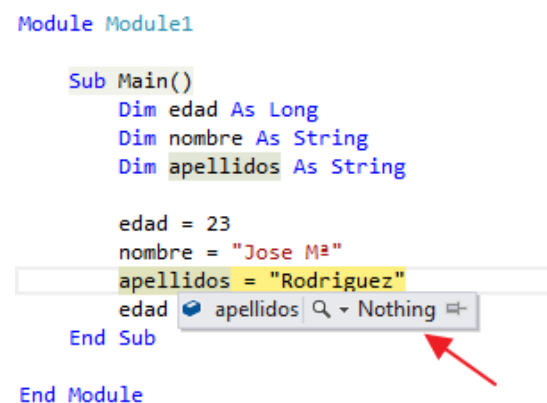
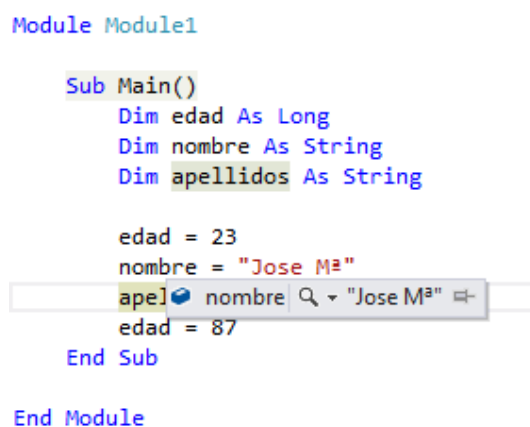


La Ventana "Inmediato" se utiliza con fines de depuración como evaluar expresiones, ejecutar instrucciones, imprimir valores de variables, etc. Permite escribir expresiones que el lenguaje de programación evalúa o ejecuta durante la depuración. En algunos casos, es posible cambiar el valor de las variables.

Sigamos con nuestra depuración. Al ejecutarse en esta ventana desaparece el cursor y se queda preparada para recibir comandos. Para comenzar la ejecución paso a paso de nuestro programa pulsaremos la tecla F11 o el botón que comentamos antes de la barra de depuración. Pulsaremos varias veces hasta llegar a la línea de la asignación del apellido:

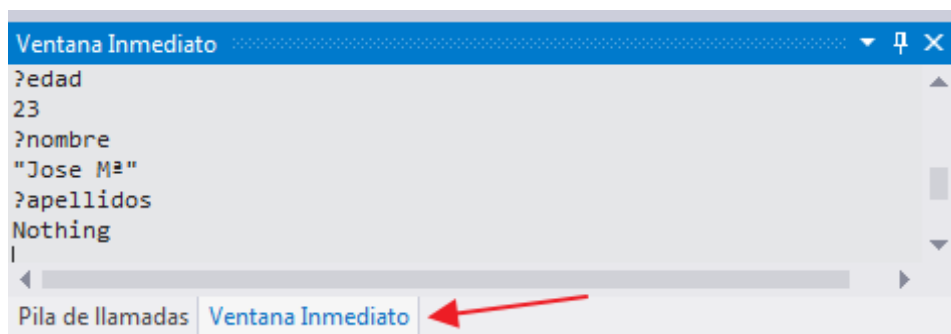


En estos momentos hemos ejecutado las primeras instrucciones y está a punto de ejecutar la asignación del apellido. La marca amarilla indica la instrucción que va a ejecutar en la siguiente pulsación así que no la ha ejecutado todavía. Veamos la información que nos ofrece el editor sobre estas variables. Haz clic en el código para activar esa ventana y pasa el ratón por encima de la variable "nombre" y luego por la variable "apellidos":



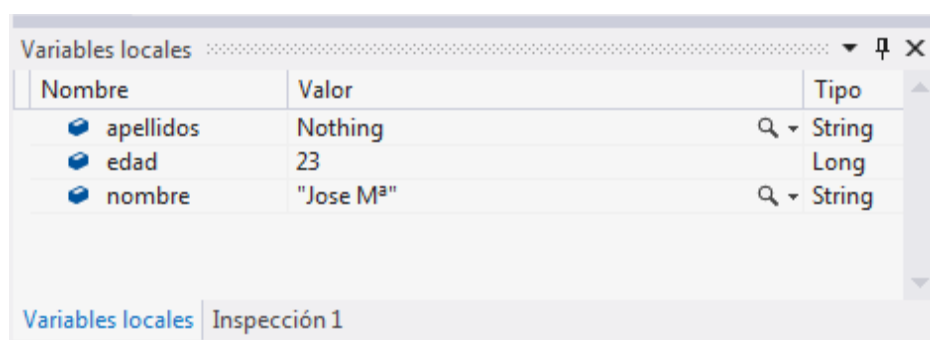
Al pasar el ratón por encima de la variable "nombre" nos indica el valor que tiene en ese momento, que tiene el esperado string "Jose Mª". Sin embargo, como la instrucción de la asignación no se ha ejecutado si queremos ver el valor de la variable "apellidos" me devuelve un "Nothing". Es decir, todavía no tiene nada y es lógico porque todavía no se ha ejecutado. Vamos ahora a la ventana Inmediato.

Escribimos: ?edad verás cómo nos escribe su valor, haz lo mismo con nombre: ?nombre y ?apellidos:



Esta ventana nos va a servir para evaluar cualquier expresión, en ese caso, al ponerle los interrogantes lo que le estábamos pidiendo es el valor de esas variables (si existen, claro). Vemos como las dos primeras tienen valor pero "apellidos" como todavía no se ha ejecutado no tiene ningún valor asignado.

También tenemos una ventana que nos muestra todas las variables con sus tipos de datos y valores actuales. Si terminas la depuración verás cómo van cambiando los datos en esta ventana y sabremos en todo momento qué valores tienen asignados las variables:



Estos dos métodos: la ejecución por líneas de código paso a paso y la posibilidad de ver el contenido de las variables son una grandísima ayuda para nuestro trabajo de programación.



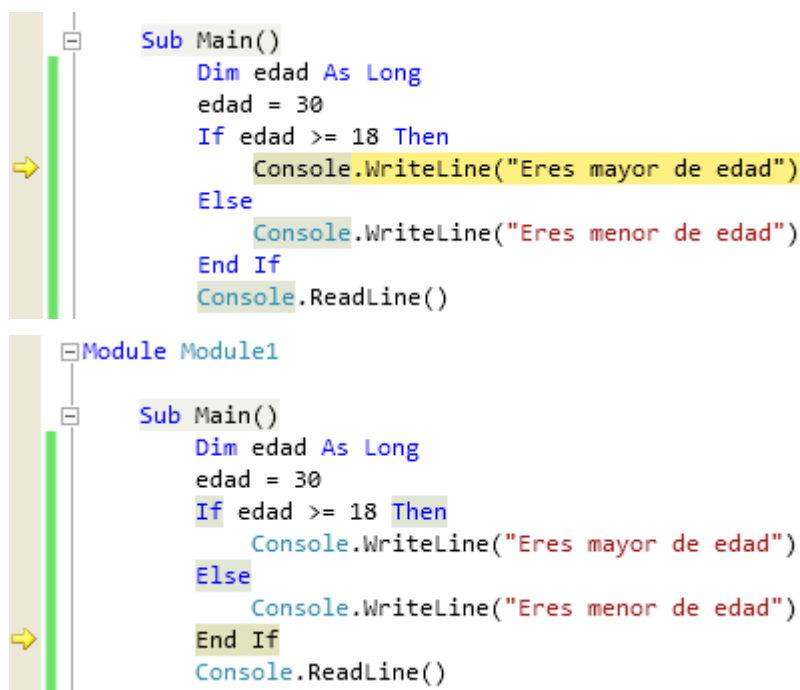
## 7.2. Ejemplo 2 con aplicación de consola

Veamos otro ejemplo de código también en consola:

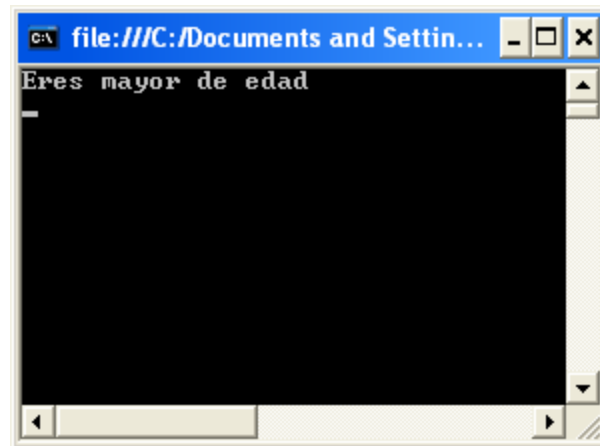
```
Dim edad As Long
edad = 30
If edad >= 18 Then
    Console.WriteLine("Eres mayor de edad")
Else
    Console.WriteLine("Eres menor de edad")
End If
Console.ReadLine()
```

Comienza con la tecla F11 la ejecución paso a paso de este programa. Ahora nos fijaremos al llegar a la comparación del "If". En este momento va a evaluar si es cierto que la variable edad es mayor o igual que 18. En este caso sí que lo es porque le asignamos antes el valor 30 así que es "True" y deberá ejecutar la línea siguiente.

Al pulsar F11 vemos como es cierto, pasa por esa línea y la siguiente. Atención: como ha hecho la comparación del If y ha sido cierta no sigue ejecutando el resto del If. Lógico porque las otras instrucciones son cuando edad<18:



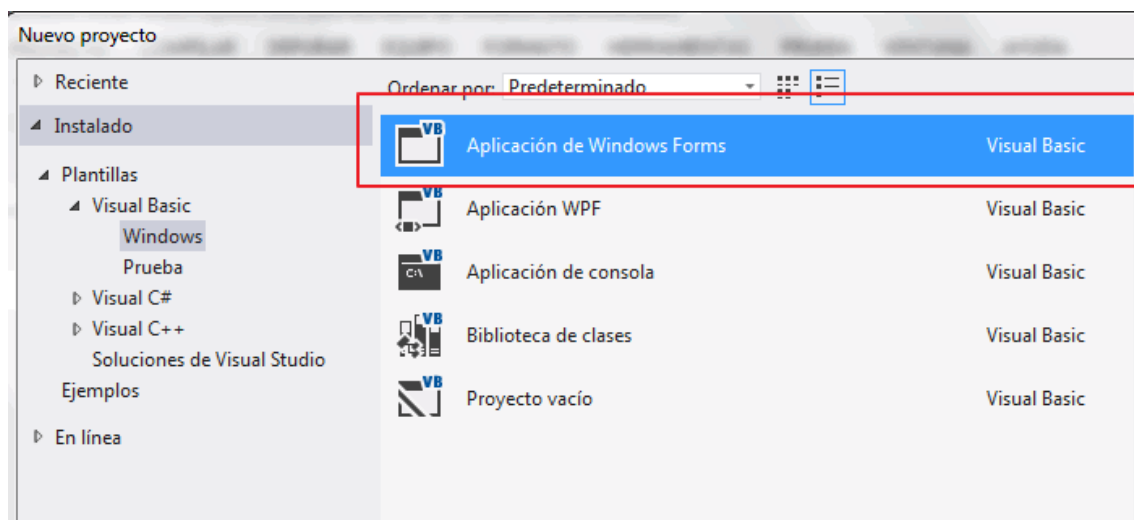
La salida del programa será:



Que es el resultado de ejecutar la instrucción de escribir en la consola un texto: "Console.WriteLine". En este caso la depuración ha sido verdaderamente útil porque hemos visto la secuencia de instrucciones que ha ejecutado al evaluar las distintas condiciones. Esto lo utilizaremos en muchas ocasiones para ver si el programa "va por donde debe ir".

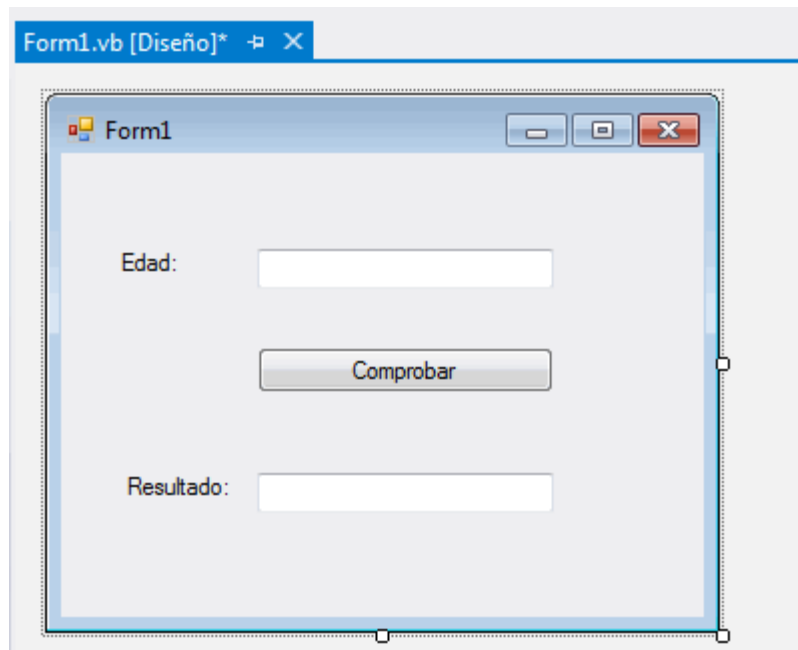
### 7.3. Ejemplo con aplicación de Windows

En este tercer y último ejemplo de esta sección crearemos una aplicación Windows:



Recuerda que utilizaremos formularios, que son uno de los objetos de la clase System.Windows.Forms.

En el formulario colocaremos estos elementos:



Al colocar los elementos sigue estos pasos:

- Colocar un objeto Label para el literal "Edad". Propiedades a modificar:
  - Propiedad **Text**. Asignar el valor "Edad"
- Añadir otro Label para el literal "Resultado". Propiedades a modificar:
  - Propiedad **Text**. Asignar el valor "Resultado"
- Añadir un cuadro de texto para que el usuario escriba su edad.
  - Propiedad **Name**. Asignarle el valor txt\_edad.
- Añadir un cuadro de texto donde escribiremos el resultado
  - Propiedad **Name**. Asignarle el valor txt\_resultado.
- Añadir un botón para que ejecute el programa
  - Propiedad **Name**. Asignarle el valor btn\_comprobar.
  - Propiedad **Text**. Asignarle el valor "Comprobar"

Normalmente estableceremos ya nombres a todos los controles que pongamos. Los controles "Label" como suelen ser etiquetas estáticas y no se manejan desde el programa pueden quedarse con el nombre genérico que le pone el IDE. Pero solo en este caso de etiquetas, a los demás siempre les pondremos un nombre descriptivo. Así en nuestro ejemplo los cuadros de texto comienzan por txt\_, y los botones por btn\_.

En nuestro programa queremos que el usuario escriba una edad y nos diga en el cuadro de texto inferior (txt\_resultado) si es mayor o menor de edad.

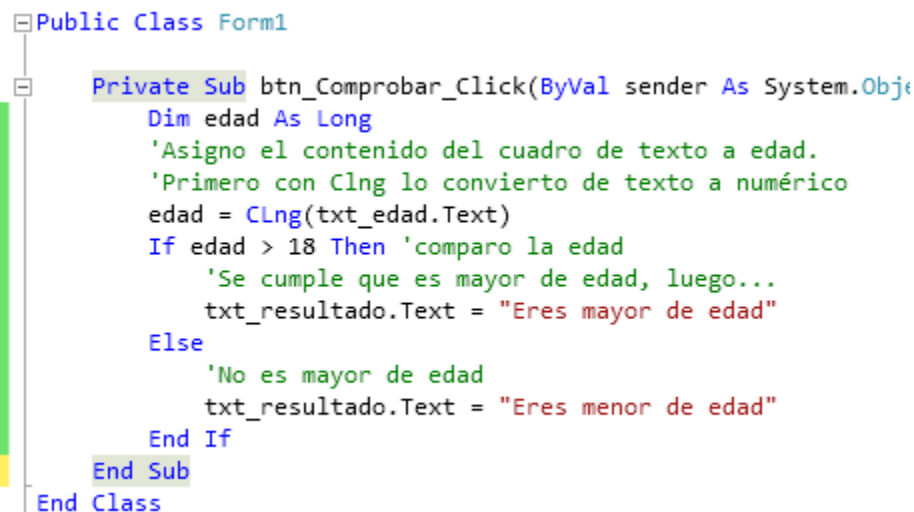
Los pasos a seguir serán:

- Lee un valor por teclado
- Conviértelo a numérico
- Evalúa si es mayor o menor de edad
- Escribe el resultado

Que sería esto:

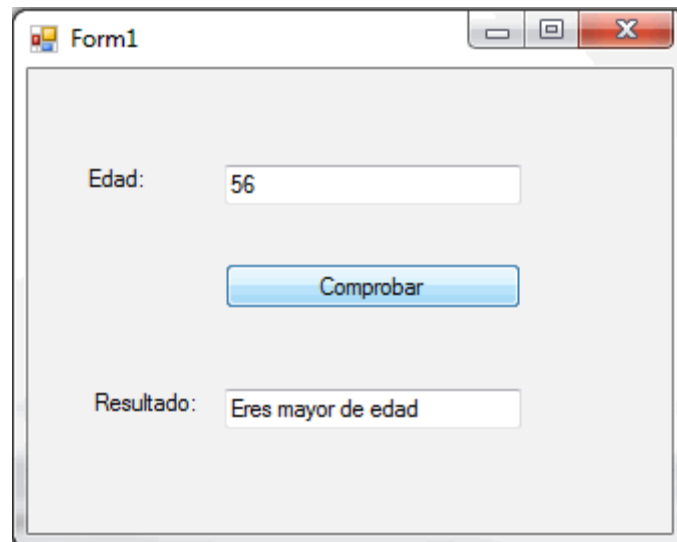
```
Dim edad As Long
'Asigno el contenido del cuadro de texto a edad.
'Primero con CInt lo convierto de texto a numérico
edad = CInt(txt_edad.Text)
If edad > 18 Then 'comparo la edad
    'Se cumple que es mayor de edad, luego...
    txt_resultado.Text = "Eres mayor de edad"
Else
    'No es mayor de edad
    txt_resultado.Text = "Eres menor de edad"
End If
```

Este es el código que debe ejecutarse cuando se haga clic en el botón así que en la pantalla de diseño del formulario haremos doble clic en el botón "Comprobar". Nos creará un procedimiento "Sub" y ahí copiaremos este código:



```
Public Class Form1
    Private Sub btn_Comprobar_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btn_Comprobar.Click
        Dim edad As Long
        'Asigno el contenido del cuadro de texto a edad.
        'Primero con CInt lo convierto de texto a numérico
        edad = CInt(txt_edad.Text)
        If edad > 18 Then 'comparo la edad
            'Se cumple que es mayor de edad, luego...
            txt_resultado.Text = "Eres mayor de edad"
        Else
            'No es mayor de edad
            txt_resultado.Text = "Eres menor de edad"
        End If
    End Sub
End Class
```

Primero vamos a ejecutar el programa entero sin hacer nada para depuración. Pulsamos la tecla F5 o el icono de inicio, al cabo de unos segundos:



Podremos ver que nuestro programa funciona. Compruébalo con varios valores de la edad. Si has escrito un valor alfanumérico que no está contemplado (por ejemplo la palabra "pepe") nuestro programa se interrumpirá y mostrará un mensaje de error (o lo que se llama que se ha producido una excepción). Esta fase es la de control de errores que todavía no hemos visto. Sigamos con nuestro programa.

Ahora vamos a ejecutarlo paso a paso, pulsemos la tecla F11 como antes. En este momento se presentará el formulario. Introducimos el valor 20 y veamos por donde sigue: vaya esto no funciona, no ha depurado nada, se ha ejecutado sin hacerme ni caso.

Veamos la explicación: ¿cuándo se ejecuta el código? en el evento clic del botón, no es una ejecución secuencial (instrucción una detrás de otra) como antes, sino que ahora determinados procedimientos se ejecutarán sólo al producirse unos eventos como es este caso. Para solucionar esto rápidamente conozcamos lo que nos ofrece el IDE para detener la ejecución de un programa cuando nosotros queremos. Ahora queremos que se ejecute al entrar en este procedimiento así que de alguna forma tenemos que ponerle un "**punto de interrupción**", es decir, en este punto te paras para pasar a modo de depuración y así poder ejecutar las líneas una a una. Para hacer esto pulsaremos con el ratón en la línea que queremos que se pare, al hacerlo se pondrá un círculo rojo en la parte izquierda del código y cambiará el color de la línea:

```
Public Class Form1
    Private Sub btn_Comprobar_Click(ByVal sender As System.Object,
        Dim edad As Long
        'Asigno el contenido del cuadro de texto a edad.
        'Primero con CInt lo convierto de texto a numérico
        edad = CInt(txt_edad.Text)
        If edad > 18 Then 'comparo la edad
            'Se cumple que es mayor de edad, luego...
            txt_resultado.Text = "Eres mayor de edad"
        Else
            'No es mayor de edad
            txt_resultado.Text = "Eres menor de edad"
        End If
    End Sub
End Class
```

Esto significa que se va a ejecutar el programa con normalidad hasta que llegue a este punto. ¿Cuándo pasará por aquí? Cuando el usuario pulse en el botón "Comprobar". Vamos a verlo, ejecuta el programa con normalidad y escribe un valor, a continuación pulsamos en el botón comprobar y:

```
Public Class Form1
    Private Sub btn_Comprobar_Click(ByVal sender As System.Object,
        Dim edad As Long
        'Asigno el contenido del cuadro de texto a edad.
        'Primero con CInt lo convierto de texto a numérico
        edad = CInt(txt_edad.Text)
        If edad > 18 Then 'comparo la edad
            'Se cumple que es mayor de edad, luego...
            txt_resultado.Text = "Eres mayor de edad"
        Else
            'No es mayor de edad
            txt_resultado.Text = "Eres menor de edad"
        End If
    End Sub
End Class
```

Esta vez sí se ha detenido justo donde queríamos. Podemos seguir ejecutando el programa con F11 hasta el final, observando la depuración.

Los puntos de interrupción tienen más opciones que ya veremos más adelante.

## 8. Más sobre variables

Después de este pequeño inciso para aprender cómo se realiza la depuración y ejecución paso a paso de los programas vamos a seguir viendo algunos detalles más de declaración de variables.

### 8.1. Declarar varias variables en una misma línea

Podemos declarar varias variables en una misma línea. Si recuerdas las anteriores declaraciones las realizábamos en líneas distintas:

```
Dim valor as Long
Dim edad as Long
Dim nombre as String
```

Podemos simplificar un poco esta definición y declarar dos variables del tipo Long en una misma línea:

```
Dim a, b As Long
```

O de esta forma:

```
Dim c As Long, d As Long
```

Pero obviamente es más sencilla la primera porque sólo indicamos el tipo de datos una sola vez.

### 8.2. Declarar varios tipos de variables en una misma línea

Si queremos declarar variables de distinto tipo en una misma línea tenemos que indicar junto a cada variable el tipo de datos que queramos que tenga:

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

En este caso, tenemos dos variables de dos tipos distintos, cada una con su **As tipo** correspondiente, pero separadas por una coma. Y por fin una declaración múltiple con distintos tipos de datos:

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

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

### 8.3. Tipo de dato por defecto de las variables

Sabemos por los comentarios anteriores que podemos declarar una variable sin decirle el tipo de datos que es, por ejemplo.

```
Dim edad
```

Cuando no se define el tipo de datos de una variable se dice que es de tipo **Object**.

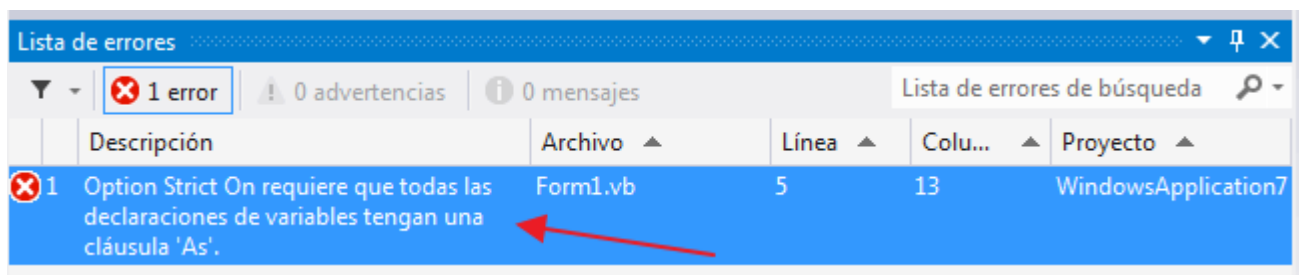
Veamos cómo actuaría VB.NET en el caso de que hagamos algo como esto:

```
Dim z
```

Si no hemos puesto o activado el **Option Strict On**, esa declaración sería válida y el tipo de datos de la variable **z** sería **Object**, es decir, sería lo mismo que hacer esto:

```
Dim z As Object
```

Recuerda que el Option Strict nos obligaba a indicarle de qué tipo de datos es la variable, y, en este caso, no se lo estamos diciendo pero como está desactivado no pasa nada. Si tenemos activado el “Option Strict” con “On”, VB.NET nos indicará que no está permitida esa declaración y nos mostrará un mensaje como:



Sigamos con la sintaxis en las declaraciones múltiples... Una cosa que no está permitida al declarar varias variables usando sólo un As Tipo, es la asignación de un valor predeterminado. Ya vimos que podíamos hacer esto para asignar el valor 15 a la variable N:

```
Dim n As Integer = 15
```

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

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

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

Por tanto, deberíamos hacerlo de esta otra forma:

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

O de esta otra:

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

Aunque esto otro sí que podemos hacerlo:

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



Es decir, si asignamos un valor al declarar una variable, éste debe estar "explícitamente" declarado con un **As Tipo = valor**. Por tanto, esto otro también se puede hacer:

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

Ya que las variables n5 y n6 se declaran con un tipo de datos, pero no se asigna un valor inicial. Por supuesto, los tipos usados no tienen por qué ser el mismo:

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

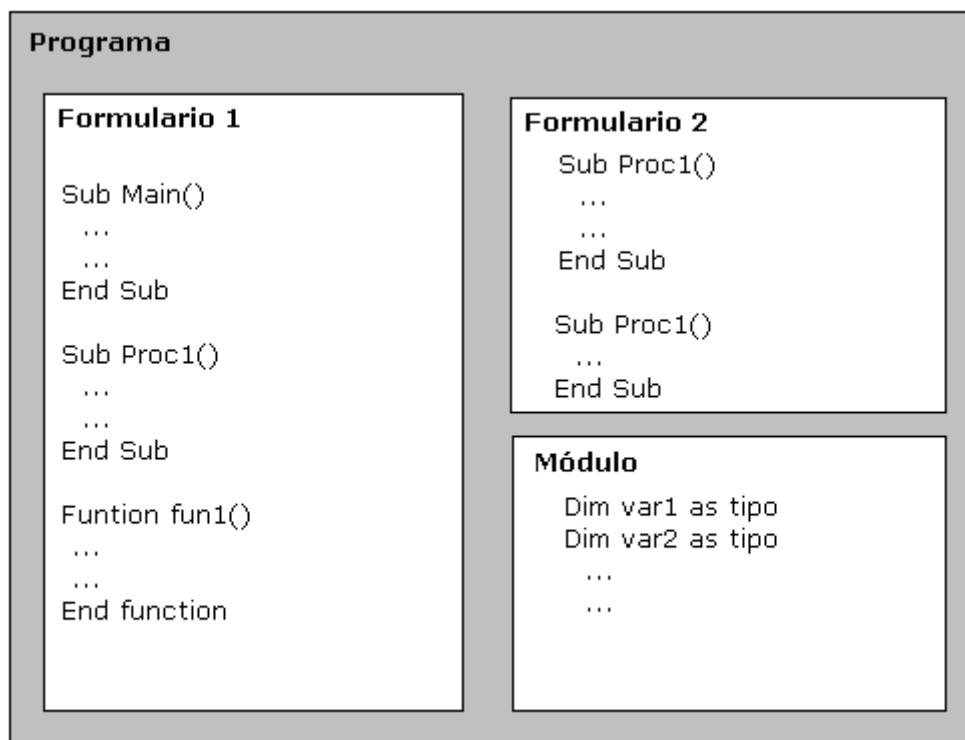
Pero... la recomendación es que no debemos complicarnos con las declaraciones de las variables de esa forma... Procura usar **Dims** diferentes para diferentes declaraciones. Si preferimos declarar cada variable con un Dim, al menos deberíamos utilizar un Dim para cada tipo de variable. Por ejemplo, el último ejemplo quedaría más legible de esta otra forma:

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

Además de que es más "legible", es más fácil de comprobar y tenemos sitio a la derecha de las declaraciones para escribir un comentario explicativo.

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

Observa un poco este esquema:



Es una forma muy simplificada de representar un programa, donde vemos que se compone de dos formularios y un módulo. Los formularios tienen a su vez unos procedimientos que son, a su vez, digamos subprogramas que realizan una determinada acción o proceso. En el primer formulario vemos cómo hay un procedimiento principal `Sub Main()`. Finalmente el módulo se encargará de declaraciones y de procedimientos globales que se utilizarán en todos los formularios.

Aunque los procedimientos los veremos enseguida, son junto con las funciones, fragmentos de código que realizan por sí mismos una acción o proceso. Nunca tendremos un programa como los ejemplos anteriores con un sólo procedimiento, que si recuerdas era el del evento clic del botón de "Comprobar". Un programa se divide en pequeñas rutinas de código formadas por una serie de líneas de código. Al construir el programa, éste irá llamando a estas rutinas para que hagan cada una su parte y produzcan el resultado. Pero claro, estos procedimientos pueden utilizar y de hecho lo hacen, sus propias variables para realizar sus cálculos.

Con esto, y viendo el gráfico anterior, y ya conociendo las variables nos planteamos una serie de preguntas:

Si declaro una variable... ¿la puedo utilizar en todo el programa?. ¿Qué relación tienen las variables con los módulos? ¿Cómo hago para que una variable sea "visible" por todos los procedimientos? ... Es decir, nos estamos planteando cuál es el alcance o visibilidad de una variable.

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

Veamos otra vez lo de los procedimientos y módulos para entenderlo mejor. Cuando creamos antes una aplicación de consola veíamos que el código que se crea es el siguiente: (utilizaremos una aplicación de consola porque se ve más sencillo para entenderlo, luego lo veremos con aplicaciones Windows)

```
Module Module1
    Sub Main()
    End Sub
End Module
```

Es decir, se crea un módulo llamado `Module1` y un procedimiento llamado `Sub Main`, que por otro lado, es el que sirve como punto de entrada al programa. En el ejemplo anterior tenemos dos "espacios" diferentes en los que poder insertar las declaraciones de las variables:

- Después de `Module Module1`
- Después de `Sub Main`.

El código insertado entre **Module** y **End Module**, se dice que es código a nivel de módulo. En este caso el único código posible es el de declaraciones de variables o procedimientos. Es decir, es el programa principal y sólo se pueden declarar procedimientos y funciones, no podemos escribir código libremente porque éste debe ir dentro de los procedimientos.

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

Si declaramos una variable a nivel de módulo, dicha variable estará disponible en "todo" el módulo, incluido los procedimientos que pudiera haber dentro del módulo. Por otro lado, las variables declaradas dentro de un procedimiento, sólo serán **visibles** dentro de ese procedimiento, es decir, que fuera del procedimiento no se tiene conocimiento de la existencia de dichas variables. Es decir, si declaro una variable justo después de Module y antes del Sub Main esa variable estará disponible en todo el programa, es decir "visible" por todos los procedimientos. Si la declaro dentro del Sub sólo estará disponible o visible para ese procedimiento o Sub. En nuestros programas utilizaremos estos dos tipos de variables. Veamos un pequeño ejemplo con dos procedimientos:

```
Option Strict On
```

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

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

    Sub Prueba()
        ' Esto mostrará que n vale 15
        Console.WriteLine("El valor de n es: {0}", n)

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

Este módulo contiene dos procedimientos de tipo Sub (hay otro tipo que es el Function). La variable **n** la hemos declarado a nivel de módulo, por tanto, estará visible en todo el módulo y por todos los procedimientos de ese módulo.

Dentro del procedimiento **Main**, hemos declarado la variable **i**, ésta sólo estará disponible dentro de ese procedimiento. Por eso al intentar usarla en el procedimiento **Prueba**, VB.NET nos da un error diciendo que la variable no está declarada. Y a pesar de que está declarada, lo está pero sólo para lo que haya dentro del procedimiento Main, por tanto, en Prueba no se sabe que existe una variable llamada **i**.

### 8.4.1. Variables que ocultan a otras variables

Veamos primero el código:

```
Option Strict On

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

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

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

Vemos una pequeña incoherencia porque la variable "n" está declarada tanto a nivel global como de procedimiento. Este ejemplo sirve para comprobar que una variable de nivel "superior" puede ser eclipsada por otra de un nivel "inferior".

La variable n declarada a nivel de módulo estará visible en todos los procedimientos del módulo, pero al declarar otra variable, también llamada n, dentro del procedimiento Prueba, ésta última "eclipsa" a la variable que se declaró a nivel de módulo y se usa la variable "local" en lugar de la "global". Aunque sólo dentro del procedimiento Prueba, ya que en el procedimiento Main sí que se ve el valor de la variable declarada a nivel de módulo, por la sencilla razón de que no hay ninguna variable declarada dentro de Main con el mismo nombre.

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

Y de momento es suficiente para entender esta parte del alcance de las variables. Más adelante seguiremos viendo el alcance de las variables.

## 8.5. Pasos por valor y referencia

### 8.5.1. Tipos de datos por valor

Cuando declaramos una variable, por ejemplo, de tipo Integer, el CLR reserva el espacio de memoria que necesita para almacenar un valor del tipo indicado. Cuando a esa variable le asignamos un valor, el CLR almacena dicho valor en la posición de memoria que reservó para esa variable. Si posteriormente asignamos un nuevo valor a dicha variable, ese valor se almacena también en la memoria, sustituyendo al anterior.

Si a continuación creamos una segunda variable y le asignamos el valor que contenía la primera, tendremos dos posiciones de memoria con dos valores, que por pura casualidad, resulta que son iguales, entre otras cosas porque a la segunda le hemos asignado un valor "igual" al que tenía la primera.

### 8.5.2. Tipos de datos por referencia

En Visual Basic .NET también podemos crear objetos que se crean a partir de una clase, (el tema de las clases lo veremos ampliamente en temas posteriores). Cuando declaramos una variable cuyo tipo es una clase, simplemente estamos creando una variable que es capaz de manipular un objeto de ese tipo. En el momento de la declaración, el CLR no reserva espacio para el objeto que contendrá esa variable, ya que esto sólo lo hace cuando usamos la instrucción New.

En el momento en que creamos el objeto, (mediante New), es cuando el CLR reserva la memoria para dicho objeto y le dice a la variable en qué parte de la memoria está almacenado, de forma que la variable pueda acceder al objeto que se ha creado. Si posteriormente declaramos otra variable del mismo tipo que la primera, tendremos dos variables que saben "manejar" datos de ese tipo, pero si a la segunda variable le asignamos el contenido de la primera, en la memoria no existirán dos copias de ese objeto, sólo existirá un objeto que estará referenciado por dos variables. Por tanto, cualquier cambio que se haga en dicho objeto se reflejará en ambas variables.

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

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

```
Sub Main()  
    ' creamos una variable de tipo Integer  
    Dim i As Integer  
    ' le asignamos un valor  
    i = 15  
    ' mostramos el valor de i  
    Console.WriteLine("i vale {0}", i)  
End Sub
```

```
' creamos otra variable
Dim j As Integer
' le asignamos el valor de i
j = i
Console.WriteLine("hacemos esta asignación: j = i")
'
' mostramos cuánto contienen las variables
Console.WriteLine("i vale {0} y j vale {1}", i, j)
'
' cambiamos el valor de i
i = 25
Console.WriteLine("hacemos esta asignación: i = 25")
' mostramos nuevamente los valores
Console.WriteLine("i vale {0} y j vale {1}", i, j)
'
Console.WriteLine("Pulsa Intro para finalizar")
Console.ReadLine()
End Sub
```

Como podemos comprobar, cada variable tiene un valor independiente del otro. Esto está claro.

Ahora vamos a ver qué es lo que pasa con los tipos por referencia. Para el siguiente ejemplo, vamos a crear una clase con una sola propiedad, ya que las clases a diferencia de los tipos por valor, deben tener propiedades a las que asignarles algún valor.

```
Class prueba
    Public Nombre As String
End Class

Sub Main()
    ' creamos una variable de tipo prueba
    Dim a As prueba
    ' creamos (instanciamos) el objeto en memoria
    a = New prueba()
    ' le asignamos un valor
    a.Nombre = "hola"
    ' mostramos el contenido de a
    Console.WriteLine("a vale {0}", a.Nombre)
    '
    ' dimensionamos otra variable
    Dim b As prueba
    '
    ' asignamos a la nueva el valor de a
    b = a
    Console.WriteLine("hacemos esta asignación: b = a")
    '
    ' mostramos el contenido de las dos
    Console.WriteLine("a vale {0} y b vale {1}", a.Nombre, b.Nombre)
    ' cambiamos el valor de la anterior
    a.Nombre = "adios"
    '
    Console.WriteLine("hacemos una nueva asignación a a.Nombre")
    '
End Sub
```

```
' mostramos nuevamente los valores
Console.WriteLine("a vale {0} y b vale {1}", a.Nombre, b.Nombre)
'
Console.WriteLine("Pulsa Intro para finalizar")
Console.ReadLine()
End Sub
```

La clase **prueba** es una clase muy simple, pero como para tratar de los tipos por referencia necesitamos una clase, he preferido usar una creada por nosotros que cualquiera de las clases que el .NET Framework nos ofrece.

Declaramos una variable de ese tipo y después creamos un nuevo objeto del tipo **prueba**, el cual asignamos a la variable **a**.

Una vez que tenemos "instanciado" (o creado) el objeto al que hace referencia la variable **a**, le asignamos a la propiedad **Nombre** de dicho objeto un valor. Lo siguiente que hacemos es declarar otra variable del tipo **prueba** y le asignamos lo que contiene la primera variable.

Hasta aquí, es casi lo mismo que hicimos anteriormente con las variables de tipo Integer. La única diferencia es la forma de manipular las clases, ya que no podemos usarlas "directamente", porque tenemos que crearlas (mediante New) y asignar el valor a una de las propiedades que dicha clase contenga. Esta es la primera diferencia entre los tipos por valor y los tipos por referencia, pero no es lo que queríamos comprobar, así que sigamos con la explicación del código mostrado.

Cuando mostramos el contenido de la propiedad **Nombre** de ambas variables, las dos muestran lo mismo, que es lo esperado, pero cuando asignamos un nuevo valor a la variable **a**, al volver a mostrar los valores de las dos variables, ¡las dos siguen mostrando lo mismo! Y esto no es lo que ocurría en el primer ejemplo.

¿Por qué ocurre? Porque las dos variables, apuntan al mismo objeto que está creado en la memoria.

**¡Las dos variables hacen referencia al mismo objeto!** y cuando se realiza un cambio mediante una variable, ese cambio afecta también a la otra, por la sencilla razón que se está modificando el único objeto de ese tipo que hay creado en la memoria.

## 9. Flujo de programa. If then Else

Vistos estos detalles sigamos con nuestro flujo de programa terminando con el de ruptura de secuencia o "If Then". Para volver a repetir los casos del If ...then ... else resumiremos otra vez su sintaxis y posibilidades:

Estas estructuras de control contienen instrucciones que se ejecutarán dependiendo del resultado obtenido al evaluar una expresión asociada a la estructura, es decir, una bifurcación. Casos posibles:

### 9.1. Decisión simple

```
If expresión Then
    'código
    '...
    '...
End If
```

Es muy aconsejable escribir el código dentro del If then con un sangrado, es decir, un nivel más a la derecha de tabulación, de esta forma el código es mucho más legible. A continuación de la evaluación escribiremos las líneas de código necesarias. Veamos un ejemplo:

```
If mivariable=20 then
    Console.WriteLine ("La variable tiene el valor 20")
End if
```

### 9.2. Decisión simple en una línea

```
If expresión Then Instrucción
```

Es decir:

```
If mivariable=20 then mivariable=20 * 3
```

En este caso puesto que sólo vamos a ejecutar una instrucción se puede poner en una sola línea. Aquí no tendríamos que escribir el "End If"

### 9.3. Decisión doble

```
If expresión then
    'código cuando es cierto
    '...
else
    'código cuando es falso
    '...
End If
```



Esto ya lo vimos antes, ejecutará las primeras instrucciones cuando se cumple la condición de evaluación y las otras cuando no:

```
If edad<18 then
    Console.WriteLine ("Eres menor de edad")
else
    Console.WriteLine ("Eres mayor de edad")
Endif
```

## 9.4. Doble decisión en una línea

If Expresión Then InstrucciónSiCerto Else InstrucciónSiFalso

Es una forma simplificada de escribirlo siempre y cuando sólo necesitemos ejecutar una operación sencilla, si por el contrario necesitamos escribir más de una línea de código (que será la mayoría de las veces) no nos valdrá y utilizaremos la anterior.

## 9.5. Decisión múltiple

```
If expresión then
    'código cuando se cumpla
    '...
elseif expresión then
    'código cuando se cumpla
    '...
else
    'código para los demás casos.
End If
```

Este caso es muy utilizado, nos hará falta cuando tengamos que evaluar más de dos opciones por ejemplo:

```
If valor>0 then
    Console.WriteLine ("Valor es mayor que cero")
elseif valor<0 then
    Console.WriteLine ("Valor es menor que cero")
else
    Console.WriteLine ("Valor igual a cero")
Endif
```

El ejemplo está muy claro queremos evaluar si un valor es mayor que cero, menor o si es cero. Tenemos que evaluar pues tres posibilidades: el ELSEIF nos permite seguir haciendo evaluaciones para bifurcar o no según se cumpla.

Este tipo de bifurcaciones no se limita a un sólo Elself, podríamos seguir haciendo más evaluaciones.

## 9.6. Utilizar más de un comparador: AndAlso y OrAlso

Recordamos también que podemos utilizar más de un comparador en las expresiones a evaluar

Cuando hacemos una comparación usando AND, VB comprueba si las dos expresiones usadas con ese operador son ciertas:

```
IF n = 3 AND x > 10 THEN
```

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

Pero podría mejorarse. Por ejemplo, si la primera no se cumple la expresión debería devolver ya un "False" y no evaluar la siguiente condición. Esto es una buena mejora.

Veamos un ejemplo. Estoy recorriendo una tabla de una base de datos (tabla: información organizada, por ejemplo, los empleados de una empresa) y quiero sacar todos los que sean de Logroño. Mi bucle va a hacer lo siguiente:

```
Mientras (No se termine el fichero) Y (población="Logroño") Haz  
    'envía cesta de navidad  
Fin del bucle
```

Es un bucle muy típico para recorrer un fichero y ver todas las fichas o registros que cumplan una condición. Se puede mejorar en lo siguiente: Supón que hemos llegado al final del fichero, esta instrucción nos va a dar un error porque como ya se ha terminado el fichero no puede hacer la comparación de la población porque no existe ningún registro. Entonces nuestro programa provocaría un error.

Esto nos hace desear una variante del AND para que evalúe la segunda expresión sólo si lo necesita, para esto utilizaremos **AndAlso**. El caso anterior se debía partir en dos instrucciones IF para que funcionase bien: una comprueba que no se ha llegado al final del fichero y si es correcta otro If comprueba si la población cumple la condición.

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

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

```
IF n = 3 ANDALSO x > 10 THEN
```

Y el segundo caso

```
Mientras (No se termine el fichero) AND ALSO (población="Logroño") Haz
...
Fin del bucle
```

Parecido ocurre con OR, aunque en este caso, sabemos que si el resultado de cualquiera de las expresiones usadas con ese operador es verdadero, la expresión completa se da por buena. Por ejemplo, si tenemos:

```
IF n = 3 OR x > 10 THEN
```

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

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

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

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

```
IF n = 3 OR ELSE x > 10 THEN
```

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

## 9.7. Prioridad de los operadores

Ya comentamos antes que los operadores tienen una prioridad cuando se ejecutan o evalúan en una expresión, recuerda que tuvimos que poner paréntesis además para agrupar las operaciones o comparaciones. Hay varios tipos de operadores: aritméticos, de comparación y lógicos que veremos por separado. Algunos no los hemos visto todavía. Esto que vemos ahora es para el caso de que una línea de código contenga varias operaciones y muestra el orden de cómo se ejecutarán:

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

- Exponenciación (^)
- Negación (-)

- Multiplicación y división (\*, /)
- División de números enteros (\)
- Módulo aritmético (Mod)
- Suma y resta (+, -)
- Concatenación de cadenas (&)

### Operadores de comparación:

- Igualdad (=)
- Desigualdad (<>)
- Menor o mayor que (<, >)
- Mayor o igual que (>=)
- Menor o igual que (<=)

### Operadores lógicos:

- Negación (Not)
- Conjunción (And, AndAlso)
- Disyunción (Or, OrElse, Xor)

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

$$X = 100 - 25 - 3$$

El resultado será diferente de esta otra:

$$X = 100 - (25 - 3)$$

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

## 10. Select Case

En muchas ocasiones tendremos que realizar una comparación o evaluación de valores del tipo If..Then pero puede llegar a ser muy engorroso si son muchos los valores. Por ejemplo, queremos asignar a una variable de tipo string el literal correspondiente al mes del año:

```
If variable_mes=1 then
    cadena_mes="Enero"
elseif variable_mes=2 then
    cadena_mes="Febrero"
elseif variable_mes=3 then
    cadena_mes="Marzo"
elseif variable_mes=4 then
    cadena_mes="Abril"
elseif variable_mes=5 then
    cadena_mes="May"
elseif variable_mes=6 then
    cadena_mes="Junio"
elseif variable_mes=7 then
    cadena_mes="Julio"
elseif variable_mes=8 then
    cadena_mes="Agosto"
elseif variable_mes=9 then
    cadena_mes="Septiembre"
elseif variable_mes=10 then
    cadena_mes="Octubre"
elseif variable_mes=11 then
    cadena_mes="Noviembre"
else variable_mes=12
    cadena_mes="Diciembre"
end if
```

Que como ves es bastante incómodo de escribir y eso que este ejemplo es muy sencillo. Veamos en funcionamiento la instrucción adecuada para esto:

```
Select Case variable_mes
    Case 1:variable_mes="Enero"
    Case 2:variable_mes="Febrero"
    Case 3:variable_mes="Marzo"
    ...
End Select
```

Así queda mucho más elegante y más claro, además su uso se justifica con sólo tener 3 posibilidades de evaluación.

Luego en casos de decisión múltiple utilizaremos este sistema.

Tiene varias opciones más ya que podemos enumerar los que queramos que cumplan la condición, por ejemplo:

```
select case valor
  case 1: ...
  case 2,3,4:...
  case 5,6,7:...
  case 8:
end select
```

Este ejemplo hubiera sido muy costoso escribirlo con los IF. Sería así:

```
If valor=1 Then
  'instrucciones...
Elseif (valor=2) Or (valor=3) Or (valor=4) Then
  'instrucciones
Elseif (valor=5) Or (valor=6) Or (valor=7) Then
  'instrucciones
Else
  'instrucciones
Endif
```

Pero podemos poner todavía más evaluaciones, por ejemplo un "Case Else" al final le dice lo que tiene que hacer en los demás valores. Por ejemplo, para ver los días que tiene cada mes del año según su índice:

```
select case mes:
  case 1,3,5,7,8,10,12
    dias=31
  case 4,6,9,11
    dias=30
  case else
    dias=28
end if
```

Observa un detalle en la sintaxis... si escribimos más de una línea de código cuando cumpla la condición las escribiremos debajo de la comparación como en el ejemplo anterior. Pero si sólo vamos a ejecutar una instrucción la podemos poner a continuación de la evaluación separada por "·":

```
case 23: instrucción
```

La lista de expresiones asociada a cada Case está, por lo tanto, separada por comas y puede tener alguno de estos formatos:

- Expresión
- Expresión Mayor-Menor
- Is comparador

Mira este ejemplo final para ver todas sus posibilidades, es realmente útil y sencillo de utilizar:

```
Select Case Variable:
  Case 1:
    'instrucciones si vale 1
  Case 30,50:
    'instrucciones si vale 30 ó 50
  Case Is < 10
    'instrucciones si es menor que 10
  Case 200 to 300
    'instrucciones si está entre 200 y 300
  Case 80 to 90,92, is >100
    'instrucciones si está entre 80 y 90 ó es el 91 o es mayor
que 100.
  Case Else
    'instrucciones en los demás casos
End Select
```

Otro ejemplo de cómo sería el código utilizando los If-Then y su equivalente en Select-Case:

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

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

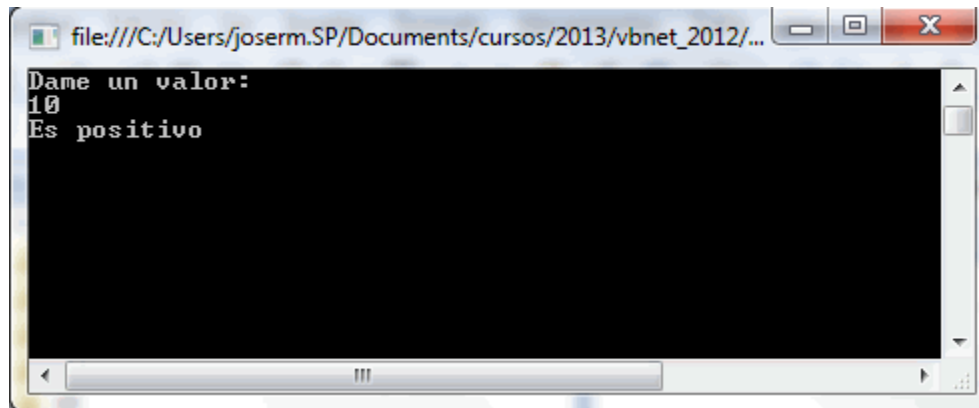
```
Select Case i
  Case 3
    '
  Case 6 To 11
    '
  Case 14, 17
    '
  Case Is > 25
    '
  Case Else
    '
End Select
```

**Nota:** No todo es perfecto con el Select-Case: su pega es que sólo puede evaluar una expresión. Los If-Then pueden evaluar varias simultáneamente con los operadores lógicos AND, OR, NOT y XOR.

## Ejercicios

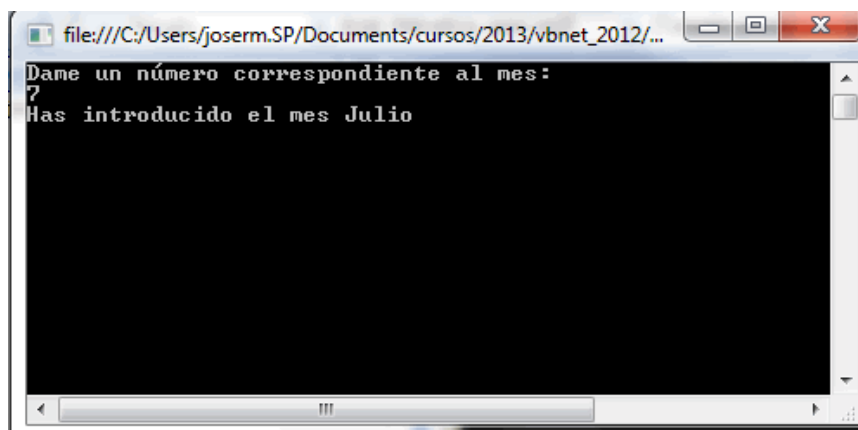
### Ejercicio 1

Crea una aplicación de consola que pida un valor y me diga si es positivo, negativo o cero



### Ejercicio 2

Haz una aplicación de consola que pida un número de mes y nos diga el literal del mes y en caso de ser mayor que 12 nos lo indique con un mensaje. Utiliza el Select Case



### Ejercicio 3

Crea una aplicación de Windows. Define una estructura:

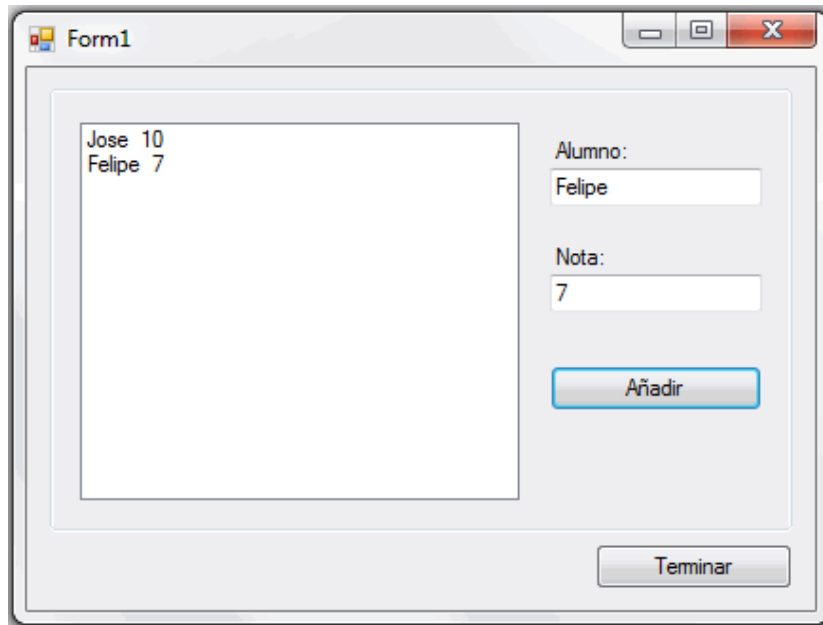
Alumno

Nombre de tipo String

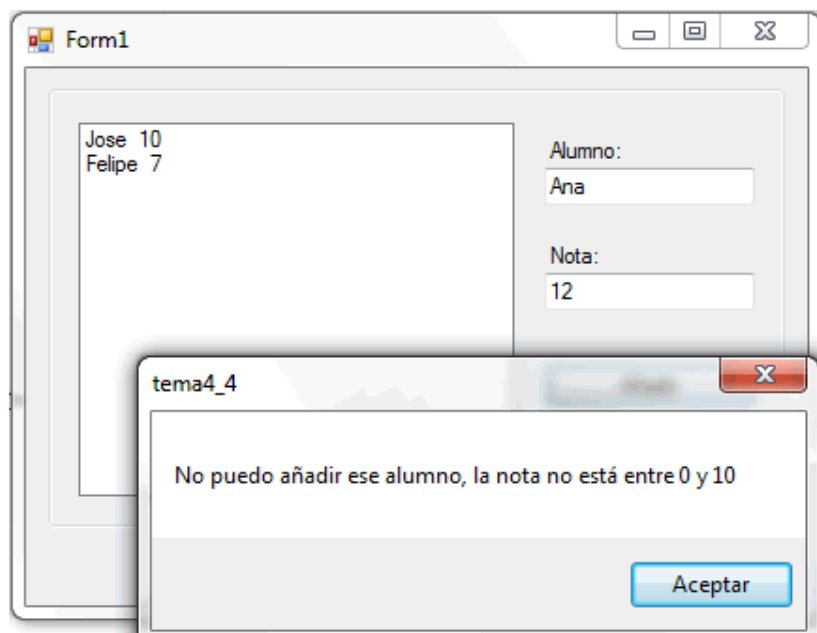
Nota de tipo real



Pon unos cuadros de texto para leer los valores e introduce la pareja de valores en un cuadro de lista sólo si los valores no están vacíos.



Además debemos comprobar el valor leído para que la nota esté entre los valores 0-10:



**Nota:** Utiliza el msgbox para escribir un mensaje y una función de conversión para el valor leído en el cuadro de texto de la nota.

Por ejemplo:

`CType(Me.txt_nota.Text, Decimal)` para asignarlo al campo nota de la estructura del alumno.